COMP 550: Algorithms & Analysis - Preliminary Notes

What is the goal of this course?

→ Improve knowledge & skills in the field of algorithms

→ Writing & using algorithms

→ Mathematically analyzing correctness & running time of algorithms.

→ problem-solving paradigms (eg, greedy vs dynamic)

→ computational complexity.

What is the "running time" of an algorithm?

→ The maximum number (as a function of input length, "n") of primitive operations that it executes.

- e.g., a $T(n)$-time algorithm, given an input of length n, executes at most $T(n)$ primitive operations on that input.

What is a primitive operation?

→ Operations (in the execution of code) that are relatively "simple" & can be executed in very small amount of time (?)

→ Rule of thumb: Pretty much any op. that can be written as one line of assembly code (RECALL: COMP 211 !!) is a primitive operation.

Examples of primitive operations?

→	Assigning a value to a variable	$x = 0$
→	performing a comparison	if $x > y$:
→	arithmetic operations	$x + y$
→	Indexing into an array	arr[3]

→ Calling or returning from a method (not necessarily performing the method itself)

RECALL COMP 455: How is running time defined?

→ The running time, aka the **time complexity** of an algorithm, is the number of steps that it takes to solve a problem in the worst case, as a function of the input length.

→ Formal DEFN: For a deterministic, decider TM M, the running time of M is the function $f: N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

- We use n to represent the length of an input (customarily)
- If $f(n)$ is the running time of M, we say that "M is an $f(n)$ Turing Machine" and that "M runs in time $f(n)$"

RECALL: What is asymptotic analysis?

→ A way to estimate the exact running time of an algorithm in order to understand the running time of the algorithm when it is run on large inputs.

→ For the running time expression of an algorithm, consider only the highest-order term (aka term with largest exponent), and disregard the coefficient of that term as well as any lower order terms (b/c they are insignificant in comparison).

- For $\boxed{\text{EX}}$, for the function $f(n) = 6n^3 + 2n^2 + 20n + 45$, we say that f is asymptotically at most n^3

Ch 1: Array Algorithms

Key assumptions?
- → Every array has length n
- → Array indices range from 1 to n (not $0-(n-1)$ like you're used to)

— Max in Array —

What is the input?
- → Array A of n distinct, positive integers

What is the goal?
- → Return the largest integer in A in $O(n)$ time.
- → Example: $A = [5, 1, 4, 10, 8, 3]$

What is the algorithm?

```
m = A[1]
for (2 ≤ i ≤ n):
    if A[i] > m:
        m = A[i]
return m
```

→ m represents the "max value". Start by setting it to $A[1]$

→ iterate through all values and update m by setting $m = A[i]$ whenever $A[i] >$ current m.

What is the running time?
- → Alg loops through for-loop $n-1$ times
  - each iteration takes $O(1)$ time because its only primitive operations
- → Therefore, $RT = (n-1) * (1) = n-1 = O(n)$

— Two - Sum —

What is the input?
- → (A, t), where A = array of n distinct integers sorted in increasing order, and $t \in \mathbb{Z}$ (t is some integer)

What is the goal?
- → Return a set of indices (i, j) s.t.:
  - $A[i] + A[j] = t$, and
  - $i < j$

What is the algorithm?
- → Ex: $A = [1, 3, 4, 7, 9, 10, 12]$ $t = 13$

```
ALG (int[] A, int t):
  i, j = 1, n
  while i < j:
      if A[i] + A[j] = t:
          return (i, j)
      elif A[i] + A[j] < t:
          i += 1
      else:
          j -= 1
```

→ Using a "2-pointer" technique

→ Start with $A[1] + j[n]$ as our first candidate. If $A[1] + j[n] = t$, we can return.

→ If $A[i] + A[j]$ is too small, move i up one to a larger value

→ If $A[i] + A[j]$ is too big, move j down one to a smaller value.

→ Keep checking & adjusting until a solution is found.

What is the RT?
- → The alg makes a maximum of n iterations. Each require $O(1)$ time. Thus, $RT = O(n)$

What would the alg & RT be like if the array wasn't already sorted?
- → Brute-force alg; $O(n^2)$-time

– Binary Search –

What is the input?

→ (A, t), where A = array of $\underline{n}$ distinct integers sorted in increasing order, and $t \in \mathbb{Z}$ (t is some integer)

What is the goal?

→ If it exists, return an index k s.t. $A[k] = t$.

→ Ex: $A = [1, 3, 4, 7, 8, 12, 15]$ $t = 8$ ANS = 5, b/c $A[5] = 8$

What is the algorithm?

```
ALG(A,t):
    i,j = 1,n
    while i ≤ j:
        m = ⌊(i+j)/2⌋
        if A[m] = t:
            return m
        elif A[m] < t:
            i = m+1
        else:
            j = m-1
```

- $i, j = 1, n$ → start w/ a pointer at the beginning & end, like in 2-sum.
- $m = \lfloor (i+j)/2 \rfloor$ → check if $A[m]$ is the value (t) we are looking for, where m = the middle index of A.
- if $A[m] = t$: → if we have found t, return index m.
- elif $A[m] < t$: → if the value @ the middle index is too small, create a new "subarray" of everything in the 'right half' of A (i = pointer at start index, which is now $m+1$).
- $j = m-1$ → if value @ middle ind. is too big, recurse on the <u>left</u> half of the array.

Why is this alg correct?

→ It always "focuses" on some subarray $A[i:j]$. $A[i:j]$ always contains t, and the size of it shrinks in each iteration.

What is the RT?

→ How many iterations of "subarrays" do we recurse through?

→ If subarray $A[i:j]$ has length ℓ at the beginning of some iteration, and length ℓ' at the end of the same iteration, then $\ell' \leq \ell/2$ (since we are "halving" the array each time).

→ For ex, if the alg sets $i = m+1$ in a given iteration:

$$\ell' = j \text{ (index of end of new subarray)} - (m+1 \text{ (new start ind for subarray)}) + 1$$

$$= j - \left\lfloor \frac{i+j}{2} \right\rfloor \leq j - \frac{i+j}{2} + \frac{1}{2} = \frac{j-i+1}{2} = \frac{\ell}{2}$$

→ If $A[i:j]$ initially has length $\underline{n}$, after k iterations, $A[i:j]$ has length at most $n/2^k$. Therefore, the alg has a total of $O(\log n)$ iterations.

- Each iteration requires $O(1)$ time.
- Thus, RT is $O(\log n)$

RT Notes →

- $O(1)$ per iteration
- # of iterations: $j - i = \underbrace{n \to \frac{n}{2} \to \frac{n}{2^2} \to \frac{n}{2^3} \to \ldots \to 1}_{k \text{ iterations}}$

$$\frac{n}{2^k} \leq 1 \Rightarrow n \leq 2^k \Rightarrow k = \log_2(n)$$

– Selection Sort –

(REMEMBER That in theory classes like this one, we say arr[1] instead of arr[0] ... an array w/ n elements has indexes $1-n$ (inclusive), NOT indexes $0-(n-1)$)

What is the input?

→ An array A of n distinct (non-repeating), unsorted integers

What is the goal?

→ Sort the elements of A by increasing value (e.g. $A[1] < A[2] < \dots A[n]$)

→ Ex: $A = [7,2,1,4,3]$ GOAL/ANS: $[1,2,3,4,7]$

What is the algorithm?

```
ALG(A):
  for i = 1; i ≤ n:
    m = i
    for j = (i+1); j ≤ n
      if A[j] < A[m]:
        m = j
    swap(A[i], A[m])
```

→ The algorithm executes $\underline{n}$ rounds, one for each $i \in [n]$

→ For each round, we compare the rest of the values (after index i, hence $j = i+1$) to the current value at $A[m=i]$.

→ If a value at an index $> i$ is smaller than $A[i]$, it needs to be moved forward. Thus, we set $\underline{m} = j$ to indicate which val($A[m]$) should be swapped with $A[i]$.

→ IDEA: Basically, in each iteration, find the smallest element in i through n. Then, swap it with i. Then, increment i & do it again.

What is the RT?

→ In round i, the alg executes at most $c(n-i)$ operations for some $c \in \mathbb{Z}^+$ (pos int)

Summing over all $i \in [n]$, the total # of operations is at most:

$$c\sum_{i=1}^{n}(n-i) = c \cdot ((n-1)+(n-2)+\dots+2+1) = O(n^2)$$

– Merge Sort –

What is the input?
→ An array A of n integers

What is the goal?
→ Sort the elements of A by increasing value (e.g. $A[1] < A[2] < \dots A[n]$)

→ Ex: $A = [2, 4, 8, 5, 1, 7, 6, 3]$

What is the algorithm?
→ IDEA: split A into its 2 halves & recursively "merge sort" each half. Then, merge the 2 halves using a 2-pointer approach.

```
MergeSort(A):
  if n = 1: return A
  k = ⌊n/2⌋
  A_L = MergeSort(A[1:k])
  A_R = MergeSort(A[(k+1):n])
  i, j, B = 1, 1, [empty list]
  while i ≤ k and j ≤ (n-k):
      if A_L[i] ≤ A_R[j]:
          B.append(A_L[i])
          i += 1
      else:
          B.append(A_R[j])
          j += 1
  if i > k:
      append all num in A_R[j:(n-k)] to B
  else:
      append all num in A_L[i:k] to B
  return A = B.as_array()
```

RT?
→ $O(n \log n)$

(Array Algorithms)

1.1 Max in Array

Input: Array A of n distinct, pos integers

Goal: Return the largest int in A

① ALG (algorithm): aka Python pseudocode

② Correctness: Intuition

Proof (Induction hypothesis)

③ Running Time (RT)

English
↓
pseudocode
↓
Python
↓
assembly
↓
machine

① ALG (A):

array indices start at 1 in this class' notation

```
    m = A(1)
for i = 2, ..., n:
    if A(i) > m:
        m = A(i)
return m
```

$m = \max(m, A(i))$

② Correctness: m is always max(A(1 = i))

③ RT = (upper bound on) "# of primitive operations" the ALG makes in the worst case

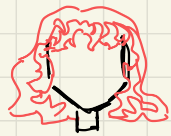
What is a primitive operation?

1. assignment ops, e.g. m = 3
2. comparison (e.g. "if x > 3")
3. arithmetic (y = x + 3)
4. Indexing (m = A(1))
5. Calls/return

→ Pretty much anything that can be done as 1 line of assembly code, is a primitive operation.

Asymptotic Notation

SECTION: Running Time



↳ (shorter) way to write something

$f(n) = 3n^2 + 7n^3 + 2n + 9$

$\downarrow$

$fn = O(n^3)$

11:31

$fn = O(n) \approx f(n) \leq n$

1. Technically, O(n) is a set that includes
   - $4n-1$, $5n$, $2n$, $n/2$, …, $\sqrt{n}$, $10\log n$, 300, …
2. Technical way to write it is $f(n) \in O(n)$, not $=$
3. Almost every RT analysis we'll do is Θ, not O
4.

SECTION: 1.2 Two Sum

Input: Array A of n distinct integers, sorted in increasing order

Ex: $A = \{1, 3, 4, 5, 7, 10, 11\}$, $t = 10$

Goal: Return (i, j) s.t. $i < j$ and $A(i) + A(j) = t$

Brute force: Try every possibility

```
ALG (A,t):
    for i ← 1,...,n-1 :
        for j ← i+1,...,n:
            if A(i) + A(j) = t:
                return (i,j)
```

$RT = O(n^2)$

(usually the case for nested for loops)

10:10

Two - Pointer

Two - Sum (Brute Force): $O(n^2)$

1.3 Binary Search (A, t) ← sorted   (This is "one sum")

Goal: Return k s.t. A(k) = t   (or nothing)

Brute Force: Scan A for T → O(n)

Ex: A = {1, 3, 4, 7, 9, 12, 15}   and   t = 8

1. Check if val of middle num in array is smaller than t

A = {1, 3, 4, 7, 9, 12, 15}
i   ↳ 7 not < 8

2. Then we can move pointer i to the right of the middle num

→ Test middle element, check if ~~2~~ too big or to small, "zoom in"/narrow down the 'subarray' accordingly

→ Binary Search Alg (A, t):

```
i, j = 1, n        # we are focusing on A(i : j)
while (i ≤ j):               j is the highest index in the array
    m (candidate) = ⌊(i+j)/2⌋
    if A(m) = t : return m
    elif A(m) < t :
        i (index!! "m" is index) = m+1
    elif A(m) > t :
        j = m-1
```

Running Time:

- O(1) per iteration
- # of iterations: $j - i = \underbrace{n \to \frac{n}{2} \to \frac{n}{2^2} \to \frac{n}{2^3} \to \ldots \to 1}_{k \text{ iterations}}$

$$\frac{n}{2^k} \le 1 \Rightarrow n \le 2^k \Rightarrow k = \log_2(n)$$

Total Running Time: O(log n)

1.4: Selection Sort

Goal: Sort A (array) (i.e., we want A[1] < A[2] <... < A[n])

$\boxed{\text{Ex}}$ A = [7, 2, 1, 4, 3]   (Goal: [1, 2, 3, 4, 7])

1. Lets find the smallest element & put it in the correct spot (e.g. next available?)

   1. swap 7 with 1   [1, 2, 7, 4, 3]
   2. swap 2 with 2   [1, 2, 7, 4, 3]
   3. swap 3 with 7   [1, 2, 3, 4, 7]
   4. swap 7 w/ 7   [1, 2, 3, 4, 7]

→ In each iteration, find the smallest element in i thru n, then swap it w/ i. Then, move i up by 1 (i = i + 1)

* Looking for the smallest element from index i to index m

ALG(A):

For i = 1, ..., n (n−1):
    m = i
    For j = i+1, ..., n:
        if A[j] < A[m]:
            m = j

$\left.\right\}$ "m = index of min(A[i:n])"

Running Time: similar to Brute Force (BT) of Two-Sum

n iterations, each takes O(n) time ⟶ **ANS: O(n²)**

(or O(n−i) but same thing)

Insertion Sort: O(n²) worst case, O(n) if A is sorted

1.5 : Merge Sort (A)

-- 1 of 2 recursive algorithms we will study. Other is DFS

Ex: A = [2, 4, 8, 5, 1, 7, 6, 3]

[2, 4, 5, 8] [1, 3, 6, 7] → got both halves sorted, now have to "merge"

i j

[1, 2

→ Going one by one thru each array and taking out the smaller value?

→ Merge takes O(n) time.

ALG (A):

if $n = 1$: return A

$k = \lfloor n/2 \rfloor$

A_L = ALG (A [1 : k])

A_R = ALG (A [k+1 : n])

i, j, B = 1, 1, empty list

index in left half arr

index in right half arr

while $i \le k$ and $j \le n-k$:

    if $A_L[i] \le A_R[j]$:

        append $A_L[i]$ to B

        i += 1

    else :

        append $A_R[j]$ to B

        j += 1

merging

if $i > k$:

    append each element of $A_R[j : n-k]$ to B

    (For el in $A_r[j : n-k]$: append el to B)

else :

    " but for A_L ... $A_L[i : k]$ to B

A = B (convert B from list to array)

return A

adding the rest

List vs Array

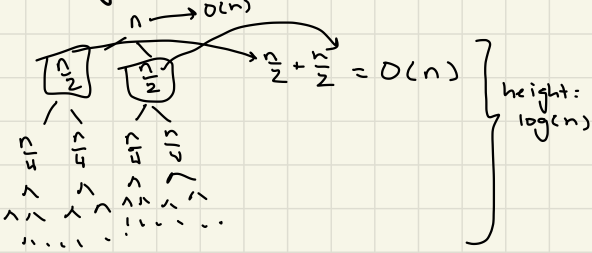
→ list : append in O(1) time

- cannot index

→ array : create in O(n) time

- cannot append
- can index in O(1) time

Running Time:



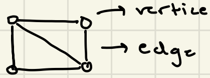
Total : O(n logn) ... "adding the overheads"

Takeaway:

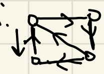
Complexity is O(n logn)

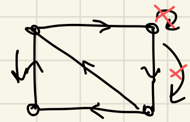
Ch.2: Essential Graph Algorithms

Undirected graph:



Directed graph:

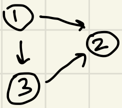




• We ignore repeat & parallel same-dir edges

→ To convert undirected to directed, just draw 2 edges in both directions, for every edge on the undirected graph

→ $G(V,E)$ where V = set of vertices $n = |V|$ and E = set of edges $m = |E|$

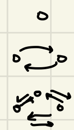


$V = \{1,2,3\}$

$E = \{(1,2),(1,3),(3,2)\}$

What is largest possible value of m?
(m = edges, v = vertices) (for a directed graph)

n	max m
1	0
2	2
3	6



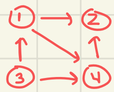
total: $\boxed{n(n-1)}$

• For undirected graph: $\boxed{\frac{n(n-1)}{2}}$ $\Big\} = O(n^2)$

In Programming, representing graphs:

1. adjacency list representation

Ex.



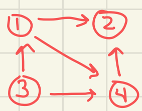
• an array
• G[1] = list of 1's "out-neighbors" (2 and 4)
• G[2] = list of 2's "out-neighbors" (none)
• G = [[2,4], [], [1,4], [2]]

→ Array: create in $O(n)$ time, index in $O(1)$ time, & can't append
→ (Linked) List: create AND append to either end in $O(1)$ time; cannot index
→ Adjacency List is technically an array of pointers to lists. This isn't important though. "Array of lists"

2. Adjacency Matrix

$$\begin{array}{r c} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ G = G(1) & [0, 1, 0, 0 \\ G(2) & [0, 0, 0, 0] \\ G(3) & [1, 0, 0, 1] \\ G(4) & [0, 1, 0, 0 \end{array}$$

Ex:



Pros and Cons of Each:

	adj. list	adj. matrix
Space:	$O(m+n)$	$O(n^2)$
Time:	$O(n^2)$	$O(n^2)$

$RT = O(m) \approx RT = O(n^2)$

Adjacency List Algs

```
is_edge (G,u,v):
    for w in G[u]:
        if w = v:
            return True
    return False
```

O(out-degree(u)) ?
= # of out-neighbors
= length of G[u]

Adjacency Matrix Algs

```
is_edge (G,u,v):
    if G[u][v]=1:
        return True
    return False
```

O(1) time... since we can index, we don't have to scan the entire D.S. (like you would w/ a list)

```
print_out_neighbors(G,u):
    for v in G[u]:
        print(v)
```

TIME:
O(out-degree(u))
aka len(G[u])

```
print_out_neighbors(G,u):
    for v in G[u]:
        print(v)
```

TIME:
O(n)
because we have to scan over 0s as well as 1s

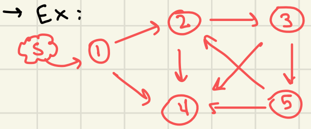
→ TAKEAWAY:

- We use adjacency lists by default, especially when you have to do a lot of printing
-

2.1 Breadth-First Search (BFS)

→ Input: (G, S) where G = directed graph and S = vertex in G that is the "start"/"source"

→ Ex:



→ GOAL: Return an array d s.t. $\forall u \in V$, $d(u)$ = distance from s to u in G

• Aka, length of shortest path

↳ aka # edges in the path

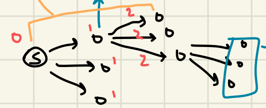
↳ sequence (list) of vertices that follows edges

(if path p has k vertices, $len(p) = k - 1$)

→ ANS: $d = [0, 1, 2, 1, 3]$

the "shortest length" of this path = 2

set the out neighbors vals to be 2



Intuition: BFS ≈ dropping water on a table (processes V in layers)

→ What is a queue? → Can add to the back & remove from the front in O(1) time

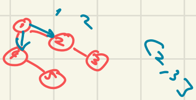
→ (Example from above) ... initially set the value of each edge = ∞. As you traverse, change the values.

[2, 4]

[4, 3]

[5] ... don't change anything by this point.

→ Once queue is empty, you're done



→ ALG(G, S):

```
d = [∞] * n, d[S] = 0
Q = queue(S)
while |Q| ≥ 1:
    u = dequeue from Q
    for v in G(u):          → look at u's out-neighbors & change
        if d[v] = ∞:           their vals if they are ∞
            d[v] = d[u] + 1
            add v to Q
return D
```

n iterations... each takes O(out-degree(u)) time

• O(out-deg(u)) ≤ m ... so you could say RT = O(m²) but that's not the best estimate.

"process u" section of alg

→ Running Time: O(m + n) ?????

• n iterations of the while loop (see orange notes)

• RT = out-deg(S) + out-deg(S's first out-neigh) + out-deg(S's 2nd neighbor) + ... =

$$\sum_{n \in V} \text{out-deg}(n) = \underline{m}$$ ... RT = O(m + n)

adding up out-degs is basically ≈ counting ms

• Creating d takes O(n) time. The rest of the alg is O(m) time. Thus, RT = O(m + n).

Ch. 2: Essential Graph Algorithms

– Breadth-First Search –

What is the input?

→ (G, s), where G is a directed graph and $s \in V$ (s is a vertice).

What is the goal?

→ Return an array d s.t. for all $v \in V$ (all vertices), $d[v]$ is the distance from s to v.

What is "distance" defined as?

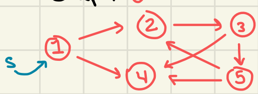
→ The shortest amount of edges that you have to "walk along" to get from node s to node v.

→ a.k.a., the # of edges in the 'shortest path'

- If path p has k vertices, $\text{len}(p) = k - 1$

Example graph?

Graph G :



ANS:

BFS $(G, 1) \Rightarrow d = [0, 1, 2, 1, 3]$

How is a graph represented for function input?

→ As an adjacency list – see pgs 13 – 14

$G = [[2,4], [3], [4,5], [\ ], [2,4]]$

What is the BFS Alg?

→ See notes on prev page for more details.

→ INTUITION: Start at node s & work through the graph in "layers"; visit s' out-neighbors, the out-nbors of those vertices, & so on.

→ Implementation:

```
d = [∞] * n, d[s] = 0
Q = queue(s)
while |Q| ≥ 1.
    u = dequeue from Q
    For v in G(u):
        if d[v] = ∞:
            d[v] = d[u] + 1
            add v to Q
return D
```

1. create a queue Q with initially just s in it.
2. set $d[s] = 0$
3. While Q is not empty, dequeue (aka, take the element/vertice which has been in Q for the LONGEST; FIFO) a vertex u from Q and "process it."

"Process vertex u":

1) loop through each out-neighbor v of u and check if v has been encountered before (if d for vertice $v = \infty$, that means it hasn't been touched since d was created).
2) If v hasn't been encountered, add v to the end of Q.
3) If v was added to Q, set $d[v] = d[u] + 1$

- Depth-First Search -

What is the idea behind DFS?

→ BFS is like water spreading across the surface of a table.

→ DFS is like running down a maze & leaving a trail of breadcrumbs

- Traversing down a graph; whenever we reach a fork in the road, we pick a direction & continue till we get stuck... at which point we backtrack along the breadcrumbs & try another direction

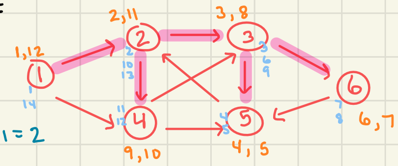
What is the input & goal?

→ Input: A directed graph G, for ex: [[2,4],[3,4],[5,6],[3,5],[2],[5]]

→ Goal: Return 2 arrays, pre[] and post[]. For a node u in G,

pre[u] = time we start exploring u

post[u] = time we stop exploring u



Example of how DFS works?

→ Let t = 1 = starting time.

1. Started at u = 1, pre[1] = 1; t = t+1 = 2
2. Then we moved to u = 2, pre[2] = t = 2 ; t += 1
3. moved to u = 3, pre[3] = t = 3; t += 1
4. moved to u = 5 pre[5] = t = 4 ; t += 1
5. Nowhere to go from node u=5, so we can write the post val: post[5] = t = 5
6. Backtrack to 3, which is where 5 came from.
7. Move to u = 6, pre[6] = t = 6 ; t += 1
8. Only place to go from 6 is 5, but 5 has alr been explored. Therefore, "stuck" at 6 so we can write the post value: post[6] = t = 7 ; t+=1
9. Backtrack to 3, nowhere else to go, so write a post value: post[3] = t = 8 ; t+=1
10. Backtrack to 2, which is where 3 came from
11. Move to u = 4 (unexplored), pre[4] = t = 9 ; t+=1
12. Stuck at 4 bc 3 and 5 alr explored, so post[4] = t = 10 ; t += 1
13. Backtrack to 2, nowhere else to go, write the post value: post[2] = t = 11 ; t+=1
14. Backtrack to 1, nowhere else to go, write the post value: post[1] = t = 12 ; t+=1

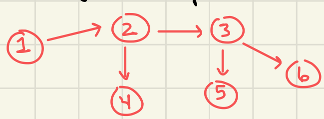
→ ANS: pre = [1, 2, 3, 9, 4, 6]

post = [12, 11, 8, 10, 5, 7]

What are the types of edges that we can label when doing a DFS?

→ DFS Tree edges: the edges that the "rat" actually ran across. Edges that were crossed to reach not-yet-explored vertices. highlighted pink in Ex above.

- The union of these edges is called the DFS Tree. From Ex above:

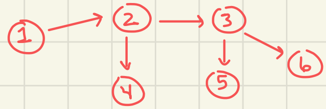


What are the types of edges that we can label when doing a DFS?

→ Forward edges: all edges (u, v) s.t. there is a path from $\underline{u \text{ to } v}$ in T.

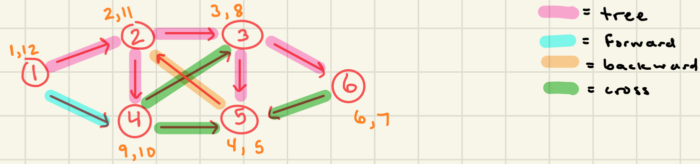
→ Backward edges: all edges (u, v) s.t. there is a path from $\underline{v \text{ to } u}$ in T.

→ Cross edges: all edges (u, v) which are not tree, forward, or back edges.



DFS Tree T from Ex 1

Visual of the Graph from Ex 1 with all edges labeled?



What is the algorithm for DFS?

→ The "DFS" function is actually just a wrapper for the Explore function, which actually does the "steps" described on prev. page.

```
Explore(G, u):          → where u = index of a vertice in graph G
    pre[u] = t
    t += 1
    for v in G[u]:      → for every out neighbor of vertice u
        if pre[v] == ∞:
            Explore(G, v)
    post[u] = t
    t += 1
```

```
DFS(G):
    pre, post = [∞] * n
    t = 1
    for u in V:         → "V"?
        if pre[u] == ∞:
            Explore(G, v)
    return pre, post
```

DFS() relies on Explore() for the logic.

Wait, is the "DFS tree" always connected?

→ No; it can have multiple separate root vertices, kind of a "DFS Forest". Each time Explore() is called from DFS() – not recursively – there is a new "tree" in the DFS forest.

What is the Running Time of DFS?

→ Similar to BFS, it starts as $O(m+n)$:

→ A single call of Explore(G,u) runs in G[u] steps -- eg, in one call, it runs for each of u's out-neighbors (b/c of line 4).

- One call of Explore is len(G[u]) time, aka out-deg(u) time.

→ Since every vertex will be explored once, and a graph G has n vertexes, and "exploring" each vertex takes out-deg(u) time, the total RT is $O(m+n)$, where m = # of edges.

Is $O(m+n)$ linear time?

→ Yes! Because if the input is size $\underline{k}$, then anything that is $O(k)$ is linear time.

→ Think about the input -- if it were a simple 1D array of n elements, then anything running in $O(n)$ time would be linear. But here, our input is an adjacency list of length $m+n$. Input G has n lists. The sum of the sizes of each list is m.

→ Running Time depends on the input. Whenever the RT ≈ the input size, it is linear time.

→ For ex, $O(n^2)$ would be "linear time" for an **adjacency matrix**, since adj. matrices are of size $n \times n$.

– Cycle Finding –

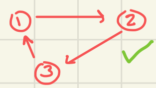
What is the Cycle-Finding problem?

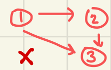
→ A problem that applies DFS. Again, the input is a directed graph G

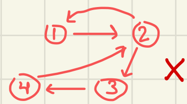
→ Goal: Return a directed cycle in G – or, if none exists, then nothing.

What is a cycle in a directed graph?

→ A "subgraph" or set of edges & vertices that is a sequence of adjacent & distinct nodes; e.g., the 1st & last vertices in the path are the same, BUT no other vertice is repeated.

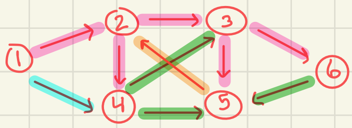






Why does DFS help us solve Cycle-Finding?

→ Recall the types of edges in a Graph being evaluated on DFS:



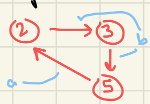
= tree : traversed during alg

= forward : edges between nodes (u,v) where $\exists$ a path from u to v in the DFS tree.

= backward : edges between nodes (v,u) s.t. "

= cross : all other edges

→ Notice that the back-edge in the above graph is what connects the tree edges to form a cycle!



a. back edge from $v,u = 5,2$

b. tree-edges showing that $v=5$ can be reached from $u=2$

→ 3 tree edges & 1 back edge **always** form a cycle.

What is the algorithm for Cycle-Finding?

→ Intuition:

- Run DFS on the graph to obtain pre- & post- values as well as a DFS Tree T.
- If there exists a back-edge (u,v), then we know that $\exists$ a path P in T that goes from v to u. The path + the back edge forms a cycle
- Return $P + (u,v)$ as a cycle in G

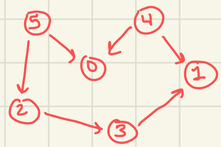
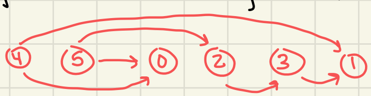
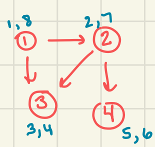
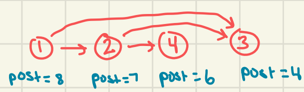
```
Find-Cycle(G):
    DFS(G)
    T = DFS Tree

    for u in V:
        for v in G[u]:
            if (u,v) is a back-edge:
                P = the v→u path in T
                return P + (u,v)
```

V is the literal "array" of vertices (rather than "G" which is the graph)

" if pre[v] < pre[u] < post[u] < post[v] "

What is the RT of Cycle-Finding?	→ Checking for a back-edge is O(1) → Running DFS & constructing T is O(m+n) → RT: O(m+n)

-Topological Ordering-

What is the input and goal?	→ Input: A Directed Acyclic Graph (DAG) G. → Goal: Return a list R that contains a topological ordering of the nodes in G.
What is a D.A.G?	→ A graph with no cycles... we can check whether topological ordering can be applied to a given dir. graph G by first running Find_Cycle() on it!
What is a topological ordering?	→ an ordering of all nodes in V s.t. every <u>edge</u> goes from left to right. → <u>Formally</u>: An ordering R of V s.t. for all $(u,v) \in E$ (all edges), u appears before v in R. • basically, a listing of the nodes $v_0, v_1, \ldots \in V$ where every node only appears in the list AFTER all the nodes pointing to it have appeared. → There can be multiple topo-sorts for a graph.
Example?	G =  R = [5, 4, 2, 3, 1, 0] OR R = [4, 5, 0, 2, 3, 1] • The first element will be a node that has no edges pointing at it (like 4 or 5) • Notice that 0 can't come in the list until 4 & 5 already have.
What is another way to think of topo sort?	→ If you draw the nodes in R in order from L to R, and then add the edges, all edges should be following the flow of left-to-right. For ex: 
What is the intuition behind the Topo-Sort algorithm?	→ Intuition: run DFS on the graph to obtain post[] values, and sort the nodes by decreasing post value → Example:  pre[] = [1, 2, 3, 5] post[] = [8, 7, 4, 6] R = [1, 2, 4, 3] ↓ 

What is the algorithm for Topo-Sort?

→ To save some running time, we can slightly modify DFS s.t. we create the final ordering R as we traverse — rather than running DFS & then sorting post[ ].

How do we "modify" DFS for this purpose?

→ Everytime we set a node u's post-val (in the Explore( ) helper function), we should then add a line of code to "add u to front of R".

- Why? b/c everytime we set a post-val for a node, that is (by nature), the biggest post-val so far. So we can thus add them to the front of R.

```
Explore_for_Topo_Sort (G, u):
    pre[u] = t ; t+=1
    for v in G[u]:
        if pre[v] = ∞ :
            Explore (G, v)
    post[u] = t ; t+=1
    Append u to front of R
```

```
DFS (G):
  /* same implementation; see notes
     on DFS */
```

```
Topo_Sort (G):
   R = [empty list]
   DFS (G)
   return R
```

What is the RT of Topo-Sort?

→ Topo-Sort alg = DFS alg with one extra line -- appending to a list (R). This only takes O(1) time.

→ Thus, R.T. of Topo Sort = R.T. of DFS : $O(m+n)$

– Strongly Connected Components –

What does it mean for a graph to be strongly connected?

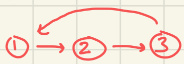
→ A directed graph where there is a directed path between every pair of vertices; every node can be reached from every other node.

- For all $u, v \in V$, $\exists$ a path from u to v AND v to u.

Example?

→ Any cycle will be strongly connected.

→ A strongly connected graph G :



→ NOT strongly connected: ① → ② ⇄ ③ ... Why? there's no path from 3 to 1.

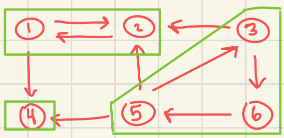
What is a strongly connected component?

→ A "subgraph" of a graph G – aka, a subset of vertices – that is strongly connected.

→ A subset of vertices that all have paths to & from one another.

Example of an SCC?

G =



- G is not a strongly connected graph, but [1, 2], [3, 5, 6], and [4] are SCCs.

Do all graphs have SCCs?

→ Actually, yes. Every directed graph can be partitioned into its strongly connected components

- Even if the SCC subsets have length 1, like [4] from the ex above.

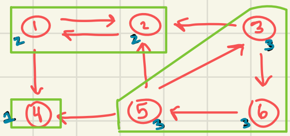
What is the problem statement for SCC?

→ Input: directed graph G

→ Goal: return an array c s.t. for all $u, v \in V$, u and v are in the same SCC if and only if $c[u] = c[v]$

- aka, for every SCC, all members of the SCC are "labeled" with the same "SCC number".

→ Ex



ANS $c = [2, 2, 3, 1, 3, 3]$

nodes 1 & 2 are an SSS

nodes 3, 5, 6 are an SCC

What is the algorithm?

SCC (G):

1. G^R = reverse of G

2. pre, post = DFS (G^R)

   c = [∞] * n

   k = 1

3. for $u \in V$ in decreasing order of post[u]:

   if c[u] = ∞ :

   BFS (G, u)

   k += 1 → k represents the "SCC number" for labeling.

4. set c[v] = k for all v reached from u → and only if c[v] == ∞

return c

1. Construct the reverse graph of G, G^R, by reversing every edge in G (flip the arrows, e.g. $(u,v) \in E(G)$ becomes $(v,u) \in E(G^R)$)

Implementation (Linear time):

```
G^R = [[]] * n          // graph with n vertices & 0 edges
for u in V(G):          // for each vertex in og graph,
    for v in G[u]:      // look at its out neighbors
        add u to G^R[v] // add reverse edge to G^R
```

2. Run DFS(G^R) to get the post values for every $u \in V$.

3. For each vertice $u \in V$ (in order of decreasing post[u]), run BFS(G, s) with a "wrapper", to find the vertices reachable from u in G.

   - BFS() with a wrapper: meaning, only run BFS() on a vertex if it hasn't already been discovered in a previous BFS() call.
   - When running BFS(G, s), ignore any node/outneighbor x if c[x] != ∞

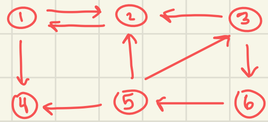
4. Whenever we have to restart BFS (iterations of step 3's for-loop), that represents a new SCC. label all nodes explored in that call with "SCC number" k (and then increment k).

Example of running $SCC(G)$?

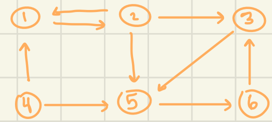
→ Lets use the graph from the earlier example of SCCs.

$c = [\infty, \infty, \infty, \infty, \infty, \infty]$

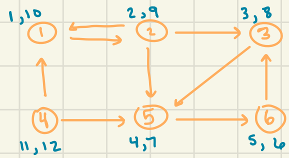
1. Obtain G^R :



→ G^R =



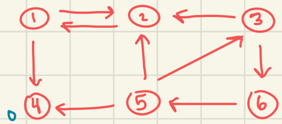
2.



⤳ post[ ] = [10, 9, 8, 12, 7, 6]
(for G^R)

post[4]=12

3. Run BFS(G, s= node w/ highest post value) BFS(G, 4)

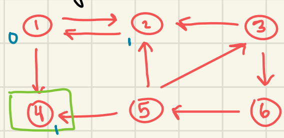


- BFS(G,4) returns $d = [\infty, \infty, \infty, 1, \infty, \infty]$ b/c node 4 has no out-neighbors

4. $c = [\infty, \infty, \infty, 1, \infty, \infty]$; k = 2

5. Run BFS (G, s= node w/ second highest post value) BFS(G, 1)

- ignore node 4 because c[4] != ∞



- BFS(G,1) returns $d = [0, 1, \infty, \infty, 1, \infty, \infty]$

Note: "ignoring" nodes in BFS would actually be implemented as not ignoring them, but instead, only adding nodes from $\underline{d}$ to a group in $\underline{c}$ iff :

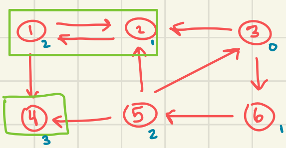
a) d[u] != ∞

b) c[u] == ∞

6. $c = [2, 2, \infty, 1, \infty, \infty]$; k = 3 → we set c[u] = k for all the nodes explored by the BFS call -- in this case, BFS(G,1), which only explored nodes 1 and 2.....well, did explore node 4 but since c[4] already filled, we don't care

7. DON'T run BFS (G, s= node w/ next highest post value) b/c next-highest post-val is node 2, but c[2] != ∞ ...it alr belongs to a group

8. Run BFS (G, s= node w/ next highest post value) BFS (G, 3)



- BFS (G,3) returns $d = [2, 1, 0, 3, 2, 1]$...but excluding explored nodes (green boxes) it is basically $d = [\infty, \infty, 0, \infty, 2, 1]$

9. $c = [2, 2, 3, 1, 3, 3]$; k = 4

10. Done!

What is the RT of SCC?

→ Constructing G^R and running DFS (for post vals): $O(m+n)$ time

→ Run BFS from one vertex u for each SCC (aka a total of at most n times)... but the amt. of vertexes it has to process decreases so..

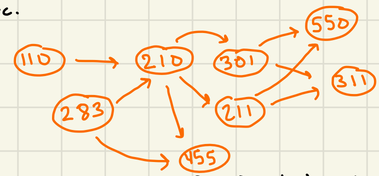
→ Total RT: $O(m+n)$

— Applications —

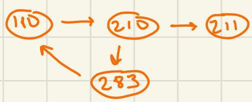
What is an application of Cycle-Finding?

→ Say you are constructing the course structure for the CS major at a university. You decide what classes are required to take other classes; what classes must be taken in sequence, etc.

(edge u,v indicates that course u is a prerequisite for course v)



→ Once you've made your list/structure, use cycle-finding alg to make sure that there isn't a "loop" of classes that depend on each other as prerequisites, meaning that none of them can be taken. For ex:



What is an application of Topo Sort?

→ As a **student**: Use Topo Sort to, given the CS major course structure, figure out a possible order in which you should take all the classes!

Ch 3: Greedy Algorithms

What is a greedy algorithm?

→ An algorithm that iteratively constructs a solution ("one piece at a time") by, in each iteration, choosing the option that **appears** the most optimal right then, without considering how current decisions affect future options.

→ Thinking short-term; best option at each moment

→ Greedy algorithms typically don't work

– 3.1: Minimum Spanning Tree –

What is a "Tree"?

→ A special type of graph or "subgraph" of a graph $G = (V, E)$

→ $T = (V, F)$

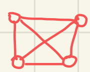
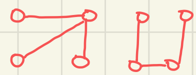
- $V(T) = V(G)$: a tree has the same set of vertices as the graph.
- $F(T) \subseteq E(G)$: a tree's edges are some subset of the edges of the graph.

What is a spanning tree?

→ A tree T of some graph G that has the same properties as above, but is also **connected**.

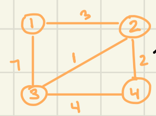
→ For this problem, we will consider undirected rather than directed graphs.

Example?

G =  possible spanning trees: 

What is the problem statement for MST?

→ Input: An undirected, connected graph G, where each edge has a distinct "weight" $w(e) \in \mathbb{Z}$ assigned to it, for ex:



"weight" of edge (2,4)

→ Goal: Return a Minimum Spanning Tree of G.

How do we represent this input in code?

→ A list of n lists of arrays of size 2, where n = # of nodes

→ Each element of the list G represents a nodes out neighbors (e.g. G[u] is the list for node u)

→ u is comprised of a "tuple" (or array of size 2) for each one of its out-neighbors, where G[u[1]] = the out-neighbor vertex v, and G[u[2]] = the weight of the edge between u and v.

Example?

→ So the ex graph above would look like this:

G = [[[2, 3], [3, 7]], → node 1 has an out neighbor 2, with edge weight 3. node 1 has out-ne. 3 w/ edge weight 7.

[[1, 3], [3, 1], [4, 2]],

[[1, 7], [2, 1], [4, 4]],

[[2, 2], [3, 4]]]

node 3:

- out-n 1 w/ w(e) = 7
- out-n 2 w/ w(e) = 1
- out-n 4 w/ w(e) = 4

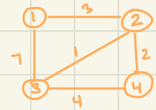
What is a Minimum Spanning Tree?

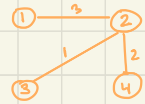
→ A spanning tree of a graph G s.t. the sum of the weights of all edges in T is minimized.

- e.g., pick the subset of edges that allows T to be connected at the minimum weight possible.

Example?

→ From ex on prev page:

G = 

MST = 

What does MST return (in code)?

→ Instead of explicitly returning a tree T in adjacency list format, we can just return a list F of edges, e.g. MST(G) = ((1,2),(2,4),(2,3))

edge from 1 to 2

Can a graph have multiple MSTs?

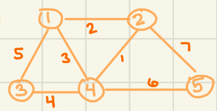
→ No, under our assumption that every edge-weight is distinct (unique)

– Prim's Algorithm –

What is Prim's Algorithm?

→ One of 3 algorithms for solving the MST problem. "Building a cut".

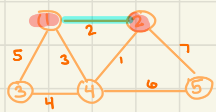
What is the idea behind it, with an example?

→ Ex: G = 

→ Intuition: 1) start at any vertex v and add it to your "bubble".

2) To "connect it" to the "outside world" (rest of the graph), select the edge (connected to v) with the lightest weight and add it to "bubble."

Add the associated vertex on other side to the bubble.

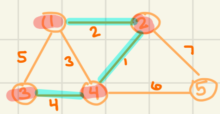


- started at vertex 1. Lightest edge-weight is for edge (1,2). added 2 to "bubble."

Continue steps 3 and 4 until all nodes have been added to bubble

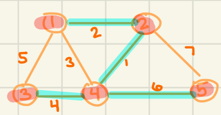
3) Continue this process: The vertex v just added becomes the one you're focused on.

- pick the lightest edge weight that "leaves the bubble"; e.g., that leads to a vertex thats not in the bubble.



- Added edge (2,4). Added vertex 4. The lightest edge from 4 is (4,1), but 1 is alr in the bubble, so we add (4,3). Added vertex 3

4) When there are no edges leading outside the bubble, "backtrack" to the previous node and check if there are edges leaving the bubble.



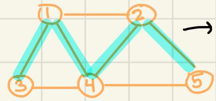
- Nowhere to go from vertex 3, so backtrack to vertex 4. add (4,5) and vertex 5.
- Done!

What is a cut?

→ A cut S is a subset of vertices in G.

→ An edge **crosses** a cut S if it has exactly 1 endpoint in S.

→ EX: $S = \{1, 2\}$



→ edges that cross S

What is the actual algorithm?

→ "Build a cut" S ... S is like the "bubble" described on prev page.

```
Prim(G):
    S = {1}
    F = [empty list]
    for i = 1, ..., n-1:
        e = lightest edge crossing S
        v = endpoint of e not in S
        add v to S; add e to F
    return F
```

begin with node 1 in the "bubble".

Find the lightest edge of all the edges, where exactly 1 endpoint of the edge is an element of S.

→ In real code, this line would actually be a for-loop (eg, "for each edge: check if it crosses S. if it does, check if its lighter than lightest edge so far", and so on)

Once you find e, add the endpoint of e that ISN'T in S, to S.

add the edge to final answer list of edges.

But how would we actually implement the "set" S?

→ Using a **binary array**. e.g.,

- an "empty set" $S = \emptyset$ ≈ `S = [0] * n`; an array of all 0s; one for each node.
- "add u to S" ≈ `S[u] = 1`
- "remove u from S" ≈ `S[u] = 0`

Why does the loop run $n-1$ iterations?

→ To connect n vertices, where n = # of nodes in G, you need $n-1$ edges. Therefore, the tree T returned by MST will always have $n-1$ edges. We need at most $n-1$ iterations to get this.

→ Alternatively, you could replace the while loop with "while $|S| < n$"; aka, while the cut S doesn't contain all the nodes.

How would we implement the for-loop to find e?

```
for u = 1, ... n:
    for v in G[u]:
        if S[u] = 1 and S[v] = 0 (or S[v]=0 and S[u]=1 ??):
            ...
```

What is the Cut Property?

→ For every cut S in G, the lightest-weight edge crossing S is in the MST of G.

→ Proof: assume for contradiction that $\exists$ a cut S s.t. the lightest edge e_1 crossing S is <u>not</u> in the MST T for G.

- if we were to add edge e_1 to T, it would become a cycle - meaning that it would no longer have the minimum amount of edges needed.
- To "fix" this, it would logically make sense to remove one edge... and why would we remove e if we can remove a heavier one?

→ Proof isn't rlly complete ... see notes/video for explanation.

Why is Prim's alg correct?

→ In each iter. of Prim's, we add the lightest edge crossing S, to F. By the Cut Property, this edge is in the MST T... so F is always a subset of T.

→ Since the alg. terminates when F has n-1 edges, and any spanning tree has exactly n-1 edges, it returns the MST of G.

What is the R.T. of Prim's algorithm?

→ Creating the binary array S, and list F, takes O(1) time.

→ The loop runs for at most n-1 iterations

→ finding e (the lightest edge crossing S) will take O(m) time (scan every edge and keep track of the lightest 1 that crosses S)

→ The rest of the stuff in the loop is O(1) time.

→ Therefore, the RT of each iteration is O(m), and there are a total of $n-1 \approx n$ iterations. So the total RT is O(mn)-time.

(O(mn) is better than $O(m^2)$, so $O(mn) \approx O(m^2)$).

- Kruskal's Algorithm -

What is Kruskal's algorithm?

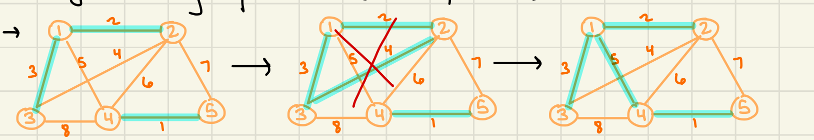
→ Another way to implement MST that doesn't focus on building a bubble, but instead on sorting the edges by weight.

How does it work?

→ Intuition: "Pick" the edges in order of increasing weight, but don't create a cycle.

→ Algorithm: Sort E by increasing weight. For each edge e in this order, add e to a list of edges F (initially empty), if F+e is acyclic (e.g. if adding edge e doesn't create a cycle in F). Return F.

Example?

→ 

- We add edges (4,5), (1,2), and (1,3) to F because they have the 3 lightest weights.
- The next lightest weight is edge (2,3)... but since adding this to F (F = blue highlight edges) would create a cycle, we don't add it.
- F = [(1,2), (1,3), (1,4), (4,5)]

What is the algorithm?

```
Kruskal (G):
  put edges in an array E
  sort E by increasing weight
  F = [ ]

  for e in E:
     if F + e is acyclic:
         add e to F
  return F
```

"F+e" is a list of edges. We can run DFS (G_1 = (V(G), F+e)) to check whether its acyclic by seeing if "G_1" has back edges. DFS works for undirected graphs.

- You could also do this check with BFS: run BFS on F with s = either endpoint (e_0, e_1) in e. If BFS returns d[e_1] (if s = e_0) = ∞, that means that e_1 and e_0 are currently not connected at all, and therefore adding e to F won't create a cycle.

What is the R.T. for Kruskal's?

→ creating & sorting list of edges: $O(m \log m)$

→ m iterations of the "for e in E" loop (once for each edge)

→ checking if $F+e$ is acyclic (DFS or BFS): $O(m+n)$

→ Total RT: $O(m \log m) + m \cdot O(m+n) = O(m^2)$

Why is Kruskal's correct?

→ because of the Cut Property (see textbook pg 16)

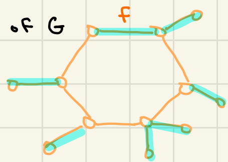
– Reverse Delete –

What is the Cycle property?

→ For any cycle C in G, the heaviest edge in C is <u>not</u> in the MST of G.

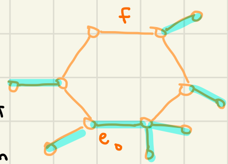
Proof?

→ Assume for contradiction that $\exists$ a cycle C whose heaviest edge $\underline{f}$ is in the MST T of G (highlighted = MST)



→ Removing $\underline{f}$ from MST T creates a cut (removing any edge from a tree breaks it up into 2 parts aka 2 cuts)

→ Since there is a cycle from one endpoint of f to the other endpoint, there will be another edge e_0 in the same cycle C that crosses the cut:



→ f is the heaviest edge, so $w(e_0) < w(f)$

→ Therefore, replacing edge f with edge e_0 in the MST would improve it (meaning T was never a valid MST in the first place)

What is Reverse-Delete?

→ the 3rd way to implement MST. Sort of a "backwards Kruskal's"

How does it work?

→ Idea: Sort E by decreasing weight. For each edge e in this ordering of E, remove e from G if $G-e$ (G without edge e) is connected. Return G.

- Basically, we start w/ the og graph G and remove edges, starting with the heaviest one and working down (by order of decreasing weight)
- Every time we want to remove a heavy edge, we first check whether the graph G would still be connected without it.
- If it would, then we can remove the edge. If not, we can't.
- At the end, we return what is left of G. This is the MST of G.

What is the algorithm?

```
Reverse-Delete (G):
    sort E by decreasing weight
    for e in E:
        if G-e is connected:
            remove e from G
    return G
```

What is the RT of Reverse-Delete?

→ Same as Kruskal's :

- $O(m \log m)$ to sort edges
- m iterations of for-loop
- checking if graph is connected : $O(m+n)$ (BFS or DFS)

→ Total RT : $O(m \log m) + m \cdot O(m+n) = O(m^2)$

3.2 : Selecting Compatible Intervals

What is an interval?

→ An array of 2 positive integers $[s,t]$ s.t. $s < t$

→ represents an event, where s = start time and t = end time.

(number line)

What is the SCI problem statement?

→ Input : an array A of n intervals (aka a 2-D array)

- Ex : $A = [[1,5],[2,4],[3,6],[7,8]]$

→ Goal : Return a list S of compatible intervals that contains as many intervals as possible.

What are compatible intervals?

→ A list of intervals s.t. no 2 intervals in the list conflict at any point in "time".

What is a real-world application of this problem?

→ For ex, imagine that A represents a list of event times at a conference. You want to know the max amt of events you can attend (so, no overlap), and the list of events to attend.

How do you solve this problem in a "greedy" way?

→ Essentially, among all intervals compatible with S, keep adding the interval e according to some criterion C.

→ Greedy = always picking the 'best option'... however we decide to define that.

Outline of the algorithm?

1) pick the interval e that [criterion C] .
2) Remove all intervals that conflict with e
3) Repeat steps 1-2 until no intervals left.

What are 4 possible choices for the "criterion" C?

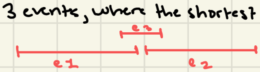
→ C = pick the interval e that...

1. Starts the earliest (intuition: start attending events as early as possible)
2. is the shortest
3. has the fewest conflicts with the remaining events (eg overlaps w/ fewest number of remaining intervals)
4. ends the earliest

What are counterexamples for options 1-3?

→ Option 1 : If the first event takes all day, then this alg might return a list of size 1 (aka only first interval), when the optimal # of events is much greater.

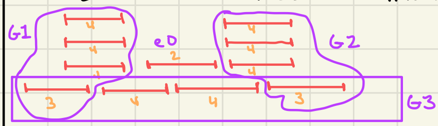
→ Option 2 : Let $|A| = 3$ events, where the shortest event overlaps with the 2 longer ones:



This alg would return $|S| = 1$, when optimally, size of $S = |2|$ (attend e1 and e2)

→ Option 3 (event with fewest conflicts w/ remaining events):

- for each event, opt. 3 assesses each event by the # of events that conflict with it
- Let $|A| = 11$ and the events in A look like this, where the # = # of conflicts:



- The alg will first select event e0. Then, its only options are one event from G1 and one from G2. So the output # of events will be 3
- However, optimally $|S| = 4$ -- all events in G3. Therefore, in this case, option 3 will not produce the correct answer.

So what is the optimal criterion?

→ Option 4: Keep picking the interval that ends the earliest.

Okay, so what is the SCI algorithm?

```
Select-Intervals (A):
    sort A by non-decreasing end time
    S = [empty list]
    for e in A:
        if e doesn't conflict with the last interval in S:
            add e to S
    return S
```

aka, increasing order of values of t

We only need to check against the last interval because we are adding intervals in order of end time. If e conflicts with any interval in S, it must also conflict with the last interval.

What is the RT of SCI?

→ Sorting the intervals: $O(n \log n)$

→ For loop: n iterations so $O(n)$

→ total RT: $O(n \log n)$

– 3.3: Fractional Knapsack –

What is the input?

→ (v, w, B), where

- v and w are arrays of n positive integers
- B is a positive integer

What does this problem represent? (The story)

→ You have n items. Each item has a value ($) and a weight (lbs.). For each item i in $1-n$, the value is $v[i]$ and the weight is $w[i]$. For ex:

$v = [3, 5, 10]$ → item one is $3 and weighs 1 lb
$w = [1, 2, 5]$ → item two is $5 and weighs 2 lb

→ You have a knapsack that can hold at most B pounds of items. GOAL: Choose which items to put in our bag s.t. the $ value is maximized.

→ Also, we are allowed to "fractionalize": We can take 1/2 (or 1/3, 1/4, etc.) of an item, which would add $ v[item]/2 and w[item]/2 lbs to the bag.

What is the goal?

→ Return an array x of size n (aka one element for each og item), where $x[i]$ = the fraction of item i that we are taking ; e.g., $0 \le x[i] \le 1$ such that :

- sum of all weights is $\le B$
- sum of the value of the items in the bag is maximized

What is the algorithm?

→ Idea: keep picking the item that gets you the highest "value per weight", $v[i]/v[w]$.

- Sort the items by non-increasing $v[i]/w[i]$
- In this order, pick as much of each item as possible w/o the total weight exceeding B (aka increase $x[i]$ by as much as possible)

```
Fractional-Knapsack(v,w,B):
    sort items by non-increasing v[i]/w[i]
    x = [0] * n
    for each item i:
        increase x[i] by as much as possible
    return x
```

x[i] must be ≤ 1 and the total weight of knapsack must be $\le B$. So take x[i]=1 if you have space for it. Otherwise, take the fraction $x[i] = \frac{\text{remaining weight}}{\text{weight of item}}$

What is the RT of fractional knapsack?

→ Sorting the items : $O(n \log n)$

→ for loop : $O(n)$

→ total RT : $O(n \log n)$

Midterm 1 Review

REVIEW NEEDED: * Merge Sort *"BFS tree" * GREEDY ALGS * PSEUDOCODE ALG FOR SCC
* RT on psol 2q4 * DFS alg intuition * SCC
* RT linear & stuff with m, n

General Notes

→ Running Time: For undirected, connected graphs: $n \le m+1$ so $O(m+n)$ is actually $O(m)$ — e.g. for BFS, DFS, etc.

→ Array indices start at 1

→ $O(\log n)$ is BETTER than $O(n)$, which is better than $O(n \log n)$

→ $G = (V, E)$ but the pseudocode for vertices in a graph must be "for u in V"

Ch. 1: Array Algorithms

	Max in Array	Two Sum
Input	Array A of distinct pos. integers	• (A, t): Array A of n distinct, sorted integers and int t.
Goal	Return largest int in A	• Return indices (i, j) s.t. $i<j$ and $i+j=t$
Idea	• Set ans = A[1]. Scan through every element of array (n iterations) and check if its bigger than ans. If so update ans = A[i]. Return ans.	• Start w/ "pointers" at beginning and end of array (i = 1, j = len(A)) • For (at most) n iterations (for x ≤ len(A) OR while i<j): • calculate sum = A[i]+A[j]. If sum = t, return i, j • If sum < t, set i += 1 and try again • If sum > t, set j -= 1 and try again
RT	• $O(n)$ – n iterations of for loop	• $O(n)$ – n iterations of for-loop

	Binary Search	Selection Sort
Input	(A, t): A = array of sorted integers t = integer (SAME as Two-Sum)	A = array of distinct integers
Goal	Return index k s.t. A[k] = t	Return A in increasing sorted order
Idea	• set 2 pointers i, j = 1, n While i ≤ j: • let m = index at middle of array. Check if A[m] = t (if so, return m) • If A[m] > t, create new "subarray" by setting j = m-1 (now, we are only checking elements from the beginning to the middle of the array; n/2 elements). • Else if A[m] > t, create new "subarray" by setting i = m+1	For i in range(1, n) (n iterations): • let m = i • For j in range (i+1, n): (the rest of the array after i) • Check if A[j] is smaller than A[m], which implies that it needs to be moved back in the array. If so, swap A[i] with A[j]. • let m = j and continue • Basically, idea is to find the smallest element besides index 1. Swap that element w/ A[1]. Now, find smallest indexes 1-2. Swap that element w/ A[2]. And so on... building sorted order bit by bit.
RT	• $O(\log n)$: In each iter, the number of elements to parse through halves.	$O(n^2)$: 2 for loops

Merge-Sort

Input Same as Selection Sort... A = array of n integers

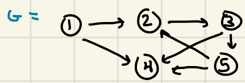
Goal Sort in increasing order

Idea • Split A into left & right half subarrays and recursively sort each half then merge them together.

RT • $O(n \log n)$: The processes of the alg take $O(n)$ time (e.g. comparing & appending etc.). Since alg is recursively called on inputs half the size of the prev one...total of $\log n$ calls * n per call = $n \log n$.

Ch 2: Essential Graph Algorithms

Breadth-First Search



s = 1

$d = [0, 1, 2, 1, 3]$

→ **Input**: Directed graph G, int s where $s \in V$

→ **Goal** : array d, where $d[i]$ is the "distance" (# of edges that have to be crossed) from node s to node i

→ IDEA: 1. Add node S to a queue. Set $d[S] = 0$

2. While queue has elements in it, pop the (least recently added) vertex from queue and "process it"

• For every out-neighbor v of u that hasn't been explored, set $d[v] = d[u] + 1$. Add v to the queue.

→ APPLICATIONS:

• For a graph G or a "subgraph" G, we can run BFS to find out if the graph or subgraph is connected.

• If running BFS(G, u_1) returns an array of all ∞s, we know that node u_1 is alone, not connected to anything else.

• If running BFS(G, s) for any s returns array where any element is ∞, graph is not connected

→ OTHER NOTES:

• "for u in V": pseudocode for iterating through every vertex in the graph

• "for v in G[u]" : pseudocode for iter. through all outneighbors of a vertex u.

• A "BFS" tree (all edges traversed while running BFS) is a spanning tree.

→ RT : $O(m+n)$

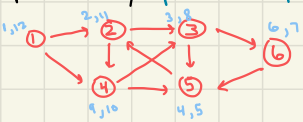
Depth-First Search

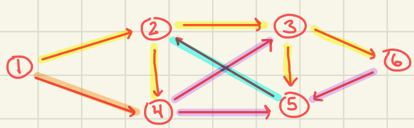
→ **Input**: directed graph G.

→ **Output**: 2 arrays pre[ ] and post[ ]

→ RT: $O(m+n)$

→ APPLICATIONS: Cycle Finding, Topo Sort





• **Tree edge**: edge traversed while running DFS (to a node when it was 1st explored)

• **Forward edge**: edge (u,v) s.t. in the TREE, there is a path from u to v

• **Back edge**: edge (u,v) s.t. in the TREE, $\exists$ a path from v to u

• **Cross edge**: any other edge

Cycle-Finding

→ **Input**: Directed, connected graph G

→ **Goal**: return a cycle in G if one exists.

→ RT: $O(m+n)$: uses DFS

→ Idea: On a graph with all edges labeled, notice that the existence of a back edge implies a cycle (between a back edge and some # of tree edges). So all we need to do is check for a back edge.

→ Algorithm: Run DFS and obtain the DFS tree T, as well as pre[] and post[].

```
FOR each vertice (for u in V):
    FOR each of its out neighbors (for v in G[u]):
        if pre[v] < pre[u] < post[u] < post[v]:
            P = path in T from v to u
            return P + (u,v)
```

these 2 lines represent iterating through all of the edges (each combo of (u,v) s.t. v is out-neighbor of u

Formula to check if edge is back edge

Topological Ordering

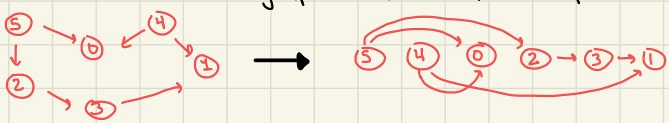
→ **Input**: a directed acyclic graph G.

→ **Output**: a list R that is a topological ordering of the nodes

→ Topological ordering : list of ALL nodes s.t.

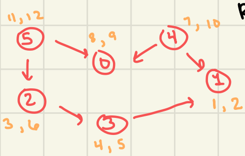
→ RT: $O(m+n)$: uses DFS

- For all edges $(u,v) \in E$, v shouldn't appear in the list before u
- Draw the nodes of a graph from L-to-R ... the topo-sort should follow this L-to-R flow.



ANS: $R = [5,4,0,2,3,1]$ or
$R = [4,5,0,2,3,1]$ or
$R = [5,4,2,0,3,1]$

→ Algorithm: run DFS, and each time you add a node to the post[] array, append it to the front of R. Sorting nodes by decreasing post value



post = [9, 2, 6, 5, 10, 12]
node: 0 1 2 3 4 5

$R = [5,4,0,2,3,1]$

Strongly Connected Components

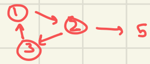
→ Input: Directed graph G

→ Output: An array c of the nodes labeled by the SCC they are in. For all $u, v \in V$, u and v are in the same SCC i.f.f. $c[u] = c[v]$

→ RT: $O(m+n)$

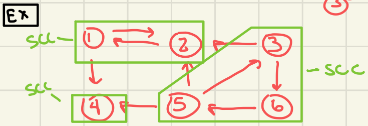
→ Strongly Connected: A "subgraph" of a graph G – aka subset of vertices V_1 – s.t. for every node in V_1 there is a directed path to every other node in V_1.

→ All cycles are SCCs →



$V_1 = [1, 2, 3]$ is an SCC b/c all nodes reachable from one another.

EX



→ SCC in directed graph is analogous to a connected component in an undirected graph.

- The undir. graph converted from a connected dir. graph – or any connected undir graph, for that matter – has exactly 1 connected component; the whole graph.

→ A graph is acyclic i.f.f. the size of every SCC is 1 vertex.

→ ALGORITHM:

1. Construct G^R, the reverse graph of G, by flipping every arrow (edge) (linear time??)
2. Run DFS on G^R to get the post values for each vertex
3. In order of highest-to-lowest post value post[u], run BFS with s = u to find which vertexes are reachable from u in G.
4. Each time BFS(G, u_x) returns, add all the nodes in d which aren't infinity, to a new SCC group.

Ch. 2 Summary

BFS → use to find if graph or part of a graph isn't connected

DFS → • use post[] vals in high-to-low order to create Topo Sort for a DAG
- use pre[] and post vals to check if a graph contains a back edge, which implies that it contains a cycle.
  - → Do this by checking if pre[v] < pre[u] < post[u] < post[v] for every node v & its out-neighbors u

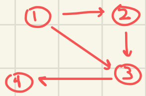
SCC → • Reverse the graph, Run DFS on reversed graph, use post[] to run BFS on each node in high-to-low post-val order. The nodes explored by a given run of BFS(G^R, u_x) are all in the same SCC.
- Return array c s.t. c[v] = c[u] if v & u are in the same SCC
- Running DFS(G^R) to get post array. BUT we run BFS on G!!!

Ch 3: Greedy Algorithms

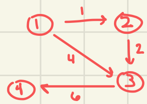
→ Choosing best option at each time that a choice has to be made.

→ Weighted undirected graphs: every edge e has weight w. In code, it's similar input as the adj. list for a dir. graph ?, except for every out-neighbor we use a tuple [vertex v, weight w]:

G = [[2,3],[3],[4],[]]

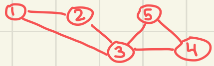


G = [[[2,1],[3,4]],[[3,2]],[[4,6]]]



→ Spanning Tree: a "path"/subgraph of a graph that reaches all nodes w/ minimum # of edges. Ex:

G =



S.T. of G =

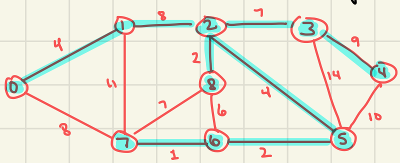
→ Minimum Spanning Tree: A ST where the sum of all weights of edges is minimized. Has exactly n-1 edges.

- Goal: return an MST as a list of edges (tuples) F that the MST contains.

→ A Cut = a subset of vertices S in G. An edge crosses S if 1 endpoint is in S.

Prim's Alg

→ Idea: Build a bubble by adding vertices to a cut & then looking for the lightest edge crossing the cut. Add lightest edges to a list F. Add vertices to cut S. Repeat n-1 times or until |S| = n.

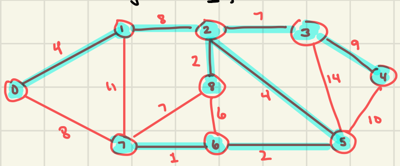


→ RT: $O(m^2)$ ... $(n-1)\cdot(m) = O(mn) = O(m^2)$

- # of edges in output MST = n-1 so n-1 iterations.
- finding lightest edge: $O(m)$

Kruskal's Alg

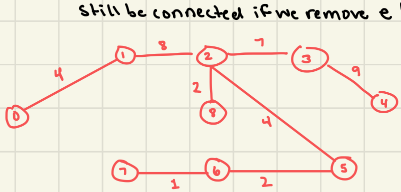
→ Idea: Sort the edges by increasing weight ($O(m \log m)$). Add the edges to F in order, starting w/ lightest. Before adding each edge e to F, ensure that it won't create a cycle by running Cycle_Finding/DFS on F. Only add it if F+e is acyclic.



→ RT: $O(m^2)$ – $O(m \log m)$ + (m iterations)($O(m)$)

Reverse-Delete

→ Idea: Sort edges by decreasing weight ($m \log m$). From heaviest to lightest, for each edge e (m iters): check whether G would still be connected if we remove e by using BFS(G-e). If so, remove e from G.



→ RT: $O(m^2)$ ... same as Kruskal's

Selecting Compatible Intervals (SCI)

→ Input : Array A of n intervals (interval = $[s,t]$ s.t. $s<t$). They represent times

→ Goal : Return a list S of compatible intervals that contains max # of intervals possible.

- compatible intervals = no overlap

→ Idea : keep selecting the interval that ends the earliest.

→ Algorithm: Sort A by non-decreasing end time (aka t from $[s,t]$). From lowest to highest, add interval x to S if it doesn't conflict w/ the last-added interval.

→ RT : $O(n \log n)$ – $n\log n$ (sort A) + n (iterations) * $O(1)$ = $n \log n + n$

Fractional Knapsack

→ Input : (v, w, B) where v = array [ ] of n \$ values, w = array [ ] of n weights, and B = integer weight limit.

- There are n items, for items $i_1, \ldots, i_n$, $v[i]$ is its value and $w[i]$ is its weight.

→ Goal : Return array x of n $\mathbb{R}$ numbers s.t. :

- $0 \leq x[i] \leq 1$
- $v[i] * x[i]$ = the value added to bag for item i
- $w[i] * x[i]$ = weight added for item i
- sum of $x[i] * w[i]$ for all i is $\leq B$
- value is maximized

→ Algorithm: Calculate ratio $v[i]/w[i]$ for all items i, and sort the items by decreasing value. For each item starting w/ highest ratio, add as much of item as possible w/o weight exceeding B.

→ RT : $O(n \log n)$ – $n\log n$ to sort, n $O(1)$ iterations.

Summary of ALL

Array

- Max in array ⟶ $O(n)$ — returns max value int
- Two Sum → $O(n)$ — SORTED ARRAY — returns indices i, j
- Binary Search → $O(\log n)$ — SORTED ARRAY — returns index of int t for BT(Array, t)
- Merge Sort → $O(n \log n)$
- Selection Sort → $O(n^2)$

Graph

- BFS ⟶ $O(m+n)$ for directed . $O(m)$ for undirected
- DFS: Cycle Finding ⟶ $O(m+n)$ or $O(m)$ for undir.
- DFS . Topo Sort ⟶ " — input is DAG
- SCC ⟶ G^R = Reverse G ; run DFS(G^R). Sort nodes in decreasing post[ ] order ; In that order, For each node, run BFS(G, s=node) and "add" all connected nodes to one SCC.
  - $O(m+n)$

Greedy

- MST (Prim's, Kruskal, R-D) ⟶ $O(m^2)$
- SC I ⟶ $O(n \log n)$
- FK ⟶ $O(n \log n)$

Ch.4: Dynamic Programming

What is Dynamic Programming?	→ A way to solve problems that involves solving a sequence of increasingly larger subproblems by using solutions to smaller subproblems. • "recursion with a table" → Recurrence of a subproblem.
How does dynamic programming compare to greedy?	→ Finding the sol'n is a little trickier BUT most of the time, dynamic alg. solutions work **better** than greedy ones when it comes to finding an "optimal" solution. → Dynamic is more formulaic
What is the format for finding & presenting a DP alg?	1. Find **subproblems**: smaller, not necessarily identical "versions" of the original problem. • Which subproblems will we solve? What will we return? 2. **Recurrence**: How do we solve each subproblem using solutions to smaller subproblems? What are the base cases? Why does the recurrence hold? • Similar idea to induction or recursion • OPT[ ] = the "table" of solutions to each subproblem. 3. **Algorithm**: Turning this recurrence / the idea into pseudocode • How do we use recurrence to populate a DP table (e.g. an array d)? • Basically turning OPT → d. Usually, we return the last element in d. 4. Remember to return a solution to the original problem itself.
What is the RT for most DP algorithms?	→ Typically (# of problems) × (time per subproblem)

— Using DP to solve "Max in Array" —

RECALL: What is Max in Array?	→ For an array A of n distinct pos. integers, return the largest int in A. • Ex: A = [3, 1, 4, 5, 2]
What are the subproblems?	→ For all i in A, we can find the max integer between A[1] and A[i]; e.g., starting at A[1] and continuously keeping track of the biggest int "so far". • For all i in range 1-n, let OPT[i] = max(A[1:i]) • We will return OPT[n] → Ex: OPT[1] = 3 OPT[2] = 3 OPT[3] = 4
What is the base case?	→ OPT[1] = A[1]
What is the recursive case?	→ For all i in range 2-n, OPT[i] = max(OPT[i-1], A[i]) • basically, every OPT[i] checks if A[i] is greater than any element before it.

Why does this recursion hold?	→ max (A[1:i]) is either the largest integer in A[1:i-1] (aka OPT[i-1]), or it is A[i]; OPT[i] "picks" the larger option.
What is the pseudocode?	(see below)

```
Max-in-Array-DP (A):
    d = [A[1]] * n
    for i in range (2,n):
        d[i] = max( d[i-1], A[i])
    return d[n]
```

What is the RT?	→ n-1 iterations; O(1) time for each; total RT = O(n)

- Longest Increasing Subsequence -

What is the input and the goal?	→ Input: Array A of n integers. • Ex: A = [3,4,1,5,2,3,6,1] → Goal: Return the length of the longest increasing subsequence (LIS) of A. • ANS: 4 ... s = [1, 2, 3, 6]
What is a subsequence?	→ A subarray of an array A that may skip some elements but may not contradict the order of the elements in A. • E.g. if B = [3,5,5,1,3,2], then some subsequences are: [3,5,3] [5] [5,2], but NOT [1,1] or [1,5]
What is an I.S.?	→ Subsequence subarray where each element MUST be greater than the last. • E.g. for ex arr A (above), possible I.S.'s are [3], [1,5,6], [1,2,3,6], [3,4,5,6].
What are the subproblems?	→ For all i in (1,...n), let OPT[i] denote the length of the L.I.S. of A that must end on A[i]. • Not the same as finding the LIS for the array A[1:i], b/c that wouldn't necessarily mean that A[i] has to be in the L.I.S. → Ex: A = [3,4,1,5,2,3,6,1]. Let "s" denote the LIS for each subproblem. • OPT[1] = 1, s = [3] • OPT[2] = 2, s = [3,4] • OPT[3] = 1, s = [1]... because the LIS must end on A[3] for i=3, we can only include all A[x] for x = 1,...,i. Both A[1] and A[2] are > A[3], so they won't form a valid increasing subsequence. • OPT[4] = 3 s = [3,4,5]
What will we return?	→ The maximum value in the OPT array. NOT OPT[n] like in problem 4.1.

What is the recurrence pattern/idea?

→ $\boxed{\text{Ex:}}$ lets look at finding OPT[4] for this example. Let $i = 4$.

$A = [3, 4, 1, 5, 2, 3, 6, 1]$ $\quad$ $OPT = [1, 2, 1, \ , \ , \ , \ , \ ]$

→ Since OPT[i-1] will represent the longest LIS ending on i, we can find the LIS for OPT[i] by finding the "best" entry A[j] to "come from" before "jumping" to A[i].

cond. 1
- basically, look at all elements A[x] where $x < i$ ...aka, elements that appear before A[i], to maintain ordering.

cond. 2
- of all of these, narrow down and only look at elements A[x] where $A[x] < A[i]$... aka, elements which are smaller than A[i], to maintain the "increasing" part of LIS
- of all of these, choose the element x with the largest value of OPT[x]. The option that brings the longest prior subsequence with it.

What is the base case?

→ OPT[1] = 1

What is the recurrence?

→ calculate OPT[i] for all $i \geq 2$ by satisfying the following recurrence:

$$OPT[i] = 1 + \max_{j \in C} OPT[j] \text{, where}$$

$$C = \{ j \mid j < i \text{ and } A[j] < A[i] \}$$

- "C" ≈ set of candidates to consider. Uses cond. 1 and 2 from above.
- if $C = \emptyset$, OPT[i] = 1.

What is the algorithm?

```
LIS(A):
    d = [1] * n
    for i = 2, ..., n:
        for j = 1, ..., i-1 :
            if A[j] < A[i] and d[i] < (1 + d[j]):
                d[i] = 1 + d[j]
    return max(d)
```

→ setting base case

- calculate d[i] according to described recurrence for OPT[i] and return max(d).

What is the RT?

→ $O(n^2)$

- Longest Palindromic Sequence (LPS) -

What is a palindromic subsequence?

→ a subsequence S where S is = to the reverse of itself.

→ Ex: A = acbba, then a PS could be [a], [b], [b, b], [a, a], or [a, b, b, a].

What is the input & goal?

→ Input: a string A of length n. Characters, not numbers.

→ Goal: return the length of the longest palindromic subsequence (LPS) of A.

→ Ex: Ans = 4, LPS = [a, b, b, a].

What are the subproblems?

→ Instead of a 1D array, OPT will be a 2D array OPT[ ][ ]

→ For all i in $(1, \dots, n-1)$ and all j in $(i, \dots, n)$,

OPT[i][j] denotes the length of the LPS in the subarray $S = A[i:j]$

- The subproblem is a "substring" – from i to j – rather than a "prefix"

→ The subproblems are: for each substring of length x in range $(1, n)$, there are $n!$ possible subproblems/substrings

- Total of $O(n^2)$ subproblems.

What will we return?

→ OPT[1][n], because this is the "subproblem" for the array A[1:n], aka simply A.

How can we visualize OPT?

→ OPT[1][1] = 1, LPS = a, subarray = A[1:1] = "a"

→ OPT[1][2] = 1, LPS = a or c, subarray = A[1:2] = "ac"

→ OPT[1][4] = 2, LPS = bb, subarray = A[1:4] = "acbb"

→ Lets create the 2D matrix for A = acbba as a table:

	j =	1 a	2 c	3 b	4 b	5 a
i = 1	a	1				$\boxed{}$
2	c	∅	1			
3	b	∅	∅	1		
4	b	∅	∅	∅	1	
5	a	∅	∅	∅	∅	1

OPT[1][j=n] is what we want to return.

any OPT[i][j] where i=j has |S| = 1 bc the substring A[i:j] is the same as A[i].

any OPT[i][j] where i<j is invalid because not a sequential substring

What are the 2 types of "recurrences" to solve?

1) For all OPT[i][j], if A[i] = A[j] – aka, a substring that starts & ends w/ the same character – then the entire A[i:j] is an LPS as long as the content between i and j – aka, A[i+1 : j−1] is also a palindrome. So LPS for OPT[i][j] is the LPS of the inner content, + 2 for A[i] and A[j]

- Formally: if A[i] = A[j], OPT[i][j] = 2 + OPT[i+1][j−1].
- But how would we obtain OPT[i+1][j−1]?

2) if A[i] ≠ A[j], we want to find the LPS of the substring w/o A[i], and the substring w/o A[j]; and select the larger LPS.

- Formally: if A[i] ≠ A[j],

$$OPT[i][j] = \max(OPT[i][j-1],\ OPT[i+1][j])$$

How do we fill the table?

→ Each entry in the table depends on the # to the left of it (aka $i-1$), below it (aka $j-1$), or to the bottom-left (diagonally) (aka OPT[i-1][j-1])

→ We can't fill out entry OPT[i][j] unless OPT[i+1][j] and OPT[i][j-1] and OPT[i-1][j-1] have already been filled.

→ So, we should fill the table row-by-row L-to-R, starting from the bottom row.

• Ex: A = [a c b b a]

i \ j		1 a	2 c	3 b	4 b	5 a
1	a	1	1	1	2	4
2	c	∅	1	1	2	2
3	b	∅	∅	1	2	2
4	b	∅	∅	∅	1	1
5	a	∅	∅	∅	∅	1

→ 2 + OPT[2][4] = 2 + 2 = 4

A[i] = A[j], so we take 2 + OPT[4][3] = 2 + 0 = 2

→ A[i] ≠ A[j], so we take max(OPT[4][4], OPT[5,5])

What is the algorithm?

```
LPS(A):
    d = [0] * (n x n)          → creating our OPT matrix
    for i = (n, ..., 1):       → starting at i = n, aka the last row, b/c we want to go from bottom to top.
        d[i][i] = 1            → the base case
        for j = (i+1, ..., n): → for each row, we work L-to-R, but RECALL we only care about values of OPT/d where j > i.
            if A[i] = A[j]:
                d[i][j] = d[i+1][j-1] + 2   → if A[i] = A[j], take the length of the LPS for all characters between A[i] and A[j]. Then add 2, for each of A[i] and A[j]. CASE 1
            else:
                d[i][j] = max(d[i+1][j], d[i][j-1])   → case 2
    return d[1][n]             → returning the LPS for the entire string, aka A[1:n].
```

What is the running time?

→ The DP table has n^2 entries, each of which take $O(1)$ time to compute. Thus, the RT is $O(n^2)$

— 4.4: 0/1 Knapsack —

What is the input?

→ RECALL 3.3: Fractional Knapsack

→ Input = (v, w, B) where

- B = An integer knapsack weight limit
- v = array of item values
- w = array of item weights

What is the goal?

→ Unlike F.K., we can't take "fractions" of items – only all or none of an item.

→ RETURN: an integer representing the maximum/optimal value that the knapsack can have.

RECALL: How did we solve Fractional knapsack?

→ Order the items by their value ratio, eg $v[i]/w[i]$ for all i. From highest-to-lowest ratio, take as much of each item as you can.

→ Ex problem: $B = 4$ $v = [3, 2, 2]$ $w = [1, 4, 3]$

What are the subproblems?

→ OPT will be a 2D-array with $i = n = |v|$ columns and $j = B+1$ rows (e.g. $j \in \{0, 1, \ldots B\}$ and $i \in \{1, \ldots n\}$).

→ $OPT[i][j]$ denotes the maximum value of the knapsack IF we can only select items $1, \ldots, i$. AND, the weight limit is j.

- for ex, for an o.g. input (v, w, B), $OPT[2][3]$ is the max value of a knapsack where $B_1 = 3$, $v_1 = v[1:3]$, and $w_1 = w[1:3]$.

What will we return?

→ $OPT[n][B]$, aka the last row & last column. At this index, we have imitated the orignal problem.

Ex problem:

i = \ j =	0	1	2	3	4
1					
2					
3					

← because $B = 4$

because $n = |v| = 3$

→ ANS

What are the base cases?

→ RECALL: Base case ≈ recursion not necessary to solve.

→ 2 Base cases:

1) $OPT[i][0] = 0$ for all i, because in these subproblems, the weight limit = 0, so we can't pack any items.

2) $OPT[1][j]$ has 2 possibilities:

- if $j < w[1]$, then it is $\underline{0}$ b/c we can't fit the item in our bag
- if $j \geq w[1]$, then it is $v[1]$.
- We only have one item to consider.

i = \ j =	0	1	2	3	4
1	0	3	3	3	3
2	0				
3	0				

What is the recurrence?

→ OPT[i][j] for all $i \geq 2$

→ 2 possible "cases" for each recurrence:

1. When considering an item i at weight j, if $w[i] > j$, then our "solution" for the optimal value doesn't change at all, b/c we know for a fact that we can't bring item i.
   - So, our solution would be the smaller subproblem where the weight limit is still j, but the list of items doesn't contain i.
   - Formally: If $w[i] > j$, then $OPT[i][j] = OPT[i-1][j]$
2. If item i could fit in the bag, we have 2 options:
   a) To still exclude item i, in which case the value is $OPT[i-1][j]$
   b) To include item i, in which case the value is $v[i] + OPT[i-1][j-w[i]]$
      - Why? Because if we are picking item i, the space in the bag is now reduced by the weight of item i. So we want to add $v[i]$ to the optimal value OPT[ ][ ] of the subproblem where j is $w[i]$ smaller, and item i hasn't been included.
   - We should choose which option, a) or b), yields a larger value

→ Formally:

$$OPT[i][j] = \begin{cases} OPT[i-1][j] & \text{if } w[i] > j \\ \max(\underbrace{OPT[i-1][j]}_{\text{ignoring } i}, \underbrace{(OPT[i-1][j-w[i]] + v[i])}_{\text{including } i}) & \text{otherwise.} \end{cases}$$

→

i = \ j =	0	1	2	3	4
1	0	3	3	3	3
2	0	3	3	3	3
3	0	3	3	3	5

What is the pseudocode?

```
Knapsack-DP(v,w,B):
    d = [0] * (n x (B+1))  ⟶ B+1 columns b/c we want to have a column for j=0 up to j=B
    For j = 1,... B:
        if j ≥ w[1]:
            d[1][j] = v[1]  ⟶ Base case
    for i = 2,...n:
        for j = 1,... B.
            if j < w[i]:
                d[i][j] = d[i-1][j]  ⟶ We must ignore item i
            else:
                d[i][j] = max(d[i-1][j], v[i] + d[i-1][j-w[i]])  ⟶ We can ignore or include item i
    return d[n][B]
```

What is the running time?

→ Table has $n*(B+1)$ entries. Each entry takes $O(1)$ time, so total RT is $O(nB)$

is the DP alg for 0/1 knapsack any faster than brute force?

→ The brute force RT is $\Omega(2^n - n)$ (trying every possibility)

→ The DP Alg isn't necessarily faster; $O(nB)$ could be larger than $O(2^n - n)$, depending on the size of B.

- $O(nB)$ is still not polynomial time

→ But often, $O(nB)$ could be less than $O(2^n - n)$? idk

– Edit Distance –

What is the "edit distance"?

*In real python code, A[1:i] actually means (1,...,i-1). But in notation for this class we can take it to mean (1,...,i), aka A[1:i+1]. So in my notes I write either of those kind of interchangeably.

→ The minimum number of "moves" we need to make to turn a string A into a string B.

- The "distance" between A and B.

→ 3 types of possible moves:

1. Insert a character (anywhere in A) "cat" → "cabt", "cat" → "acat"
2. Delete a character (anywhere from A) "cat" → "ca", "cat" → "ct", "cat" → "at"
3. Replace one character (in A) with another – counts as one move. "cat" → "aat", "cat" → "cbt"

What is the problem statement?

→ INPUT: (A, B), where A and B are strings of length m and n, respectively.

→ GOAL: Return a nonnegative integer representing the **edit distance** from A to B. Basically # of moves to turn A into B

What are use cases for this problem?

→ Autocorrect suggestion algorithms. looking at real words w/ a small edit distance from the typed word that has a typo.

→ DNA: comparing how similar 2 strands are.

What are the subproblems?

→ Ex: A = "star" and B = "water"

→ We will shrink both A and B down to prefixes and find the edit distance for each combination.

→ For all $i \in \{0, 1, \dots m\}$ ($m = \text{len}(A)$) and $j \in \{0, 1, \dots n\}$, let $OPT[i][j]$ denote the edit distance from $A' = A[1:i+1]^*$ to $B' = B[1:j]^*$

→ We will return $OPT[m][n]$.

Why do we start OPT[][] with index 0?

→ A[0] and B[0] denote an **empty string** (whilst A[1], for ex, = "s"). Helpful for solving.

→ Ex:

i \ j		0 ""	1 w	2 a	3 t	4 e	5 r
0	""	0	1	2	3	4	5
1	s	1					
2	t	2					
3	a	3					
4	r	4					$\boxed{}$ → ANS

What are the base cases?

→ $OPT[0][j] = j$

- editing empty string into str of length j will take j insertions

→ and $OPT[i][0] = i$

- editing str of length i into the empty string will take i deletions.

How would we fill out row $i = 1$?

		j = 0 ""	1 w	2 a	3 t	4 e	5 r
0	""	0	1	2	3	4	5
i = 1	s	1	1	2	3	4	5
2	t	2					
3	a	3					
4	r	4					

replace s w/ w

replace s w/ w, then add a (one possibility)

each time, we need to first replace s with one of [w, a, t, e, r], and then insert the other 2-4 chars

What about row $i = 2$?

		j = 0 ""	1 w	2 a	3 t	4 e	5 r
0	""	0	1	2	3	4	5
1	s	1	1	2	3	4	5
i = 2	t	2	2	2	$\boxed{*}$		
3	a	3					
4	r	4					

delete "s", replace (t, w)

- repl(s, w)
- repl(t, a)

* Here, we have to turn st → wat.

OPTIONS:

1) turn st → wa (+2) (aka OPT[i][j-1])
   add t (+1)

2) turn s → wat (+3) (aka OPT[i-1][j])
   delete t (+1)

3) turn s → wa (+2) (aka OPT[i-1][j-1])
   "replace" t with t (+0)

→ For the 3rd option, replacing t w/ t actually = doing nothing, so it costs 0 b/c $A[i] = B[j]$.

What is the recurrence?

→ For all $i \geq 1$ and $j \geq 1$.

→ For each OPT[i][j] we have to edit $A[1:i]$ s.t. its last character equals $B[j]$. There are 3 ways to do this:

1) Edit $A[1:i]$ into $B[1:j-1]$, then insert $B[j]$
   - st → wa → wat
   - OPT[i][j-1] = moves to edit $A[1:i]$ into $B[1:j-1]$
   - So this would be OPT[i][j-1] + 1

2) Edit $A[1:i-1]$ into $B[1:j]$, then delete $A[i]$
   - st → watt → wat
   - OPT[i-1][j] + 1

3) Edit $A[1:i-1]$ into $B[1:j-1]$ and replace $A[i]$ with $B[j]$
   - s → wa → wat
   - if $A[i] \neq B[j]$, cost is OPT[i-1][j-1] + 1
   - if $A[i] = B[j]$, cost is OPT[i-1][j-1]

→ Formally,

$$\text{OPT}[i][j] = \min \begin{cases} \text{OPT}[i][j-1] + 1 \\ \text{OPT}[i-1][j] + 1 \\ \text{OPT}[i-1][j-1] + \delta_{ij} \end{cases}$$

where $\delta_{ij} = 0$ if $A[i] = B[j]$, and $= 1$ otherwise.

What is the pseudocode?

```
Edit-Distance (A,B):
    d = [0] * ((m+1) x (n+1))
    for j = 1, ... n :
        d[0][j] = j
    for i = 1, ... m :
        d[i][0] = i
    for i = 1, ..., m :
      for j = 1, ... n :
         d[i][j] = min(d[i][j-1] +1, d[i-1][j] +1)
         if A[i] = B[j] :
            d[i][j] = min(d[i][j], d[i-1][j-1])
         else :
            d[i][j] = min(d[i][j], d[i-1][j-1] +1)
    return d[m][n]
```

What is the RT? → $O(mn)$

	j =	0 ""	1 w	2 a	3 t	4 e	5 r
i = 0	""	0	1	2	3	4	5
1	s	1	1	2	3	4	5
2	t	2	2	2	2	3	4
3	a	3	3	2	3	3	4
4	r	4	4	3	3	4	$\boxed{3}$

– Independent Set in Trees –

How do we apply DP to tree problems?

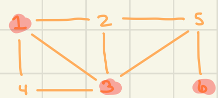
→ Each subproblem corresponds to a subtree of the tree T.

→ For each subtree, we can define 2 subproblems tied together by their recurrences.

What is an independent set?

→ A subset S of the vertices of an undir. graph G s.t. :

- $\forall\ u,v \in S\ ,\ \{u,v\} \notin E(G)$
- For every 2 nodes in S, those 2 nodes are not an edge in G.

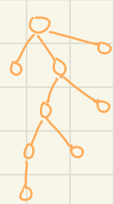
→ Ex: G = 

- one subset is $\{1, 3, 6\}$ b/c the three nodes aren't connected to each other (directly).

Recap: What is a tree?

→ An undirected graph T with n vertices where:

- There are exactly $n-1$ edges
- T is acyclic
- T is connected



What is a maximum independent set (MIS)?

→ An independent set whose weight is as large as possible.

(If weights not given, every node has weight = 1.)

→ RECALL COMP 455: The problem of finding the MIS of an undir. graph G is Turing-hard; nobody knows if a poly-time alg exists.

What is the input to I.S.T.?

→ A tree with weighted nodes. Specifically, (T, w) where

- T = (V, E) is a tree rooted at vertex 1
- w is an array of length |V|, where w[u] is a positive int. denoting the weight of vertex u.
- Unlike MST, in IST, node weights aren't necessarily distinct.

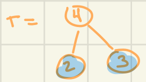
What is the goal?

→ Return the weight of a MIS in T.

Examples?

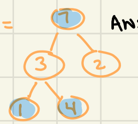
→ The labels on nodes represent weights, not node #:

→ T =



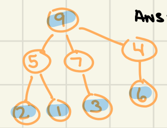
ANS: 5 (2+3)

T =



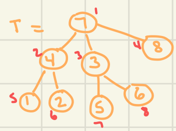
ANS: 12

T =



ANS: 21

→ EX: T =



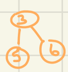
= node #

What are the subproblems?

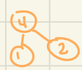
→ We can't do prefixes like in arrays, but the ≈ of that is subtrees.

→ For all nodes in T (for all u ∈ V), let T(u) denote the subtree rooted at node u.

- Ex: T(3) =



and T(2) =

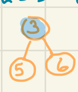


→ We define 2 subproblems for each vertex subtree T(u):

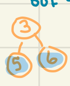
1) $OPT_{in}[u]$: denotes the weight of the MIS in T(u) but we must include node u.

2) $OPT_{out}[u]$: denotes the weight of the MIS in T(u) but we must exclude node u.

→ Ex: $OPT_{in}[3] = 3$



$OPT_{out}[3] = 11$



What will we return?

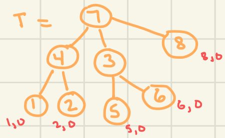
→ We will return $\max(OPT_{out}[r], OPT_{in}[r])$, where r is the root of the whole tree T, e.g. node 1 in ex above.

What are the base cases?

→ If u is a leaf - aka a node w/ no children, like node 7 from ex above, T(7) = 5, so then T(u) is a graph w/ only one node:

- $OPT_{in}[u] = w[u]$ (we include node u)
- $OPT_{out}[u] = 0$

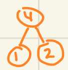
→ Given the base case, we can find $OPT_{in}(u)$, $OPT_{out}(u)$ for all leaves:

T = 

How will we find OPT_{in} and OPT_{out} for the nodes which are not leaves?

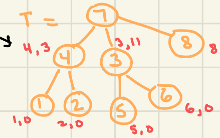
→ For the second layer of nodes u', OPT_{in} will be $w[u]$ because we can't include any of its children.

→ OPT_{out} will be the sum of the OPT_{in} values for each child, since the children are not connected:

T(2) = 

- $OPT_{in}[2] = 4$
- $OPT_{out}[2] = 1+2 = 3$

added vals for second layer

T = 

What is the recurrence for $OPT_{in}[u]$?

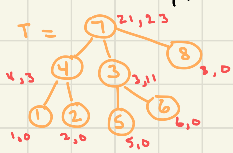
→ For all u in T which are not leaves, $OPT_{in}[u]$ will be the weight of u, plus the $OPT_{out}[v]$ value for each of u's children

- Why? B/c if including u, we can't include u's children. But we can include u's "grandchildren" so for all of u's children we add up the weights of the MIS' that don't include the child.
- Formally, $OPT_{in}[u] = w[u] + \sum_{v \in ch(u)} OPT_{out}[v]$
  $ch(u)$ aka u's children

What about $OPT_{out}[u]$?

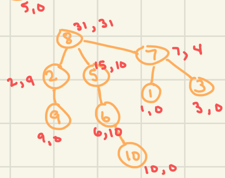
→ We can't include u, but that doesn't mean we are forced to include u's children. Instead, for each child v, we can check whether including v (& thus excluding v's children) or excluding v will give us a higher weight.

- For each child, decide if it should be included (independently).
- Formally, $OPT_{out}[u] = \sum_{v \in ch(u)} \max(OPT_{in}[v], OPT_{out}[v])$

→ T = 

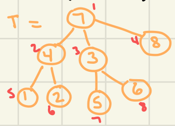
so ans = max(21, 23) = 23 !

→ Another Ex:



How would we implement the solution in code?

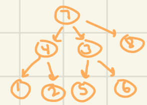
→ The input T will be expressed as a list of lists of each nodes neighbors, and w is a list of each nodes weights. Eg, for ex 1:



T = [[2,3,4],[1,5,6],[1,7,8],[1],[2],[2],[3],[3]]

w = [7,4,3,8,1,2,5,6]

→ First, we should convert T into a dir. graph where we pick any node as the root, and have all the edges "point down":



- In code, this becomes a list G where G[u] is a list of u's children.

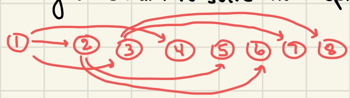
G = [[2,3,4],[5,6],[7,8],[ ],[ ],[ ],[ ],[ ]]

- This makes it much easier to work with. Now we know where to start w/ the "leaves" (aka all nodes u where G[u] = [ ] (aka empty).

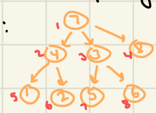
How will we ensure that we solve the subproblems in the right order?

→ We need to ensure that when we solve subproblem rooted at node u, we have already solved the s.p. for each of u's children.

→ SOLUTION: Use a topological ordering! We want to solve the subproblems in reverse topological order.



- Ex:



- One possible topo sort: [1,2,3,4,5,6,7,8]. The reverse of this = the order in which we'll perform the recurrence.

What is the pseudocode?

```
MIS-Tree(T,w):
  Convert T to dir. graph (?)
  d_in, d_out = [0]*n, [0]*n
  node_order = Topo_sort(V)
  node_order = reverse(node_order)
  for u ∈ V in "node_order" order:
      d_in[u] = w[u]
      for v ∈ T[u]:
          d_in[u] += d_out[v]
          d_out[u] += max(d_in[v], d_out[v])
  return max(d_in[1], d_out[1])
```

obtain the reverse topo ordering in which we will solve the subproblems

V = set of vertices in T

For all of u's children. If u is a leaf, then the line above this one covers the "base cases".

return the max of OPT_{in}, OPT_{out} for the root of the tree.

- Q: are we always returning the max from u = 1? does it have to be the root of the o.g. tree, or the root of the dir. graph we convert T into?

→ There are 2n subproblems (2 for each vertice) and each takes O(n)-time.

What is the running time?

→ However, since T is undir, the RT is not $O(n^2)$ because computing the recurrence for each n is O(|ch(u)|)-time.

→ Since T is an undir. TREE, there are exactly n-1 edges. So total RT = O(n).

→ Similar concept to BFS RT for undir. graphs.

— Common DP Patterns —

What are common DP patterns if the input is...

an array A of length n?

1. $\forall\, i \in [n]$ (aka for all $i = 1, \dots, n$): $OPT[i]$ = OPT / the "optimal solution" given the "input" is now $(A[1:i])$.
   - Max-in-Array
2. $\forall\, i \in [n]$: $OPT[i]$ = OPT given the "input" is now $(A[i:n])$.
3. $\forall\, i \in [n]$: $OPT[i]$ = OPT given "input" is now $(A[1:i])$, that somehow involves $A[i]$
   - Longest Increasing Subsequence
   - (Kind of) MIS in trees
4. $\forall\, i \in [n]$ and $\forall\, j \in [i, n]$ (aka all $j = i+1, i+2, \dots n$): $OPT[i][j]$ = OPT / optimal solution given the "input" is $(A[i:j])$.
   - LPS

(A, K), where A = array and K = positive integer?

5. $\forall\, i \in [n]$ and $\forall\, j \in \{0, 1, \dots K\}$. $OPT[i][j]$ = OPT given the "input" is now $(A[1:i], j)$
   - 0/1 Knapsack

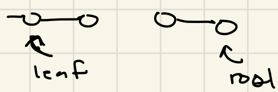
(A, B), where A and B are arrays of length $\underline{m}$ and $\underline{n}$?

6. $\forall\, i \in [m]$ and $\forall\, j \in [n]$: $OPT[i][j]$ = OPT given the input is $(A[1:i], B[1:j])$
   - Edit Distance

(T), where T = a rooted tree with vertex set V?

7. $\forall\, u \in V$: $OPT[u]$ = OPT given the input is (T_u) (tree rooted at u)
8. $\forall\, u \in V$: $OPT[u]$ = OPT given the input is (T_u), that somehow involves u.
   - MIS in Trees

HW:



pick $A[i] \rightarrow$ can't pick $A[i+1]$

- special version of MIS in trees where T is a path

Ch.5: Shortest Paths

What is this chapter about?

→ Ch 5.1–5.3 consider variants of the Single-Source Shortest Path (SSSP) problem

→ 5.4 considers the All-Pairs Shortest Path (APSP) problem.

What is the SSSP problem?

→ Input: (G, s) where G is a directed graph with edge lengths ℓ.

- s is a vertex in G ($s \in V$) that is the "source" vertex.

→ Goal: Return an array d of length n s.t. for all $v \in V$, $d[v]$ is the shortest path from s to v.

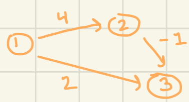
Have we done this problem before?

→ Yes! If $\ell = 1$ for all edges (all edges have same length), then running BFS(G, s) would return the SSSPs.

How do we represent edge lengths in our dir-graph input?

→ We embed the edge length in the adjacency list of edges.

→ Ex:



$G = [[(2,4),(3,2)], [(3,-1)], []]$

node 1 has an edge pointing to node 3 of length 2

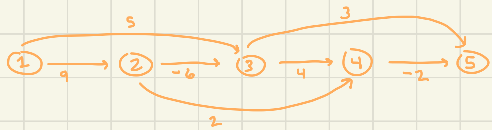
– DAG DP –

What is the problem?

→ Input: (G, s) where G is a DAG (directed acyclic graph) w/ edge lengths.

→ Goal: return the SSSP. e.g., for all $v \in V$, the shortest path from node s to node v.

→ Ex:



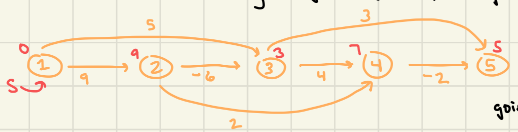
$s = 1$

- Whenever the input to a problem is a DAG, it is helpful to look at the graph in topological order (like Ex above).

What are the subproblems?

→ For all $v \in V$, let OPT[v] denote the distance from s to v.

→ INTUITION: comparing lengths of ways to get from s to v:



- ANS: [0, 9, 3, 7, 5]
- We choose "3" for d[3] b/c going directly from ① → ③ takes 5 but going ①→② ②→③ takes $9 + (-6) = 3$

→ We will return $d = OPT$.

What is the base case?

→ $OPT[s] = 0$ (distance from s to itself).

What is the recurrence?

→ Look at all the choices to get from $s \to v$.

- AKA, check OPT[u] for all nodes u which are an in-neighbor (aka "point to") v. This represents the shortest path from s to an in-nbr of v
- Add the distance from $u \to v$
- Choose the smallest option.

→ Formally, for all $v \neq s$, $OPT[v] = \min(OPT[u] + \ell((u,v)))$ for all $u : (u,v) \in E(G)$

- in code, the length of edge (u,v) is the 2nd element in the tuple (v, ℓ) for G[u]

What is the pseudocode?

→ We want to solve the subproblems in **topological order**.

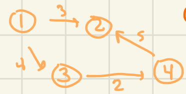
→ Also, for every $v \neq s$, we need to look at the lengths of all of v's in-neighbors. Initially, this info is not stored at G[v], but at G[in-neighbor of v].

- To make it easier, we will also save a graph G' where every edge is reversed and G'[v] tells us the in-neighbors of vertice v.
- Kind of "pre-computing" the in-neighbors of v.

```
DAG-DP(G,s).
  d = [∞] * n
  d[s] = 0                          → base case
  ordering = Topo_Sort(G)
  G' = G with each edge reversed
  for v ∈ V in "ordering" order:
    for u in G'[v]:                 → G'[v] lists v's in-neighbors
                                      in G, and the edge lengths.
      d[v] = min(d[v], d[u] + ℓ(u,v))
  return d
```

How would we create G'?

→
```
G' = [[]] * n
for u in V(G):
  for v in G[u]:
    add u,ℓ to G[v]
```



G = [[(2,3),(3,4)], [], [(4,2)], [(2,5)]]

G' = [[], [(1,3),(4,5)], [(1,4)], [(3,2)]]

What is the running time?

→ Topo sort w/ DFS is $O(m+n)$

→ Computing G' is $O(m+n)$

→ Computing d[v] takes O(# of in-neighbors (v))-time. The sum of in-degrees (aka edges!) is m, so the total RT is $O(m+n)$

How else could we use the logic of the DAG DP problem?

→ To find the longest paths instead of shortest; change $d[-\infty]$ to $d[\infty]$, and find max() instead of min()

→ Every DP has an underlying DAG

- The vertices are like subproblems
- The edges ≈ dependencies in the recurrence.

→ In this way, DAG DP is kind of a representation of all DP problems.

– Bellman – Ford –

What is a negative cycle?

→ A cycle in a dir. graph where the sum of the edge lengths is < 0.

→ If we consider finding SSSP for a graph w/ negative cycles, the problem becomes NP-hard (no poly-time alg discovered yet).

Why do negative cycles make SSSP harder?

→ It becomes difficult to say what the "shortest path" for $s \to v$ is, because you could choose to cycle infinitely through a negative cycle of vertices, because it allows the length to get smaller & smaller to negative infinity.

What is the problem statement?

→ Input: (G, s), where G = a dir. graph w/ **no negative cycles**, and s = the source vertex.

→ Goal: Return array d representing the SSSP.

What are the subproblems?

→ Unlike 5.1, in this problem G is not necessarily acyclic, so we can't utilize Topo-Sort to help us shrink & order the problems.

→ For all $v \in V$ and $j \in \{0, 1, \ldots n-1\}$ where n = # of vertices, let OPT[v][j] denote the length of the shortest path from $s \to v$, where we can have at most j edges in our path.

- j ≈ our "budget" of edges. Kind of like 0/1 Knapsack

→ j goes from $0 \to n-1$ because a path starting at $\underline{s}$ has max $n-1$ edges; any more edges than that would mean you are repeating edges.

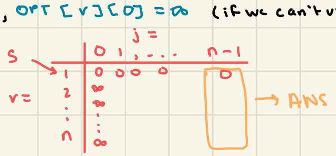
What we will return?

→ The column / list at OPT[ ][n-1] (aka, s.p. for each v when allowed to use as many edges as you want).

What are the base cases?

→ OPT[s][j] = 0 (path from $s \to s$ will use no edges). for all j.

→ For all $v \neq s$, OPT[v][0] = ∞ (if we can't use any edges, the length to get to v is ∞)



What is the recurrence?

→ For all $v \neq s$ and $j \geq 1$, the path will either:

- have at most $j-1$ edges, aka $OPT[v][j-1]$. basically, the length of the path $s \to v$ where we don't use the "option" to add 1 more edge. OR,
- have j edges, in which case we use any of the paths that lead to an in-neighbor of v and have $j-1$ edges, and then we add on the edge from in-neighbor $\to v$ as our "j^{th}" edge.

→ We want the min. of these 2 choices. Formally,

$$OPT[v][j] = \min\left(OPT[v][j-1], \min_{u:(u,v)\in E}\left(OPT[u][j-1] + \ell(u,v)\right)\right).$$

↳ min for all nodes u which are in-neighbors of v.

In what order do we solve the subproblems?

→ $OPT[v][j]$ depends on $OPT[v][j-1]$, so we will fill it out column-by-column, left-to-right.

- For every column j, calculate every $d[v][j]$ according to the recurrence.

What is the pseudocode?

```
Bellman-Ford (G, s):
  d = [∞] * (n * n)
  d[s][0] = 0                      → base case
  G' = G w/ each edge reversed
  For j = 1,...n-1:                → so that we iterate column-by-column
    for v ∈ Vertices of G:
      d[v][j] = d[v][j-1]          → start by setting OPT = OPT[v][j-1], and then check if we can do better
      for u in G'[v]:              → all of v's in-neighbors
        d[v][j] = min(d[v][j], d[u][j-1] + ℓ(u,v))
  return d[·][n-1]
```

What is the Running Time?

→ There are n columns. Each takes $(m+n)$-time, b/c we're basically performing DAG DP on each.

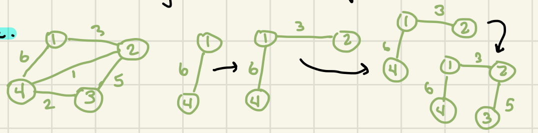
→ Total RT $= O(n)(m+n) = mn + n^2 \approx O(mn)$-time (b/c $m > n$)

— Dijkstra's Algorithm —

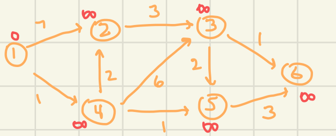
What is the problem statement?

→ Find the SSSP, like in the other problems, but this time, all edge lengths ℓ are nonnegative. Find shortest path from $s \to v$ for all $v \in V$.

→ Similar to Prim's (ch.3): keep picking the heaviest edge that adds a new vertice to the bubble.



→ Ex: G =



$d = [\infty, \infty, \infty, \infty, \infty, \infty]$

What is the algorithm?

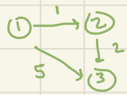
→ Set all SSSP lengths initially to be ∞

→ Starting with node $\underline{s}$, add s to your "bubble" and then process it

What does it mean to "process" an edge?

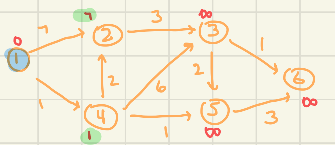
→ "Process it" = relax(u,v): see if we can decrease our "current estimate" of distance from $s \to v$ by finding a node $\underline{u}$ that is an in-neighbor of v, and calculating length(s,v) to be length(s,u) + length(u,v).

Ex:



- The SSSP for $1 \to 3$ is $\underline{3}$ if we go through node 2 rather than directly from 1 to 3.

- after processing node $s = 1$, we relax edges (1,4) and (1,2):



$d = [0, 7, \infty, 1, \infty, \infty]$

Which node do we process next?

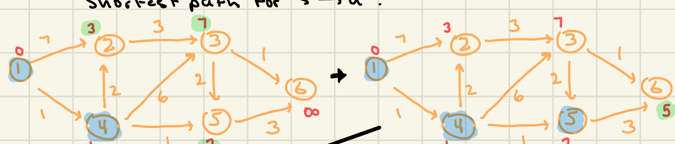
→ The vertex with the smallest $d[v]$ value so far; e.g. node that is currently least distance from s.

- Add this node to the bubble, then "process it" to relax its edges.

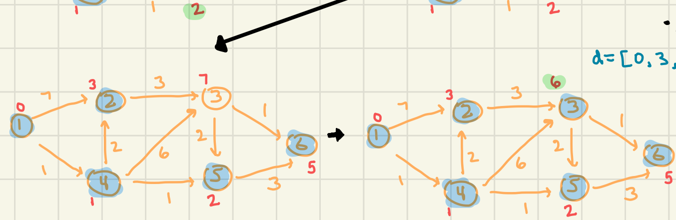
How do we relax the edges of the node added next?

→ For every out-neighbor u of node v, check if $(d[v] + \ell(v,u)) < d[u]$

- aka, is the path $s \to v$ + path $v \to u$ shorter than the current shortest path for $s \to u$?



- ANS: $d = [0, 3, 6, 1, 2, 5]$



Summary: What is the intuition for this alg?	→ Starting with node $\underline{s}$ and then by choosing the node with the smallest $s \to v$ path-length, and until the bubble doesn't contain all vertices: • add the node (v) to the "bubble" • "process" v by relaxing the edges, aka, for each outneighbor u of v, check if the length of $s \to v$ plus the distance from $v \to u$ is <u>smaller</u> than the current value set for length $(s \to u)$ • Finally, return the array of shortest paths.
What is the pseudocode?	
What is the RT?	→ $O(n^2)$

— Floyd-Warshall —

What is the APSP Problem?

→ "All Pairs Shortest Path"

→ Input: a directed graph G with edge lengths (negative allowed) ℓ, where G has no negative cycles.

→ Goal: return an $n \times n$ array d, where for all $u, v \in V$, $d[u][v]$ is the length of the shortest path from u to v.

- Basically same as Bellman-Ford, except no specified s.

Why can't we just run Bellman-Ford n times?

→ We could, but the RT would be n^4

→ With DP, we can do this faster.

What are the subproblems?

→ Instead of shrinking by the amount of edges we can use to get from u to v (like in B-F), we shrink by reducing the set of vertices, r, that we are allowed to use to get from u to v.

→ For all $u \in \{1, \dots n\}$, all $v \in \{1, \dots n\}$, and all $r \in \{0, 1, \dots n\}$, $OPT[u][v][r]$ will denote the length of the s.p. from $u \to v$ with only vertices $\{1, \dots r\}$ available as intermediate vertices.

- total of $n \cdot n \cdot (n+1) \approx n^3$ subproblems

What will we return?

→ The "table $r = n$"; aka, the table of u & v path lengths when $r = n$.

- Think of $OPT[u][v][r]$ as a set of n $n \times n$ tables where each table has the s.p.s from all u to all v when we are allowed to use $[r]$ vertices.

→ RET: $OPT[\cdot][\cdot][n]$.

What is the base case?

→ If $r = 0$, we can't use any intermediate vertices, so $OPT[u][v][0]$ will be ∞ UNLESS:

- $u = v$, in which case $OPT[u][v][0] = 0$. OR
- if (u, v) is an edge in G, in which case $OPT[u][v][0] = \ell(u, v)$.

$r = 0$ {, $r = 1$ {, ..., $r = n$ {

u= \ v=	1	2 . . . n
1	0	
2		0
⋮		
n		

u= \ v=	1	2 . . . n
1	0	
2		0
⋮		
n		

u= \ v=	1	2 . . . n
1	0	
2		0
⋮		
n		

What is the recurrence?

→ The table for $r=n$ will rely on the table for $n-1$, and so on.

(Relying on previous table)

→ If $r \geq 1$, then we are allowed to use nodes $1-r$ along the way from $u \to v$.

There are 2 options for OPT[u][v][r]

1) Don't use node r, in which case OPT[u][v][r] = OPT[u][v][r-1]

2) Use node r, in which case we want the distance from node u to node r, plus the distance from node r to node v.

- We want to obtain these values from the sp wasn't allowed as an intermediate vertex, aka:

$$\underbrace{\text{OPT}[u][r][r-1]}_{\text{dist}(u,r)} \quad \text{and} \quad \underbrace{\text{OPT}[r][v][r-1]}_{\text{dist}(r,v)}$$

→ We want the minimum of these options. Formally

$$\text{OPT}[u][v][r] = \min \begin{cases} \text{OPT}[u][v][r-1], \\ \text{OPT}[u][r][r-1] + \text{OPT}[r][v][r-1] \end{cases}$$

What is the pseudocode?

```
Floyd-Warshall(G):
  d = [∞] * (n x n x (n+1))
  For u in range(1,...n):                      basic case: Filling out table r=0
        d[u][u][0] = 0
        for v ∈ G[u]:
            d[u][v][0] = ℓ(u,v)
  for r in range(1,...n):                      for r,u,v ∈ V
    for u in range(1,...n):
      for v in range(1,...n):
        d[u][v][r] = min(d[u][v][r-1], d[u][r][r-1]+d[r][v][r-1])
  return d[·][·][n]  ──→  return table n
```

What is the RT?

→ $n \times n \times n$ subproblems so $O(n^3)$-time.

Midterm 2 Study Guide - Dynamic Programming

L.I.S.

$A = [3, 4, 1, 5, 2, 3, 6, 1]$

- **Subproblems**: OPT[i] = LIS ending on A[i]
- **Return**: max element in OPT
- **Base Case**: OPT[1] = 1

OPT[1] = 1

OPT[2] = 2

OPT[3] = 1

OPT[ ] = [1, 2, 1, 3, 2, 3, 4, 1]

ANS = 4

→ **Intuition**: For OPT[i], look at all elements A[1:i] (aka all elements up to element i)

- Narrow down & only look at elements x in A[1:i] where A[x] < A[i] (b/c need an increasing sequence)
- Of all of these, check which one has the max OPT value, OPT[x].
- OPT[i] = OPT[x] + 1.

→ **Recurrence**: $OPT[i] = 1 + \max_{j \in C}(OPT[j])$ for $C = \{j \mid j < i \text{ and } A[j] < A[i]\}$.

→ **ALG**:

```
d = [1] * n   (because every LIS will have atleast length 1)
for i = 2, ..., n:
    for j = 1, ..., i-1:
        if A[j] < A[i]:
            d[i] = max(d[i], d[j] + 1)
return max(d)
```

→ **RT**: $O(n^2)$ b/c 2 for loops

L.P.S.

→ Problem Statement: Find longest sequence of characters in A s.t. the sequence is a palindrome.

- The sequence doesn't have to be subsequent in A. For ex, abba is a P.S. of abracadabra

A = acbba

OPT:

i \ j	a	c	b	b	a
a	1	1			
c		1			
b			1		
b				1	
a					1

→ **RT**: $O(n^2)$

- **Subproblems**: OPT[i][j] = length of LPS for A[i:j]. E.g., OPT[2][5] = A[2:5] = cbba
- **Return**: OPT[1][n]
- **Base Cases**: for all i = 1, ..., n, OPT[i][i] = 1
- **Intuition**:

→ If A[i] = A[j], we have a PS of at least length 2 just by the subsequence "A[i] A[j]". But if the chars between i and j also have a palindrome, our LPS would be even longer. So:

- CASE 1: If A[i] = A[j], $OPT[i][j] = 2 + \underbrace{OPT[i+1][j-1]}_{\text{the substring between } i \text{ and } j,\ A[i+1:j-1]}$

→ If A[i] ≠ A[j], then we want to compare the LPS for the substring without A[i] — aka OPT[i+1][j] — and the substring w/o A[j] — aka OPT[i][j-1]. We keep whichever is larger. So:

- CASE 2: If A[i] ≠ A[j], OPT[i][j] = max(OPT[i+1][j], OPT[i][j-1]).

Midterm 2 Study Guide - Dynamic Programming

0/1 Knapsack

→ **Problem Statement**: (v, w, B) where B = weight limit, and for n item choices, $v[i]$ = value of item i and $w[i]$ = weight of item i.

- Return the maximum value of the knapsack where we can only take ALL or NONE of each item.

$B=4$ $v=[3,2,2]$ $w=[1,4,3]$

OPT:	0	1	2	3	4
1	0	3	3	3	3
2	0	3	3	3	3
3	0	3	3	3	5

- **Subproblems**: $i=1,...n$ and $j=0,...B$ $OPT[i][j]$ = max value of knapsack if we can only choose items $1,...i$ and the weight limit is j.
- **Base Case**: for all i where $j=0$, $OPT[i][0]=0$
- **Base case**: For all j where $i=1$, if $w[i] \leq j$, $OPT[1][j]=v[i]$. Else, if $w[i]>j$, $OPT[1][j]=0$.
- **Return**: $OPT[n][B]$ (bottom right square)

Intuition: Fill out table row by row, aka each subproblem ≈ introducing a new possible item.

→ If $w[i]>j$, we can't possibly add item i to our bag, so our max value is the same as the 1 where item i wasn't an option:

- CASE 1: $OPT[i][j]=OPT[i-1][j]$

→ If $w[i]<j$ and we still want to add item i to our bag, we have a value of at least $v[i]$. After adding item i, we have $(j-w[i])$ pounds of space left. $OPT[i-1][j-w[i]]$ will give us the maximum value we can obtain to fill the rest of the bag. We add $v[i]$ to this.

- CASE 2A: $OPT[i][j]=OPT[i-1][j-w[i]]+v[i]$

→ If $w[i]<j$ and we still DON'T want to add item i to our bag, our max value is the same as if item i wasn't an option:

- CASE 2B: $OPT[i][j]=OPT[i-1][j]$

→ We want the max of 2A and 2B for Case 2.

→ **Recurrence**: For all $i \geq 2$:

$$OPT[i][j] = \begin{cases} OPT[i-1][j] & \text{if } w[i]>j \text{, and} \\ \max(OPT[i-1][j],\ OPT[i-1][j-w[i]]+v[i]) & \text{otherwise.} \end{cases}$$

→ **RT**: $O(n^2)$

Midterm 2 Study Guide - Dynamic Programming

Edit Distance

→ **P.S.**: Given two strings of length m and n (A, B), return the min. # of "moves" needed to turn str A into str B... aka the 'edit distance'

→ "Moves": Insert a character anywhere in A, delete a character anywhere from A, or replace any 1 character in A w/ another.

→ **Subproblems**: For $i = 0, \dots m$ and $j = 0, \dots n$, OPT[i][j] = edit distance to turn A[1:i] into B[1:j].

↳ e.g. OPT[2][3] = turning "st" into "wat"

→ **Return**: OPT[m][n]

j=0 represents B as the empty string, ""

OPT	""	w	a	t	e	r
""	0	1	2	3	4	5
s	1	1	2	3	4	5
t	2	2	2	2	3	4
a	3	3	2	3	3	4
r	4	4	3	3	4	$\boxed{3}$ — ANS

→ **Base Cases**: Editing an empty str A into a string B of length j will take j insertions, so OPT[0][j] = j for all j.

- Same concept for editing str A of length i into empty string B via i deletions (OPT[i][0] = i for all i).

Ex: OPT[3][4], "sta" → "wate"

→ **Recurrence**: for $i \geq 1$ and $j \geq 1$, we have to edit A[1:i] s.t. its last character = B[j]. 3 OPTIONS to do this:

1) Edit A[1:i] into B[1:j-1] and then insert B[j] sta → wat → insert "e" → wate
   - AKA, OPT[i][j-1] + 1 (for the insertion)

2) Edit A[1:i-1] into B[1:j] then delete A[i] sta → wate a → delete "a" → wate
   - AKA, OPT[i-1][j] + 1 (for the deletion)

3) Edit A[1:i-1] into B[1:j-1] then replace A[i] with B[j] sta → wat a → replace "a" with "e" → wate
   - If A[i] ≠ B[j], then OPT[i][j] = OPT[i-1][j-1] + 1
   - if A[i] = B[j], then we don't actually need to spend a "move" on the replacement, so OPT[i][j] = OPT[i-1][j-1]

→ Take the min of these 3 options. Fill table T-D, L-to-R.

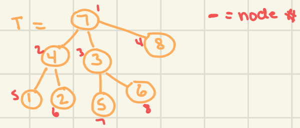
→ **RT**: O(mn)

Midterm 2 Study Guide - Dynamic Programming

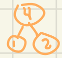
Ind. Set in Trees

→ Input: (T, w) where T = a tree rooted at vertex 1 and w = an array of length n (# of nodes) where $w[u]$ = the positive int "weight" of node u.

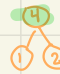
- Basically a tree w/ weighted nodes.
- Tree = acyclic, connected undir. graph with $n-1$ edges.



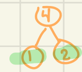
→ Goal: return weight of the **independent set** (subset of nodes where NONE of them have an edge to each other) with the maximum total weight of all the nodes.

→ Subproblems: For all $u \in V$, $T(u) \approx$ the subtree rooted at node u. E.g. $T(2)$ =  . For all $T(u)$ subtrees:

- $OPT_{in}[u]$ = the weight of the MIS of $T(u)$ that MUST include node u.
- $OPT_{out}[u]$ = weight of MIS of $T(u)$ that CANNOT include node u.



$OPT_{in}[2] = 4$



$OPT_{out}[2] = 2 + 1 = 3$

DP Patterns

→ Input = Array of length n:

1. $\forall\ i \in [n]$ (aka for all $i = 1, \ldots n$) : $OPT[i]$ = OPT / the "optimal soln" given the input is now $(A[1:i])$.
2. $\forall\ i \in [n]$: $OPT[i]$ = OPT given the "input" is now $(A[i:n])$.
3. $\forall\ i \in [n]$: $OPT[i]$ = OPT given "input" is now $(A[1:i])$, that somehow involves $A[i]$
   - LIS: required to end on element $A[i]$
4. $\forall\ i \in [n]$ and $\forall\ j \in [i, n]$ (aka all $j = i+1, i+2, \ldots n$):
   $OPT[i][j]$ = OPT / optimal solution given the "input" is $(A[i:j])$.
   - LPS: finding LPS for $A[i:j]$

→ Input = (A, k), Array of length n and int k:

5. $\forall\ i \in [n]$ and $\forall\ j \in \{0, 1, \ldots k\}$. $OPT[i][j]$ = OPT given the "input" is now $(A[1:i], j)$
   - 0/1 knapsack: find max value if we have weight limit j and items $1-i$.

→ Input = (A, B), Arrays of length m and n:

6. $\forall\ i \in [m]$ and $\forall\ j \in [n]$: $OPT[i][j]$ = OPT given the input is $(A[1:i], B[1:j])$
   - Edit Distance: edit dist to turn $A[1:i]$ into $B[1:j]$.

→ Input = rooted tree T with vertex set V:

7. $\forall\ u \in V$: $OPT[u]$ = OPT given the input is (T_u) (tree rooted at u)
8. $\forall\ u \in V$: $OPT[u]$ = OPT given the input is (T_u), that somehow involves u.

Midterm 2 Study Guide - Shortest Paths

DAG DP

→ **P.S.**: Input = (G, s) where G = directed, acyclic graph with edge lengths ℓ and $s \in V$.

Find the lengths of the shortest paths from $s \to v$ for all $v \in V$. Return in an array $\underline{d}$.

→ **Subproblems**: $OPT[v]$ = length of S.P. from $s \to v$ for all $v \in V$. We return $d = OPT$.

→ **Intuition**: Each time we consider a node u, we consider all of its in-neighbors (nodes pointing to it). For each in-neighbor x, the potential path length for $s \to u$ = $OPT[x] + len(x, u)$ - the SP to x + the length of path from x to u.

- We must "consider" a node only AFTER we have "considered" (aka computed OPT) each of its in-neighbors, so we should consider the nodes in **TOPO-ORDER**.

→ **Base Case**: $OPT[s] = 0$

→ **Recurrence**: Choose the minimum $(OPT[x] + \ell(x,u))$ for all in-neighbors of v.

- $OPT[v] = \min\limits_{u:(u,v) \in E} OPT[u] + \ell(u,v)$

→ **ALGORITHM**:

1) initialize $d = [\infty] * n$ and $d[s] = 0$
2) Construct G' = Reverse graph of G to get all in-neighbors of node u in graph G.
3) In Topo_Sort Order: For v in V:

    for each out-neighbor of v in G' (for $u \in G'[u]$):

        $d[v] = \min(d[v], d[u] + \ell(u,v))$ ⟿ updates for each pos. in-neighbor

→ **RT**: $O(m+n)$

Midterm 2 Study Guide - Shortest Paths

Bellman - Ford

→ **P.S.**: Return array of shortest paths from node s for (G, s) where G = directed graph with no negative cycles. Edge lengths CAN be negative.

→ **Subproblems**: A path from s to any u can have max $n-1$ edges. For all $v \in V$ and $j \in \{0, 1, ... n-1\}$, $OPT[v][j]$ = The length of the path from $s \to v$ using at most j edges.

→ **Return**: List at $OPT[\ ][n-1]$ (aka no budget on num. of edges)

→ **Base Case**: $OPT[s][j] = 0$ for all j. $OPT[v][0] = \infty$ for all $v \neq s$ (can't form path w/ 0 edges).

→ **Recurrence**: For all $v \neq s$ & $j \geq 1$, the path has 2 Options:

1) Do not utilize the ability to use all j edges. Only use $j-1$ edges, aka same soln as that where budget = $j-1$
   - CASE 1: $OPT[v][j] = OPT[v][j-1]$
2) Utilize all j edges. Our "j^{th} edge" will be the one pointing to v, so it must come from an in-neighbor u of v. Therefore, for all in-neighbors u of v, find $OPT[u][j-1]$ (the SP with $j-1$ edges), and add $\ell(u,v)$ b/c its the j^{th} edge.
   - CASE 2: $\min(OPT[u][j-1] + \ell(u,v))$ for all u = in-neighbor of v.

→ Take the minimum of the 2 cases.

→ Fill out table column-by-column, left-to-right. Because $OPT[v][j]$ depends on prev column, $OPT[v][j-1]$.

→ **RT: $O(mn)$**

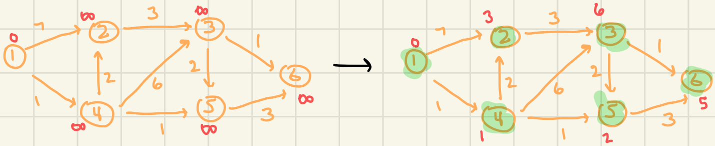
Dijkstra's

→ **PS**: SSSP for (G, s) where all edge lengths are non-negative.

→ **Intuition**: We initialize all the SP lengths to be ∞ (aka $d = [\infty] * n$). $d[s] = 0$ b/c path from $s \to s$.

1) Choose u, the node w/ lowest d value. u must also NOT be in set S. At first, this will be s ($u = s$).
2) Add u to our set of processed vertices S.
3) Look at all out-neighbors x of u. $d[u]$ = SP from $s \to u$, so a possible path from $s \to x$ could be $d[u] + \ell(u, x)$
   - If this possible path is < current SP for x, update the SP. aka

     $d[x] = \min(d[x], d[u] + \ell(u,x))$ where u is the node we are currently processing.
4) Repeat steps 1-3 until all nodes are in set S. Return d.



→ **RT**: $O(n^2)$

Midterm 2 Study Guide – Shortest Paths

Floyd-Warshall

→ **PS:** Given graph G, find APSP: an $n \times n$ table depicting the SP from $u \to v$ for all $u, v \in V$ (aka all possible pairs of nodes)

→ **Subproblems**: For all $u, v \in V$ and $r \in \{0, \dots n\}$, OPT[u][v][r] = length of SP from $u \to v$ where we can only use nodes $\{1, \dots r\}$ to get from u to v.

→ **Return**: The table OPT[ ][ ][n] — aka all nodes allowed as intermediate.

→ **Base Case**: When $r = 0$,

- OPT[u][v][0] = 0 for all u, v when u = v,
- OPT[u][v][0] = $\ell(u,v)$ for all u, v when $\exists$ an edge $u \to v$,
- and OPT[u][v][0] = ∞ otherwise

→ **Recurrence**: Fill the 3D array table by table, aka r = 0, r = 1, so on. The OPT[ ][ ][0] table was our base case.

For all OPT[u][v][r] when $r \geq 1$, we have 2 options:

1) Don't use the newly allowed intermediate vertex r, in which case OPT[u][v][r] = OPT[u][v][r-1].

2) Use node r in the middle of the path from $u \to v$. In this case, we want to add up length (SP from $u \to r$) + length (SP from $r \to v$). Specifically, we want these lengths from BEFORE r was allowed as an intermediate vertex. AKA OPT[u][v][r] = OPT[u][r][r-1] + OPT[r][v][r-1]

→ Take the min of the 2 options ↑.

→ **RT**: $n \times n \times n$ matrix so $O(n^3)$

Ch. 6: Flows and Cuts

What is the maximum flow problem?

→ RECALL: Shortest Path problems are about finding the fastest way to get a truck from point s to point t

→ Maximum flow problems ≈ sending as many trucks as possible from s to t.

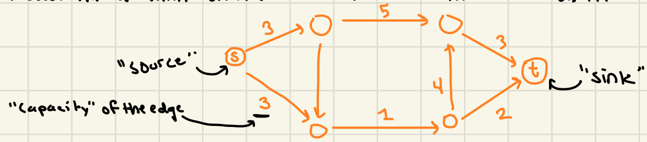
What is a flow network?

→ an input (G, s, t) where

- G = connected, directed graph where each edge e has a "capacity" $c(e) \in \mathbb{Z}^+$ (positive int)
- s = a node in G that represents the **source vertex**.
  - Assume that no edges point to s.
- t = a node in G that represents the **sink vertex**.
  - Assume that no edges are pointing out of t.
  - Assume that $s \neq t$

→ EX: thing of the graph as a map of roads to checkpoints. We want to maximize the amt. of stuff we can send in trucks from point $\underline{s}$ to point $\underline{t}$.

- Each Road has a "limit" on the amount of trucks that can be on it.



RECALL: What is a cut?

→ A subset S of vertices

- $\delta^{out}(S) = \{(u,v) \in E : u \in S, v \notin S\}$ denotes the set of edges leaving / crossing the cut (b/c they point from a vertex in S to a vertex not in S).
- $\delta^{in}(S) = \{(u,v) \in E : u \notin S, v \in S\}$ denotes the set of edges entering S.

→ S is an **s-t cut** if $s \in S$ AND $t \notin S$

What is a flow?

→ A function $f: E \to \mathbb{R}$ ... a function that gives a <u>number</u> to each edge in $E(G)$.

What are $f^{out}(\,)$ and $f^{in}(\,)$?

→ For a cut S (which could also just be a single vertex, e.g. $S = \{u\}$), $f^{out}(S)$ represents the amount of flow leaving S, e.g: For all edges LEAVING S, the sum of the "flows" for each of those edges.

$$f^{out}(S) = \sum_{e \in \delta^{out}(S)} f(e)$$

→ $f^{in}(S) = \sum_{e \in \delta^{in}(S)} f(e)$ ...the amount of flow entering S.

→ When S is a single vertex, e.g. $S = \{u\}$, we write $f^{out}(u)$ instead of $f^{out}(\{u\})$.

What does it mean for a flow to be feasible?

1. **Capacity Constraints:** For every edge e, the amount of "flow" on the edge is not greater than its capacity.
   - For all $e \in E$, $0 \leq f(e) \leq c(e)$
2. **Conservation:** For every vertex except s and t, the amount of flow entering v = the amount of flow leaving v.
   - $\forall$ $v \in V$ where $v \neq s, v \neq t$: $f^{in}(v) = f^{out}(v)$.

What is the value of a flow f?

→ Defined as $|f|$, the total amount of flow leaving vertex s.

- $|f| = f^{out}(s)$

What is a maximum flow?

→ Given (G, s, t), it is a flow f where $|f|$ is maximized.

What is the capacity of a cut S?

→ The sum of the capacities of all edges leaving the cut.

- $c(S) = \sum_{e \in \delta^{out}(S)} c(e)$

What is a minimum s-t cut?

→ An s-t cut S s.t. the capacity $c(S)$ is minimized.

How would you represent a flow in code?

→ The same way we represent edges that have lengths, weights, or capacity.

→ Ex: $1 \xrightarrow[4]{5} 2 \xrightarrow[2]{4} 3 \xrightarrow[5]{10} 4$, $s = 1$ and $t = 4$. Let the red numbers = the flow for each edge. Then we would represent G as:

$G = [[(2,5)], [(3,4)], [(4,10)], []]$ and f as:

$F = [[(2,4)], [(3,2)], [(4,5)], []]$.

- Ford-Fulkerson -

What is the input and goal?

→ INPUT: A flow network (G, s, t)

→ GOAL: Return a **maximum s-t flow**. AKA a flow w/ the max sum of all flow leaving s.

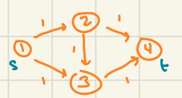
What is a simple example?

→ Ex: $G = 1 \xrightarrow[4]{5} 2 \xrightarrow[4]{4} 3 \xrightarrow[4]{10} 4$, $s = 1$, $t = 4$.

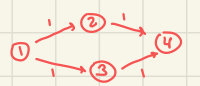
- $f([2 \to 3])$ can be at most 4 due to capacity constraint, and $f^{in}(2)$ must $= f^{out}(2)$ so $f([1 \to 2])$ must also be 4. Finally $f(3 \to 4)$ will also have to be 4.
- ANS: $F = [[(2,4)], [(3,4)], [(4,4)], []]$

What are some other examples?

→ G =

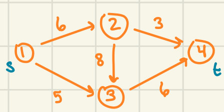


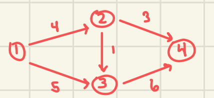
ANS: F =



$|f| = 2$

→ G =





$|f| = 9$

→ NOTE: $|f|$ cannot possibly be greater than $f^{in}(t)$.

What is one natural approach to this problem?

→ Use DFS (or other pathfinding alg) to find a path in G from $s \to t$.

→ For each possible path P, for each edge (u,v) in P,

- set $\Delta_P = \min_{e \in P} c(e)$, aka the minimum capacity of all edges in P.
- increase the flow of each edge in P by Δ_P, aka $f(e) = f(e) + \Delta_P$
- In G, decrease the capacity of each edge in P by Δ_P, so as to represent the "remaining capacity" of e after we have considered a possible path P... aka $c(e) = c(e) - \Delta_P$

→ Repeat above process for all paths until we can't make any more progress.

Will this approach always be correct?

→ NO! It works if G happens to be an s-t path, but not in general.

→ Why? Because once we consider one s-t path P that isn't actually optimal, it affects how we treat the other paths & then our final answer too.

- We have to be able to undo the changes made to $f(e)$ and $c(e)$ when we consider a path P.

What is the correct approach to the Ford-Fulkerson problem?

→ Rather than searching for s-t paths to consider in G, search for them in the **residual network** G_f of G.

- basically, use another graph G_f to track updates made after considering a path P. Namely, the "remaining capacities", and the things that we can undo.

→ We need to find the residual network for G, given our current "working" flow f.

How do we find the residual network?

- It tracks the leftover capacities & how much we can undo.

```
Residual (G, f):
  G_f = (V(G), E_f = ∅)
  for e = (u,v) ∈ E(G):
    if f(e) < c(e):
      add (u,v) to E(G_f)
      set c(e) in G_f = c(e) - f(e)
    if f(e) > 0:
      rev = (v,u)
      add rev to E(G_f)
      set c(rev) in G_f = f(e)
  return G_f
```

forward edge → add (u,v) to $E(G_f)$

backwards edge → rev $= (v,u)$

→ start off by having no edges in G_f

→ (for each edge u,v in the og graph G):

→ if the flow of e is less than the real capacity of e in G, we should add that edge to our residual graph

- In G_f, we should set $c(e)$ to be the remaining capacity.

→ if we are sending flow on edge e, we account for allowing us to "undo" the change by also adding the backwards / reversed edge to G_f

- Why? B/c if we end up using (u,v) in our final flow, then we want to decrease the amt. of flow being sent on (u,v)
- So we should set capacity of the reversed edge to be $f(e)$... aka the amount that we can undo

So what will our actual ALG for Ford-Fulkerson be?

→ Very similar to our first approach, except we do not modify G and instead search for s-t paths P in G_f

→ For each s-t path P in G_f, set Δp = the min. capacity of all edges in P

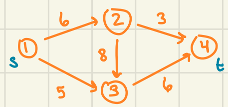
- then, augment f along P by the min. residual capacity $c_f(e)$ over all edges in P ($e \in P$)... aka, "increase" $f(e)$ by Δp for all edges in P.
- then, update G_f by setting G_f = Residual(G, f).

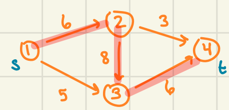
What is the algorithm?

```
Ford-Fulkerson(G, s, t):
    f = [ ] * number of edges
    for all e ∈ E(G):
        f(e) = 0
    G_f = G
    while G_f has an s-t path P:
        Δp = min_{e∈P} c_f(e)
        for all e = (u,v) in P:
            if e is a forward edge:
                f(u,v) = f(u,v) + Δp
            else:
                f(v,u) = f(v,u) - Δp
        G_f = Residual(G, f)
    return f
```

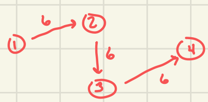
Example problem?

→ Let G = [[(2,6), (3,5)], [(3,8), (4,3)], [(4,6)], [ ]]. Initially, $G_f = G$ and our first s-t path is highlighted below:

G =


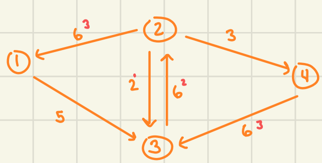
G_f =


- $P = 1 \to 2 \to 3 \to 4$
- $\Delta p = 6$
- $f(e)$ for $e = 1\to2$, $2\to3$, and $3\to4$ = $f(e) + \Delta p = 6$ (b/c $f(e)=0$ for all e, initially)
- F =



→ New Residual graph G_f:

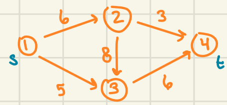
1. $c(e) - f(e) = 8 - 6 = 2$
2. $c(rev) = f(e) = 6$
3. $f(e) = 6$ is NOT < $c(e) = 6$, so we don't add $e = (u,v)$ to G_f. We only add rev



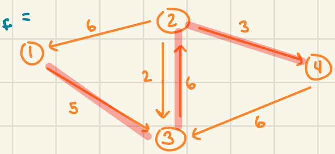
What is the next step?

→ Look for a new s-t path P in G_f:

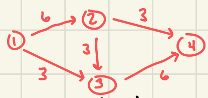
G =



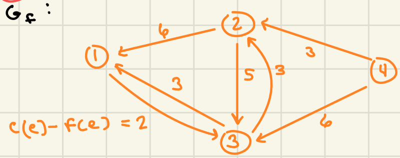
G_f =



- P = 1 → 3 → 2 → 4
- $\Delta_P = 3$
- f(e) for e = 1→3, 3→2, and 2→4:
  - 1→3 is fwd. edge, so f(1→3) = 0 + 3 = 3
  - 3→2 is backward. f(v,u) = f(2→3) = 6
    - f(2→3) = 6 − 3 = 3
  - f(2→4) = 0+3 = 3
  - F =

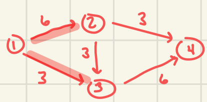


→ New Residual graph G_f:



What is the answer?

→ There are no more s-t paths in G_f, so we return f with |f| = 6 + 3 = 9 :



How can we check if our flow value is correct?

→ Let S = { v | v is a node in G that is reachable from s in the last residual graph G_f }

↓ aka nodes that are directly reachable by s... from a single edge

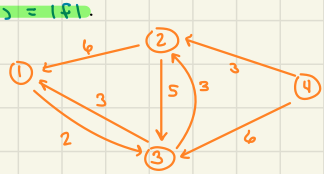
- A set of vertices (a cut)

→ Let c(S) = the sum of the capacities of all edges leaving the cut — in the o.g. graph.

→ If we solved the problem correctly, then c(S) = |f|.

Example?

→ In the Ex above, here was our final G_f:



- S = { 3, 1 }

  c(S) =

What is the RT?

→ In each iteration, the value of the flow, |f|, increases by Δ_P (and Δ_P always ≥ 1)

→ So the ALG makes at most v iterations, where v = value of max. flow

→ Using DFS or a similar pathfinding alg amounts to a total of O(m) time per iteration, so the total RT is O(mv)

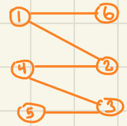
6.2, 6.3: Bipartite Matching; Bipartite Vertex Cover

What is a bipartite graph?

→ An undirected graph G where all the vertices in G can be split into 2 groups L and R s.t. every edge in G has exactly one endpoint in L.

- aka every edge goes from a node in $\{L\}$ to a node in $\{R\}$
- no edges whose endpoints are both in the same group.

→ Ex:



$L = \{1, 4, 5\}$

$R = \{6, 2, 3\}$

→ Given a bipartite graph, assume that we can label each vertex w/ either L or R in $O(m + n)$ time.

What is a matching?

→ A subset of edges where no 2 edges in the set share an endpoint.

- Ex: In graph above, $\{1, 6\}$ and $\{4, 2\}$ are matching, but $\{1, 6, 2, 4\}$ is not.

What is a use case for this concept?

→ Let the nodes in $\{L\} \approx$ kids & the nodes in $\{R\} \approx$ gifts. Each child should get at most one gift, and we want to find the max. # of kids who can get the gifts they want.

– Bipartite Matching –

→ An application of the **maximum flow** problem.

What is the input & Goal?

→ Input: a bipartite graph $G = (L \cup R, E)$

→ Goal: Return a maximum (largest size) **matching** in G.

What can we say about a matching?

→ If M_1 and M_2 are matchings, the set $M_3 = M_1 \cap M_2$ is also a matching.

- RECALL: "$\cap$" = intersection = elements in BOTH x and y.

→ PROOF: $\forall\, e_1, e_2 \in M_1 \cap M_2$, e_1 and e_2 are both in M_1. This implies that e_1 & e_2 don't share an endpoint.

- If an edge is in M_3, it is in both M_1 & M_2, so they can't share an endpoint.

What is the algorithm idea?

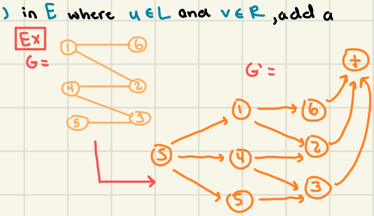
→ We want to convert this graph to a directed graph with capacitized edges – aka a flow – so that we can use **Ford-Fulkerson**.

1) $G \longrightarrow (G', s, t)$ 2) f = max flow 3) convert f to a matching

What are the steps to convert G into a flow?

→ To construct $G' = (V', E')$ from $G = (V, E)$ and $V = L \cup R$

1. add s and t as new vertices. $V' = V \cup \{s, t\}$
2. E consists of a set of edges, all of which span between a node in L and a node in R. So to make this directed, for every edge (u, v) in E where $u \in L$ and $v \in R$, add a directed edge $(u \rightarrow v)$ (from L to R) in E'.
3. For all $u \in L$, add an edge (s, u) to E'.
4. For all $v \in R$, add an edge (v, t) to E'.

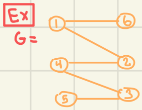


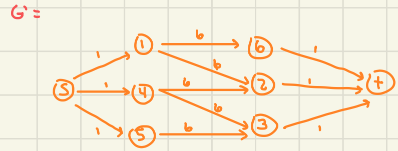
How do we add the capacities?

5. Set all of the new edges (e.g. those leaving s, and those entering t) to have $c(e) = 1$

6. Set all of the old edges (e.g. those going from $L \to R$) to have capacity $c(e) = n$ = # of vertices in o.g. graph.

Ex





What do we do now?

→ Run Ford-Fulkerson to find the maximum flow!

→ Then, return all the edges in G where $f(e) > 0$ (aka edges that f "sends flow" on).

What is the Algorithm?

```
Bipartite-Matching (G):
    G' = (V' = V, E' = E)
    direct every edge in E' from L to R
    for all e ∈ E':
        c(e) = n (aka |V(G)|)
    add s,t to V'
    for all u ∈ L:
        add (s,u) to E' with capacity 1
    for all v ∈ R:
        add (v,t) to E' with capacity 1
    f = Ford-Fulkerson (G', s, t)
    return M = { all e ∈ E' where f(e) = 1 }
```

What is the RT?

→ Constructing G' and M takes $O(m+n)$-time

→ Running F_F on (G', s, t) takes $O(mn)$-time

→ Total RT = $O(m+n) + O(mn) = O(mn)$

– Bipartite Vertex Cover –

RECALL: What is the capacity of a cut?

→ The sum of the capacities of all edges leaving the cut.

- $c(S) = \sum_{e \in \delta^{out}(S)} c(e)$

RECALL: What is a minimum s-t cut?

→ An s-t cut S s.t. the capacity $c(S)$ is minimized.

- A subset S that includes {s} and excludes {t}
- The max flow $|f| \leq c(S)$ for an s-t cut.

What is the max-flow-min-cut theorem?

→ The max. flow value over all feasible s-t flows for a graph G, $|f|$, is always going to be equal to the capacity $c(S)$ of the s-t cut S with the minimum capacity.

- $\max |f| = \min_{s\text{-}t \text{ cuts } S} c(S)$

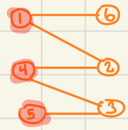
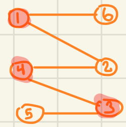
How can we find the minimum s-t cut of a flow?

→ Using Ford-Fulkerson!

- Run F-F to find the maximum flow of a flow network
- The set of vertices reachable from s in the last **Residual Graph** G_f is actually our min. s-t cut!!
- Simply run BFS(last Residual Graph, s) to return the set of all nodes reachable from $\underline{s}$.

What is a vertex cover?

→ (VC) given a graph G, a vertex cover is a subset S of vertices s.t. every edge in G has at least one endpoint in S.

→ Ex:  and  are both VCs of this graph. Every edge touches a node in S.

$S = \{1, 4, 5\}$ $S = \{1, 4, 3\}$

What is the Bipartite Vertex Cover problem?

→ Input: a bipartite graph $G = (L \cup R, E)$

→ Goal: Return a **minimum** vertex cover of G (aka least amt. of vertices possible)

How do we solve this problem?

→ Using Ford Fulkerson! We want to use the minimum s-t cut findable using F-F, and construct our max V-C from it.

1) Convert G into (G', s, t) following the same process as that in 6.2
2) Run Ford-Fulkerson on G', s, t
3) Let G_f = the last Residual Graph given by F-F. Run BFS on (G_f, s) to obtain all nodes reachable by s in G_f. This becomes our S = the minimum s-t cut in G.
4) Convert S into our final answer, the min VC in G.

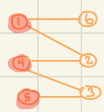
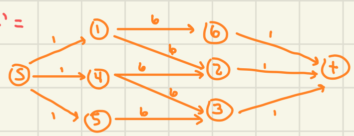
How do we use the min. s-t cut to obtain the min. VC?

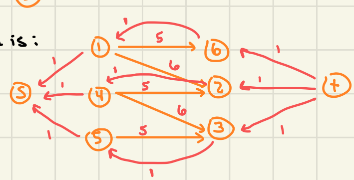
→ Given the b.p. graph G, with sets of nodes L and R, AND given our min. s-t cut S, the set of nodes representing the min VC is:

- All of the nodes in L which are NOT IN S (aka $L - S$ aka $L \setminus S$, formally), AND
- All of the nodes in R which are ALSO IN S (aka $R \cap S$)

→ AKA: Ans $= (L \setminus S) \cup (R \cap S)$

Example?

→ $G =$  and $G' =$ 

→ After running F-F, our last Residual Graph is: 

→ Min s-t cut = BFS(G_f, s) = $S = \{s\}$

→ $(L \setminus S) = \{1, 4, 5\}$. $(R \cap S) = \{\}$

→ ANS: VC cover = $\{1, 4, 5\}$

What is the RT of Min vertex cover?

→ Alg is almost Identical to 6.2, so RT = $O(mn)$

SUMMARY: How did we solve problems 6.2 and 6.3?

→ To solve **bipartite matching**, instead of solving it directly, we converted it to a diff problem - finding max flow - and translated the answer to get our solution.

- AKA, we **reduced** max. bipartite matching to max. flow

→ To solve **min. vertex cover**, we reduced it to minimum s-t cut.

Ch 7: NP Hardness

What have the previous chapters been about?

→ Strategies & patterns for solving problems in **polynomial time**.

What is ch.7 about?

→ Showing that a problem **cannot be solved** in polynomial time

What does it mean for a problem to be NP-hard?

→ We don't know if it can be solved in polynomial-time

- Maybe it can... but no one has found an algorithm yet. And we believe that it probably can't.

How do we prove that a problem is NP-hard?

→ With a comparison. For ex, let A and B be 2 "decision problems". When we say $A \le B$, it means "A is at most as hard as B", and "B is at least as hard as A".

- aka, comparing the relative difficulties of various problems.

Why is this comparison meaningful?

→ B/c if one day someone discovers that, for ex, problem A is actually hard (not just NP-hard), then it proves that B is also hard.

What is a decision problem?

→ A problem that is comprised of an input specification, and a yes/no problem Q.

- As opposed to **optimization problems**, like LIS, Knapsack, MST, etc. etc., where the output is a specific solution.

How do we translate optimization problems into decision ones?

→ By introducing an additional input K, and asking if the optimal value is **at least** K — for minimization problems — or **at most** K — for maximization problems.

→ To solve the "decision version", you just solve the optimization version & compare the output to K.

Examples?

	Optimization Version →	Decision Version
LIS:	• Input = list A • Output = length of LIS in A	• Input : (A, K) • Q = Is the length of the LIS in $A \ge K$?
MST:	• Input : graph G • Output : weight of the MST	• Input: (G, K) • Q = Is the weight of the MST in $G \le K$?
0/1 Knap sack:	• Input : (v, w, B) • Output: max value to fit in backpack	• Input: (v, w, B, K) • Q = is the value of the optimal solution $\ge K$?
Maximum Flow:	• Input: (G, s, t) • Output: value of max flow	• Input: (G, s, t, K) • Q = is the value of the max s-t flow $\ge K$?

7.1: Reductions, P, and NP

What is the class P, informally?

→ The set of all **decision problems** that can be solved in polynomial time

$P = \{ A \mid A \text{ is a decison problem that can be solved in poly-time} \}$

Example problems in P?

→ LIS, MST, bipartite matching, vertex cover, etc.

→ For Ex: given an array A and an integer k, we can determine whether A has an increasing subsequence whose length is $\geq k$ in polynomial time.

What is the class NP, informally?

→ The set of all decision problems that can be solved using **brute force**.

- aka pretty much every problem we look at, including every one we will look at in this class.

→ NP = nondeterministic poly time

What is known & not known about the classes P & NP?

→ KNOWN: All problems in P are in NP... $P \subseteq NP$

→ UNKNOWN: is $P = NP$? Aka, are all the NP-hard problems actually easy, but we haven't solved them yet?

- We believe $P \neq NP$, but we don't know.

How do we prove that a problem is not solvable in poly-time?

→ A reduction:

1. Imagine problems A and B. We **KNOW** already that $A \notin P$ (not poly-time solvable). We don't know about B
2. Assume for contradiction that $B \in P$ & has a polytime algorithm, ALG_B.
3. We "reduce" every input into problem A into an input to problem B s.t.
   - Any/every time the input would be accepted by A, the transformed input is also accepted by B.
   - Any time the input would be rejected by A, the transformed input is also rejected by B.

→ To specify the nature of these "input transformations", we have to write an algorithm that takes ANY input of A and converts it to some input to B s.t. the rules above (in step 3) are satisfied.

4. Now that we have this alg, instead of using our normal, brute force, NP hard alg for A, we solve inputs to problem A by first "transforming" the input, then running it through ALG_B, then outputting the answer given by ALG_B.

What is a polynomial time reduction?

→ An algorithm that does the "input transformation" described above, but specifically in **polynomial time**

→ Formally: an alg f that transforms every instance X of A into an instance $f(X)$ of B s.t. X is a "yes" instance of A iff. $f(X)$ is a "yes" instance of B.

→ $A \leq B$ ≈ A is polytime reducible to B.

How does a poly-time reduction prove that a problem is hard?

→ Take the ex from prev. page, where we know that A is hard & want to prove that B is hard.

→ By assuming that B has a poly-time alg, we were able to create a polytime reduction that maps all A-inputs to B-inputs and prove that $A \le B$. Using this poly-time alg, we were then able to actually solve A in polynomial time:

```
ALG_A(x):
    y = transform_input(x) ←—— a poly-time transformation
    return ALG_B(y)
```

→ However, we already know that A can't be solved in poly-time, so the above can't actually be possible. Therefore, B must also be hard. Otherwise it would mean that A is easy, which it isn't.

RECAP: What does it mean if $A \le B$ is true?

→ B is at least as hard as A

→ If A is hard, then it implies that B is hard.

→ If B is easy, then it implies that A is easy.

RECAP: How do we prove that $A \le B$?

→ Describing a polynomial-time algorithm $f: A \to B$ that satisfies the following:

1. Forward direction: If x is "yes" inst. of A, $f(x)$ is a "yes" inst. of B.
2. Backward direction: If $f(x)$ is a "yes" inst. of B, x is a "yes" inst. of A.

What does it mean if a problem B is NP-hard?

→ For all problems $x \in NP$, $x \le B$ (aka a polytime alg for B would allow us to solve every problem $x \in NP$ in poly time)

→ To show that B is NP-hard, we just have to choose some problem already "known" to be NP-hard, and reduce it to B, aka $A \le B$ for some $A \in NP$.

What does it mean if a problem B is NP-complete?

→ $B \in NP$ AND B is NP-hard.

Summary: What are we doing in this chapter?

→ We want to show that a problem B is hard. But we can't do that, so instead, we say that B is "at least as hard" as some other problem A. And we prove this by proving that $A \le B$, by giving a poly time reduction $f: A \to B$.

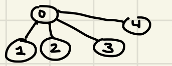
7.2: Independent Set to Vertex Cover

What is an Independent Set?

→ For an undirected graph G, an "independent set" is a subset S of nodes s.t. none of the edges in G connect 2 of the nodes in the subset.

- i.e., none of the nodes in S are directly connected to each other.

→ For ex, if G =



, then $S = \{1, 2, 3, 4\}$ is an ind. set b/c for every edge of G $\{u, v\}$, $(u \notin S \lor v \notin S)$ is true.

i.e., there are no edges connecting $v = 1, 2, 3,$ or 4 to one another.

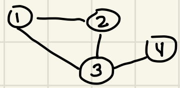
What is the Independent Set problem?

→ Input is (G, k) where the input is an undir. graph G and k = an integer

→ Problem Q: Does G contain an independent set of size at least k?

RECALL: What is a Vertex Cover?

→ For an undirected graph, a "vertex cover" of that graph is a subset of nodes/vertices where every edge in G touches one of those nodes. For example, if G =



, then $S = \{1, 3, 4\}$ is a vertex cover b/c every edge touches either $v = 1$, $v = 3$, or $v = 4$ (or both)

but $S = \{1, 2\}$ is not.

What is the vertex cover problem?

→ Given (G, k): Does G have a VC of size at most k nodes?

What is the goal?

→ Prove that VC is NP-hard by showing that $IS \leq VC$

- aka, write a poly-time alg f that converts every input to I.S. into an input to VC s.t. $x \in$ lang. IS i.f.f. $f(x) \in$ lang. VC

What is our poly-time reduction alg f?

→ $\boxed{f(G, k): \text{Given } (G, k), \text{return } (G, n-k).}$

```
Independent_Set(G, k):
    y = f(G, k)
    return Vertex_Cover(y)
```

How do we prove the correctness of an alg f?

→ To prove this, we must prove the forward & backward directions. For ex, to prove that f above is correct:

1) Forward: if $x = (G, k)$ is a yes for I.S., prove that $f(x) = (G, n-k)$ is a yes for V.C.

Why does this work?

2) Backward: if $x = (G, k)$ is a no for I.S., prove that $f(x) = (G, n-k)$ is a no for V.C.

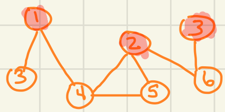
- ALTERNATIVELY, show: if $f(x) = (G, n-k)$ is a yes for VC, prove that $x = (G, k)$ is a yes for IS

↳ this one is usually easier to prove.

What is the forward-direction proof for I.S. $\leq$ V.C.?

→ If (G,k) is a "yes" inst. of IS, then G has an ind. set S where $|S| \geq k$.

→ Ex: let $G =$



and $k = 3$. Then (G,k) is a "yes" if we let $S = \{1,2,3\}$

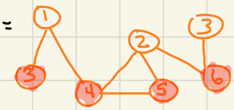
→ If S is an I.S., that means that none of the edges that touch a node in S touch another node in S. So, every edge touching a node in S is also touching at least one node in G that isn't in S.

→ None of the edges "in" S connect the nodes to one another. Therefore, if we let $X = \{v \mid v \text{ is a node in } G \text{ and } v \notin S\}$ – aka every node not in S, aka $V(G) - S$ — then X has to be a vertex cover of G because the edges touching nodes in X will cover, by defn, every node in X. But they will also cover every node in S b/c the nodes in S must be connected to the graph somehow

→ SUMMARY: For (G,k), if accepted by IS, then we can find a vertex cover of at most $V(G) - k$ aka $n-k$ nodes. Thus $(G, n-k)$ is accepted by VC.

What is the backward-direction proof for I.S. $\leq$ V.C.?

→ Ex: let $G =$



and $n-k = 4$. Then $(G, n-k)$ is a "yes" of VC if we let $S = \{3,4,5,6\}$

→ If G is a "yes" of VC, that means $\exists$ a VC S of G where $|S| \leq n-k$

→ The complement set $S' = V/S$ (aka every node not in S) is an IS of G and $|S'| \geq k$. Therefore, (G,k) is a yes for IS.

SUMMARY: What does this proof imply?

→ Independent Set $\leq$ Vertex Cover

- If I.S. is hard, which we think it is, then VC is also hard.

What are some properties of poly-time reductions?

1. Reflexive: for all problems A, $A \leq A$
   - Why? Because $f(x) = x$ is a correct reduction from A to A. e.g. the alg just outputs the ans returned by A.
2. Transitive: for all A, B, C, if $A \leq B$ and $B \leq C$, then $A \leq C$
   - Why? To reduce A to C, we'd simply call the function f to transform A's input to a B-input. Then, transform that B-input to a C-input. Then, run the input through ALG_C & return the ans.

What is NOT a property of poly-time reductions?

→ Symmetric: for all A, B, if $A \leq B$, it does NOT necessarily mean that $B \leq A$.

- For ex, $VC \leq Halt_{TM}$ but $Halt_{TM}$ is NOT reducible to VC.

7.3: 3-SAT to Independent Set

RECALL COMP 455: What is the 3SAT problem input?

→ A set of n boolean variables $\{x_1, x_2, \ldots x_n\}$, and

→ A set of ℓ clauses, where each clause is a boolean expression consisting of exactly $\underline{3}$ literals OR'ed together.

- A "literal" = a boolean var OR its negation, e.g. $\bar{x}_1, x_1, x_2, \bar{x}_2$, etc.

What is the problem Q?

→ Let L = a boolean expression where each clause in the input is AND'ed together.

- Ex: variables $= \{x_1, x_2, x_3\}$ and

  clauses $= \{(x_1 \vee x_2 \vee \bar{x}_3), (\bar{x}_1 \vee \bar{x}_2 \vee x_3), (\bar{x}_1 \vee \bar{x}_2 \vee x_3)\}$

  Then $L = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$

→ Q: Is there an assignment $\varphi : \{x_1, \ldots x_n\} \to \{T, F\}$ (aka an assignment of each variable to either T or F) s.t. L evaluates to True?

$$3SAT = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable Boolean formula} \}.$$

- Ex: Yes, because $x_1 = T \quad x_2 = F \quad x_3 = T$ lets L = True

→ Basically, an assignment to each variable s.t. for each $\underline{\text{clause}}$, at least one literal in the clause = T

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$
$$(T \vee F \vee F) \wedge (F \vee T \vee T) \wedge (F \vee T \vee T)$$
$$T \wedge T \wedge T = T$$

Why is the 3SAT problem significant?

→ Because the **Cook-Levin Theorem** proves & states that, by definition (e.g. NOT using a reduction), that 3SAT is NP-hard.

- It is a KNOWN NP-hard problem.

→ Therefore, any problem that we can reduce 3SAT to, we can then assert that it is NP-hard.

- e.g., we will prove $3SAT \le IS$, which proves that I.S. is NP-hard.
- We already proved that $IS \le VC$, so this will also implicitly prove that VC is NP-hard.

What is the goal?

→ Prove that IS is NP-hard by showing that $3SAT \le IS$.

→ RECALL: I.S. $= \{ (G, k) \mid G \text{ has an Ind. set of size} \ge k \}$

What is our poly-time reduction f?

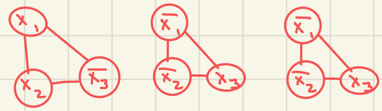
→ Ex: let $L = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$

→ f(3SAT input):

1) For each clause C_j, add a "triangle" to G by creating one vertex per literal in C_j and connecting the 3 vertices together.

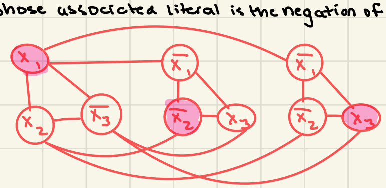
- Thus, if there are ℓ clauses, G will initially have 3ℓ nodes and 3ℓ edges.

$G =$



2) Add "conflict edges": For every node v in G, add an edge between v and all nodes whose associated literal is the negation of v.

$G =$



3) Let $k = \ell =$ the number of "triangles" aka the number of clauses.

4) Return (G, k)

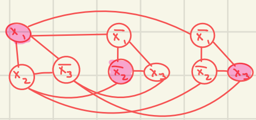
What is the forward direction proof?

→ If input is a "yes" for 3SAT, then $\exists$ an assignment φ that satisfies all clauses.

- Ex: in ex above, $\varphi = \{x_1 = T, x_2 = F, x_3 = T\}$ satisfies all clauses

→ For each triangle in G, say that we pick any literal in the triangle that evaluates to true, and add it to our independent set.

- Ex:



IS $S = \{x_1, \overline{x_2}, x_3\}$

→ This selection S obviously has size $\geq k$.

→ S is an independent set because:

- We only choose one node from each triangle, so obviously the edges connecting each triangle won't interfere with our independent set.
- The conflict edges are also not an issue BECAUSE we assigned each literal to T or F. And for each triangle, we added a literal that evaluated to $\underline{T}$ to our IS S. If a literal b eval. to true, then it may have "conflict edges" to all nodes "$\overline{b}$". But since "$\overline{b}$" would then eval. to false, we would only ever have one of b or $\overline{b}$ in our I.S., b/c they can't both be true.

What is the backward direction proof?

→ If we have an IS of size $K = \ell$ where the nodes are literals, and this input $(G, k = \ell)$ was a "yes" inst. of Independent Set:

G =

We can use this graph & the IS S (in this ex, $S = \{x_1, \overline{x}_2, x_3\}$) to construct a "yes" instance of 3SAT:

- For each literal in S, set it to be TRUE in the boolean assignment φ:

φ: $x_1 = T$

$\overline{x}_2 = T$ so $x_2 = F$

$x_3 = T$ or F, it doesn't matter

→ We can prove that this is a satisfying assignment for 2 reasons:

1) The assignments will not contradict each other, b/c given the fact of the "conflict edges", if S is an IS, it will not contain both $\underline{b}$ and $\underline{\overline{b}}$ for any literal b.

- aka "φ is well defined"

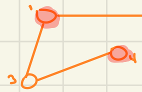
2) The assignment satisfies every clause b/c each triangle represents a clause, and each triangle adds at least 1 node to S.

7.4: Vertex Cover to Dominating Set

What is a dominating set?

→ A subset S of vertices s.t. for all nodes $u \in V$, u is either in S, or u has a neighbor in S

- Basically "covering every vertex", unlike VC, which tries to "cover every node".

→ Ex: 

$S = \{1, 4\}$ is a dominating set.

What is the Dominating Set problem?

→ Input: (G, k)

→ Problem: Does G have a dominating set of size at most k?

- e.g., with $\leq k$ nodes, can we "cover every vertex"?

RECALL: What is the Vertex Cover problem?

→ For (G, k), Does G have a VC of size at most k nodes?

→ Vertex Cover: Subset S of vertices s.t. every edge has at least 1 endpoint in S.

What is the goal?

→ Prove that DS is hard by showing that Vertex Cover $\leq$ Dominating Set

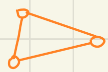
- Given a "yes" instance of VC, write a poly-time function to output a "yes" instance of DS.

Why can't our function $f(G,k)$ just return (G,k)?

→ Returning the VC input almost satisfies the forward direction: If G has no isolated vertices (e.g. every node has ≥ 1 edge), then a Graph w/ a VC of size k will have a DS of size k.

- But if G has isolated vertices, like  , then it could be a yes for VC but a no for DS.

→ This reduction also doesn't satisfy the backward direction:

$G =$  where $k = 1$ would be a NO for VC, but a yes for DS.

How do we solve this problem?

→ Intuition for reductions: "$A \leq B$" ≈ solve A given a 'solver' for B.

→ Given a "solver" for Dominating Set, we need to turn our graph G — which covers k edges — into a graph G' which covers k nodes.

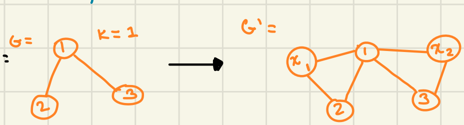
- Somehow convert every edge to a vertex?

What will be our polytime reduction f?

1. Given G, construct a new graph $G' = G$, initially. Then, for each edge $e \in E(G)$:
   - add a new vertex x_e to G'
   - add the edges (u, x_e) and (v, x_e)

→ Intuitively, we are placing a new vertex "next to" each edge e.

2. Set $k' = k + |I(G)|$, where $I(G)$ is the set of isolated vertices in G.
3. Return (G', k')

→ Ex: 

What is the forward direction proof?

→ Given a "yes" instance of VC, let S be a vertex cover of G s.t. $|S| \leq k$.

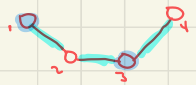
→ We claim that $S' = S \cup I(G)$ will always be a dominating set of G'.

- aka, for any vertex $u \in V(G')$, $u \in S'$ or u has a neighbor in S'.

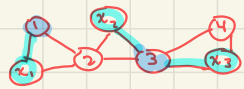
→ Why can we claim this?

1. If $u \in I(G)$ (the isolated vertices in G) or $u \in S$, then obviously $u \in S'$.
2. If $u \in V(G)$ and $u \notin S$ – aka, an "old"/og vertex from graph G that wasn't in the V.C. S, then u has at least one neighbor in S', because we know that S is a VC of G, meaning all edges attached to u must be "covered" – meaning that at least one neighbor of u is in S and therefore in S'
3. If $u \in V(G')$ and $u \notin V(G)$ – aka the new vertices added in the reduction:

- Given a "yes" inst of VC, we know $\exists$ set S which covers all edges (aka one endpoint of every edge is in S):



- All the new edges are added "along" existing edges. Since we know that those edges are covered by nodes in S, the same nodes in S will end up touching every new vertex x_e added:



What is the backwards direction proof?

→ Let $S' = (G', k + |I(G)|)$ be a "yes" instance of DS, where S' is a DS of G' where size $\leq k$ nodes.

- We can't just claim "S' is a VC of G" like we did in the fwd. proof, b/c G' has vertices that G doesn't and those could be in the DS S'.

→ We will convert S' to a subset S of nodes in G, which is also a VC of size k. This will prove that "yes" instances of DS(F(VC_Input)) (running DS with the transformed inputs given by the reduction) correspond to "yes" instances of VC(VC_Input).

How will we convert S' to S?

1. Start with $S = \emptyset$ (empty)
2. For all u in S':
   - if $u \in V(G)$, add u to S
   - otherwise (if u is a new vertex added by the reduction): add either neighbor of u to S.
   - If u is in $I(G)$, do not add it to S.

→ SKIPPED: Rest of backward proof showing why "S' is DS of G'" ⇒ "S is a VC of G"

What is true about the DomSet and VC problems?

→ DomSet $\in$ NP (not solvable in poly time, but is brute forcable)

→ VC $\in$ NP-complete (aka VC $\in$ NP AND VC $\in$ NP-hard)

- NP-Hard = every problem in NP reduces to VC.

7.5: Directed to Undirected Hamiltonian Cycle

RECAP: What are we doing in chapter 7?

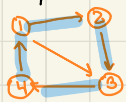
→ We want to show that a problem B cannot be solved by a polynomial-time alg... aka that B ∈ NP-Hard

→ We can't actually prove that, but if ∃ a problem A that IS believed to be NP-Hard, and we can show that $A \le B$ (A is poly-time solvable given a "solver" for B), then we can prove that B is at least as hard as A.

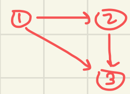
What is the DirHamCycle problem?

→ Input: Directed graph G. Does G contain a Hamiltonian cycle (a cycle that visits each node exactly once)?

Ex "yes" instance G:



Ex "no" instance G':



What is the HamCycle problem?

→ Input: Undirected graph G. Does G contain a Hamiltonian cycle?

What is the goal?

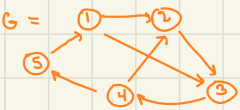
→ Prove that HamCycle is at least as hard as DirHamCycle by proving

DirHamCycle ≤ HamCycle

How do we construct our reduction F?

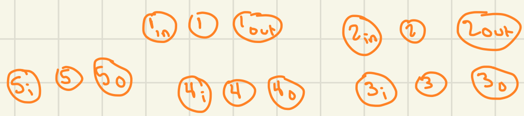
→ Given a Directed graph G, if G has a HC, return an undir graph G' that has a Ham Cycle.

→ Ex: G =

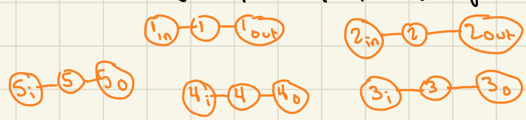


→ REDUCTION: Construct G' as follows:

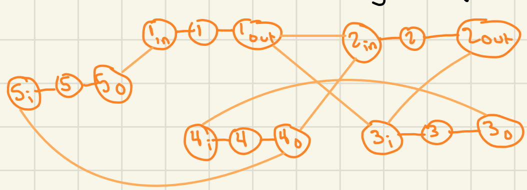
1. For each node $u \in V(G)$: add 3 nodes $\{u_{in}, u, u_{out}\}$ to G'.



2. For each node $u \in V(G)$: add 2 edges (u_{in}, u) and (u, u_{out}) to G'. (aka, every vertex gets replaced by a path of length 2)



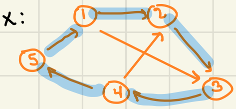
3. For each edge $(u,v) \in E(G)$ (in the og graph): Add the edge (u_{out}, v_{in}) to G'. "u_{out}" represents the node sending out-neighbors from "u".



What is the forward direction proof?	→ If G has a dir. Ham cycle C, we can construct a HC C' in G' by simply following C, but instead of jumping from node u to node v and so on, we "enter" a node at u_{in}, then u, then u_{out}, THEN v_{in} for the next vertex in C, and so on.

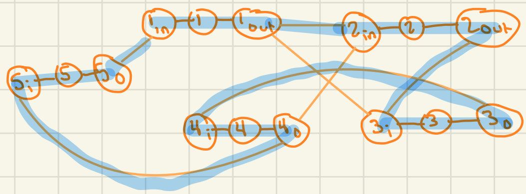
- C' goes $u \to u_{out} \to v_{in} \to v$ every time C goes $u \to v$

→ Ex:

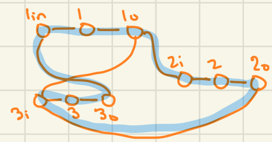


C could $= 1 \to 2 \to 3 \to 4 \to 5 \to 1$, so

$C' = 1_{in} \to 1 \to 1_{out} \to 2_{in} \to 2 \to 2_{out} \to 3_{in} \to 3 \to 3_{out} \to 4_{in} \to 4 \to 4_{out} \to 5_{in} \to 5 \to 5_{out} \to 1_{in}$



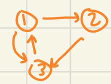
What is the backward direction proof?	→ Suppose G' has a Ham Cycle C'. Like



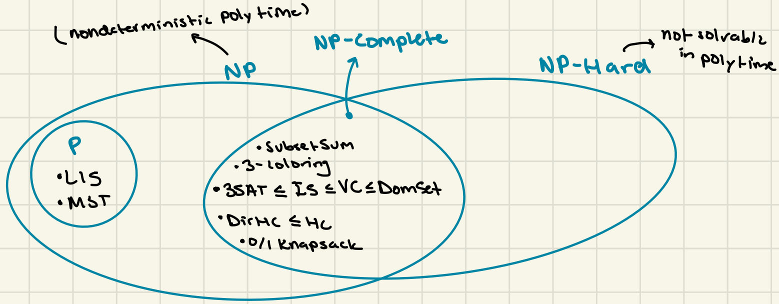
→ Consider any vertex $u \in$ o.g. graph G. In C', there must exist a subpath (u_{in}, u, u_{out}). After visiting u_{out}, C' must visit some vertex v_{in}. It MUST go to v, v_{out} after that, consecutively.

→ Therefore, we can construct a HC C in G by taking C and removing every vertex not in G (aka the "in" and "out" vertexes)

- Ex above: $C' = 1_{in} \to 1 \to 1_{out} \to 2_{in} \to 2 \to 2_{out} \to 3_{in} \to 3 \to 3_{out} \to 1_{in}$
- so $C = 1 \to 2 \to 3 \to 1$, which is an HC in G:



RECAP of ch.7?



→ 3-SAT has been proven NP-Complete/Hard w/o a reduction (Cook-Levin Thm.)

Approximation Algorithms

Whats the deal w/ unsolvable problems?	→ For many problems (like 3SAT, VC, etc.), there probably is <u>no</u> poly-time alg to solve them • We don't know, b/c we haven't found one yet. But maybe. • These problems are in NP or NP-hard.
But what if we still <u>need</u> (101) to solve them?	→ We have to give up one of the two: • polynomial time • optimality (accuracy of ans)
What are approximation algorithms?	→ Algorithms to solve **hard** problems within polynomial time, by sacrificing optimality (ability to ALWAYS yield the exact correct answer), & instead delivering an **approximation**. → Approximation algorithms: • Don't always return an optimal solution – but often quite close • Always run in poly-time • Are relatively simple
What is a α-approximation algorithm?	→ For a given problem, such as a minimization problem, let OPT denote the true, actual optimal solution. Let ALG denote the solution given by the Approx. Alg. → DEFN: a α-approx-alg for a given problem always returns a solution whose value ALG is within a factor α of OPT.
What does this mean for minimization vs maximization problems?	→ **Minimization:** There exists a $\alpha \geq 1$ s.t., on every instance of the algorithm, $$OPT \leq ALG \leq \alpha \cdot OPT$$ → E.g., a 2-approx alg returns a value ALG that is at most $2 \cdot OPT$ away from the right answer → **Maximization:** There exists a $\alpha \leq 1$ s.t., on every instance of the algorithm, $$OPT \geq ALG \geq \alpha \cdot OPT$$
What is the approximation ratio of an algorithm?	→ The smallest val of α s.t. the algorithm is a proven α-approximation algorithm. → To prove an α-approx alg's correctness, show that the alg satisfies one of the 2 inequality statements above (depending on the type of the problem) – on every instance. → e.g., a 2-approx alg is also a 3-approx alg, but the ratio is 2 b/c that's the smallest possible.

– 8.1: Vertex Cover –

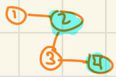
RECALL: What is the VC problem? (not decision type)

→ A **minimization problem**: for a graph G, return the smallest possible VC : aka a subset S of V s.t. for all $e \in E$, e has at least one endpoint in S

What is our goal?

→ Describe a 2-approx alg for VC : The alg returns a subset of nodes ALG s.t.,

For any graph G, $|OPT(G)| \leq |ALG(G)| \leq 2 \cdot |OPT(G)|$

- Aka, $|ALG|$ is at most $2\times$ the size of OPT, $|OPT|$.
  - For ex, if $G =$  , $|OPT| = 2$ and $2 \leq |ALG| \leq 4$

What will our alg be?

→ A Greedy algorithm: For each edge $e \in G$, add both endpoints of e to S, and remove the endpoint vertices $\{u,v\}$ from G. Repeat this process until G has no edges.

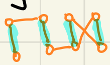
- NOTE that when we remove a vertex u from a graph, we also remove all edges touching u.

```
VC-Matching(G,k):
    S = empty set
    while G has an edge e = {u,v}:
        add u to S
        add v to S
        remove u and v from G
    return S
```

How do we prove its correctness?

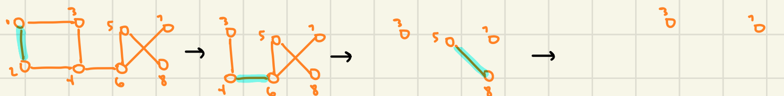
→ Prove the **Theorem**: For all possible G, $|ALG(G)| \leq 2 \cdot |OPT(G)|$

→ Proof: Consider the set of edges "chosen" by ALG (aka the edges that we actually "look at" before removing its endpoints, as opposed to edges that get automatically deleted when we remove a node u from G).

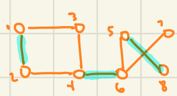
- Those edges form a **matching** M: a subset of edges s.t. no 2 edges in M share an endpoint. For ex: 

How do we know that the set of edges considered is a Matching?

→ When we consider an edge, we remove its endpoints — meaning that we automatically remove every edge that would share an endpoint with it!



$S = \{\ \}$ → $S = \{1, 2\}$ → $S = \{1, 2, 4, 6\}$ → $S = \{1, 2, 4, 6, 5, 8\}$

- Edges considered : 

What do we know about the OPT subset VC for a graph G?

→ $|OPT| \geq |M|$: OPT has at least $|M|$ nodes, because OPT must be a subset that covers every edge, and the edges in M share NO nodes; there's no overlap. At least one endpoint of each $e \in M$ must be in OPT.

- Where M = the maximum matching of G.

What do we know about the ALG subset returned by our alg?

→ In the alg, we add **2 nodes** for each edge considered. We have already shown that alg considers at most $|M|$ edges. Therefore, we add at most $2 \times |M|$ nodes to our subset ALG in our algorithm.

How does this come together to prove that our approx. alg is correct?

→ $|ALG| \leq (2 \cdot |M|) \leq (2 \cdot |OPT|)$

or "=" according to the lecture btw

because $|M| \leq |OPT|$!

- 8.2: Load Balancing -

What is a scheduling problem?

→ We have n jobs and m machines, and we need to assign each job to a machine.

→ Every job has a corresponding length $\ell[n]$

→ For a given jobs-to-machines assignment, for each machine m, the **load** on m = the sum of the lengths of every job assigned to m

→ For a given jobs-to-machines assignment, the **makespan** of the assignment is the maximum load created by that assignment - aka, the load of the machine w/ the heaviest load.

What is the Load Balancing problem?

→ INPUT: (ℓ, m), where ℓ is an array of size n; $\ell[i]$ for $i = 1, ... n$ is the length of job i.

And m = the # of machines.

→ GOAL: Return an assignment of jobs-to-machines s.t. the **makespan** is minimized (minimize the max load)

→ Ex: if $\ell = [3, 1, 2]$ and $m = 2$, the OPT solution is to assign job 1 to one machine, and jobs 2 & 3 to the second machine:

→ The Decision-version of this problem (e.g., given ℓ, m, and a makespan k, does there exist an assignment s.t. the makespan $\leq k$?) is **NP-Hard**.

What is our goal?

→ Describe a 2-approx Algorithm for L.B.: On every instance of (ℓ, m), the makespan M_{ALG} of our assignment ALG returned by the algorithm should be **at most** $2 \cdot M_{OPT}$

What is our algorithm?

→ Greedy: for each job j, pick the machine M_i w/ the current smallest load and assign j to M_i.

algorithm to return val. of makespan

```
Load-Balancing (l, m):

    min_load = ∞

    index = 0

    machines = [0] * m

    for job in range 1, ... l:

        for i in range 1, ... m:

            if machines[i] < min_load:

                min_load = machines[i]

                index = i

            machines[i] += l[job]

    return makespan = min(machines)
```

What is an online algorithm?

→ An alg that can make its final decisions & determine an answer w/o knowing what the future holds. This alg is an example.

What theorem do we have to prove for correctness?

→ The val. of the makespan of ALG returned by our alg - which will call T, is $\leq 2 \cdot OPT$

→ Proof:

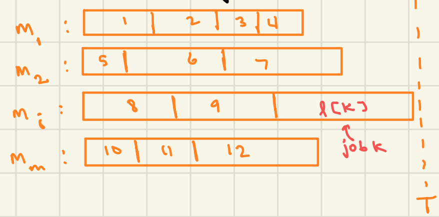
① OPT - the true minimal makespan - is at least the sum of all of the loads of all jobs, divided by the # of machines m:

$$OPT \geq \frac{\sum_{j=1}^{n} l[j]}{m}$$

- The absolute ideal assignment would be one where we can split the jobs evenly over all machines s.t. each machine has the same load. Obviously, we can't possibly have a smaller makespan than that.
- E.g., if $\sum_{j=i}^{n} l[i] = 20$ and we have 5 machines, the makespan is $20/5 = 4$, at minimum - we would only be able to achieve this if the loads of the indiv. jobs allow it.

$\boxed{\text{Ctd. next page}}$

② The makespan T can be thought of as a vertical line:



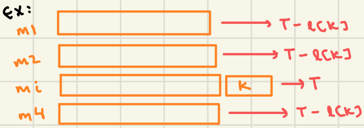
→ If T is ALG's makespan, let m_i = the machine w/ the max load. Let job k be the last job added to m_i. Aka, after job k was added, the value of T was determined.

→ At the time that the alg was deciding on where to place job k, the load of every machine had a load of at least $T - \ell[k]$

- Why? B/c alg. chose to add job k to machine m_i BECAUSE it had the smallest load at that point in time. m_i's load after adding job k became m_i–load + $\ell[k]$, which ended up being T.
- If any other machine had a load less than $T - \ell[k]$, alg. would have chosen to place job k on it.

→ Given this, the sum of all the loads must be at least $m \times (T - \ell[k])$.

Ex:



$$\sum_{j=1}^{n} \ell[j] \geq m \cdot (T - \ell[k])$$

③ $OPT \geq \ell[k]$ – where k is the last job that gets computed.

→ Now combine observations 1, 2, and 3:

$$m \cdot (T - \ell[k]) \leq \sum_{j=1}^{n} \ell[j]$$

$$T \leq \frac{\sum_{j=1}^{n} \ell[j]}{m} + \ell[k]$$

rearranging observation 2

→ Since ① gave that $\frac{\sum_{j=1}^{n} \ell[j]}{m} \leq OPT$ and ③ gave that $\ell[k] \leq OPT$,

we can say that:

$$T \leq \frac{\sum_{j=1}^{n} \ell[j]}{m} + \ell[k] = T \leq OPT + OPT = T \leq 2 \cdot OPT \text{ !!}$$