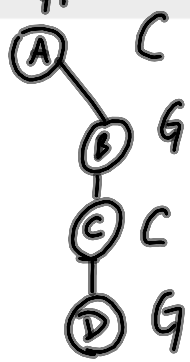
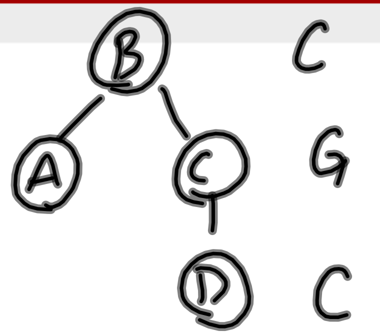
(10 minutes for "discussion" problems)

connected

every pair of vertices
u, v
u ⇝ v
and v ⇝ u

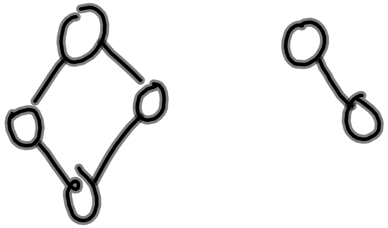
BFS from A:

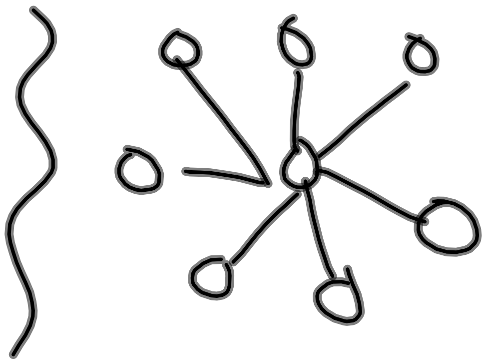


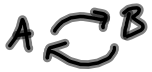


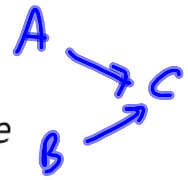
In a connected bipartite graph, is the bipartition unique? Justify your answer.

if not connected?







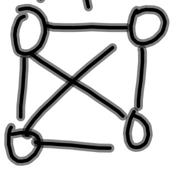


Suppose have a directed acyclic graph G and we want to find out if there is a simple path that visits every vertex. Give a *linear time* algorithm that determines if a given graph has such a path.

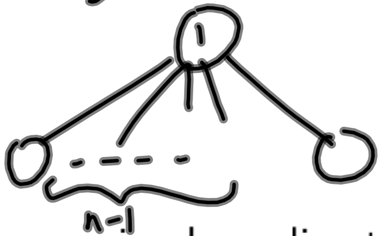
- $P \leftarrow$ Topological Sort/Order on G
- If P is a path, return P
- Else return false

//should explain why

ex K_4



1. $BFS(K_n)$:



2. $DFS(K_n)$
Path/Chain length n

The graph K_n is defined as a simple undirected graph with n vertices and nC_2 edges. Note that this means that every pair of vertices has one edge between them.

1. What does a breadth-first search tree look like for K_n?
2. What does a depth-first search tree look like for K_n?

Warm-Up

*: largest ≤

for each interval i
binary search for * s_i in finish times

Give an $O(n \log n)$ time algorithm that computes $p(i)$ for all intervals. You may assume that the intervals are already sorted by finish time.

Solve "The Big Problem" recursively

Goal: OPT(n)

Tautology: friend either takes class i or won't

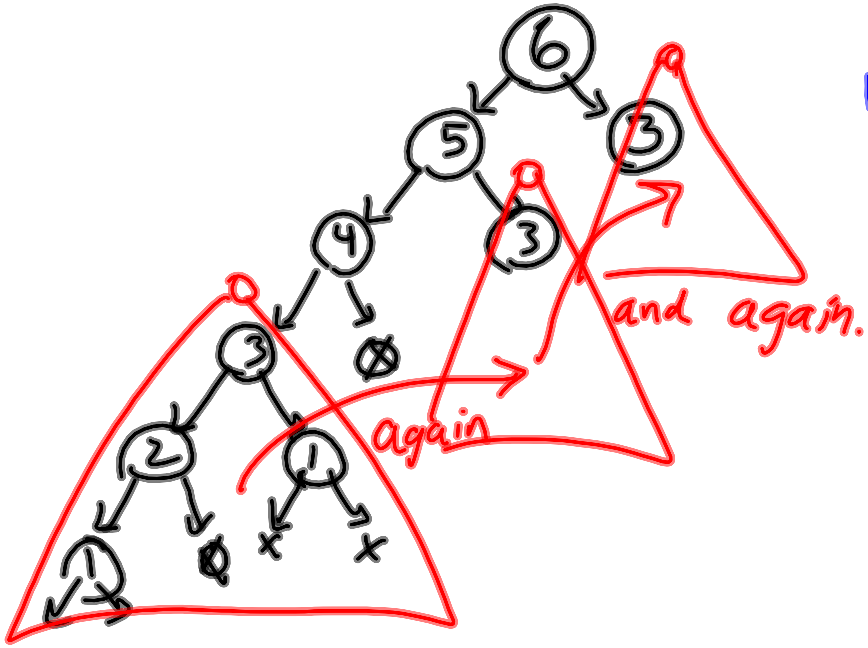
`OPT(i)` // optimal # credits obtainable among intervals $1 \ldots i$.

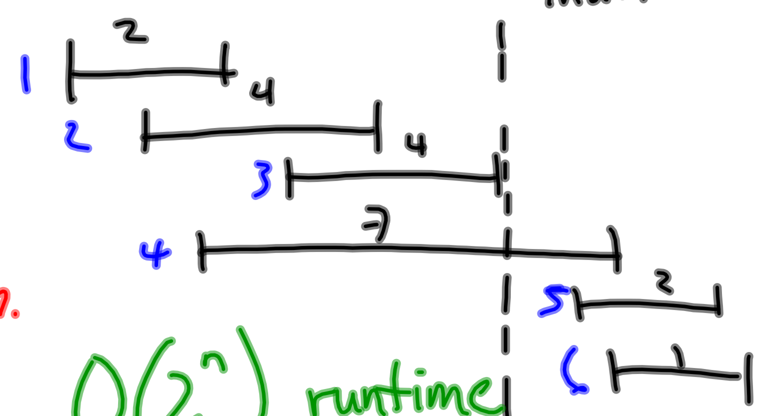
// Does your friend take class i or not?

```
{
    if i < 1:
        return 0
    value_if_not_taken = OPT(i-1)      // 1..i-1 since i not taken
    value_if_taken = OPT(p(i)) + Vi
    return max(value_if_not_taken,
               value_if_taken)
}
```

So, how does that recursion look?

Key: (i) = rec. call index i





$O(2^n)$ runtime if we're not careful

So let's be careful. ☺

declare OPT[1... n] (array of ints)
Set OPT[i] = -1 for all i (sentinel value)
Call modified OPT(n).

O(n) {
OPT(i)
    if i<1 return 0
    if OPT[i] ≠ -1, return OPT[i]
    else OPT[i] = max((see -2 slides))
        return OPT[i]

+ precompute of p(i)

O(n log n)

iteratively.

- compute $p(i)$ values $//O(n\log n)$
- declare $OPT[0..n]$
- $OPT[0] = 0$ // base case

$\Big\}\ O(n)$

for $i=1$ to n

   // fill in $OPT[i]$ as per recursion $\Big\}\ O(n)$

   $OPT[i] = \max(OPT[i-1],\ V_i + OPT[p(i)])$ $//O(1)$

$O(n) + O(n\log n) \longrightarrow O(n\log n)$

Filling in the table

i	$p(i)$	v_i	OPT(p(i)) + v_i	OPT($i-1$)	OPT(i)	Take?
0	N/A	N/A	N/A	N/A	0	
1	0	2	2	0	2	
2	0	4	4	2	4	
3	1	4	2+4=6	4	6	
4	0	7	0+7=7	6	7	
5	3	2	6+2=8	7	8	✓
6	3	1	6+1=7	8	8	X

Reconstruct the solution

```
i ← n
while i > 0
    // should ~~ take class i?
    if OPT[i] == OPT[i-1]
        // no
        i ← i-1
    else
        output interval i
        i ← p(i)
```

Something very important

Dynamic Programming is not about filling in tables.
Dynamic Programming is about smart recursion.

Free Food Exercise: Recursive Solution

OPT(i): optimal cost on days 1.. i

n days
O(1) each
(when properly
memoized)
O(n) total
time

if $i < 1$, return 0
if day i is free, return OPT(i-1)
else return $\min \begin{pmatrix} \$20 + \text{OPT}(i-7) \\ \$6 + \text{OPT}(i-1) \end{pmatrix}$

groceries
XOR
cafeteria

Free Food Exercise: Iterative Solution

```
declare OPT[-6 ... n]
OPT[-6 ... 0] = 0
for i = 1 to n
    // fill in OPT[i]
    if day i free
        OPT[i] = OPT[i-1]
    else
        OPT[i] = min(20 + OPT[i-7], 6 + OPT[i-1])
```

Free Food Exercise: Output

```
i ← n
while i > 0
    if day i free
        i = i-1
    else if OPT[i] == 20 + OPT[i-7]
        output groceries: day i-6
        i = i-7
    else
        i = i-1
```

Subset Sum

ex: $\{2, 3, 4\}$ $T=8$ NO

$\{\overset{\times}{2}, 3, 4\}$ $T=7$ yes

$\{\overset{\checkmark}{2}, 3, 4\}$ $T=5$

Recursive Solution Take One

$OPT(i) =$ Subset of $1..i$ solve?

if_i_not_used $= OPT(i-1)$

if_i_used X "is there a subset of $S[1..i-1]$ that adds to $T - S_i$?"

Recursive Solution Take Two

Goal: SubS[n,T]

SubS(i,j): "is there a subset of S[1..i] that adds to j?"

if j==0, True

if i==0, false

//non-base:

use_ith = SubS(i-1, j-s[i]) //or false if j<s[i]

dont_use_ith = SubS(i-1, j)

return use_ith OR dont_use_ith

if memoized and iterative, running time: $\Theta(nT)$

Iterative Solution

declare SubS[0..n, 0..T]
for i = 0 to n, SubS[i, 0] = true
for j = 1 to T, SubS[0, j] = false

for i = 1 to n
{
 for j = 1 to T
  if j < s[i], SubS[i, j] = SubS[i-1, j]
  else
   SubS[i, j] = SubS[i-1, j] OR SubS[i-1, j-s[i]]
*}

return SubS[n, T]
 //or, a few slides later, output the subset

Visualizing

target

Subset to use

	0	1	2	3	4	5	6
{} X	T	F	F	F	F	F	F
{2}	T	F	T	F	F	F	F
{2, 3}	T	F	T	T	F	T	F
{2, 3, 4}	T						

Something very important

Something very important

Dynamic Programming is not about filling in tables.
Dynamic Programming is about smart recursion.

Finding the right subset

// assume SubS[n, T] is true...

i ← n, j ← T

while i > 0:

if Subs[i-1, j] i = i-1

else

output s[i]

i = i-1

j = j - s[i]

Start the homework!