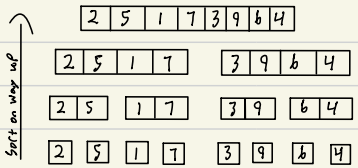
CSC 140 Notes

limitations of Asymptotic Notation

- We assume all ops take the same time
- We don't model memory and caching
- We ignore constants and lower order

Θ is tightly bound $\cong =$
O is upper bound $\cong \leq$
o is tight upper bound $\cong <$
Ω is lower bound $\cong \geq$
ω is tight lower bound $\cong >$

Merge sort (Divide & conquer)



*If runtime isn't a pow of 2 look for closest one to find Aprox runtime *

Solving with recursion tree method

$$T(n) = \begin{cases} d & n \leq n_0 \\ aT(\frac{n}{b}) + f(n) & n > n_0 \end{cases}$$

$f(n)$ = work done at each level
d = cost of base case
a = # of subproblems
b = division ratio

Geometric series

$$= \sum_{i=1}^{H-1} \text{cost at level}$$

$$= \left(\frac{1 - (\text{cost at level})^{H-1}}{1 - \text{cost at level}} \right)$$

ex) $T(n) = 7T(\frac{n}{2}) + n^3$

$H = \log_b n = \log_2 n$
$a = 7$
$b = 2$
$f(n) = n^3$
$L = a^H = 7^{\log_2 n} = n^{\log_2 7}$
base cost $= d \cdot L = d n^{\log_2 7}$
cost at level $= n^3 (\frac{1}{8})^i$
recursive cost $= \sum_{i=0}^{H-1}$ cost at level
$= n^3 \frac{(\frac{7}{8})^{\log_2 n} - 1}{\frac{7}{8} - 1}$
$= 8n^3 \left(1 - \frac{7^{\log_2 n}}{8^{\log_2 n}}\right)$
$= 8n^3 \left(1 - \frac{n^{\log_2 7}}{n^3}\right)$
$= 8n^3 - 8n^{\log_2 7}$
Total cost $= 8n^3 - 8n^{\log_2 7} + dn^{\log_2 7}$
$= \Theta(n^3)$

Examples

$T(n) = 5T(\frac{n}{4}) + n \log n$

$n^{\log_4 5} \cong n^{1.001} > n \log n$
$T(n) = \Theta(n^{\log_4 5})$

$T(n) = 2T(\frac{n}{2}) + n$

$n^{\log_2 2} = n = cn$
$T(n) = \Theta(n \log n)$

$T(n) = 2T(\frac{n}{3}) + n \log n$

$n^{\log_3 2} < n \log n$
check regularity
$2(\frac{n}{3}) \log \frac{n}{3} \leq kn \log n$
True for $k = \frac{2}{3} > 1$
$T(n) = \Theta(n \log n)$

$T(n) = 2T(\frac{n}{2}) + n \log n$

$n^{\log_2 2} = n < n \log n$
but not asymptotically
can't use Master theorem

Master's theorem

Compare $L = n^{\log_b a}$ with $f(n)$

1) if L is polynomially larger $f(n) = O(n^{\log_b a - \varepsilon})$
then $T(n) = \Theta(n^{\log_b a})$

2) if both are asymptotically the same $f(n) = \Theta(n^{\log_b a})$
then $T(n) = \Theta(f(n) \log n)$

3) if $f(n)$ is polynomially larger $f(n) = \Omega(n^{\log_b a + \varepsilon})$
then $T(n) = \Theta(f(n))$ if
$af(\frac{n}{b}) \leq kf(n)$ for $k < 1$
regularity condition

Mergesort(A,p,r)

```
if p==r return
q = (p+r)/2
Mergesort(A,p,q)
Mergesort(A,q+1,r)
Merge(A,p,q,r)
```

Merge(A,p,q,r)

```
n1 = q-p+1
n2 = r-q
create two temp arrays
L[n1+1]   R[n2+1]
Large = Max(L[n1-1], R[n2-1])+1
L[n1] = R[n2] = Large
i = j = 0
for k=p to r
  if L[i] < R[j]
    A[k] = L[i]
    i++
  else
    A[k] = R[j]
    j++
```

Time complexity

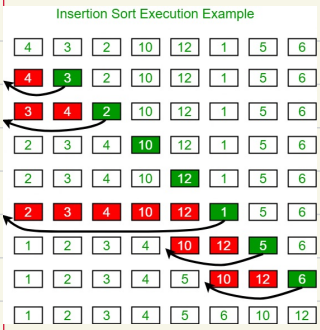
$T(n) = 2T(\frac{n}{2}) + cn$

$L = n^{\log_b a} = n^{\log_2 2} = n$

$n = n$
so, case 2 of Master theorem

$\Theta(n \log n)$

Insertion sort



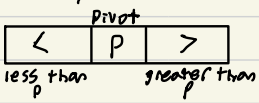

```
Insertion sort(A,n)
for i=1 to n-1
  key = A[i]
  j = i-1
  while(j>=0 And
                A[j]>key)
    A[j+1] = A[j]
    j--
  A[j+1] = key
```

Time complexity

$\Theta(n^2)$

Quicksort

- sorts array based on pivot



- uses two pointers
  - i: index of last element less than pivot
  - j: index of current element

- if work done at each level is balanced then $\log n$ levels. if not then n^2 levels

```
Quicksort(A,p,r)
if p>r return
q=partition(A,p,r)
Quicksort(A,p,q-1)
Quicksort(A,q+1,r)
```

```
Partition(A,p,r)
x=A[r]  //pivot is last element
i=p-1
for j=p to r-1
  if A[j] < x
    swap(A[j],A[i+1])
    i++
swap(A[r],A[i+1])
return i+1  //returns index
```

Time complexity

Best case: $\Theta(n \log n)$

Worse case: $\Theta(n^2)$

Max Sub Array

find sub array with max sum

-1	3	-5	4	9

L = Max Left
R = Max Right
C = Max Crossing

```
MaxSubArray(A,p,r)
if p==r  return A[p]
q=(p+r)/2
L=MaxSubArray(A,p,q-1)
R=MaxSubArray(A,q+1,r)
C=MaxCrossingSubArray(A,p,q,r)
return Max(L,R,C)
```

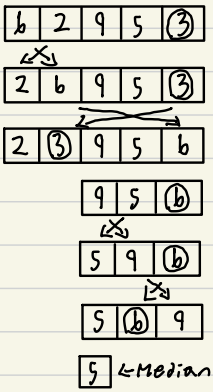
Time Complexity

$T(n) = 2T(\frac{n}{2}) + \theta(n)$

$n^{\log_2 2} \Rightarrow n = \theta(n)$

$\theta(n \log n)$

Median



i = median $\lfloor \frac{n+1}{2} \rfloor$

```
Select(A,p,r,i)
if p=r return A[p]
q=Partition(A,p,r)
if i=q return A[q]
if i<q
   return Select(A,q,r,i)
else
   return Select(A,q+1,r,i)
```

```
Partition(A,p,r)
x=A[r]  //pivot is last element
i=p-1
for j=p to r-1
  if A[j] < x
     Swap(A[j],A[i+1])
     i++
Swap(A[r],A[i+1])
return i+1   //returns index
```

Time Complexity

Best case: $\theta(n)$
Worse case: $\theta(n^2)$

Greeding Algorithms

- Makes a series of choices where you pick the best route at the given moment and commit

Optimization problem
- Maximum and Minimum problems
- Objective functions
- Constraints

Knapsack problem

- Pick a subset of items to place in a knapsack without exceeding total space available in the sack
  - less than or equal size
  - as full as possible
- Objective function: value in knapsack
- Constraints: sum of weights of taken items cannot exceed the capacity
- forms:
  - Fractional: can take parts of items
  - 0-1: take all or nothing

Functional knapsack (W, Nr, C, n)

W = array of weight
Nr = array of value
C = capacity
n = # of items
load = sum of weight

Find Nr[i]/W[i] for every item i
Sort the items in both arrays by Nr[i]/W[i]
load = 0
for i to n-1 AND load < C
  if W[i] ≤ C-load // completely fits
    take all of item i
    load += W[i]
  else
    take C-load of item i
    load = C
    break

Time complexity

$\Theta(n \log n)$

Activity selection problem

- Select activities with no overlap

Greedy Act Selection(S, f, n)

Sort Activities in S and f by ascending finish time in a
S = array of start time
f = array of finish time
n = # of acts
k = index of last taken act in A

Add a[0] to solution set A
for i=1 to n-1
  if S[i] ≥ f[k]
    Add a[i] to A
    k = i;

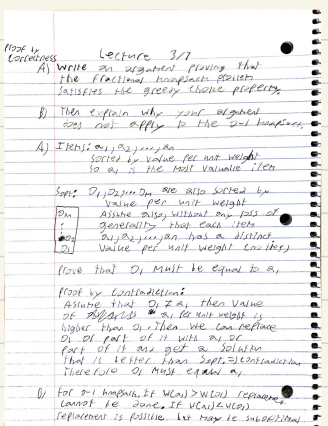
Time complexity

$\Theta(n \log n)$

Proof of correctness

Consists of:
1) Greedy choice property
   - there exists at least one optimal solution that begins with a greedy choice
2) Optimal structure

a_1 | a_2 | a_3 | a_4 | ... | a_n — remaining sub problems



Proof by Correctness — Lecture 3/7

A) Write an argument proving that the fractional knapsack problem satisfies the greedy choice property.

B) Then explain why your argument does not apply to the 0-1 knapsack.

A) Items: $a_1, a_2, \ldots, a_n$ sorted by value per unit weight so a_1 is the most valuable item

Sort: $O_1, O_2, \ldots, O_m$ are also sorted by value per unit weight

Base: Assume wlog without any loss of generality that each item $O_1, O_2, \ldots, O_m$ has a distinct value per unit weight (unlikely)

Prove that O_1 must be equal to a_1

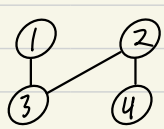
Proof by contradiction:
Assume that $O_1 \neq a_1$, then value of O_1 is $\neq a_1$ for unit weight is higher than O_1. Then we can replace O_1 or part of it with a_1 or part of it and get a solution that is better than Sopt. ⇒ Contradiction. Therefore O_1 must equal a_1

B) For 0-1 knapsack, if $W(a_1) > W(O_1)$ replacement cannot be done, if $W(a_1) \leq W(O_1)$ replacement is possible. Let maybe be subproblem

Graph coloring problem

- Choose color for each vertex where each is a different void
- NP complete problem
- Activity selection gives sub optimal solution
- edges represent conflicts

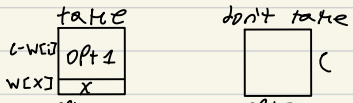
a	s	f
1	1	4
2	4	8
3	2	5
4	6	7



Dynamic programming

0-1 knapsack problem

- two decisions:



i \ j	1	2	3	4	5	6	7	8
No items	0	0	0	0	0	0	0	0
1	0	3	3	3	3	3	3	3
1+2	0	3	4	4	7	7	7	7
1+2+3	0	3	4	4	7	7	9	10

t[i][j]: Optimal solution for a knapsack of capacity j using items 1, 2, ..., j

Build this chart

```
DP_0-1_knapsack(w, v, n, c)
Create a 2-D array t with n+1
        rows and c+1 columns
for j to c   t[0][j]=0
for i=1 to n
  for j to c
    if w[i] > j
      t[i][j]=t[i-1][j]
    else
      t[i][j]=Max(v[i] + t[i-1][j-w[i]],
                  t[i-1] )
```

```
DecideItems  //gives items picked
i=n   j=c
for i=n down to 1
  if t[i][j] > t[i-1][j]
    take item i
    j=j-w[i]
```

Time complexities

0-1 knapsack:
$\theta(nc)$

DecideItems:
$\theta(n)$

Longest common subsequence

X=ABCDE
Y=BDAE
BDE is longest sequence

X[i] \ Y[j]	None	B	D	C	A	B
None	0	0	0	0	0	0
A	0	0	0	0	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
D	0	1	1	2	2	3
E	0	1	2	2	2	3

- Correctness not affected by order
- Performance affected by order
- Solution is always in last box

```
LCS(X,Y)
 M=X.len   n=Y.len
Create 2-D Array t  m+1 rows n-1 col
Fill 1st row and col with 0
for i=1 to m
  for j=1 to n
    if X[i] == Y[j]
      t[i][j] = t[i-1][j-1]+1
    else
      t[i][j]=Max(t[i-1][j], t[i][j-1])
```

```
LCS Tracer
while (i≥0 & j≥0)
  if (X[i]==Y[j])
    Add t[i][j] to LCS
    i--
    j--
  else
    if(t[i-1][j] <= t[i][j-1]
      i--
    else
      j--
```

Time complexities

LCS:
$\theta(mn)$

Tracer:
$\theta(m+n)$

Backtracking & Branch and bound

An upper bound of a 0-1 knapsack is found by solving the problem as if it's a fractional knapsack

Need 2 Arrays of size n
A_{curr} and A_{best} with following attributes:
- A.takenV (sum of taken value)
- A.untakenV (sum of untaken value)
- A.takenW (sum of taken weights)

backtracking
Branch-and-bound with simple upper bound
Brute force

```
FindSolutions(i) // i := level of tree
if (A_curr.takenW > C) return
if(i==n+1)
  if(A_curr.takenW ≤ C)
    if A_curr.takenV > A_best.takenV
       A_best = A_curr
  return
if totalSumofVals - A_curr.untakenV ≤ A_best.Val
  return

DontTake(i)
FindSolution(i+1)
undoDontTake(i)
Take(i)
FindSolution(i+1)
UndoTaken(i)

Take(i)
A_curr.takenV += V[i]
A_curr.takenW += W[i]
A[i] = true

DontTake(i)
A_curr.untakenV += V[i]
A[i] = false

undoTake(i)
A_curr.takenV -= V[i]
A_curr.takenW -= W[i]

undoDontTake(i)
A_curr.untakenV -= V[i]
```

Example

	W	V	V/W
item 1	3	21	7
item 2	6	30	5
item 3	5	15	3
item 4	10	20	2
item 5	6	6	1
		(94)	

cap = 19

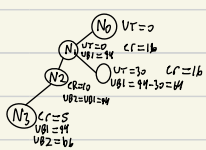
UB1: Assume you can take all unselected items
$\Theta(1)$

UB2: Assume that you can take any item that fits
$\Theta(n)$

UB3: look at remaining items and solve that with fractional knapsack
$\Theta(n)$

At Node N1
$UB2 = V_1 + \sum_{i \geq 2} V_i$
$= 21 + 73 = 94$

$UB2 = V_1 +$ Sum of vals of all items with weight ≤ 16
$= 21 + 73 = 94$



At N2 $UB2 = UB1 = 94$

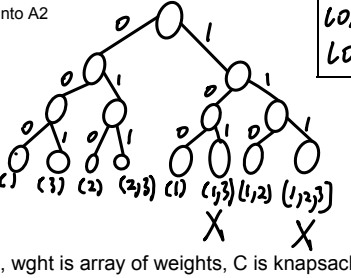
At N3
UB1 = Σ taken + Σ undecided
= 94
UB2 = Σ taken + Σ unselected that fit
= 66

Question: Backtracking and Branch-and-Bound [25 Points]

Consider the following backtracking algorithm for solving the **0-1 Knapsack Problem**:

$A_1 = A_{curr}$ $A_2 = A_{best}$

```
Backtracking_Knapsack (val, wght, C, n)
Create two arrays A1 and A2 each of size n and initialize all of their entries to zeros
Initialize the attributes takenVal, untakenVal and takenWght in each of A1 and A2 to zero
 totValSum = 0
for i = 1 to n
      totValSum += val[i]
FindSolutions (1)
Print: Contents of A1 //prints the actual contents of A1
Print: Contents of A2 //prints the actual contents of A2


FindSolutions (i)
1 if (A1.takenWght > C)
2     Print: Backtrack at i //prints actual value of i
3     return


4     if (i == n+1)
5         Print: Check leaf with soln A1 //prints actual content of A1
6         if (A1.takenVal > A2.takenVal)
7               Print: Copy A1 into A2 //prints actual array contents

8                   Copy A1 into A2
9         return


10   DontTakeItem(A1, i)
11   FindSolutions (i+1)


12   TakeItem(A1, i)
13   FindSolutions (i+1)
14   UndoTakeItem(A1, i)
```



Output

Print check leaf
Print check leaf
Copy A1 into A2
check leaf
check leaf
check leaf
Copy A1 into A2
check leaf
Backtrack at 4
check leaf
Backtrack at 4
Contents A1 : 111
Contents A2 : 110

Check leaf 000
Check leaf 001
Copy 001 into 000
Check leaf 010
Check leaf 011
Copy 011 into 001
Check leaf 100
Backtrack at 4
Check leaf 110
Copy 110 into 011
Backtrack at 4
Contents of A1: 111
Contents of A2: 110

val is array of values, wght is array of weights, C is knapsack capacity and n is number of items.

(a) Show **next to the above code** the output that it prints for the following input instance:

Item 1: weight = 7 Kg, value = $8
Item 2: weight = 2 Kg, value = $3
Item 3: weight = 5 Kg, value = $6
Capacity = 10

Assume that each print statement appears on a separate line in the output and prints the contents of each array as a **bit string**. For example, the print statement on Line 7 of **FindSolutions()** may print something like this: Copy 101 into 001. (10 points)

(b) In Assignment 4, you have implemented a Branch-and-Bound (B&B) algorithm for this problem using the following three upper bounds:
UB1: Sum the values of taken items and undecided items
UB2: Sum the values of taken items and the undecided items that fit in the remaining capacity
UB3: Sum the values of taken items and the solution to the remaining sub-problem solved as a Fractional Knapsack problem.
Given the following instance of the problem, Enter in the table below the value of each of the above bounds at each of the nodes specified in the table. Each row specifies a tree node, and the items that are not mentioned in that row have not been decided yet at that node. (9 points)

Item 1: weight = 3 Kg, value = $24 -> 8
Item 2: weight = 4 Kg, value = $20 -> 5
Item 3: weight = 5 Kg, value = $15 -> 3
Item 4: weight = 9 Kg, value = $18 -> 2
Item 5: weight = 7 Kg, value = $7 -> 1
Capacity = 15

SUM = 24+20+15+8+7 = 84

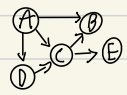
Node	UB1	UB2	UB3	
Item 1 not taken	60	38	47	
Item 1 taken and Item 2 not taken	64	64	53	cr=12
Item 1 taken and Item 2 taken	84	66	65	cr=8

Node		UB1	UB2	UB3
Item 1 not taken	A	60	60	47
Item 1 taken and Item 2 not taken	B	64	64	53
Item 1 taken and Item 2 taken	C	84	66	65

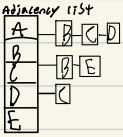
(c) Assuming that **k** backtrackings happen on Line 2 at the same level **i**, give a mathematical formula that expresses the number of pruned nodes **p** as a function of **i**, **k** and the number of items **n**. For example, each backtracking at i=n will prune 2 nodes, and each backtracking at i=n-1 will prune 6 nodes, and so forth (draw a little tree to verify this). Give an algebraic formula for **p** as a function of **i, k** and **n**. (6 points) .

Graph Algorithms

graphs can be directed and weighted



Adjacency list



Adjacency matrix

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	0	0	0
C	0	1	0	0	1
D	0	0	1	0	0
E	0	0	0	0	0

- list are good
  - when you want to know all neighbors
- Matrix
  - determining if two vertex are neighbors

Given a graph V vertices. what is the maximum number of possible edges

$E \leq \frac{V(V-1)}{2}$ if undirected

V(V-1) due to after edge V there is only V-1 edges left to choose

divide by 2 to not double count edges

$E \leq V(V-1)$ if directed

Complete graphs have all possible edges

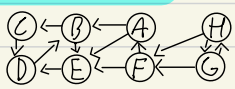
$\Theta(E)$ but its E in undirected 2E in directed both still $\Theta(E)$

Breadth first search

- uses a queue

```
BFS(G,s)                                   Θ(V)
for each vertex u in G
    u.d = ∞
    u.π = Null          //π = parent
s.d = 0                 //s = source  d = distance
create a queue Q
Q.enqueue
while Q is not empty
    u = Q.dequeue()  → Θ(V)
    for each neighbor in adj list of u
Θ(E)  if v.d == ∞    //∞ = unvisited
        v.d = u.d + 1
        Q.Enqueue(v)
        v.π = u
```

Depth-First search



undiscovered: w
discovered: G
finished: B

- don't use for distances
- $E = V-1$

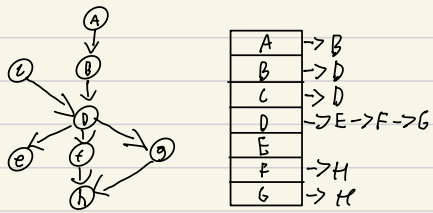
```
DFS(G)
for each vertex u in G  Θ(V)
    u.color = white
    u.π = Null
time = 0
for each vertex u in G
    if u.color = white  Θ(V)
        Explore(G,u)  O(V)
```

```
Explore(G,s)
s.color = Gray
time++
s.d = time
for each neighbor v adj list
Θ(E) if v.color = Gray
        print "cyclic graph"
    if v.color = White
        v.π = s
        Explore(G,v)
s.color = black
time++
s.f = time
```

Time complexity

$\Theta(V+E)$

Topological Sort



DAG

Given a directed acyclic graphs find an ordering of the vertices such that if there is an edge from X to Y, X appears before Y in this order

```
Topological sort(G)  -> Θ(V+E)
for each vertex u in G  -> Θ(V)
    u.depCount = 0
for each vertex u in G  -> V   Θ(V+E)
  for each neighbor v in u -> E
     u.depCount  += 1
Create a readyList R and solutionList S
for each vertex u in G  -> Θ(V)
  if u.depCount = 0
     R.AddToTail(u)
While R is not empty -> Θ(V+E)
   u = R.ExtractHead() -> V
   S.AddTail(u)
   for each neighbor v of u
      v.depCount -= 1  -> E
      if v.depCount == 0
         R.addTail(v)
```

← slower

Topological w/ DFS(G) Θ(V+E)

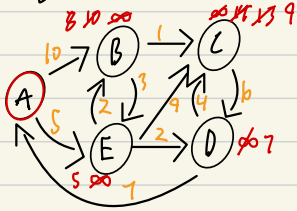
Do DFS whenever a vertex u is finished

S.AddToHead(u)

← faster

Dijkstra's Algorithm

Single Source Shortest Paths



	d	π
A	0	
B	∞ 10 8	A E
C	∞ 14 13 9	E D B
D	∞ 7	E
E	∞ 5	A

- looking for weighted distance
  + edge weights must be positive

Dijkstra(G,S) $\Theta(V \log V + E \log V)$

```
for each vertex u in G
  u.d = ∞
  u.π = NULL
s.d = 0
Create a Priority Queue Q
for each u in G
  Q.insert(u)        -> V inserts
While Q is not empty
  u = Q.ExtractMinimum()   -> V extracts
  for each neighbor v of u
    if u.d + w(u,v) < v.d
      v.d = u.d + w(u,v)   -> O(E) x decrease key time
      v.π = u
```

Priority Queue

for an unsorted array

$T(V,E) = \cancel{O(V)} + O(V^2) + O(E)$
$= O(V^2) + O(E)$
$= O(V^2)$ as $E = V^2$ for complete graph

for a heap

$T(V,E) = O(V \log V) + O(V \log V) + O(E \log V)$
If every vertex is reachable from the source than $E = \Omega(V)$
$= O(E \log V)$

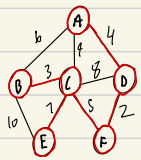
for a dense graph

Assume $E = O(V^2)$
unsorted $= O(V^2)$
Heap $= O(V^2 \log V)$

for a sparse graph

$E = O(V)$
unsorted array $= O(V^2)$
Heap $= O(V \log V)$

Prims Minimum Spanning tree



Weighted undirected
$E_{max} = \frac{V(V-1)}{2}$
$E_{tree} = V-1$

- Prims Alg is greedy and is based upon links
+ connects all vertexs with min weight

	key	TT
A	0	Null
B	∞ 6 3	Null A C
C	∞ 9 8 5	Null A D F
D	∞ 4	Null A
E	∞ 7	Null C
F	∞ 2	Null D

Prim(G)
Select a source S
for each vertex u in G
  u.d = ∞
  u.π = Null
S.key = 0
Create a Priority Queue Q
for each vertex u
  Q.Insert(u) -> V times
While Q is not empty
  u = Q.ExtractMin() -> V times
  for each neighbor v of u
    if v ∈ Q AND w(u,v) < v.key
      v.key = w(u,v) -> E x decrease key
      v.π = u

only difference with Dijkstra's

Priority Queue

for an unsorted array

$T(V,E) = \cancel{O(V)} + O(V^2) + O(E)$
$= O(V^2) + O(E)$
$= O(V^2)$ as $E = V^2$ for complete graph

for a heap

$T(V,E) = O(V\log V) + O(V\log V) + O(E\log V)$
If every vertex is reachable from the source than $E = \Omega(V)$
$= O(E\log V)$

for a dense graph

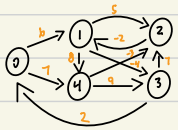
Assume $E = O(V^2)$
unsorted $= O(V^2)$
Heap $= O(V^2 \log V)$

for a sparse graph

$E = O(V)$
Unsorted array $= O(V^2)$
Heap $= O(V\log V)$

Bellman Ford Algorithm

- uses dynamic programming
- simple path has V-1 edges
- considers first paths with 1 edge
  + then up to 2 edges
    keeps going to V-1 edges
  + can use 0 edges as base case



# of Edges

vertex #	0	1	2	3	4
0	0	0	0	0	0
1	∞	6	6	2	2
2	∞	∞	4	4	4
3	∞	∞	2	2	-2
4	∞	7	7	7	7

_ = changes

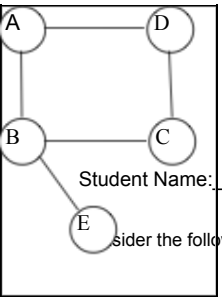
BellmanFord(G,S) $\theta(VE)$

```
create a VxV matrix A
set 1st row to zero
set 1st col to ∞ except for src
for j=1 to V-1 // # of edges        θ(V)
    for i=1 to V-1 // vertex #      θ(V)
        A[i][j] = ∞
        for each incoming edge (k,i)
Aggrigate   if A[k][j-1] + w(k,i) < A[i][j]
E θ(E)          A[i][j] = A[k][j-1] + w(k,i)
```

- Need to initialize all values to ∞

$T(V,E) = O(V(V+E))$
$= O(V^2 + VE)$
$E = \Omega(V)$ so
$= O(VE)$
Dense graph $\theta(V^3)$

- no way to get improvement of # of Edges = to V unless there is a negative edge weight



CSC 140: Advanced Algorithm Design and Analysis
Second Quiz, Spring 2018
Friday, 4/20/2018

Student Name:_____ Student ID:_____

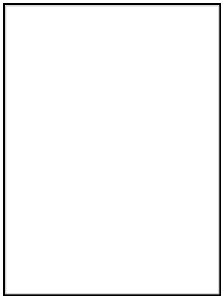
sider the following pseudo-code for the **BFS algorithm** studied in class.

```
BFS (G, r)  //G is a graph, r is the root (source) of the search
1 for each vertex u in G (in alphabetical order)
2       u.d = ∞
3       u.π = NULL
4 r.d = 0
5 Create an empty queue Q
6 Q.Enqueue(r)
7 Print: Enqueue r
8 while Q is not empty
9       u = Q.Dequeue()
10       Print: Dequeue u
11       for each neighbor v in the adjacency list of u (in alphabetical order)
12              Print: Check v
13              if v.d == ∞
14                    v.d = u.d+1
15                    v.π = u
16                    Q.Enqueue(v)
17                    Print: Enqueue v
```

(a) Show the output printed by the above BFS code when applied to the graph below with the source being **A**. Note that the *for* loops on Lines 1 and 11 go through the vertices in **alphabetical order**. Note also that the print statements print the actual vertex names (A, B, C, etc.) not the program variable names. Assume that each print statement appears on a separate line. The first line in the output is given to you. (20 points)

Output

Enqueue A
Deq A
Check B
Enq B
Check D
Enq D
Deq B
Check C
Enq C
Check E
Enq E
Deq D
Check A
Check C
Deq C
Check B
Check D
Deq E
Check B



(b) Fill in the table below with the number of times each of the specified lines is executed when each of the given graphs is the input to the algorithm. Give the exact count in terms of the number of vertices **V** not the asymptotic value. Note that due to the symmetry in the graphs below, the answers will be the same regardless of the source.

(30 points)

Graph	Line 9	Line 13	Line 16
Totally disconnected graph with **V** vertices and no edges	1	0	0
Undirected graph shaped as a **single circle** that goes through all **V** vertices	V	2V	V-1
Complete undirected graph with **V** vertices	V	V(V-1)	V-1

(c) Write the pseudo-code for an algorithm that runs after the above BFS algorithm and prints for each vertex the actual shortest path (not the distance) computed by the above BFS algorithm. The path is a sequence of vertices like <A, B, C>. The path for each vertex must be printed in the right order not in reverse order. Your algorithm must be **as efficient as possible**. (20 points)

$\Theta(V^2)$

```
for each vertex u in G
    list.Clear()
    p = u.π
    while p != NULL
        list.AddToHead (p)
        p = p. π
    list.Print()
```

NP completeness

Intro

An informal definition of a problem is NP complete if there is no known polynomial time alg that solves every instance of the problem exactly

= conditions

- to show a problem is NP complete you show its the same as an NP complete problem through reduction

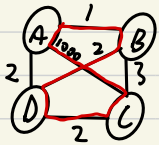
Optimization problem

Decision problem

Subset problem
Given a set S and a target number T, determine if there is a subset S that sums to T
Ex)
$S = \{2,3,4\}$
$T = 5 \rightarrow$ Yes $\{2,3\}$
$T = 8 \rightarrow$ No
Similar to 0-1 knapsack

* simple cycle that goes through all vertexs (TSP tour) *

Traveling salesman problem
Given an a complete unweighted graph, find a simple cycle that goes through all vertices and has the lowest possible cost



Through greedy alg

Traveling salesman problem
Given an a complete unweighted graph and a target T, determine if the graph has a simple cycle that goes through all vertices with cost less than or equal T

verification, reduction, complexity case

TSP-decision
If someone gives you a proposed soln. (seq of vertex) can you verify that its correct (legal and meets T)

It is verifiable in polynomial time

Complexity Class

Class P: problems that can be solved in poly time

Class NP: problems verified in poly time

Class NP-complete: the hardest probs in NP

X is NP-complete if all probs in NP are reducible to y

Prob X is reducible to prob y if you can transform any instance of x into an instance of y

X: to be proven
y: known to be NP-complete

$X \leq_p y$ X
$y \leq_p X$ ✓

A problem L is NP-complete if
1) $L \in NP$
2) for every prob $L' \in NP$, $L' \leq_p L$

if a prob satisfies condition 2 but not 1 then its NP Hard

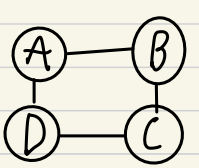
Ex) TSP-opt

Decision probs can be solved in poly time

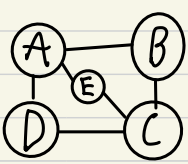
reduction examples

Hamiltonian cycle prob

Given a graph, determine if the graph has a simple cycle that goes through all vertexs



Hamiltonian



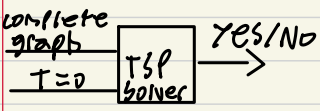
Not hamiltonian

Given that Hamiltonian is NP-complet prove that TSP-decision is NP-complete

condition 1 is met

Condition 2
We need a reduction from Ham to TSP
Show Ham $\leq_p$ TSP

Assume you have a solver for TSP



Given a graph g construct a complete graph g' with the following edge weights

$$w(u,v) = \begin{cases} 0 & \text{if } (u,v) \in E \\ 1 & \text{if } (u,v) \notin E \end{cases}$$

g' has a TSP tour of cost 0 iff g is ham

Subset sum and set partition problem

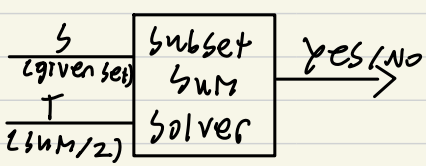
Set partition
Given a set S of integers determine if S can be partitioned into two subsets with equal sums

Assuming that set partition is known to be NP-complete prove subset sum is NP-complete

Prove
1) Subset sum is verifiable in poly. time

Set partition $\leq_p$ subset sum

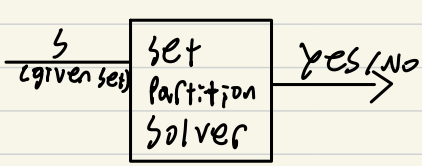
Assume that you have a solver for subset sum



Prove
1) Set partition is verifiable in poly. time

Set partition $\geq_p$ subset sum

Assume that you have a solver for set partition



* opt = NP-Hard
decision = NP-complete *

Construct $S' = S \cup \{a\}$
Where $a = 2xT - sum$
Then S' can be partitioned
IFF S has a sum that
sums to T

Vertex cover & set cover

Vertex cover (optimization)
Given an undirected graph
find the min # of vertexs
that cover all edges

Vertex cover (decision)
Given an undirected graph
and target T determine
if all edges can be
covered with T vertices
or less

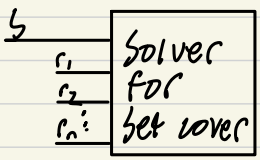
Set cover (opt)
Given a set S and m
subsets of S find
the min # of subsets
whos union gives S

Set cover (decision)
Given a set S and m
subsets determine if
S can be constructed
using T or less of
given subsets

Given vertex cover is
known to be NP-complete
Prove set cover is
NP-complete

1) Set cover $\in$ NP

2) Need to reduce vertex
cover to set cover



reduling any instance of
vertex cover into an
instance of set cover:
Given a graph G, let S
be the set of edges E.
for each vertex u, create
a subset S_u that consist
of all the edges that are
covered by u
The target T will be
the same for both
so S can be covered by
T subset iff edges E
can be covered using T
vertices

Problem Types

Max	Min	Decision by Nature
0-1 knapsack	Set cover	Subset sum
Clique	Vertex cover	Ham cycle
	graph coloring	Set partition
	TSP	SAT

CSC 206: Algorithms and Paradigms
CSC 140: Advanced Algorithms
Sample Questions on Algorithm Usage

Map each of the following problems to one of the problems studied in class, and then give the **running time** of the **most efficient** algorithm studied in class for solving the problem. Briefly explain your mapping and describe any **adjustments or transformations** that may be needed to solve the given problem. If the problem is an NP-complete or NP-hard problem for which we did not study a solution, do the mapping and then state that it is NP-Complete or NP-hard.

(a) Suppose that you are developing an **airline reservation application**. You have the costs of the flights that connect **A** airports around the world. Of course, only a limited number of airport pairs have direct flights connecting them. On a given day **d**, there are **F(d)** flights, and each flight **f** between airport **i** and airport **j** has a cost **C(f)**. Which algorithm would you use to implement each of the following functionalities?

1. The user enters a date and the names of two airports (for example, Sacramento and Paris), and the application outputs the route with **the cheapest cost** (for example, <Sacramento, LA, New York, Paris>, with flight numbers).

Problem: Shortest weighted paths
Mapping:
Airport maps to vertex
Flight maps to edge
Flight cost maps to edge weight
Algorithm: Dijkstra
Algorithm design technique: Greedy
Running time: O(ElogV) = O(F(d)logA)
A heap implementation is used because the graph is sparse

2. The user enters a date and the names of two airports (for example Sacramento and Paris), and the application outputs the route with the **minimum number of stops** (for example, <Sacramento, San Francisco, Paris>, with flight numbers).

Problem: Shortest unweighted paths
Mapping:
Airport maps to vertex
Flight maps to edge
Algorithm: BFS
Algorithm design technique: greedy
Running time: O(V+E) = O(A+F(d))

(b) Suppose that you are the manager of a new airline company. There are **P** airports around the world that you would like to cover, but it is too expensive to have a direct flight between every pair of airports. So, you would like to find the least expensive way of connecting all **P** airports directly or indirectly together. For **each** pair **(i, j)** of airports, you have an estimate **C(i, j)** of the cost of

offering a direct flight between **i** and **j;** if offering a direct flight between a pair of airports is infeasible, the cost is set to **infinity**. Which algorithm would you use to determine which pairs of airports should have direct flights between them such that all **P** airports are connected together with the minimum total cost?

Problem: Minimum spanning tree (network)
Mapping:
Airport to vertex
Direct flight maps to edge
Complete graph (there is an edge between every pair of vertices)
Direct flight cost maps to edge weight
Algorithm: Prim's
Algorithm design technique: greedy
Running time: $O(V^2) = O(P^2)$
An unsorted array implementation is used because the graph is dense, in fact, complete. The graph is complete, because we are given the estimated cost for **every** pair of airports.

(c) Suppose that you are a manager at some company and you have a certain budget of $**B** for buying new equipment. Each department has provided you with a list of devices that they would like to buy, and you have put all of these requests together in one big list of **D** devices**.** For each device **i**, you have been provided with its price **P(i)** as well as a score **S(i)** indicating its importance or added value to the company. Which algorithm would you use to decide which of the **D** devices to buy in order to maximize the added value without exceeding the given budget?

Problem: 0-1 Knapsack
Mapping:
Device maps to an item
Device price maps to item weight
Device added value maps to item value
Budget maps to the knapsack capacity
Algorithm: Dynamic programming
Running time: $\theta(nC) = \theta(DB)$

(d) Suppose that you would like to write a program that generates an exam schedule for **N** courses. For each pair **(i, j)** of courses, you know if there is at least one student taking both courses. Which algorithm would you use to find the **minimum number of time slots** that are needed to construct the exam schedule such that no student has two exams scheduled in the same time slot?

Problem: Graph coloring
Mapping:
Course maps to vertex
Conflict between two courses because they have one or more common students maps to an edge
Time slot maps to color
Note that this is the general graph coloring problem not the special case of interval graph coloring.
This is because the conflicts are not caused by interval overlapping.
Problem is NP-hard
We did not study an algorithm for this problem, but it can be solved using branch-and-bound