Platforms & Tools

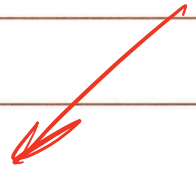
D2L (DEN):
- Syllabus ✓
- Lecture Notes ✓
- Lecture Videos ✓
- HW Assignments ✓
- HW Submissions ✓
- Any other reference material ✓
- online Exams ✓

Piazza: Discussion Board ✓

Roles & Responsibilities

- Instructor — lectures & Discussions
- TAs — HWs & Exams
- Graders — HW grading
- Course Producers — *
- CS Dept. Advisors
- DEN Support

Textbooks



- Algorithms Design by Jon Kleinberg & Eva Tardos

- Supplemental Textbook:
  Introduction to Algorithms,
  3rd edition, by Cormen et al.

Your Responsibilities

1. – Attending lectures & Discussions
2. – Completing reading assignments
3. – Doing HW problems
4. – Doing as many other problems from textbook as possible

Your Grade

Exam 1	30%	Oct. 1
Exam 2	30%	Nov. 5
Exam 3	35%	Dec. 3
Final Proj	5%	Dec. 8
	100%	

Grading Scale

90 - 100	A	60 - 64.99	C^+
86 - 89.99	A^-	55 - 59.99	C
80 - 85.99	B^+	50 - 54.99	C^-
70 - 79.99	B	45 - 49.99	D
65 - 69.99	B^-	Below 45.99	F

- Scale will be adjusted if median falls below 75.

- At least the top 20% of the class will receive an A^-.

- At least the next 10% of the class (between top 20% and top 30%) will receive an A^-.

Prerequisites

- Discrete Math – Mathematical Induction
- Sorting methods
- Basic data structures: Arrays, stacks, queues, linked lists
- Basics of graphs: Trees, cycles, DAG, adjacency list/matrix, etc.
- Graph search algorithms: BFS, DFS

High level Syllabus

Today!

- Introduction
- Review of some prereq's + asymptotic notation
- Major algorithmic techniques
  - Greedy
  - Divide & Conquer
  - Dynamic Programming

→ Exam 1

- Network Flow
- Computational Complexity Theory
- Approximation Algorithms
- Linear Programming

→ Exam 2

Covers only 2 topics

→ Exam 3

↑ Comprehensive

Corrections

1- ~~An algorithm is a set of instructions in machine language.~~

2- ... ~~Algorithmic science advanced on Wall Street ...~~

3- ... ~~Invite 6 million algorithms for a listen ...~~

When studying a problem, we go through the following steps:

1- Come up with a concise problem statement

2- Present a solution

3- Prove correctness

4- Perform complexity analysis

Stable Matching

Stable Matching Example

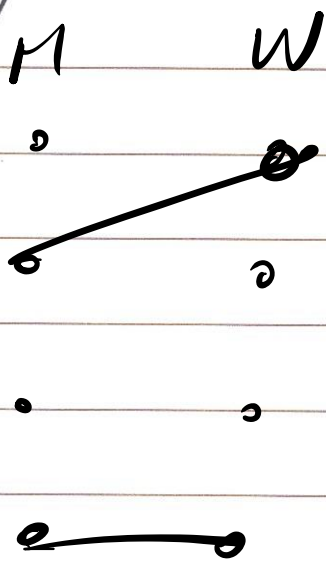
Problem: We are interested in matching n men with n women so that they could stay happily married ever after.

Step 1: Come up with a concise problem statement.

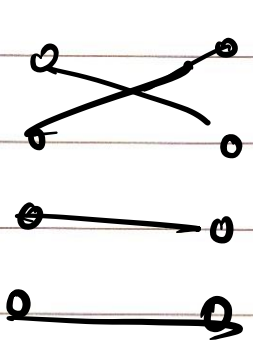
We have a set of n men, $M = \{m_1, \ldots m_n\}$

We have a set of n women, $W = \{w_1, \ldots w_n\}$

Def. A *Matching* S is a set of ordered pairs.



Def. A *perfect matching* S' is a matching with the property that each member of M and each member of W appear in exactly one pair in S'.



A perfect matching

Add notion of preferences

Each man $m \in M$ ranks all women

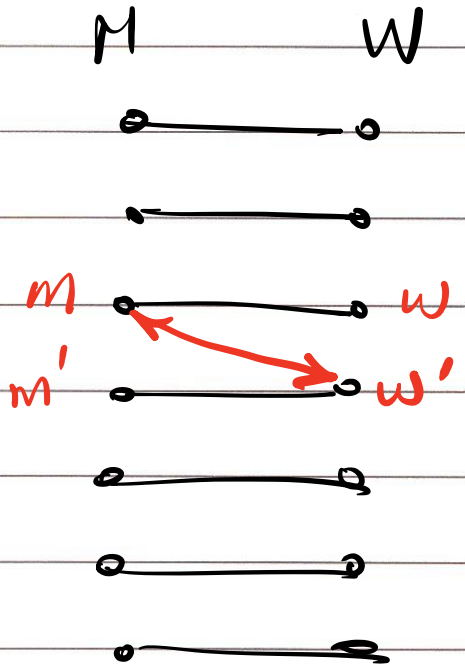
- m prefers w to w' if m ranks w higher than w'.

- Ordered ranking of m is his preference list

$$P_{m_i} = \{w_{i_1}, w_{i_2}, \ldots w_{i_n}\}$$

Same for women, i.e. each woman $w \in W$ ranks all men...

S



Such a pair (m, w') is called an instability

Def. Matching S is stable if

1- It is perfect

2- There are no instabilities WRT S

Step 1: Input: Preference lists for a set of n men & n women.

Output: Set of n marriages w/ no instabilities

Step 2: Gale-Shapley Alg.

Step 3 Proof of Correctness

(1) From the woman's perspective, she starts single, and once she gets engaged and she can only get into better engagements

(2) From the man's perspective, he starts single, gets engaged, and may get dropped repeatedly only to settle for a lower ranking woman.

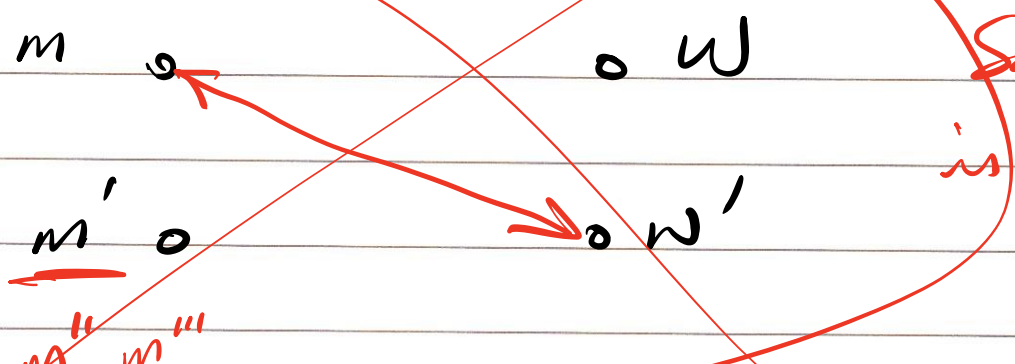
(3) Solution will terminate in at most n^2 iterations

(4) Solution is a perfect matching

(5) solution is a stable

Proof by Contradiction

Assume an instability exists in our solution involving two pairs $(m, w), (m', w')$



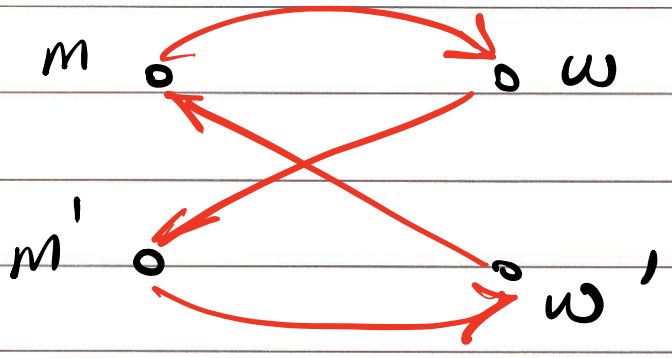
Say (m, w') is an instability

Q: Did m propose to w' at some point in the execution?

If no, then w must be higher than w' on his list → contradiction!

If yes, he must have been rejected in favor of m'' and due to (1) either $m'' = m'$ or m' is better than m''

→ contradiction!



$(m, w), (m', w')$

$(m, w'), (m', w)$

Step 4 Complexity Analysis

1- Identify a free man $O(1)$

2- For a man m, identify the highest ranked woman to whom he has not yet proposed. $O(1)$

3- For a woman w, decide if w is engaged, and if so to whom $O(1)$

4- For a woman w and two men m & m', decide which man is preferred by w $O(1)$

5- Place a man back in the list of free men. $O(1)$

1. Identify a free man

	get	put
Array	O(1)	O(1)
linked list	O(1)	O(1)
queue	O(1)	O(1)
stack	O(1)	O(1)

2. Identify the highest ranked woman to whom m has not yet proposed.

Keep an array Next[1..n], where Next[m] points to the position of the next woman he will be proposing to on his pref. list.

Men's preference list: ManPref[1..n, 1..n], where
ManPref[m, i] denotes the i^{th} woman on man m's preference list.

To find next woman w to whom m will be proposing to:

w = ManPref[m, Next[m]]

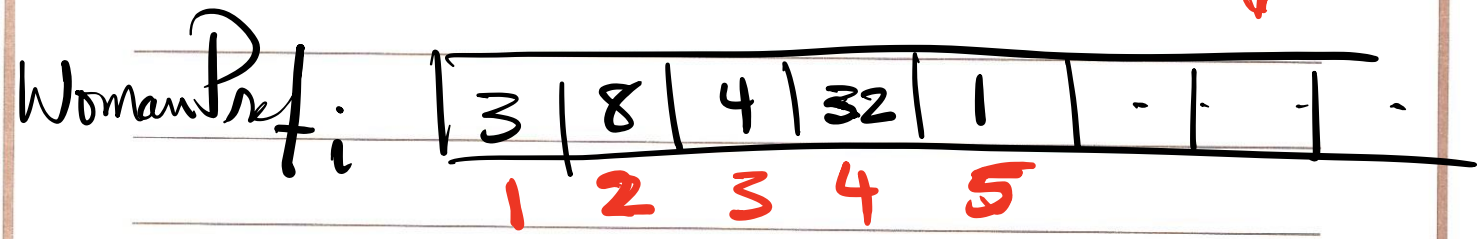
takes O(1)

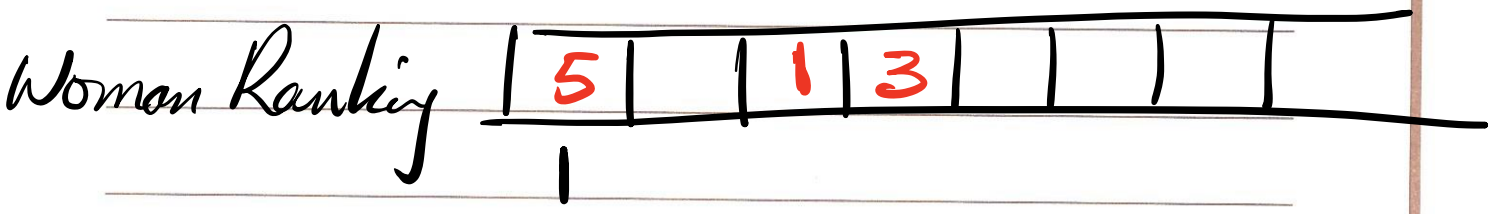
3_ Determine woman w's status

Keep an array called Current[1..n]
where Current[w] is Null
if w is single & set to m
if w is engaged to m.

takes O(1)

4_ Determine which man is preferred by w.



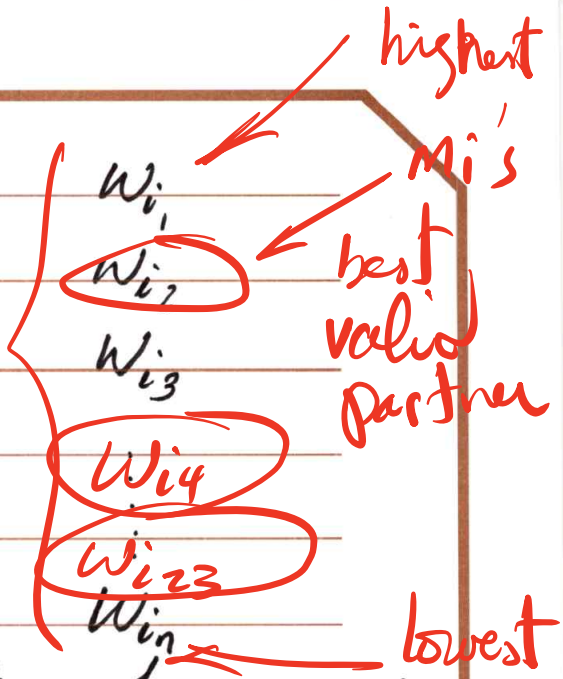


Preparation before entering GS iterations

Create a Ranking array where Ranking[w,m] contains the rank of man m based on w's preference

Preparation	+	GS iteration
$O(n^2)$		$O(n^2)$

Overall Complexity = $O(n^2)$



Def. Woman w is a valid partner of a man m if there is a stable matching that contains the pair (m, w)

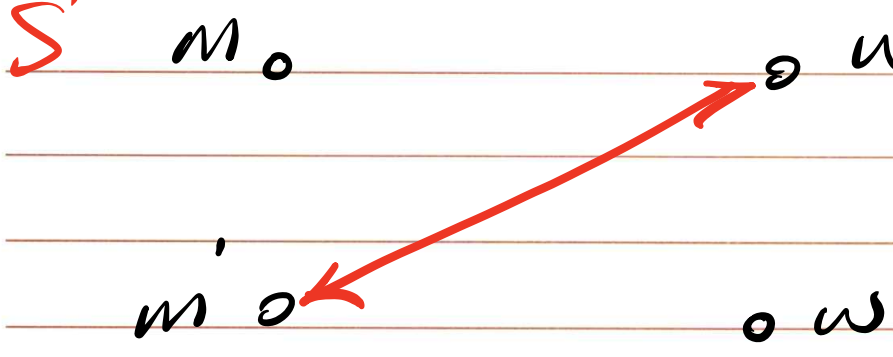
Def. m's best valid partner is

Claim: Every execution of the G-S algorithm (When men propose) results in the same stable matching regardless of the order in which men propose.

Plan: to prove this, we will show that when men propose, they always end up with their best valid partner.

Proof by contradiction:

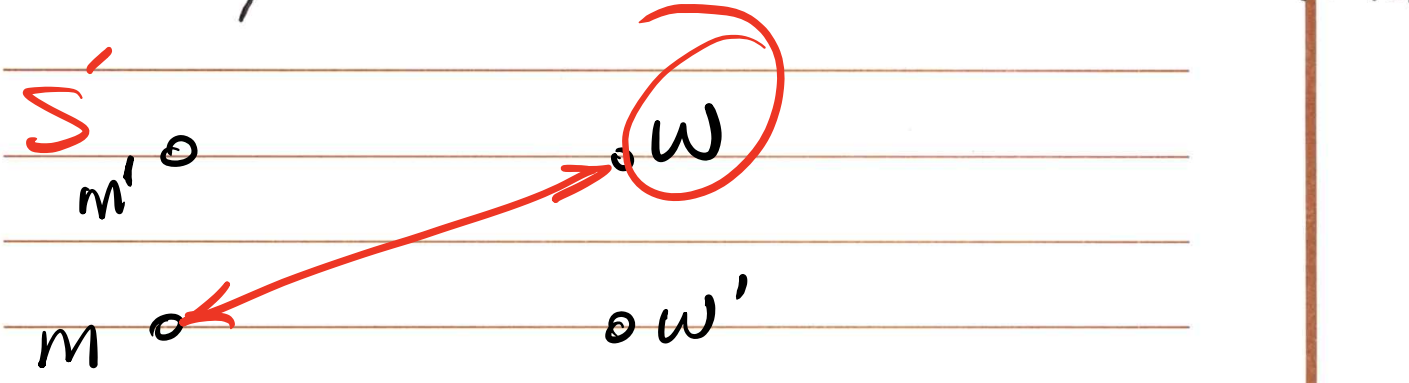
Say m is the first man rejected by a valid partner w. in favor of m'



Claim: When men propose, women end up with their worst valid partners

Proof: By contradiction ....

Suppose we end up with a matching S where for a pair (m, w) in S, m is not w's worst valid partner.
So there must be another matching S' where w is paired with a man m' whom she likes less.



Discussion 1

1. Prove that every execution of the G-S algorithm (when men are proposing) results in the same stable matching regardless of the order in which men propose.

Solution is the textbook. This is done by showing that regardless of the order of men proposing, men always end up with their best valid partners.

2. Prove that when we run the G-S algorithm with men proposing, women end up with their worst valid partners.

This solution is also provided in the textbook.

3. True or False:
In every stable matching that Gale–Shapley algorithm may end up with when men propose, there is a man who is matched to his highest-ranked woman.

This is false. Here is a counterexample:

m1	m2	m3	w1	w2	w3
w1	w1	w2	m3	m2	m1
w2	w2	w1	m2	m1	m2
w3	w3	w3	m1	m3	m3

Given the above preference lists, when men propose we end up with the following pairs: (m1,w3), (m2,w2), and (m3,w1) where none of the men end up with their highest-ranked woman.

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

This is True. We can prove this by contradiction. Let's say we have already found the bipartition X,Y where all edges in the graph go between X and Y. Now let's assume that we have another bipartition X', Y' different from X, Y. If X', Y' is different from X, Y, then there must be at least one node *i* that used to be in X and now is not in X' but has moved into Y'. We can run a BFS starting from node *i.* Say *i* is at level 1 in the BFS tree. All nodes at level 2 that must have belonged to Y should now be in X', and all nodes at level 3 which must have belonged to X should now be in Y', and so on. In other words, all node that used be in X are now in Y' and all nodes that were in Y are now in X'. So X', Y' is not different from X, Y.

CSCI 570 - Fall 2021 - HW 1

Due: *Sep 3rd*

1. Reading Assignment: Kleinberg and Tardos, Chapter 1.

2. Solve Kleinberg and Tardos, Chapter 1, Exercise 1.

   False. Suppose we have two men m, m_0 and two women w, w_0. Let m rank w first; w rank m_0 first; m_0 rank w_0 first; and w_0 rank m first. We see that such a pair as described by the claim does not exist.

   Rubric (5pt):

   - 2 pt: Correct T/F claim
   - 3 pt: Provides a correct counterexample as explanation

3. Solve Kleinberg and Tardos, Chapter 1, Exercise 2.

   True. Suppose S is a stable matching where m and w are not paired with each other. Suppose that contains the pairs (m, w_0) and (m_0, w) instead, for some $m_0 \neq m, w_0 \neq w$. Clearly, the pairing (m, w) is preferred by both m and w over their current pairings, contradicting the stability of S. Thus, by contradiction every stable matching must contain (m, w).

   Rubric (5pt):

   - 2 pt: Correct T/F claim
   - 3 pt: Provides a correct explanation

4. State True/False: An instance of the stable marriage problem has a unique stable matching if and only if the version of the Gale-Shapely algorithm where the male proposes and the version where the female proposes both yield the exact same matching.

   True.
   Proving the given statement requires proving the claims in two directions ('if' and 'only if').
   Claim: *"If there is a unique stable matching then the version of the Gale-Shapely algorithm where the male proposes and the version where the female proposes both yield the exact same matching."*

   The proof of the above claim is clear by virtue of the correctness of Gale-Shapely algorithm. That is, the version of the Gale-Shapely algorithm

where the male proposes and the version where the female proposes are both correct and hence will both yield a stable matching. However since there is a unique stable matching, both versions of the algorithms should yield the same matching.

Next, we prove the converse: *"If the version of the Gale-Shapely algorithm where the male proposes and the version where the female proposes both yield the exact same matching then there is a unique stable matching."*

The proof of the converse is perhaps more interesting. For the definition of best valid partner, worst valid partner etc., see page 10-11 in the textbook.

Let S denote the stable matching obtained from the version where men propose and let S_0 be the stable matching obtained from the version where women propose.

From (page 11, statement 1.8 in the text), in S, every woman is paired with her worst valid partner. Applying (page 10, statement 1.7) by symmetry to the version of Gale-Shapely where women propose, it follows that in S_0, every woman is paired with her best valid partner. Since S and S_0 are the same matching, it follows that for every woman, the best valid partner and the worst valid partner are the same. This implies that every woman has a unique valid partner which implies that there is a unique stable matching.

Rubric (10pt):

- 4pts: Correct ' $\Longrightarrow$ ' proof
  (a) 1pt: Correctly state forward claim
  (b) 3pt: Correctly justify forward claim
- 6pts: Correct ' $\Longleftarrow$ ' proof
  (a) 1pt: Correctly state backwards claim
  (b) 5pts: Correctly justify backwards claim

5. A stable roommate problem with 4 students a, b, c, d is defined as follows. Each student ranks the other three in strict order of preference. A matching is defined as the separation of the students into two disjoint pairs. A matching is stable if no two separated students prefer each other to their current roommates. Does a stable matching always exist? If yes, give a proof. Otherwise give an example roommate preference where no stable matching exists.

   A stable matching need not exist. Consider the following list of preferences. Let a, b, c all have d last on their lists. Say a prefers b over c, b prefers c over a, and c prefers a over b. In every matching, one of a, b, c should be paired with d and the other two with each other. **Without loss of generality**, suppose a gets paired with d (thus, b, c with each other). Now, a prefers c over his current partner d and c also prefer to

be with each other. Thus every matching is unstable no stable matching exists in this case.

Rubric (6pt):

- 1pts: Correct answer (i.e. matching need not exist)
- 5pts: Correct counter-example as explanation
  (a) 2pt: Describe the preference lists
  (b) 3pts: Explain why no stable matching exists

6. Solve Kleinberg and Tardos, Chapter 1, Exercise 3.

   We will give an example (with $n = 2$) of a set of TV shows/associated ratings for which no stable pair of schedules exists. Let a_1, a_2 be set the shows of A and let b_1; b_2 be the set of shows of B. Let the ratings of a_1; a_2; b_1; b_2 be 1, 3, 2 and 4 respectively. In every schedule that A and B can choose, either a_1 shares a slot with b_1 or a_1 shares a slot with b_2. If a_1 shares a slot with b_1, then A has an incentive to make a_1 share a slot with b_2 thereby increasing its number of winning slots from 0 to 1. If a_1 shares a slot with b_2, then B has an incentive to make b_2 share a slot with a_2 thereby increasing its number of winning slots from 1 to 2. Thus every schedule that A and B can choose is unstable.

   Rubric (6pt):

   - 1 pt: Correct claim that there are configurations of shows without stable matching
   - 5 pt: Provides a correct counterexample

7. Solve Kleinberg and Tardos, Chapter 1, Exercise 4.

   We will use a variation of Gale and Shapley (GS) algorithm, then show that the solution returned by this algorithm is a stable matching.

   In the following algorithm (see next page), we use hospitals in the place of men; and students in the place of women, with respect to the earlier version of the GS algorithm given in Chapter 1.

   This algorithm terminates in $O(mn)$ steps because each hospital offers a position to a student at most once, and in each iteration some hospital offers a position to some student.

   The algorithm terminates by producing a matching M for any given preference list. Suppose there are $p > 0$ positions available at hospital h. The algorithm terminates with all of the positions filled, since, any hospital that did not fill all of its positions must have offered them to every student. Every student who rejected must be committed to some other hospital. Thus, if h still has available positions, it would mean total number of available positions is $> n$, the number of students. This contradicts the assumption given, proving that all the positions get filled.

```
1: while there exists a hospital h that has available positions do
2:   Offer position to the next highest ranked student s in the preference list
     of h that wasn't offered by h before
3:   if s has not already accepted a position at another hospital h' then
4:     s accepts the position at h.
5:   else
6:     Let s be matched with h'.
7:     if s prefers h' to h then
8:       then s rejects the offer of h
9:     else
10:      s accepts the position at h freeing up a position at h'
11:    end if
12:  end if
13: end while
```

The assignment is stable. Suppose, towards a contradiction, that the M produced by our adapted GS algorithm contains one or more instabilities. If the instability was of the first type (a student s' was preferred over a student s by a hospital h but was not admitted), then h must have considered (offered) s before s' who wasn't offered, which is a contradiction because h prefers s' to s. Thus, the instability was not of the first type. If the instability was of the second type (there is student s, s' currently at hospitals h, h' and there's a swap mutually beneficial to h and s'), then h must not have admitted s' when it considered it before s, which implies that s' prefers h' to h, a contradiction. Thus, the instability was not of the second type. Thus, the matching was stable. Thus at least one stable matching always exists (and it is produced by the adapted GS algorithm as above).

Rubric (15pt):

- 8pts: Algorithm
  (a) 1pt: Loop condition (line 1)
  (b) 2 pt: hospitals offer next highest ranked student (line 2)
  (c) 2pts: case that s is free (lines 3-4)
  (d) 3pts: cases if s is at another h'
- 7pts: Proof
  (a) 1 pt: Algorithm terminates in finite steps (optional to mention in O(mn) steps)
  (b) 2pt: All positions get filled
  (c) 2pts: Explain why no instability of first type
  (d) 2pts: Explain why no instability of second type

8. N men and N women were participating in a stable matching process in a small town named Walnut Grove. A stable matching was found after

the matching process finished and everyone got engaged. However, a man named Almazo Wilder, who is engaged with a woman named Nelly Oleson, suddenly changes his mind by preferring another woman named Laura Ingles, who was originally ranked right below Nelly in his preference list, therefore Laura and Nelly swapped their positions in Almanzos preference list. Your job now is to find a new matching for all of these people and to take into account the new preference of Almanzo, but you don't want to run the whole process from the beginning again, and want to take advantage of the results you currently have from the previous matching. Describe your algorithm for this problem.
Assume that no woman gets offended if she got refused and then gets proposed by the same person again.

First, Almanzo leaves Nelly and proposes to Laura:

- If Laura rejects Almanzo and decides to stay with her current fiance (her current fianc is ranked higher than Almanzo in her preference list), then Almanzo just goes back and proposes to Nelly again and gets engaged with her. Algorithm stops. This is where you take advantage of the previous matching.
- Else, Laura accepts Almanzo proposal, she will get engaged with him and leave her current fiance (say Adam). And here comes the main part of the matching puzzle.
  (a) Note that Adam may or may not simply propose to Nelly and be engaged with her. The reason is Nelly may be far below Laura in his preference list and there are more women (worse than Laura and better than Nelly), to whom he prefers to propose first. Furthermore, the fact that Nelly becomes single can create more instabilities for those men, who once proposed to Nelly and were rejected by her because she preferred Almanzo. Thus, Nelly is ranked higher than the current fiances of those once unlucky men and they are ready to propose to Nelly again for a second chance.
  (b) Thus, one way is to put Adam and all those rejected-by-Nelly men in the pool of single men, and put Nelly and the fiancees of the rejected-by-Nelly men in the pool of free women. Similar to Nellys case, moving any woman from the engaged state to the single state may create more instabilities; thus one needs execute step (b) recursively to build the pools of free men and women. In the worst case, all men and all women will end up in the single pools; in the best case (when Laura accepted Almanzo's proposal), only Adam and Nelly are in the 2 single pools, one for men and another for women.
  (c) Execute G-S algorithm until there is no free man. For each man, if there is a woman who once rejected him and is now free, he will try to propose to her again. Otherwise, he will propose to

those women that he has never proposed to, moving top down in his preference list. In the latter case, more men (subsequently more women) may go to the single pools.

Note: The above solution is described in details in order to clarify the problem. For this problem, students can simply write pseudo code without having to discuss the details of each case as above.

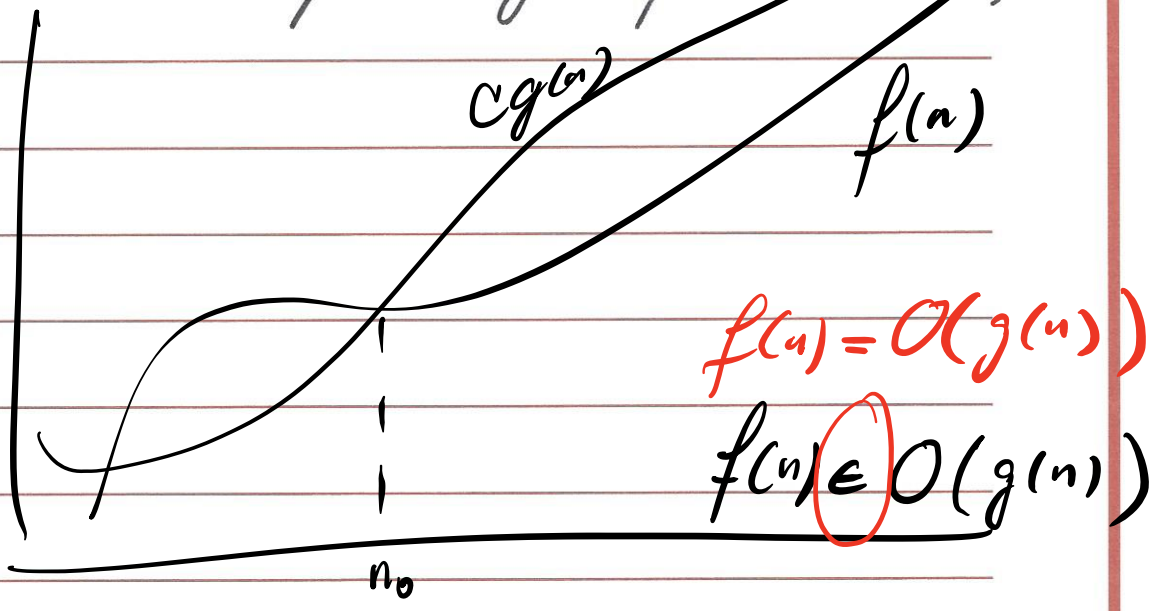
Rubric (15pt):

- 2pt: Cover the early termination case (if Laura rejects Almanzo, the matching stays as is).
- 2pts: Mention the instability of Adam once Laura leaves him
- 3pts: Mention the instability of men that were rejected by Nelly.
- 5pts: Recursively building the pools of free men and women
- 3pts: Executing GS **while exapanding single pools as needed at each step**.

Completely acceptable if any of these points is not explicitly described but the algo/pseudocode covers it.

Review of the Asymptotic Notations

Formally, $O(g(n)) = \{ f(n) \mid$ there exist positive constants C and n_0 such that $0 \leq f(n) \leq C\,g(n)$ for all $n \geq n_0 \}$



$f(n) = O(g(n))$

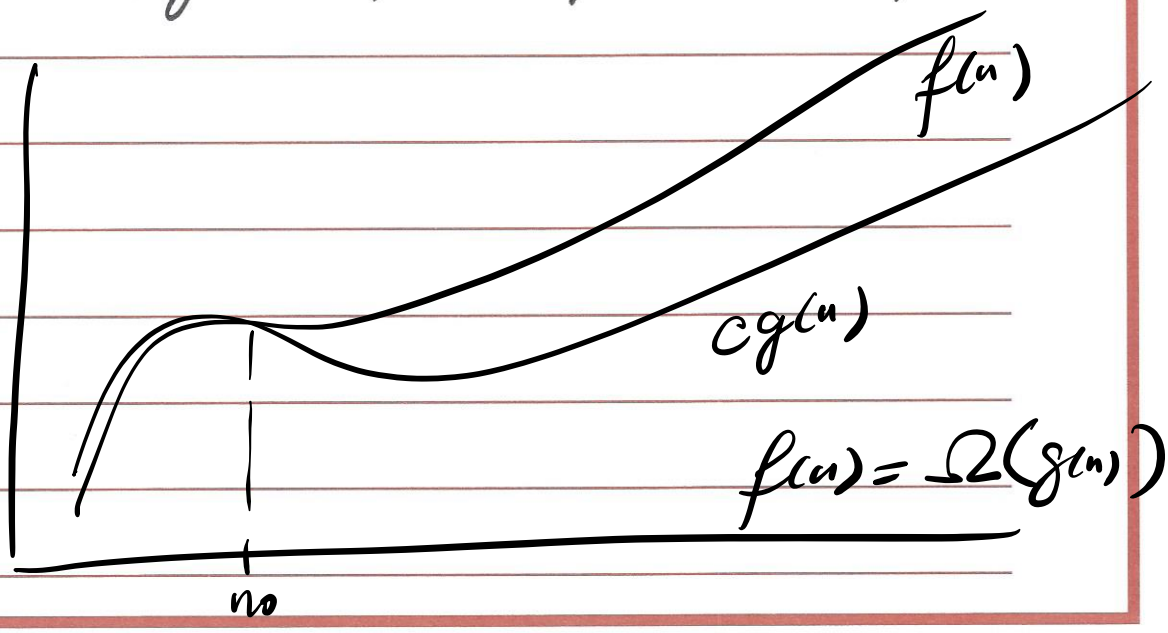
$f(n) \in O(g(n))$

T Any quadratic function is $O(n^2)$

T Any linear " " "

F Any cubic " "

$\Omega(g(n)) = \{ f(n) \mid$ there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for $n \geq n_0 \}$

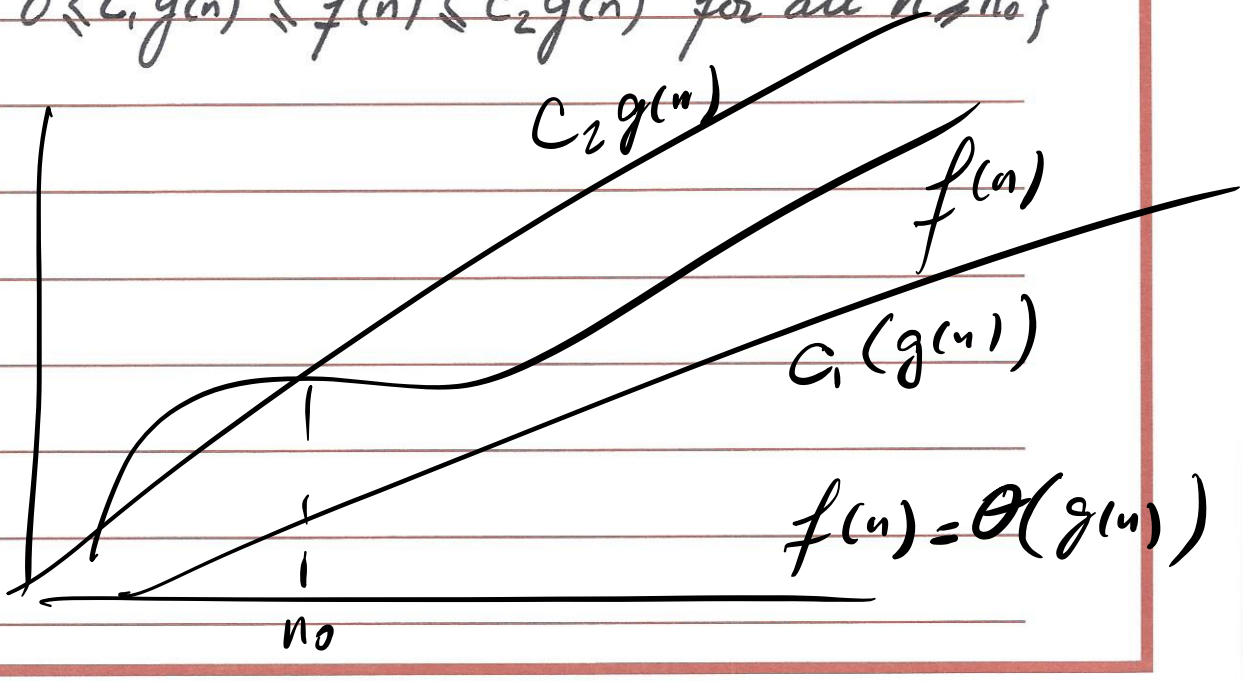


T Any quadratic func. is $\Omega(n^2)$

F Any linear " "

T Any cubic " "

$\Theta(g(n)) = \{ f(n) \mid$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$



T Any quadratic func. is $\Theta(n^2)$

F Any linear " " "

F Any Cubic " "

	Worst Case	Best Case
Linear Search	$O(n), \Theta(n), \Omega(n)$	$O(1), \Theta(1), \Omega(1)$
Binary "	$O(\lg n), \Theta(\lg n), \Omega(\lg n)$	$O(1), \Theta(1), \Omega(1)$
Insertion Sort	$O(n^2), \Theta(n^2), \Omega(n^2)$	$O(n), \Theta(n), \Omega(n)$
Merge Sort	$O(n \lg n), \Theta(n \lg n), \Omega(n \lg n)$	$O(n \lg n), \Theta(n \lg n), \Omega(n \lg n)$
⋮		

Worst case performances:

Algorithm A: $\Theta(4^n n^3 \lg n)$

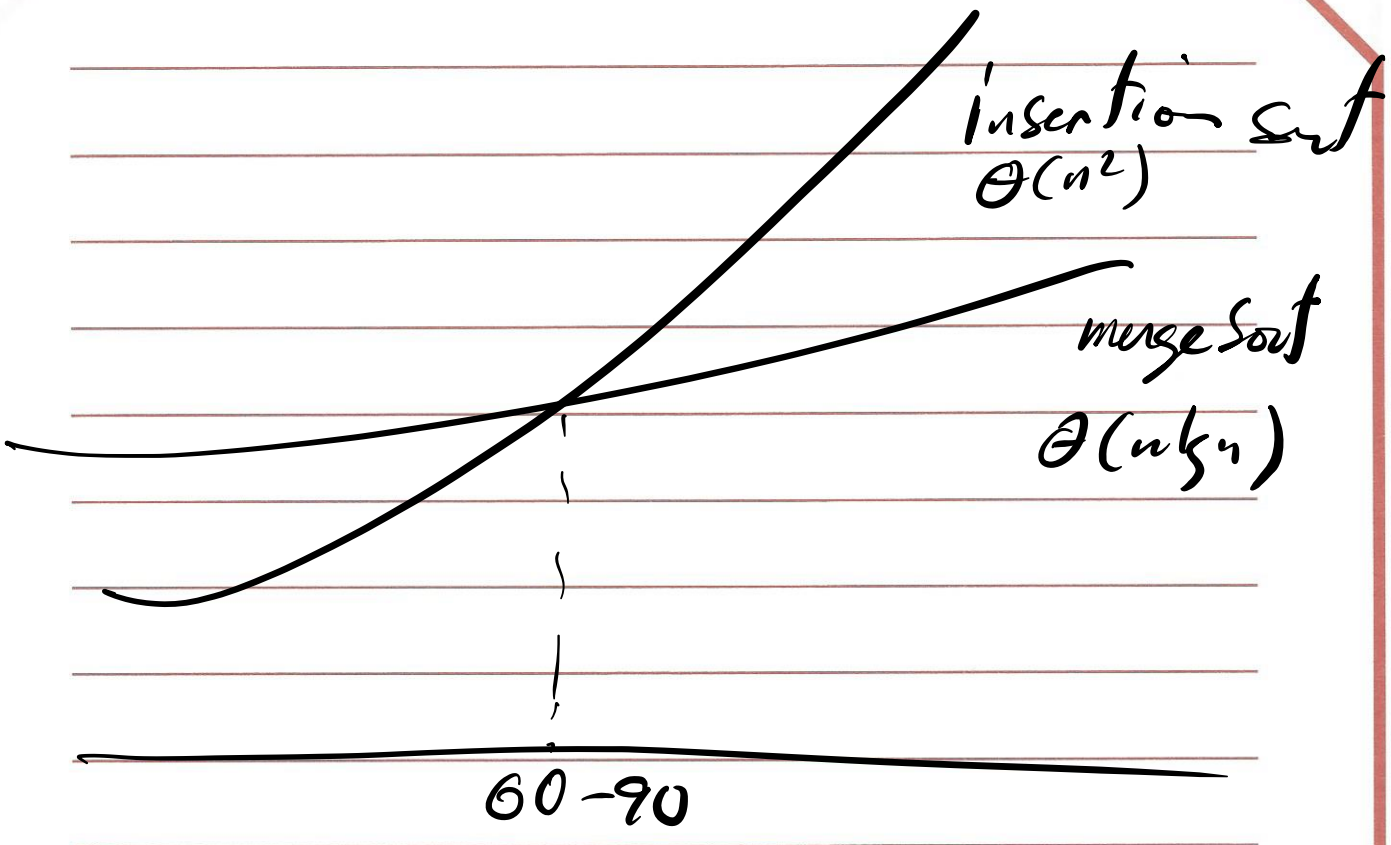
Algorithm B: $\Theta(3^n n^8 (\lg n)^2)$

- Exponential component
- Polynomial "
- Logarithmic "

fastest growing

4

slowest growing


insertion sort
$\Theta(n^2)$
merge sort
$\Theta(n \lg n)$
60-90

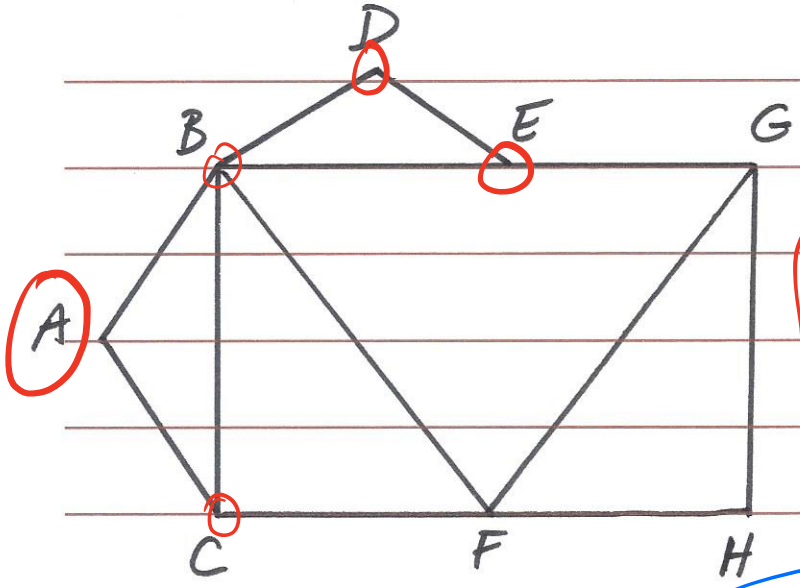
Review of BFS & DFS

Q: What are we searching for?

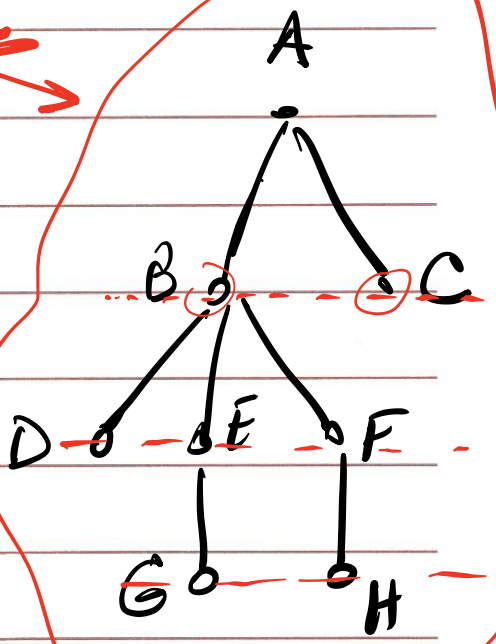
- Find out if there is a path from A to B.

- Find all nodes that can be reached from A.

BFS



BFS Tree →

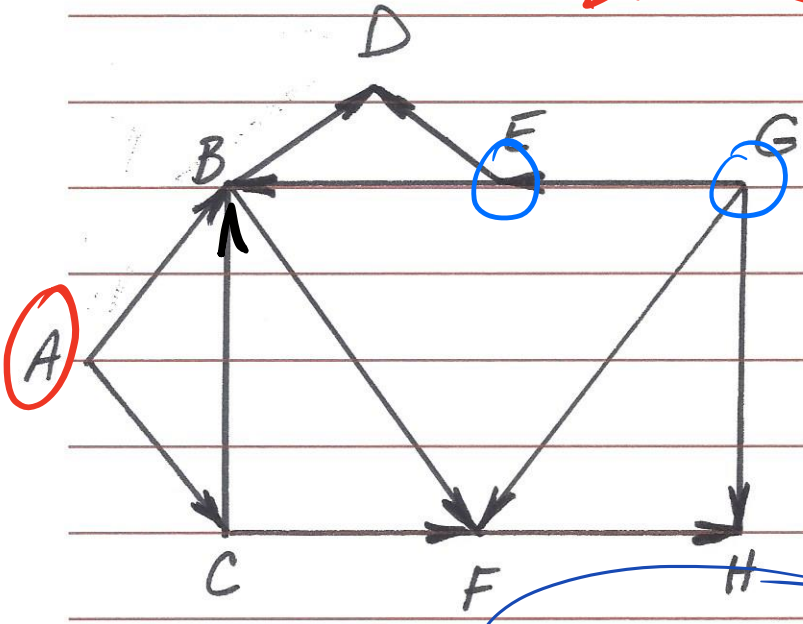


Takes $O(m+n)$

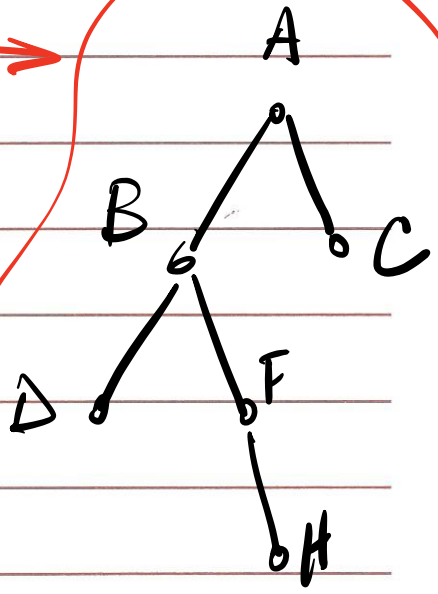
m: no. of edges

n: " " nodes

DFS



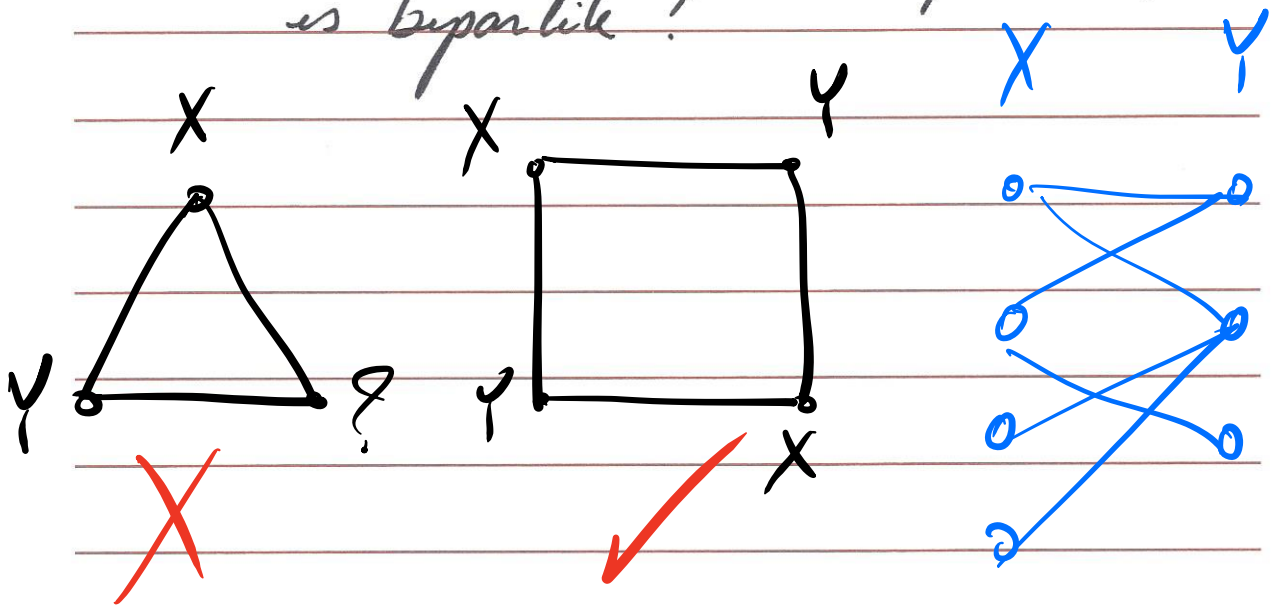
DFS Tree →

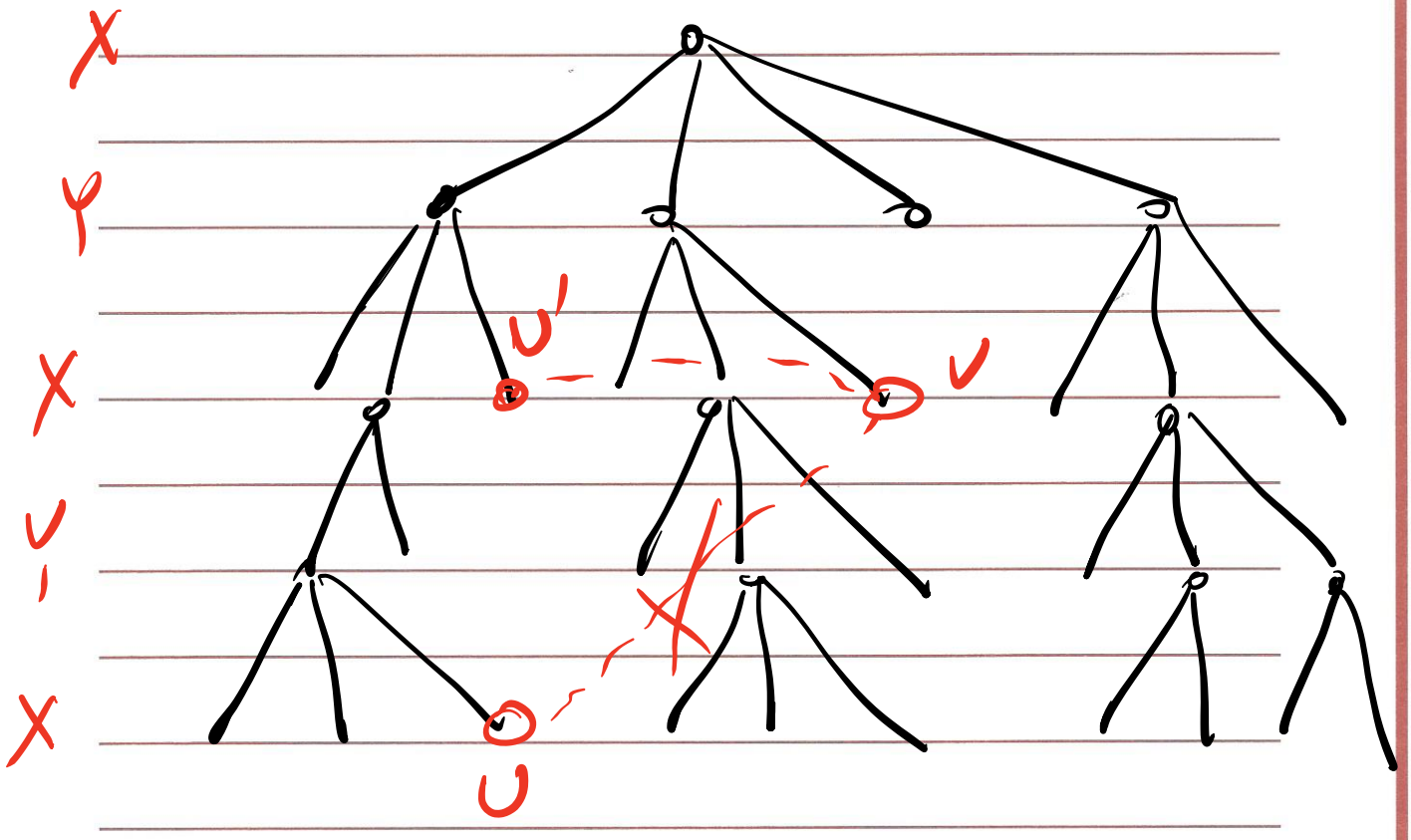


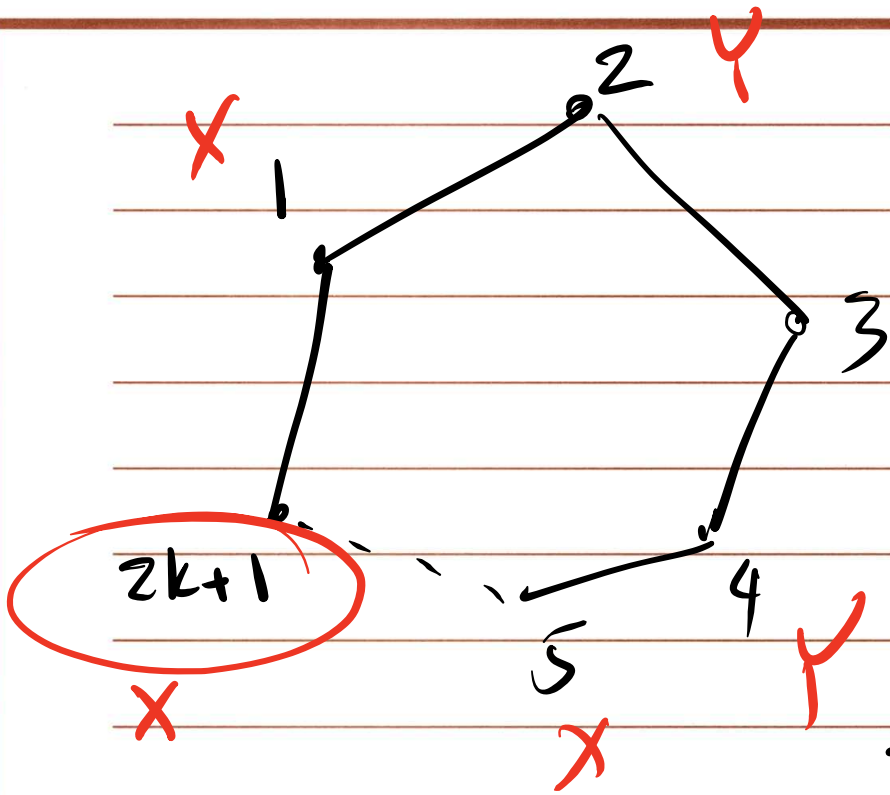
takes $O(m+n)$ $O(m \log n + n)$

$\Theta(m+n)$ ~~$\Theta(m \log n + n)$~~

Q: How do you determine if a graph is bipartite?

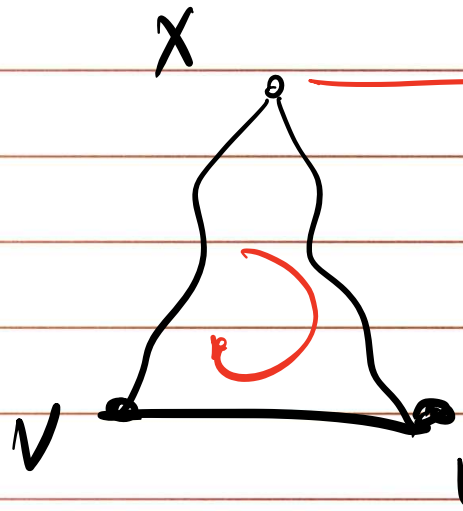






FACT:

If a graph G is bipartite it cannot have an odd cycle.



level j

length of cycle = 2*(j-i) + 1

odd!

Solution:

$O(m+n)$ Run BFS starting from any node, say S. Label each node Red or Blue depending on whether they appear at an odd or even level on the BFS tree.

$O(m)$ Then, go through all edges and examine the labels at the two ends of the edge. If all edges have a Red end and a Blue end, then the graph is bipartite.

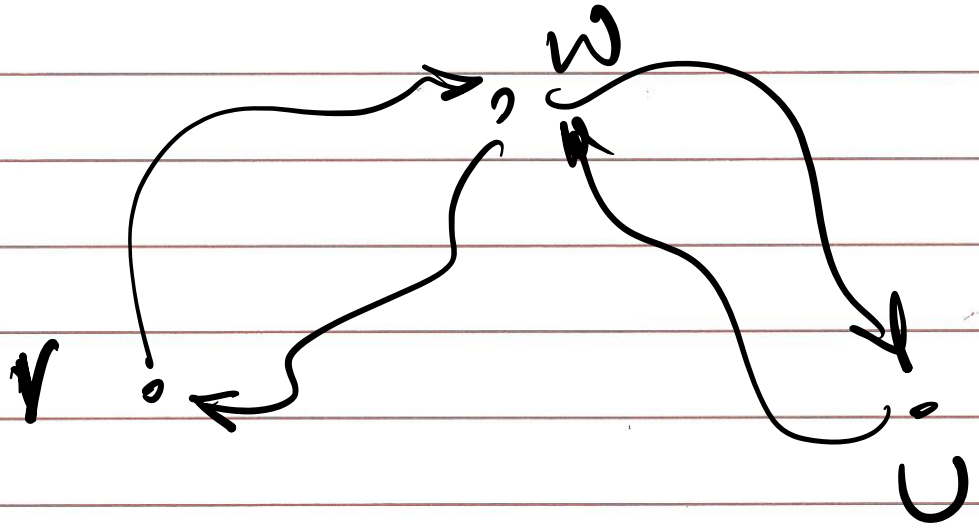
Otherwise, the graph is not bipartite.

overall complexity $= O(m+n)$

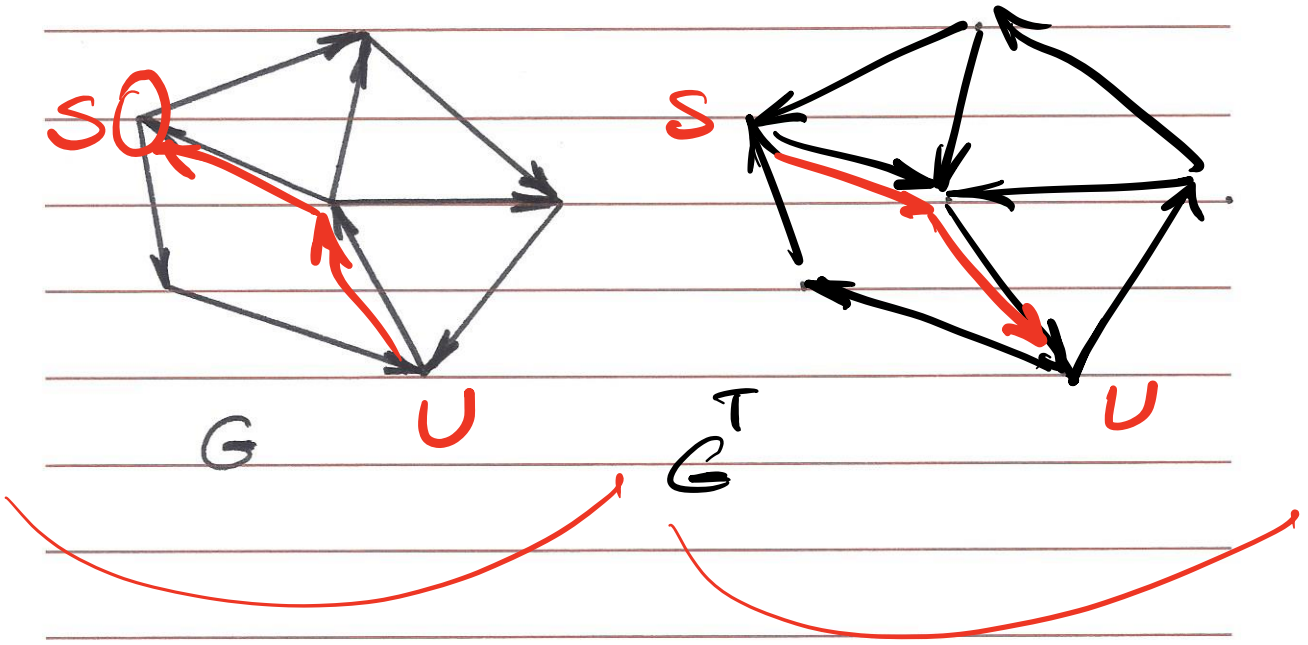
Def. A directed graph is strongly connected if there is a path from any point to any other point in the graph.

Q: How do you know if a given directed graph is strongly connected?

Brute Force Sol.: Run BFS/DFS n times, once from each node $\rightarrow O(n^2+mn)$



Transpose of a directed graph



Mutually Reachable Nodes

Solution:

1. Use BFS or DFS to find all nodes reachable from s (an arbitrary node) in G. If some nodes are not reachable from s, stop. The graph is not strongly connected.

O(m+n)

Otherwise, continue with step 2.

2. Create G^T (Transpose of G)

O(m+n)

3. Use BFS or DFS to find all nodes reachable from s in G^T. If some nodes are not reachable from s, then the graph is not strongly connected.

O(m+n)

Otherwise, the graph is strongly connected.

Overall Complexity = O(m+n)

Discussion 2

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$\log n^n,\ n^2,\ n^{\log n},\ n \log \log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n)).* Is it true that $2^{f(n)} = O(2^{g(n)})$?

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$\log n^n$, n^2, $n^{\log n}$, $n \log \log n$, $2^{\log n}$, $\log^2 n$, $n^{\sqrt{2}}$

$2^{\log n} = n^1$

$\log n^n = n \log n$

$\log^2 n,\ n^1,\ n \log\log n,\ n \log n,\ n^{\sqrt{2}},\ n^2,\ n^{\log n}$

$n^2 > n \log n$

$n^{1.001} > n \log n$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n)* = *O(g(n))*. Is it true that $2^{f(n)} = O(2^{g(n)})$?

$f(n) = 2n \qquad g(n) = n$

$2^{2n} \neq O(2^n)$

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
  while (n > 0)
    find_max(L, n); //finds the max in L[0…n-1]
    n = n/4;
```

O(n) ← Find_Max runs in linear time WRT n

$\log_4 n$ (

n = 3/4 → 0

Carefully examine to see if this is a tight upper bound (Big θ)

Runs in $O(n \log n)$ ✓

more careful examination:

$cn + cn/4 + cn/16 + \cdots$

$< 2cn = O(n)$

$n + n/2 + n/4 + n/8 + \cdots$

$2n$

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

$\Omega(1)$ ✓ $\Omega(n)$ ✓

n	output
0	a
1	aa
2	aaaa
⋮	
n	aa...aa (2^n)

$\Theta(2^n)$

Carefully examine to see if this is a tight lower bound (Big θ)

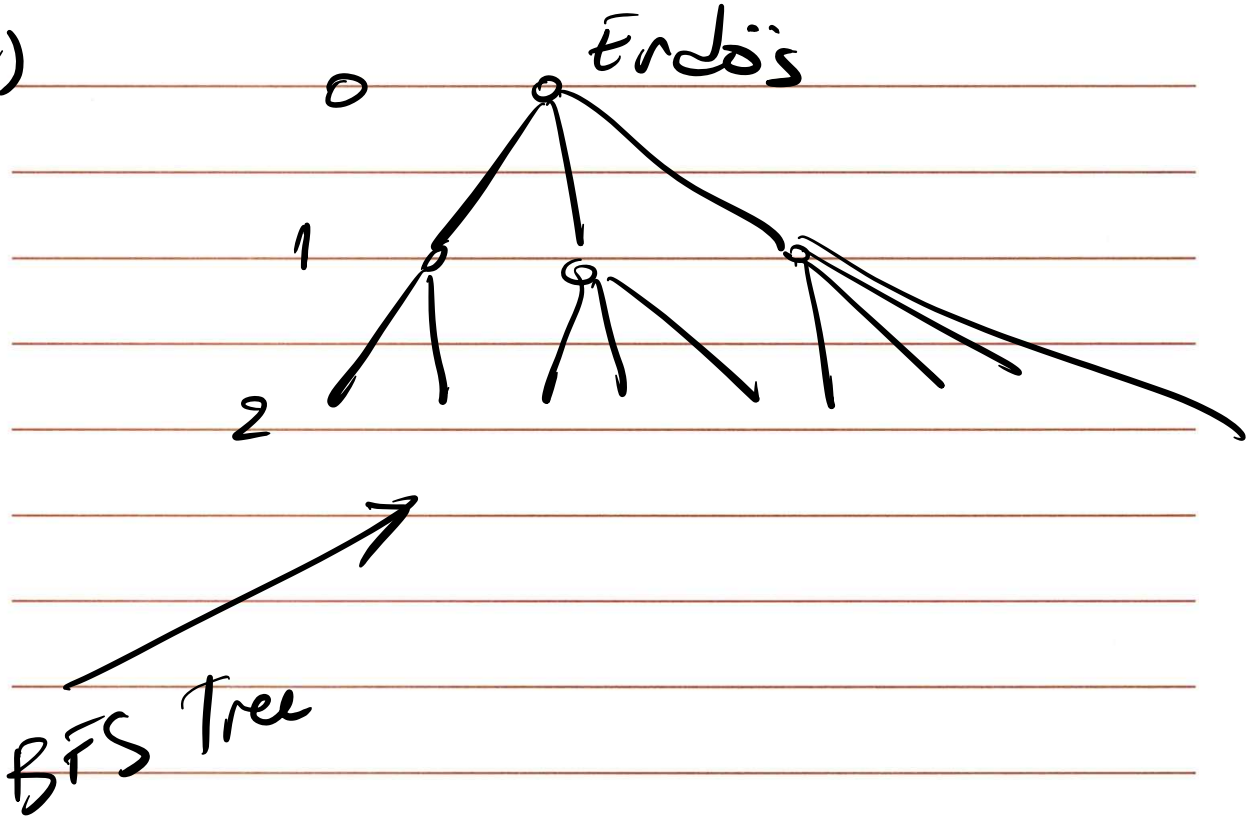
5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

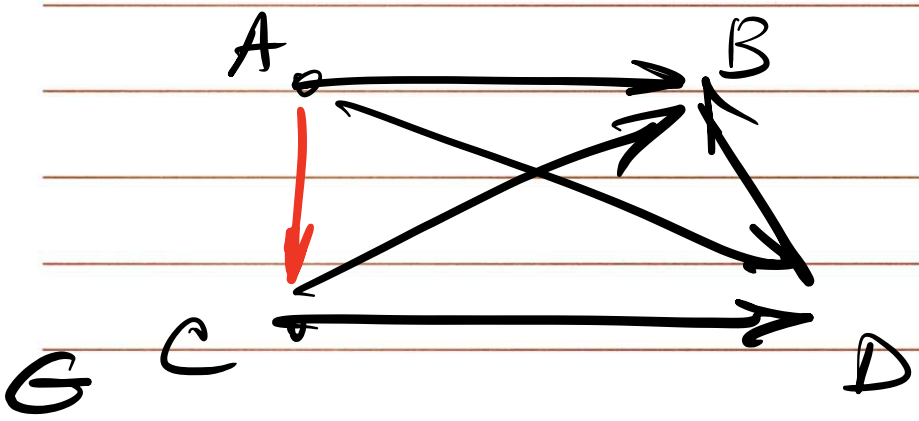
a) use an undirected graph where
- nodes represent Mathematicians
- edges " co-authorship

b) Run BFS from Erdös (or the math') and find the other node

c)



6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

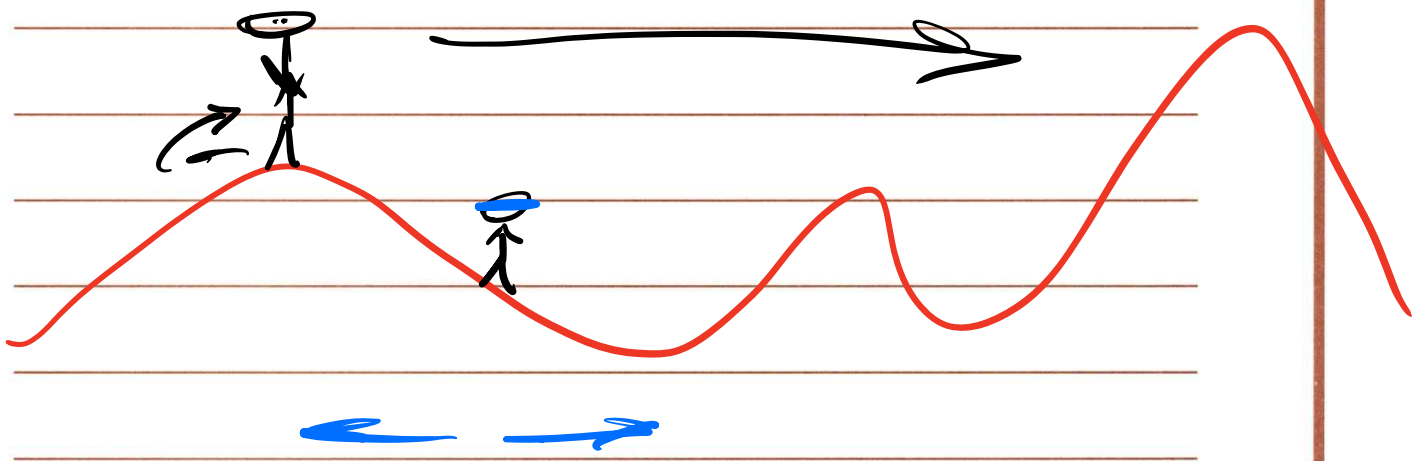


topological orderings of G:

A C D B

~~C A D B~~

takes O(m+n)

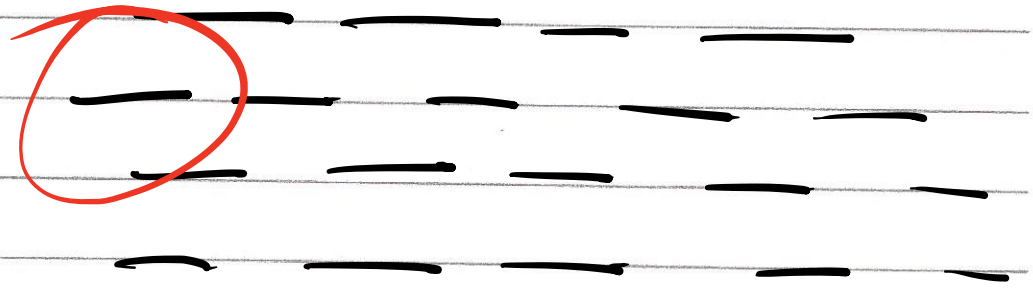


Interval scheduling Problem

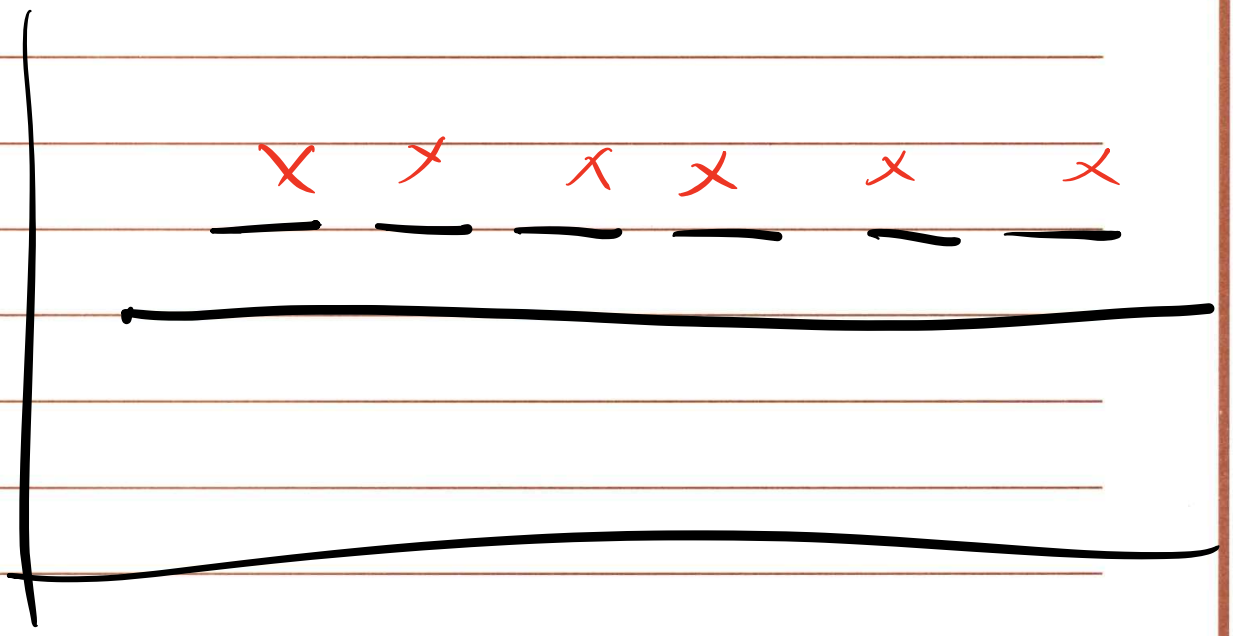
Input: Set of requests {1..n}

i^{th} request starts at s(i) and ends at f(i)

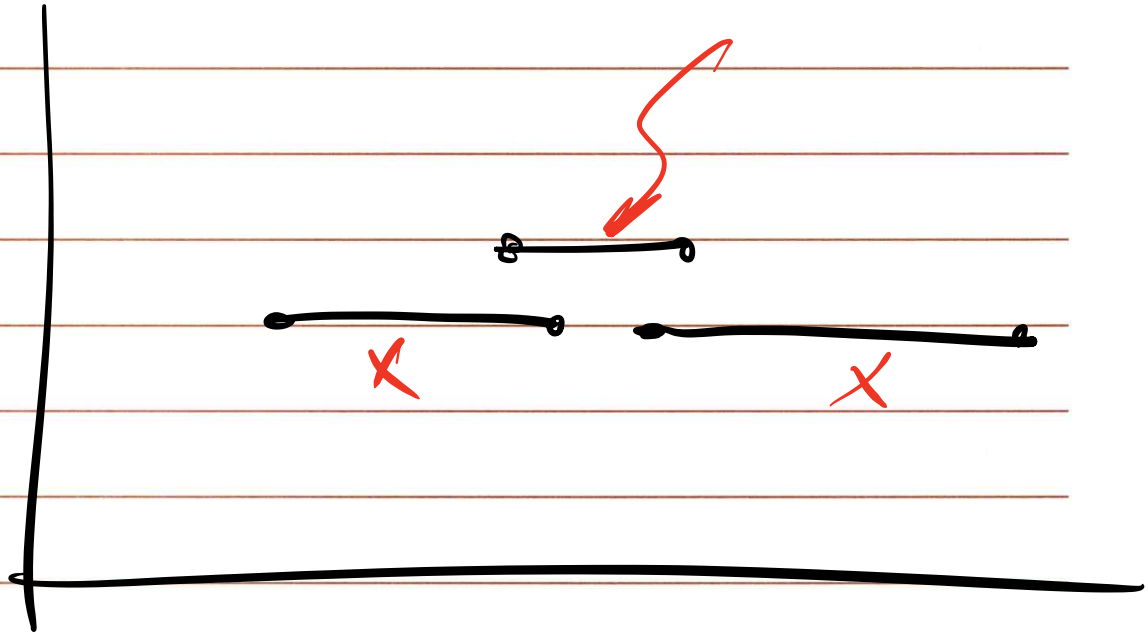
Objective: To find the largest compatible subset of these requests



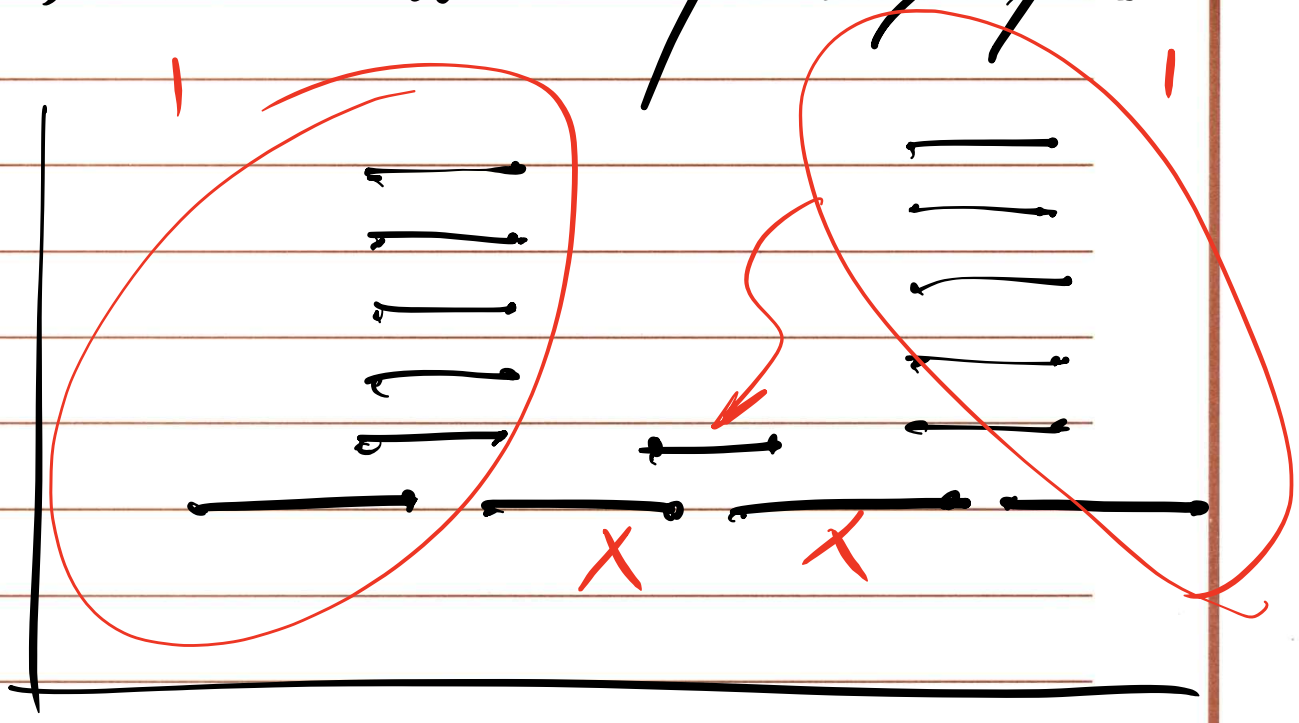
try #1 Earliest start time first.



try #2 Smallest requests first



try #3 Smallest no. of overlaps first.



try #4 Smallest finish time first.

Solution:

Initially R is the complete set of requests
& A is empty

While R is not empty

Choose a request $i \in R$ that has the smallest finish time

Add request i to A

Delete all requests from R that are not compatible w/ i

endwhile

Return A

Proof of Correctness

(1) Show that A is a compatible set

(2) Show that A is an optimal set

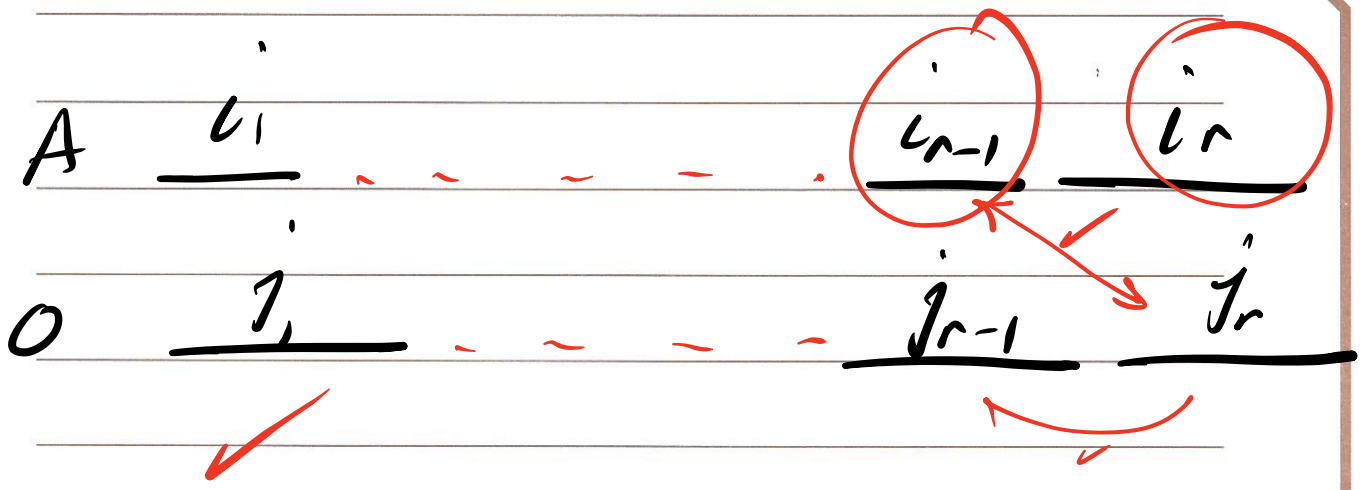
Say A is of size k

Say there is an opt. solution O

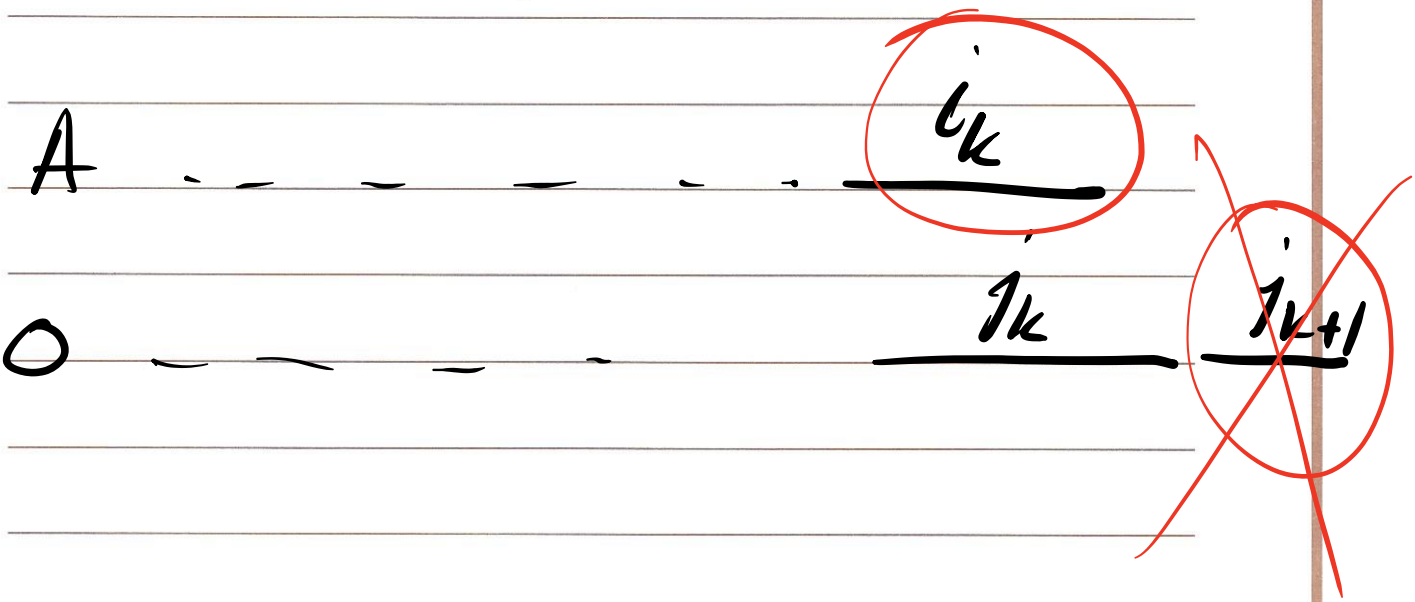
Requests in A: $i_1, \ldots, i_k$

" " O: $j_1, \ldots, j_m$

We will first prove that for all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$



We can then easily prove that $|A| = |O|$

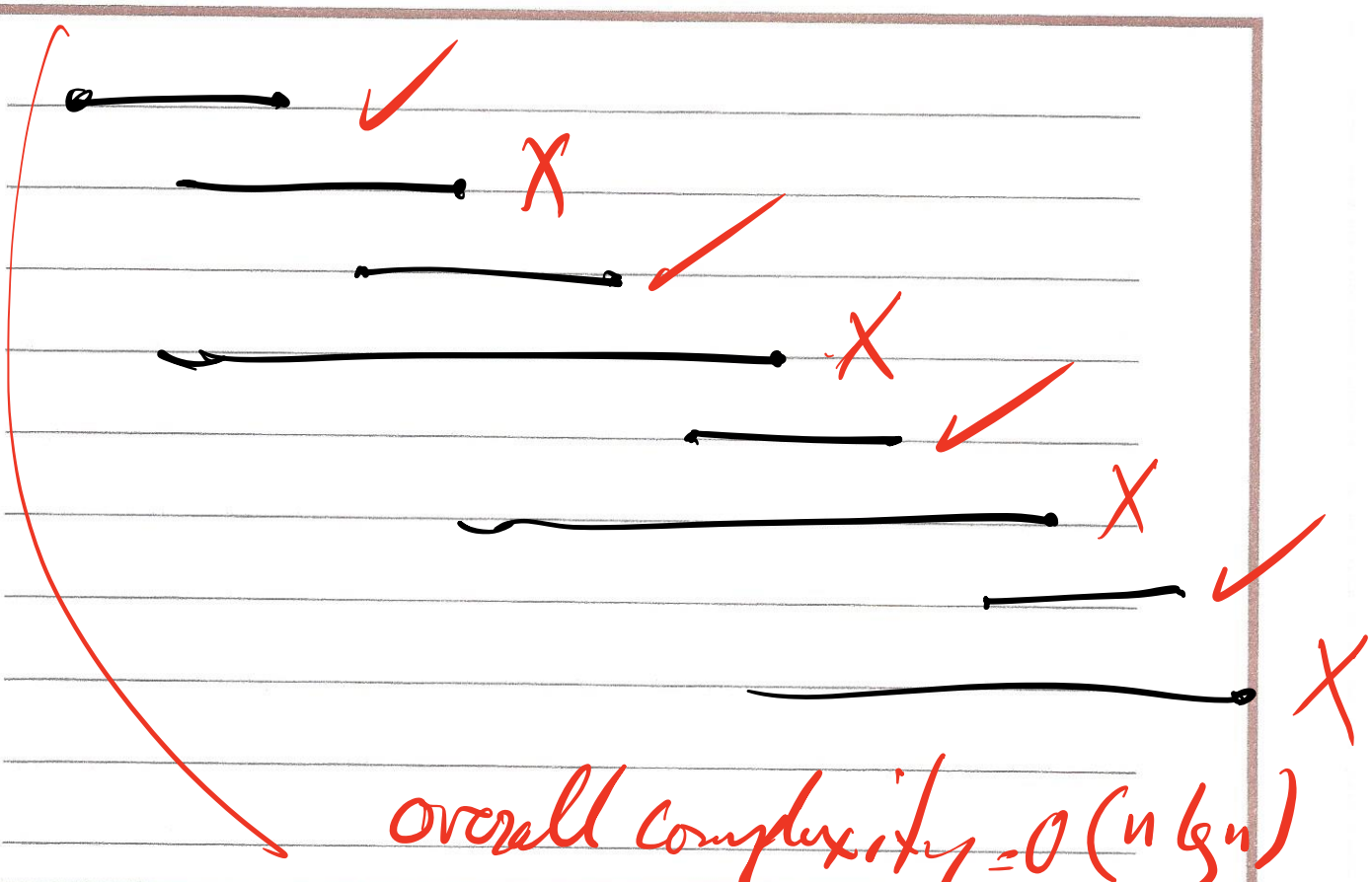


Implementation

- Sort requests in order of finish time and label in this order: $O(n \log n)$

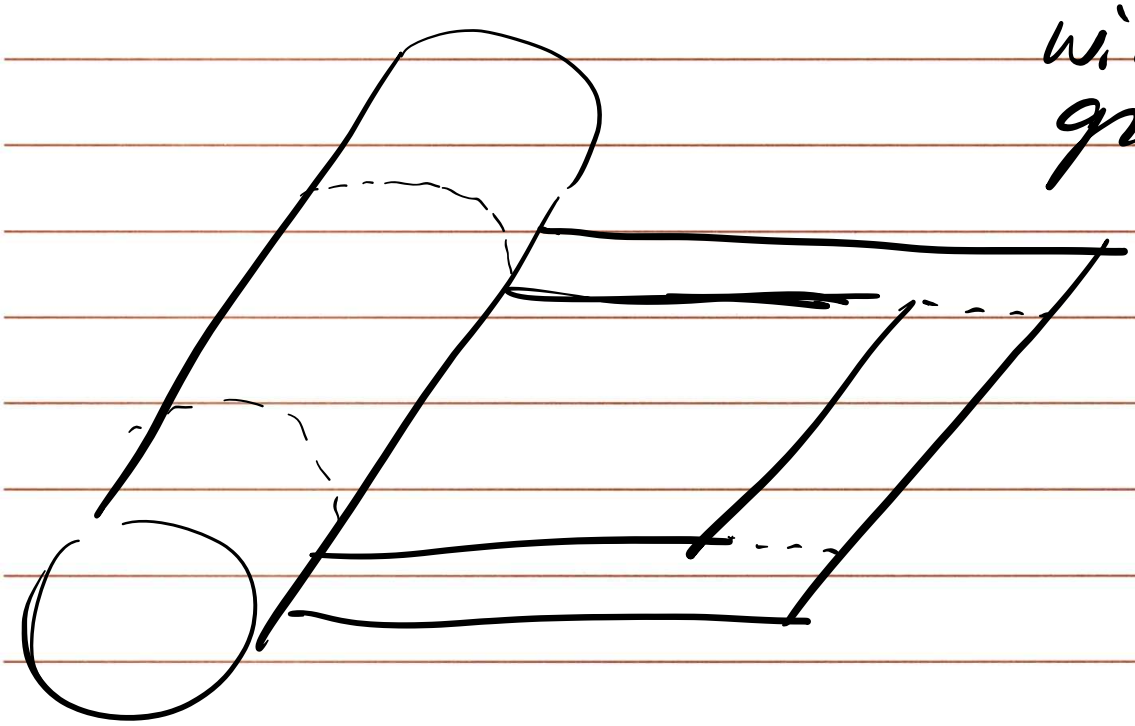
  $f(i) \leq f(j)$ where $i < j$

- Select requests in order of increasing $f(i)$, always selecting the first. Then iterate through the intervals in this order until reaching the first interval for which $s(j) \geq f(i)$ $O(n)$

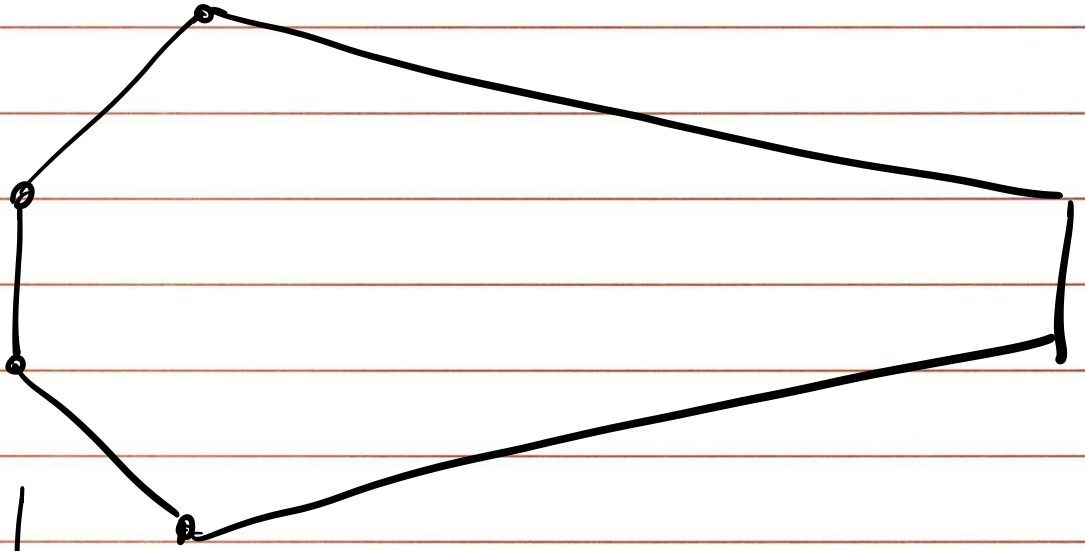


overall complexity $= O(n \log n)$

Orders: Qty
width
grade



Coffin Sched.



warm up.

Fractional Knapsack

Knapsack has a weight capacity of W.

We are given as input, a set of n objects with weight w_i and value v_i.

Objective: Fill up the knapsack to its weight capacity such that the value of items in knapsack is maximized.

Ex. knapsack weight cap: 10

items	1	2	3	4	5
Values	10	20	15	2	8
weights	4	10	5	1	2

Come up with a greedy solution to this problem on your own and solve the above example numerically. Then try to prove that your solution is optimal.

Discussion 2

1. Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if *f(n) = O(g(n))*

$$\log n^n,\ n^2,\ n^{\log n},\ n \log\log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

Solution: First separate functions into logarithmic, polynomial, and exponential
Logarithmic: $\log^2 n$
Exponential: $n^{\log n}$
Polynomial: everything else

So we just need to order the ones that are polynomial now:
Since $2^{\log n} = n$ (assuming log base 2), and $\log n^n = n \log n$, we will have the following order:

$$n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2$$

Now we put the whole list in this order: Logarithmic, polynomial, exponential which gives us:

$$\log^2 n\ ,\ n\ ,\ n \log\log n\ ,\ n \log n\ ,\ n^{\sqrt{2}}\ ,\ n^2\ ,\ n^{\log n}$$

2. Suppose that *f(n)* and *g(n)* are two positive non-decreasing functions such that *f(n) = O(g(n)).* Is it true that $2^{f(n)} = O(2^{g(n)})$?

Solution: Not true. Will not work for f(n) = 2n and g(n) = n

3. Find an upper bound (Big O) on the worst case run time of the following code segment.

```
void bigOh1(int[] L, int n)
   while (n > 0)
      find_max(L, n); //finds the max in L[0…n-1]
      n = n/4;
```

Carefully examine to see if this is a tight upper bound (Big θ)

Solution: The call to find_max takes linear time with respect to n, and we know that the while loop terminates after log n (base 4) iterations. So, it is easy to say that this code runs in O(n log n). But if you look closer and add up the cost of the calling find_max at every iteration, we will get:

1st call:	cn
2nd call:	cn/4
3rd call:	cn/16
….	

The sum of this series adds up to less than 2cn -- without doing any math. (Since we know that cn + cn/2 + cn/4 + cn/8 + … + c adds up to 2cn and the above sum is less than that)

So, O(n log n) is not a tight upper bound. The tight upper bound is $\theta (n)$

4. Find a lower bound (Big Ω) on the best case run time of the following code segment.

```
string bigOh2(int n)
   if(n == 0) return "a";
   string str = bigOh2(n-1);
   return str + str;
```

Carefully examine to see if this is a tight lower bound (Big θ)

Solution: We quickly see that the recursive call bigOh2 calls itself with a problem size one less than the current size. So, the function will call itself exactly n times. So, we can say that the best case run time is $\Omega (n)$. But if we look more carefully, we will see the string concatenation operation will cost a lot more:

n	output string
----	----------
0	a
1	aa
2	aaaa
3	aaaaaaaa
….	
n	2^n

So, $\Omega (n)$ is not a tight lower bound. In fact, the tight lower bound here is exponential $\theta (2^n)$ $(2+4+8+\ldots+2^n = 2^{n+1} = \theta (2^n))$

5. What Mathematicians often keep track of a statistic called their Erdős Number, after the great 20th century mathematician. Paul Erdős himself has a number of zero. Anyone who wrote

a mathematical paper with him has a number of one, anyone who wrote a paper with someone who wrote a paper with him has a number of two, and so forth and so on. Supposing that we have a database of all mathematical papers ever written along with their authors:

a. Explain how to represent this data as a graph.
b. Explain how we would compute the Erdős number for a particular researcher.
c. Explain how we would determine all researchers with Erdős number at most two.

Solution:

a) Create a graph where nodes represent mathematicians and edges represent co-authorship.
b) Run BFS in that graph starting from Erdős and see at which level that particular mathematician appears in the BFS tree. That will be his/her Erdős number.
c) All mathematicians within the first 2 levels of the BFS tree (staring from Erdős) have Erdős numbers of at most two.

6. In class, we discussed finding the shortest path between two vertices in a graph. Suppose instead we are interested in finding the *longest* simple path in a directed acyclic graph. In particular, I am interested in finding a path (if there is one) that visits all vertices.
Given a DAG, give a linear-time algorithm to determine if there is a simple path that visits all vertices.

Solution: Find a topological order (See textbook for details on finding topological ordering in DAGs). See if there is a path that goes through the nodes of the graph in that topological order. If there is, then this will be the longest path we are looking for. In fact, if this path exists, it gives us a strict precedence order from the starting node to the end node on the path, and therefore the graph can only have one topological ordering. [Any other ordering of the nodes in the graph will violate this precedence order because if we move the position of any node in that order, we will end up with at least one edge that goes in the opposite direction of the topological order.] So, if our topological order does not provide such a path, then we can conclude that such a path does not exist in the graph.

CSCI 570 - Fall 2021 - HW 2 Solution

Due September 9th

1 Graded Problems

1. What is the worst-case runtime performance of the procedure below?

```
c = 0
i = n
while i > 1 do
    for j = 1 to i do
        c = c + 1
    end for
    i = floor(i/2)
end while
return c
```

Solution:

There are i operations in the for loop and the while loop terminates when i becomes 1. The total time is

$$n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \cdots \leq (1 + 1/2 + 1/4 + \cdots) \cdot n \leq 2n = O(n).$$

Rubric (5 pts):

- 2 pts: if bound correctly found as $O(n)$
- 3 pts: Provides a correct explanation of the runtime
- 2 pts total if bound given is $O(n \log n)$ (i.e., not a tight upper bound)

2. Arrange these functions under the O notation using only $=$ (equivalent) or $\subset$ (strict subset of):

   (a) $2^{\log n}$

   (b) 2^{3n}

   (c) $n^{n \log n}$

   (d) $\log n$

   (e) $n \log \left(n^2\right)$

   (f) n^{n^2}

   (g) $\log(\log(n^n))$

   E.g. for the function n, $n+1$, n^2, the answer should be

   $$O(n+1) = O(n) \subset O(n^2).$$

   Solution:

   First separate functions into logarithmic, polynomial, and exponential. Note that

   $$2^{\log n} = n, \quad n^{n \log n} = 2^{n(\log n)^2}, \quad n^{n^2} = 2^{n^2 \log n},$$

   we have:

   (a) Logarithmic: $\log n$, $\log(\log(n^n))$

   (b) Polynomial: $2^{\log n}$, $n \log(n^2)$

   (c) Exponential: 2^{3n}, $n^{n \log n}$, n^{n^2}

   - Since
     $$\log n \leq 1 \cdot \log(n \log n) = \log(\log(n^n)),$$
     so $\log n = O(\log(\log(n^n)))$. On the other hand,
     $$\log(\log(n^n)) = \log(n \log n) \leq \log(n^2) = 2 \cdot \log n,$$
     so $\log(\log(n^n)) = O(\log n)$. Thus
     $$O(\log n) = O(\log(\log(n^n))).$$

   - Since every logarithmic grows slower than every polynomial,
     $$O(\log(\log(n^n))) \subset O(2^{\log n}).$$

   - $2^{\log n} = O(n) \subset O(n \log n) = O(2 \cdot n \log n) = O(n \log n^2)$. Thus
     $$O(2^{\log n}) \subset O(n \log(n^2)).$$

- Since every exponential grows faster than every polynomial,

$$O(n\log(n^2)) \subset O(2^{3n}).$$

- Since

$$O(3n) \subset O(n(\log n)^2) \subset O(n^2 \log n),$$

so

$$O(2^{3n}) \subset O(2^{n(\log n)^2}) = O(n^{n\log n}) \subset O(2^{n^2\log n}) = O(n^{n^2}).$$

Therefore,

$$O(\log n) = O(\log(\log(n^n))) \subset O(2^{\log n}) \subset O(n\log(n^2)) \subset O(2^{3n}) \subset O(n^{n\log n}) \subset O(n^{n^2})$$

Rubric (10 pts):

- -2 pts for each "inversion" in the order
  For example, $O(2^{3n}) \subset O(n^{n^2}) \subset O(n^{n\log n})$ has one inversion and $O(n^{n\log n}) \subset O(n^{n^2}) \subset O(2^{3n})$ has two inversions
- -2 pts for any $=$ mistaken to be $\subset$ or vice versa

3. Given functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

   (a) $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$

   (b) $f_1(n) + f_2(n) = O\left(\max\left(g_1(n), g_2(n)\right)\right)$

   (c) $f_1(n)^2 = O\left(g_1(n)^2\right)$

   (d) $\log_2 f_1(n) = O\left(\log_2 g_1(n)\right)$

Solution:

By definition, there exist $c_1, c_2 > 0$ such that

$$f_1(n) \leq c_1 \cdot g_1(n) \text{ and } f_2(n) \leq c_2 \cdot g_2(n)$$

for n sufficiently large.

(a) True.

$$f_1(n) \cdot f_2(n) \leq c_1 \cdot g_1(n) \cdot c_2 \cdot g_2(n) = (c_1 c_2) \cdot (g_1(n) \cdot g_2(n)).$$

(b) True.

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq (c_1 + c_2)(g_1(n) + g_2(n)) \\ &\leq 2 \cdot (c_1 + c_2) \max(g_1(n), g_2(n)). \end{aligned}$$

(c) True.

$$f_1(n)^2 \leq (c_1 \cdot g_1(n))^2 = c_1^2 \cdot g_1(n)^2.$$

(d) False. Consider $f_1(n) = 2$ and $g_1(n) = 1$. Then

$$\log_2 f_1(n) = 1 \neq O(\log_2 g_1(n)) = O(0).$$

Rubric (4 pts for each subproblem):

- 1 pts: Correct T/F claim
- 3 pts: Provides a correct explanation or counterexample

4. Given an undirected graph G with n nodes and m edges, design an $O(m+n)$ algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one.

   Solution:

   Without loss of generality assume that G is connected. Otherwise, we can compute the connected components in $O(m+n)$ time and deploy the below algorithm on each component.

   Starting from an arbitrary vertex s, run BFS to obtain a BFS tree T, which takes $O(m+n)$ time. If $G = T$, then G is a tree and has no cycles. Otherwise, G has a cycle and there exists an edge $e = (u, v) \in G \setminus T$. Let w be the least common ancestor of u and v. There exist a unique path T_1 in T from u to w and a unique path T_2 in T from w to v. Both T_1 and T_2 can be found in $O(m)$ time. Output the cycle e by concatenating P_1 and P_2.

Rubric (15 pts):

- No penalty for not mentioning disconnected case.
- 7 pts: for detecting whether G contains a cycle
- 5 pts: for finding (the edges in) a cycle if G contains one
- 3 pts: describing that the runtime is $O(m+n)$ in each step (and thus total)

2 Practice Problems

1. Solve Kleinberg and Tardos, **Chapter 2, Exercise 6.**

   Solution:

   (a) The outer loop runs for exactly n iterations, the inner loop runs for at most n iterations, and the number of operations needed for adding up array entries $A[i]$ through $A[j]$ is $i+j-1 = O(n)$. Therefore, the running time is in $n^2 \cdot O(n) = O(n^3)$.

   (b) Consider those iterations that require at least $n/2$ operations to add up array entries $A[i]$ through $A[j]$. When $i \leq n/4$ and $j \geq 3n/4$, the number of operations needed is at least $n/2$. So there are at least $(n/4)^2$ pairs of (i, j) such that adding up $A[i]$ through $A[j]$ requires at least $n/2$ operation. Therefore, the running time is at least $\Omega((n/4)^2 \cdot n/2) = \Omega(n^3/32) = \Omega(n^3)$.

   (c) Consider the following algorithm:

   for $i = 1, 2, \ldots, n-1$ **do**
   $B[i, i+1] \leftarrow A[i] + A[i+1]$
   end for
   for $j = 2, 3, \ldots, n-1$ **do**
   for $i = 1, 2, \ldots, n-j$ **do**
   $B[i, i+j] \leftarrow B[i, i+j-1] + A[i+j]$
   end for
   end for

   It first computes $B[i, i+1]$ for all i by summing $A[i]$ with $A[j]$. This for loop requires $O(n)$ operations. For each j, it computes all $B[i, i+j]$ by summing $B[i, i+j-1]$ with $A[i+j]$. This works since the value $B[i, i+j-1]$ were already computed in the previous iteration. The double for loop requires $O(n) \cdot O(n) = O(n^2)$ time. Therefore, the algorithm runs in $O(n^2)$.

2. Solve Kleinberg and Tardos, **Chapter 3, Exercise 6.**

   Solution:

   Proof by Contradiction: assume there is an edge $e = (x, y)$ in G that does not belong to T. Since T is a DFS tree, one of x or y is the ancestor of the other. On the other hand, since T is a BFS tree, x and y differs by at most 1 layer. Now since one of x and y is the ancestor of the other, x and y should differ by exactly 1 layer. Therefore, the edge $e = (x, y)$ should be in the BFS tree T. This contradicts the assumption. Therefore, G cannot contain any edges that do not belong to T.

Scheduling to Minimize Lateness

Scheduling to Minimize Lateness

- Requests can be scheduled at any time
- Each request has a deadline
- Notation: $L_i = f(i) - d_i$

  L_i is called lateness for request i.

Goal: Minimize the Maximum Lateness $L = \max_i L_i$

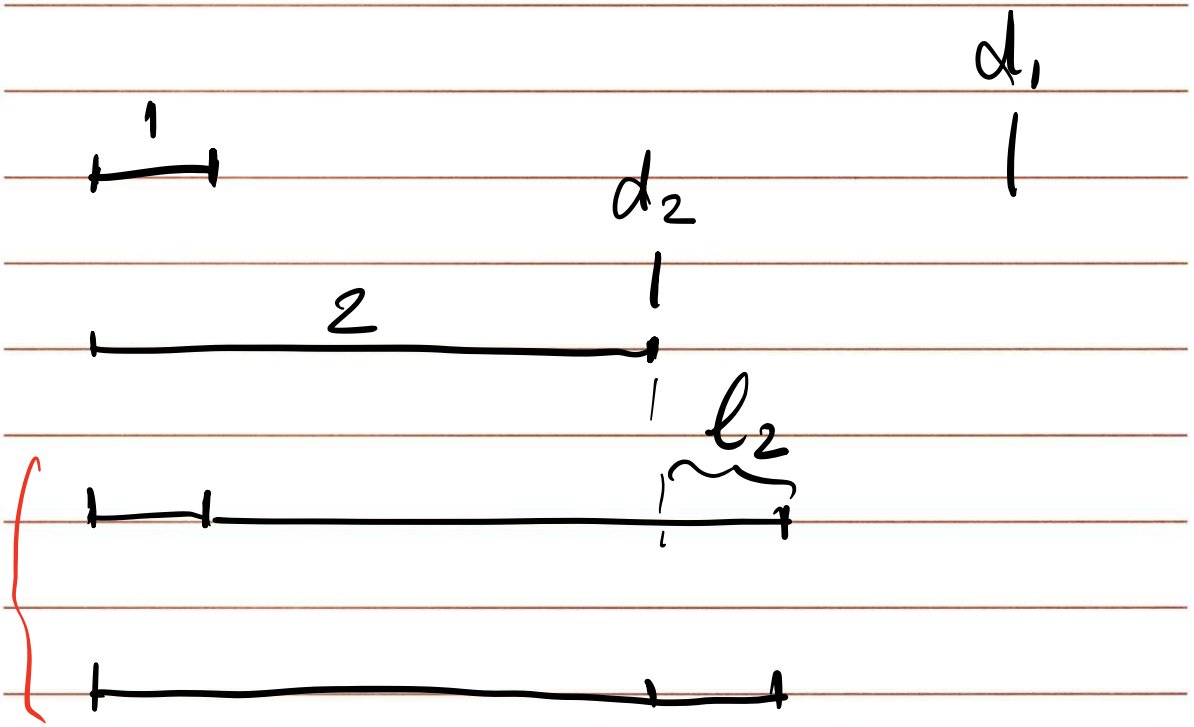
Sol 1.

job 1 late by 5 hrs
" 2 " " 6 "

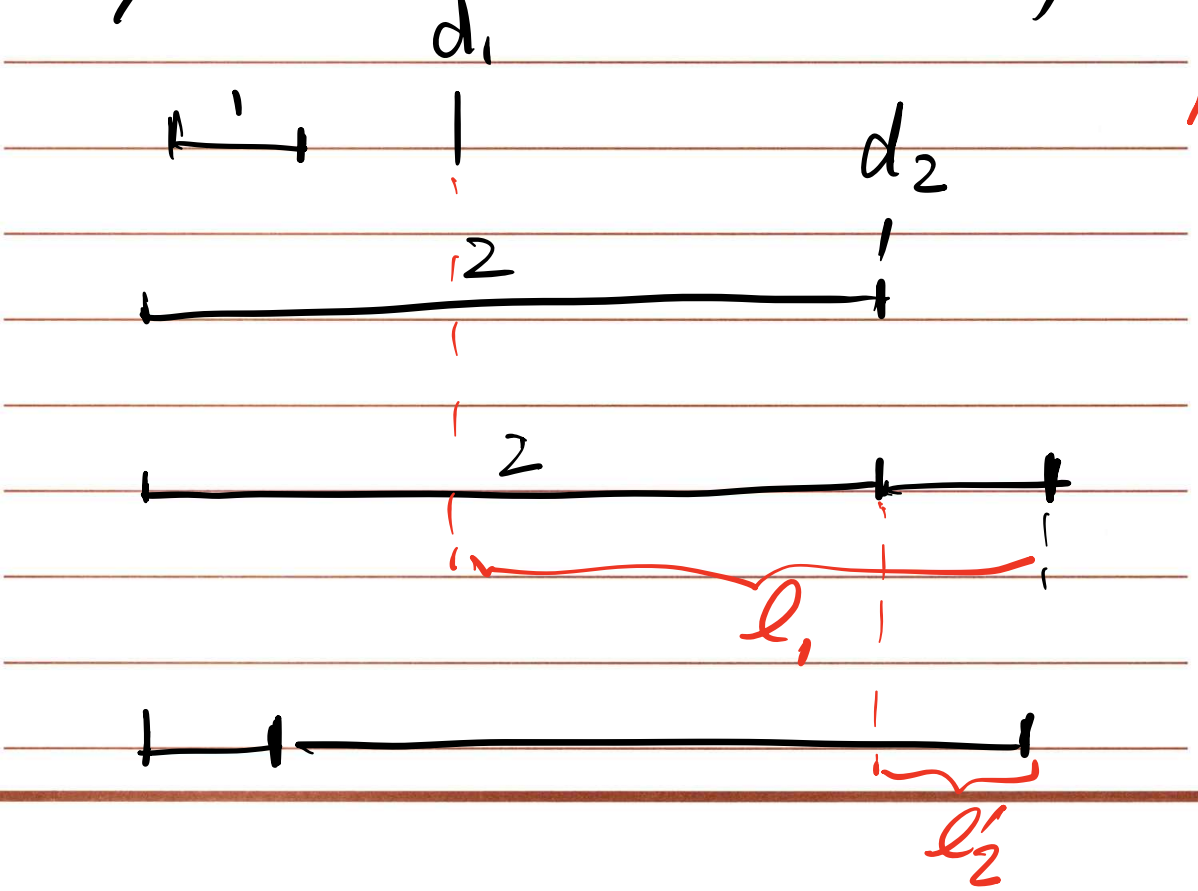
Sol 2.

job 1 late by 0 hrs
job 2 " " 7 "

try #1 Schedule shortest req's first



try #2 Shortest slack time first

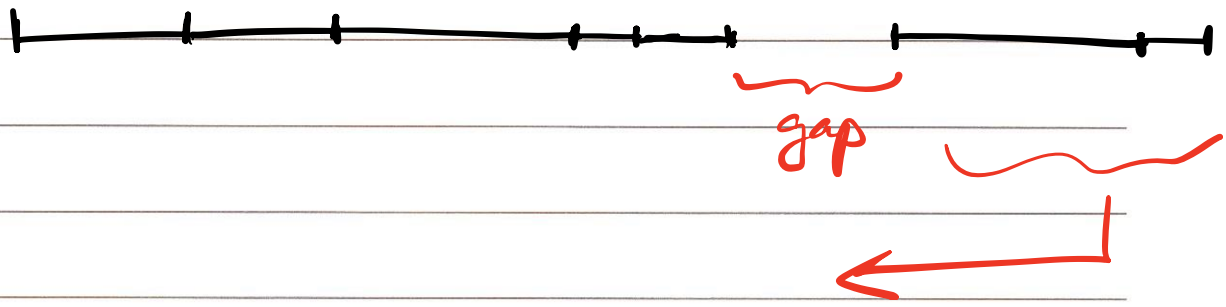


Solution:

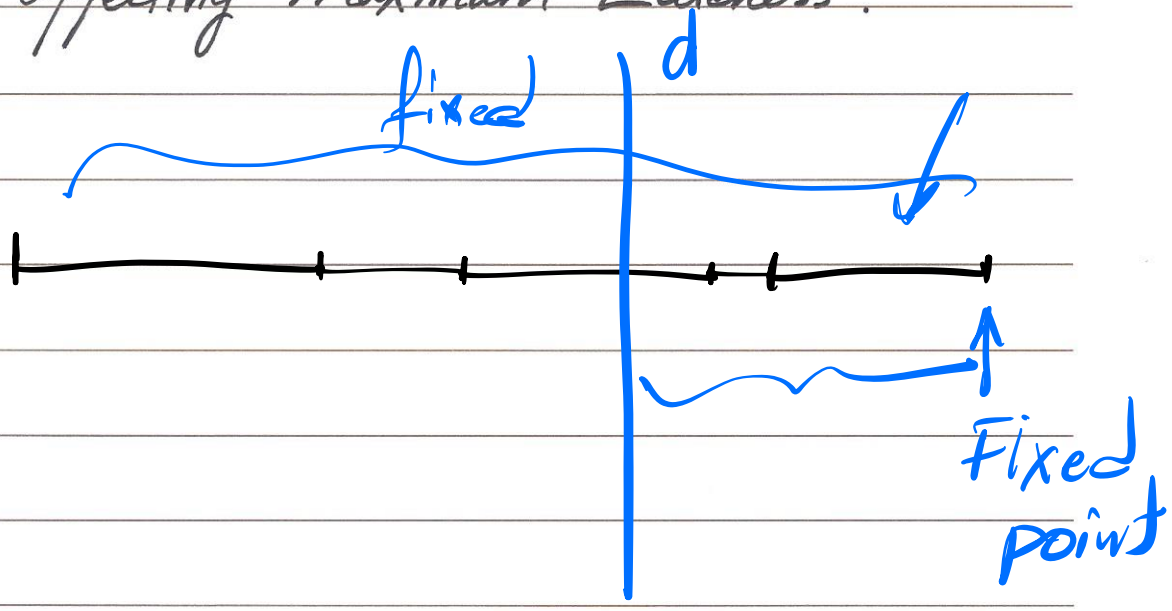
Schedule jobs in order of their deadline without any gaps between jobs.

Proof of Correctness

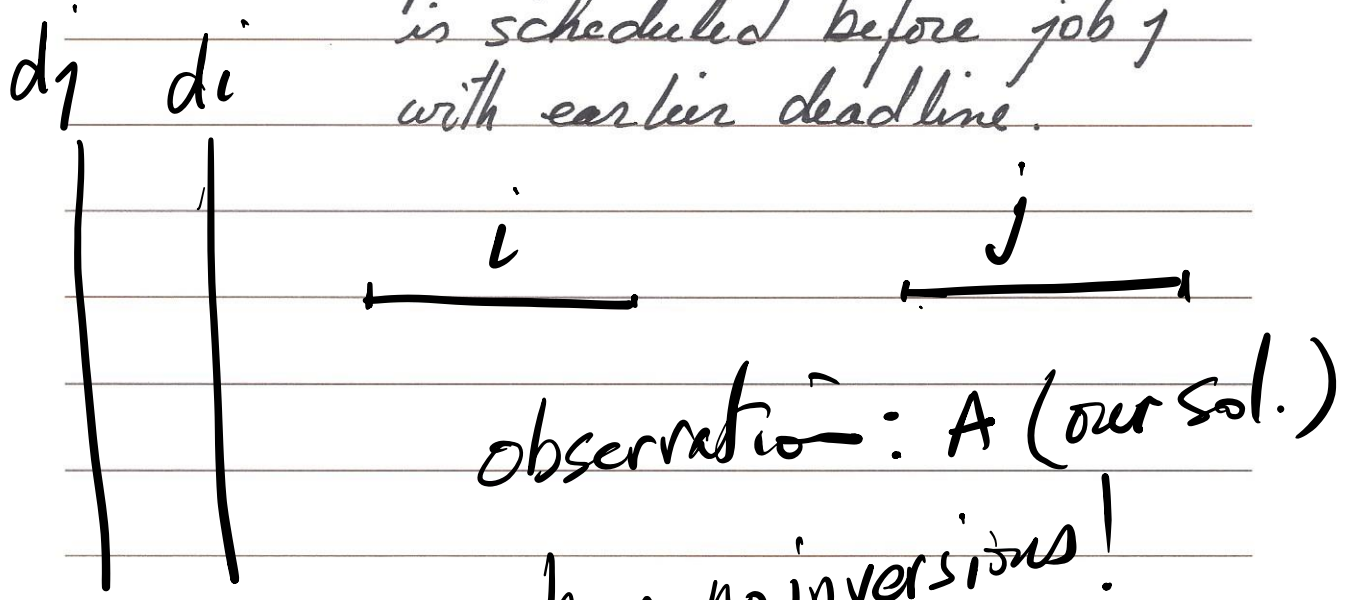
① There is an optimal solution with no gaps.



② Jobs with identical deadlines can be scheduled in any order without affecting Maximum Lateness.



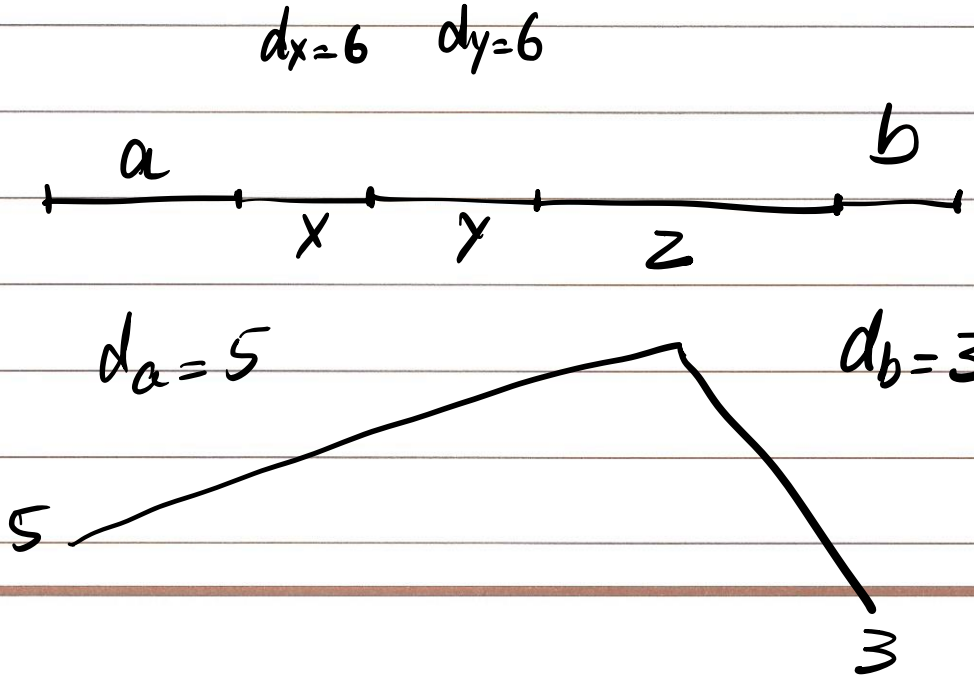
③ Def. Schedule A' has an inversion if a job i with deadline d_i is scheduled before job j with earlier deadline.

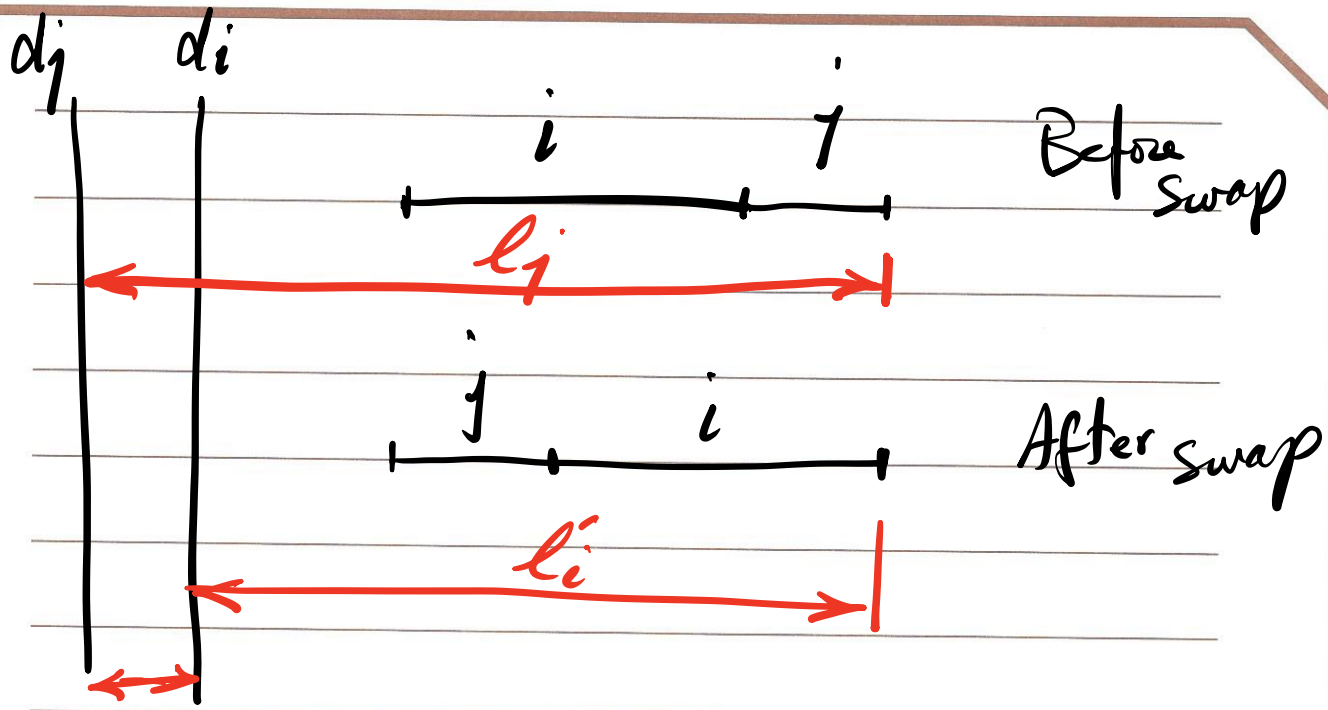


observation: A (our sol.) has no inversions!

(4) All schedules with no inversions and no idle time have the same Maximum Lateness.

(5) There is an optimal schedule that has no inversions and no idle time.





So, if there is an optimal solution that has inversions, we can eliminate the inversions one by one as shown above until there are no more inversions. This solution will also be optimal.

(6) Proved that there exists an optimal schedule with no inversions and no idle time.

Also proved that all schedules with no inversions and no idle time have the same Maximum Lateness.

Our greedy algorithm produces one such solution ⇒ It will be optimal

Priority Queues

A priority queue has to perform these two operations fast!

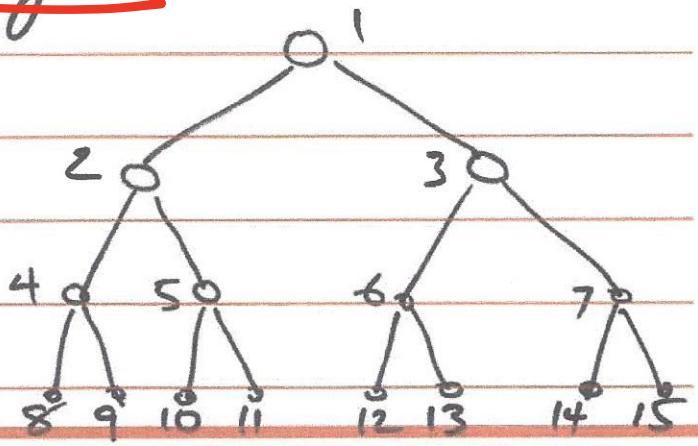
→ 1- Insert an element into the set

→ 2- Find the smallest element in the set

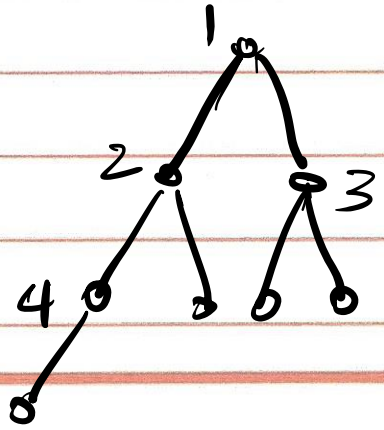
	insert	FindMin
Array implementation	$O(1)$	$O(n)$
Sorted " "	$O(n)$	$O(1)$
linked list "	$O(1)$	$O(n)$
sorted linked list "	$O(n)$	$O(1)$

Background

Def. A binary tree of depth k which has exactly $2^k - 1$ nodes is called a full binary tree.



Def. A binary tree with n nodes and of depth k is complete iff its nodes correspond to the nodes which are numbered 1 to n in the full binary tree of depth k.



Traversing a complete binary tree stored as an array

Parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
if $i = 1$, i is the root

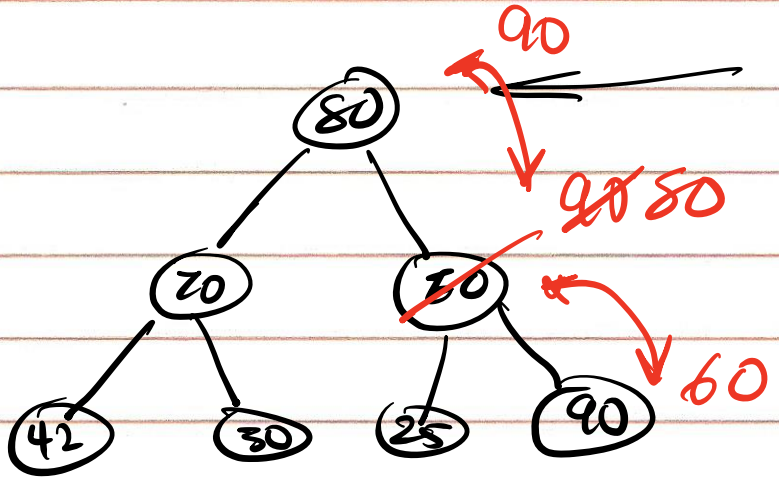
Lchild(i) is at $2i$ if $2i \leq n$
otherwise it has no left child

Rchild(i) is at $2i+1$ if $2i+1 \leq n$
otherwise it has no right child

Def. A binary heap is a complete binary tree with the property that the value (of the key) at each node is at least as large as the values at its children (Max heap)

Find Max

takes O(1)



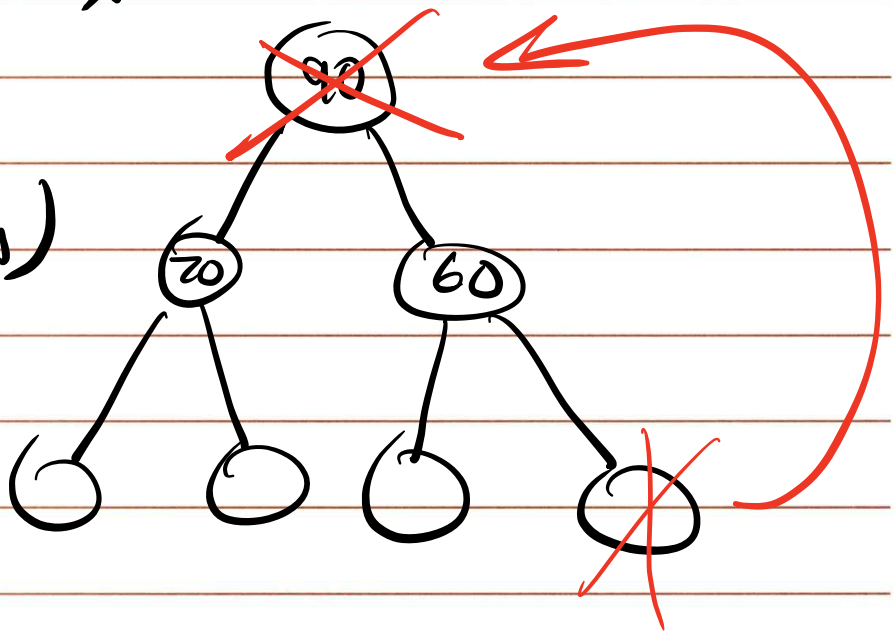
insert

insert 90

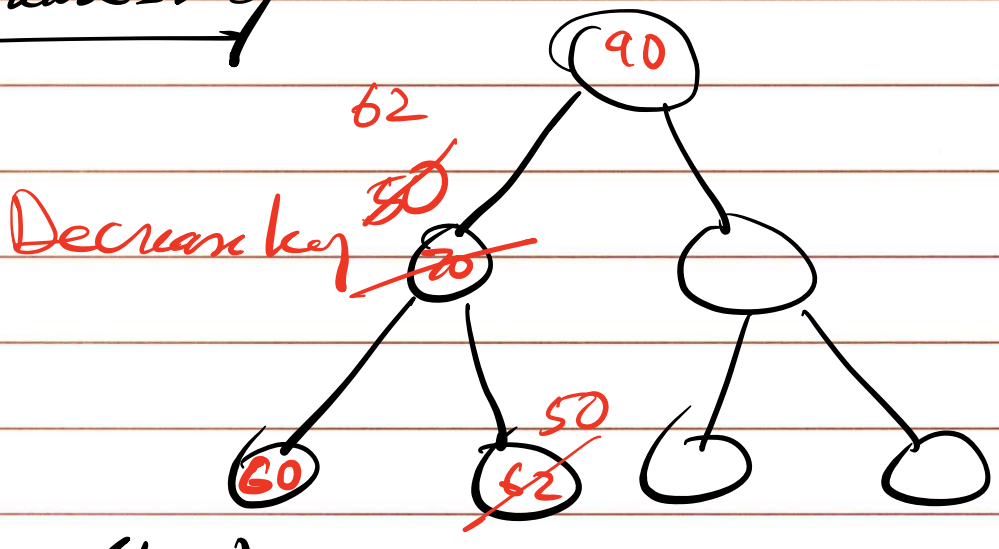
takes O(lg n)

Extract_Max

takes O(lgn)

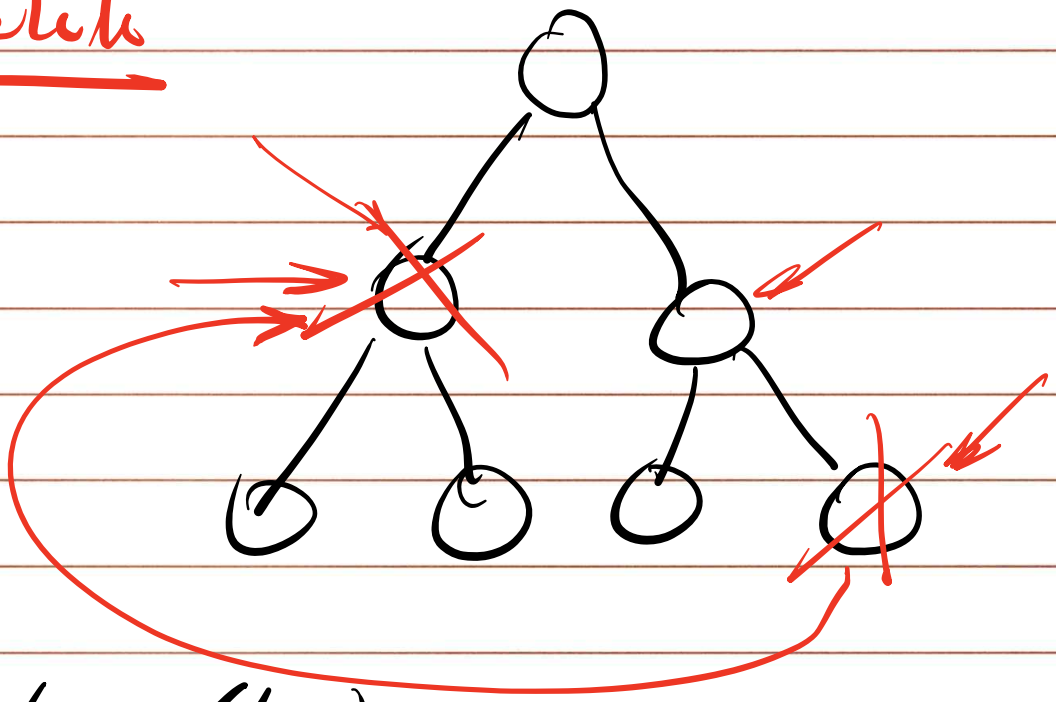


Decrease_key



takes O(lgn)

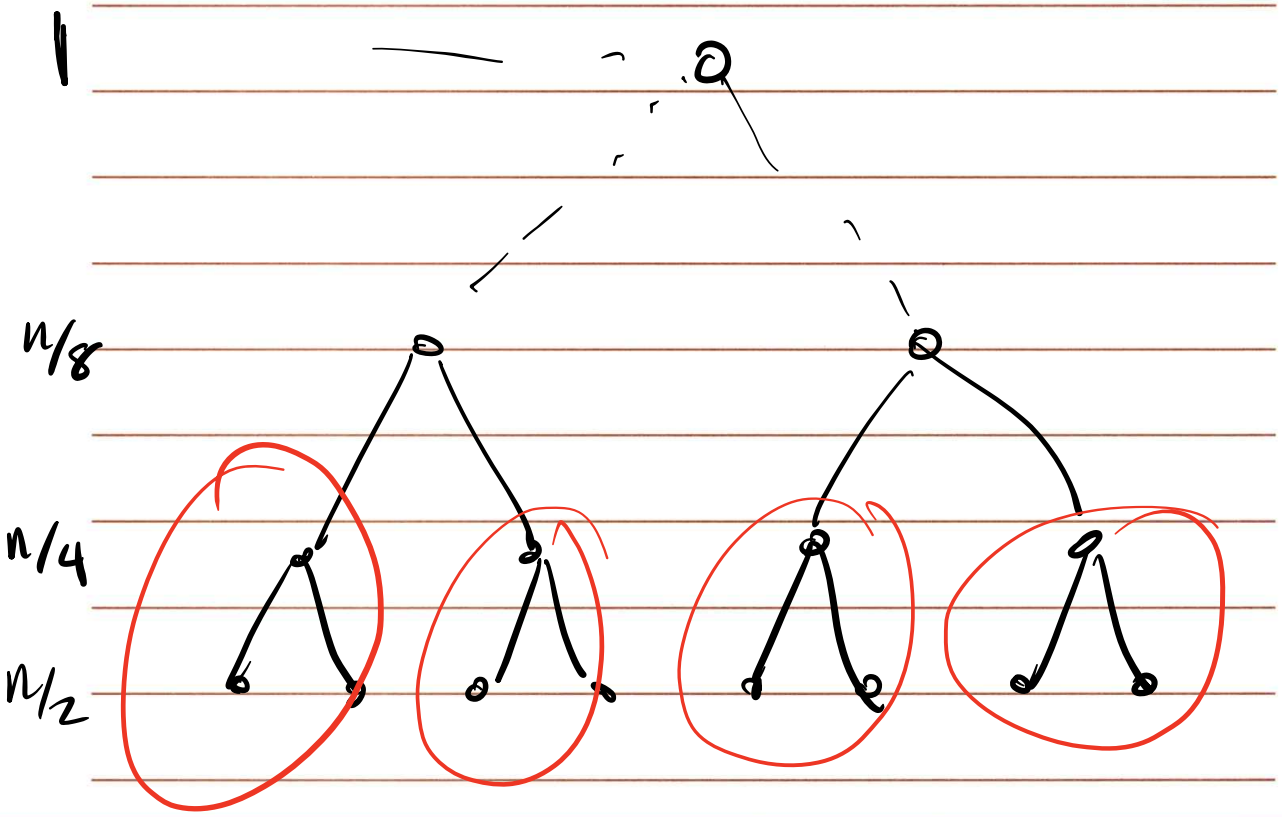
Delete



takes $O(\lg n)$

Construction of a binary heap

takes $O(n \lg n)$ Using n insert op's.



$n/4 * 1$

$n/8 * 2$

$n/16 * 3$

$\vdots$

$1 * kn$

$$T = n/4 * 1 + n/8 * 2 + n/16 * 3 + \cdots$$

$$T/2 = n/8 * 1 + n/16 * 2 + n/32 * 3 + \cdots$$

$$T - T/2 = \underbrace{n/4 + n/8 + n/16 + \cdots}_{n/2}$$

$T/2 = n/2 \longrightarrow \boxed{T = n}$

construction this way takes $O(n)$

Merge of 2 binary heaps of size $\frac{n}{2}$.

takes linear time using linear time construction of the heap.

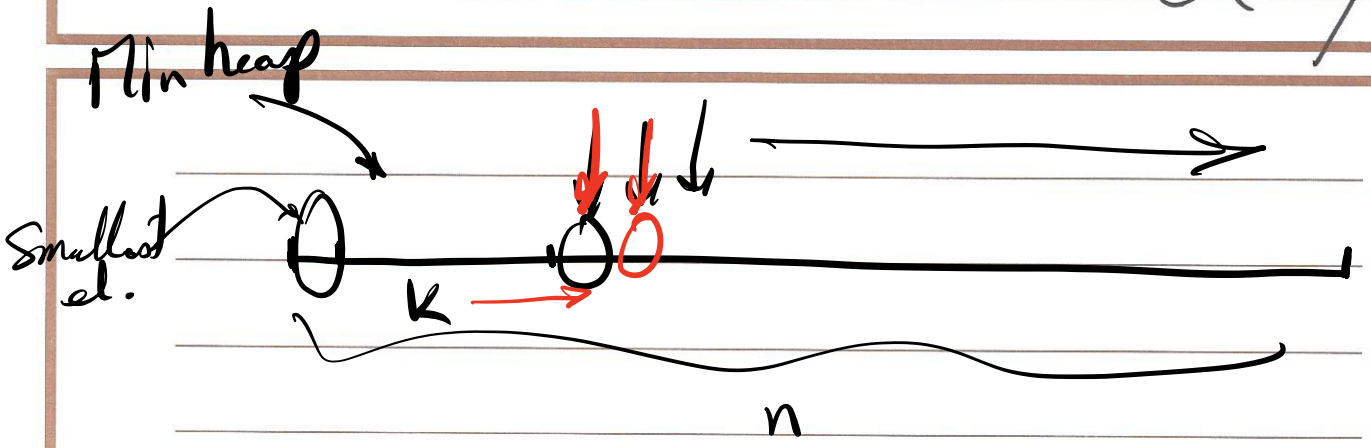
Problem Statement

Input: An unsorted array of length n

Output: Top k values in the array ($k < n$)

Constraints:

- You cannot use any additional memory
- Find an algorithm that runs in time $O(n \lg k)$



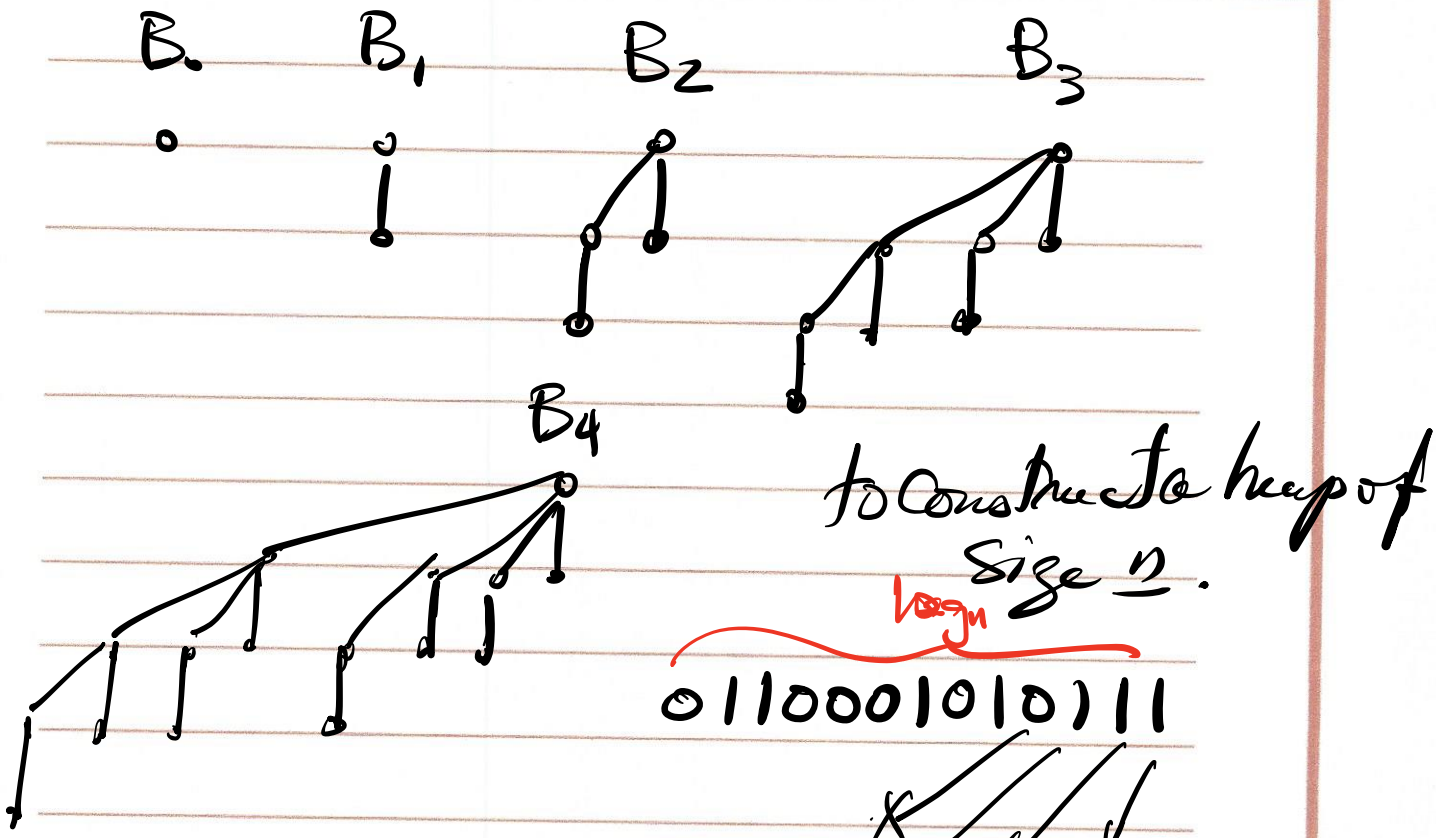
$(n-k) * \lg k$

initially $O(k)$ to construct the min-heap

$$O(k) + O((n-k)\lg k) = O(n \lg k)$$

Def. A binomial tree B_k is an ordered tree defined recursively

- Binomial tree B_0 consists of one node
- Binomial tree B_k consists of 2 binomial trees B_{k-1} that are linked together such that root of one is the leftmost child of the root of the other.



to construct a heap of size n.

$\log n$

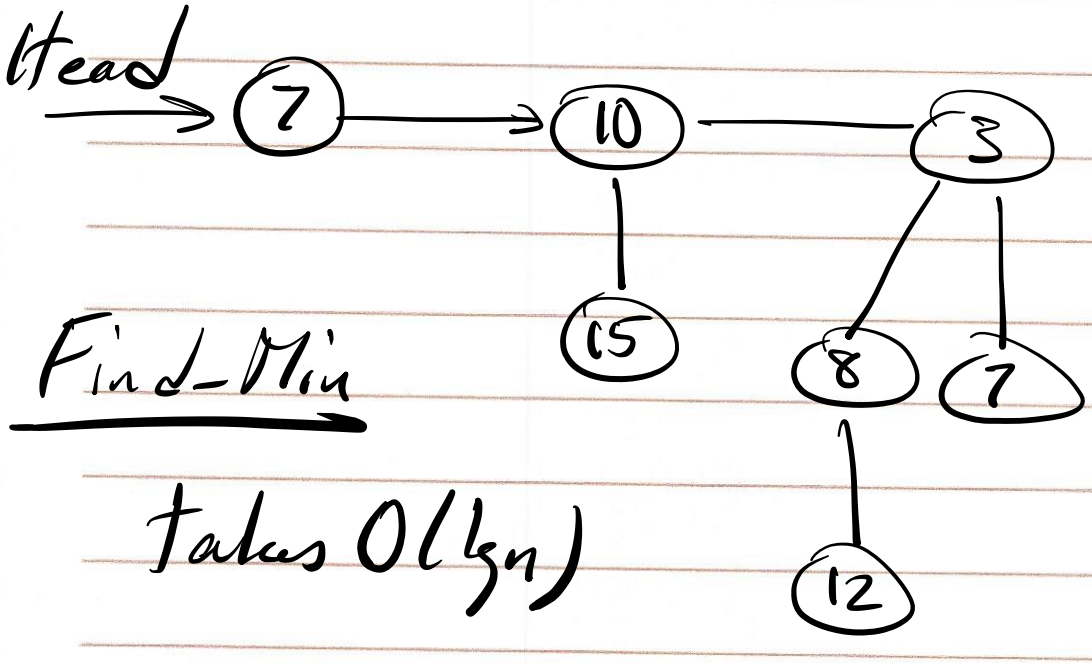
01100010101 11

B_2 B_1 B_0

Def: A binomial Heap H is a set of binomial trees that satisfies the following properties:

1- Each binomial tree in H obeys the min-heap property

2- For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.

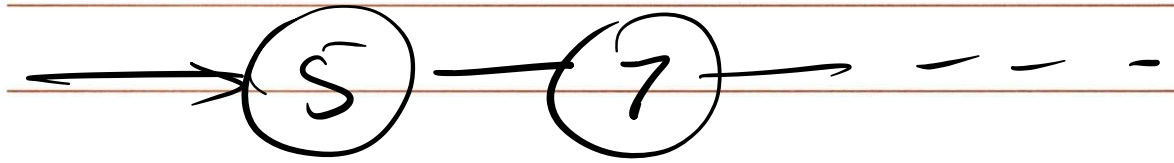


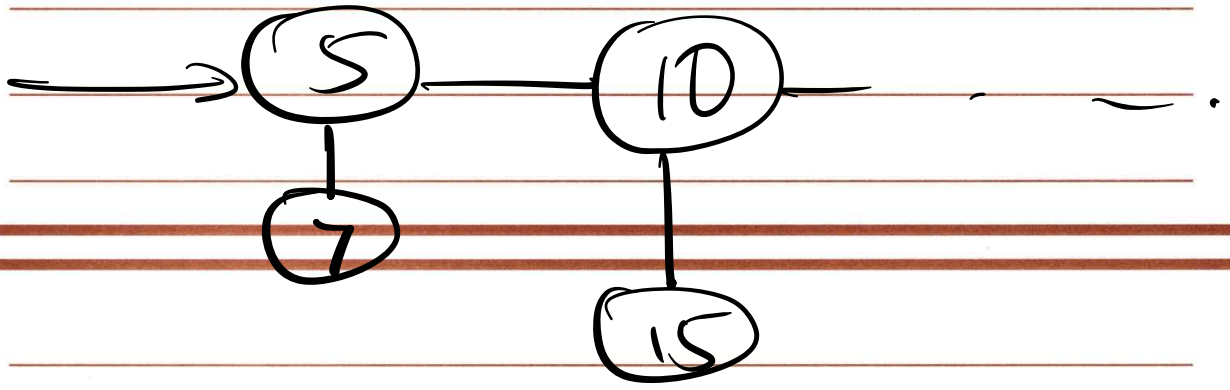
Find_Min

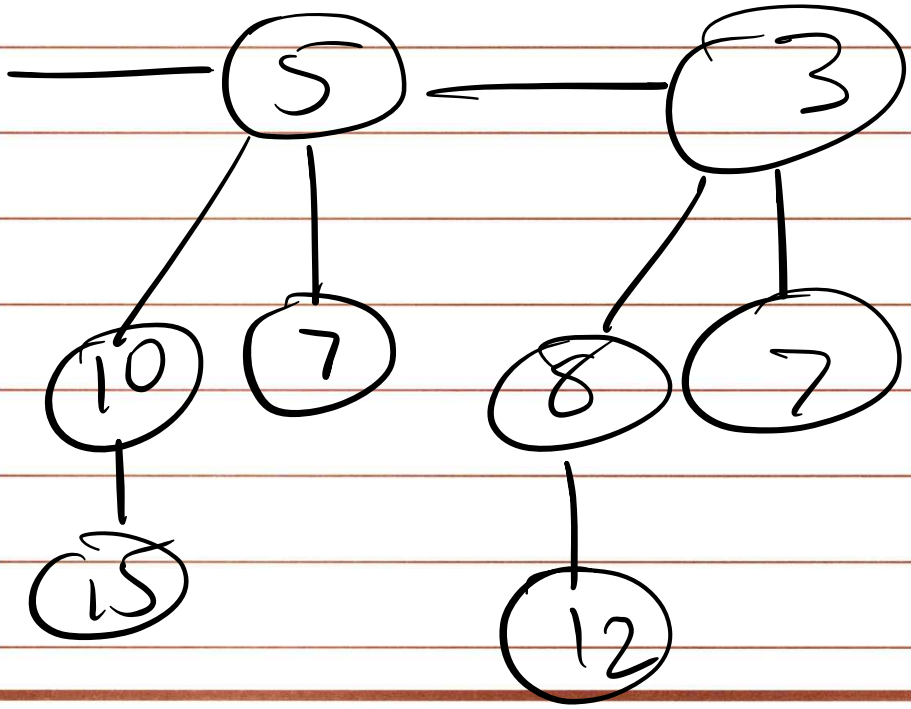
takes $O(\lg n)$

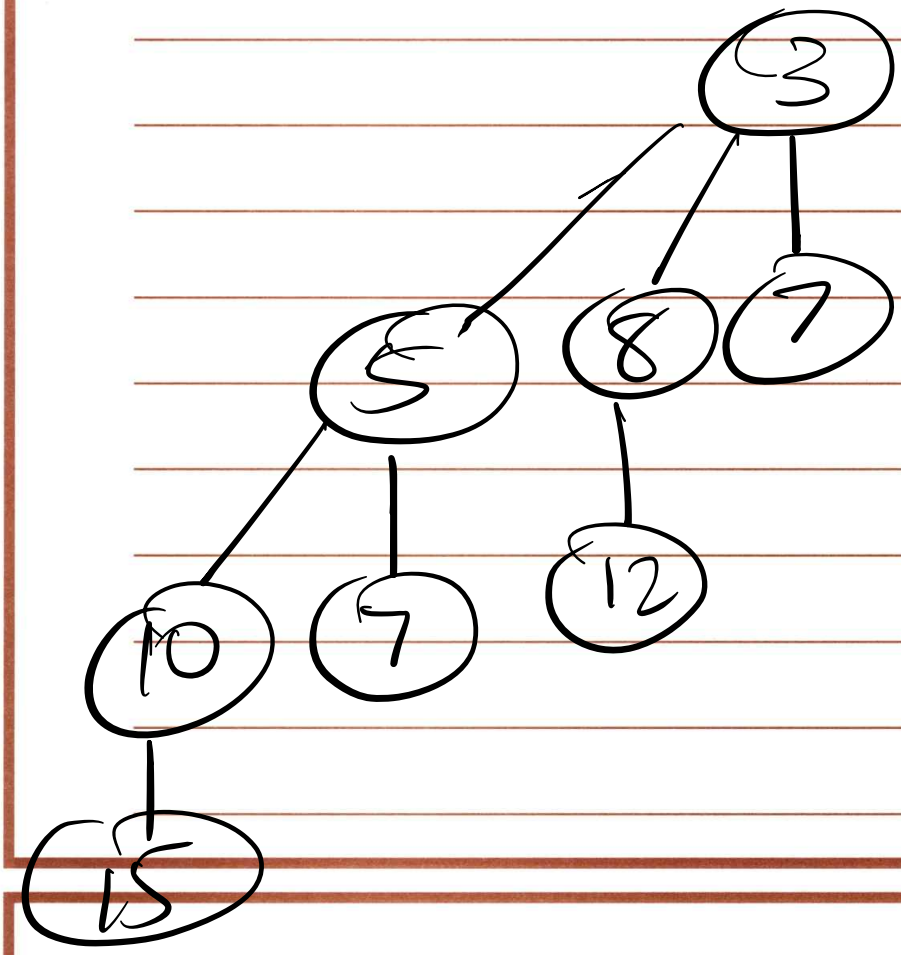
insert → takes O(log n)

insert (5)

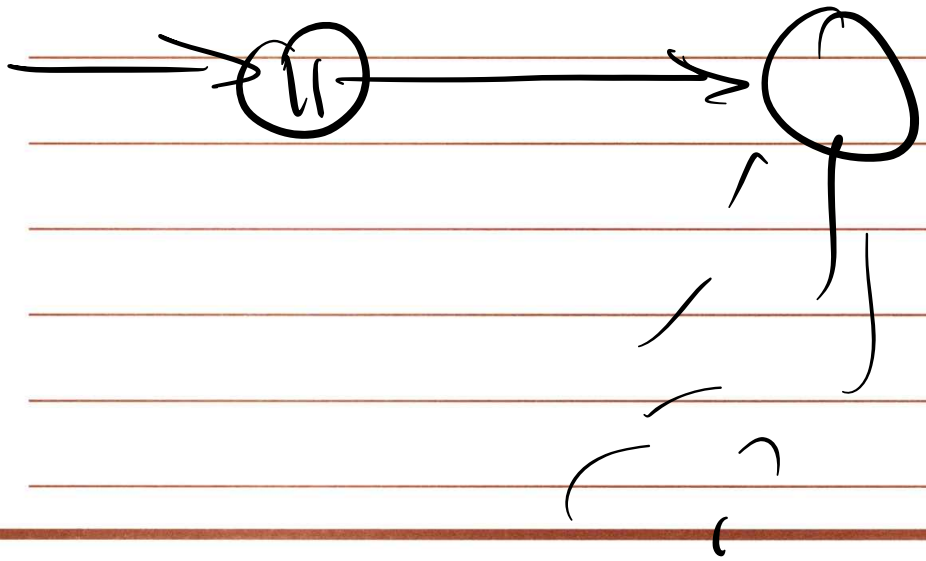








insert 11



Amortized Cost Analysis

Ex. 1

```
for i = 1 to n
    push or Pop
endfor
```

push takes O(1)
Pop takes O(1)

Our worst case run time complexity = O(n)

Ex. 2

```
for i = 1 to n
    push or pop or multipop
endfor
```

O(n)

push takes O(1) worst-case
pop takes O(1) worst-case
multipop takes O(n) worst-case

Our worst case run time complexity = $O(n^2)$

Aggregate Analysis

- We show that a sequence of n operations (for all n) takes worst-case time $T(n)$ total.

- So, in the worst case, the amortized cost (average cost) per operation will be $T(n)/n$

observation: Multipop takes $O(n)$ time if there are n elements pushed on the stack

A sequence of n pushes takes $O(n)$
multipop takes $O(n)$ worst-case

$T(n) = O(n)$
when amortized over n operations,
average cost of an operation $= O(n)/n = O(1)$

Ex. 2

```
for i = 1 to n
    push or pop or multipop
end for
```

push takes O(1) amortized

pop " O(1) "

multipop " O(1) "

our worst case runtime complexity = O(n)

Accounting Method

- We assign different charges (amortized costs) to different operations

- If the charge for an operation exceeds its actual cost, the excess is stored as credit.

- The credit can later helps pay for operations whose actual cost is higher than their amortized cost.

- Total credit at any time = total amortized cost − total actual cost

Total
- ^Credit can never be negative.

Ex. 2 for $i = 1$ to n

push or pop or multipop

endfor

try # 1

assign a ~~cost~~ charge of 1 to each operation

op.	charge	Actual cost	total credit
push	1	1	0
push	1	1	0
multipop	1	2	-1

try # 2

assign charges as follows:

push 2 → O(1)

pop 0 → O(1)

multipop 0 → O(1)

op.	charge	Actual cost	total credit
push	2	1	1
push	2	1	2
multipop	0	2	0

Ex. 2

```
for i = to n
    push or pop or multipop
end for
```

	Amortized cost
push	O(1)
pop	O(1)
multipop	O(1)

worst-case runtime complexity = O(n)

- Fibonacci heaps are loosely based on binomial heaps.

- A Fibonacci heap is a collection of min-heap trees similar to Binomial heaps, however, trees in a Fibonacci heap are not constrained to be binomial trees. Also, unlike binomial heaps, trees in Fibonacci heaps are not ordered.

- Link to Fibonacci heap animations:

www.cs.usfca.edu/ngalles/JavascriptVisual/FibonacciHeap.html

Amortized Costs

	Binary Heap	Binomial Heap	Fibonacci Heap
Find-Min	$O(1)$	$O(\lg n)$	$O(1)$
Insert	$O(\lg n)$	"	$O(1)$
Extract-Min	"	"	$O(\lg n)$
Delete	"	"	$O(\lg n)$
Decrease-key	"	"	$O(1)$
Merge	$O(n)$	"	$O(1)$
Construct	$O(n)$	$O(n)$	$O(n)$

Discussion 3

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.
Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.
What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

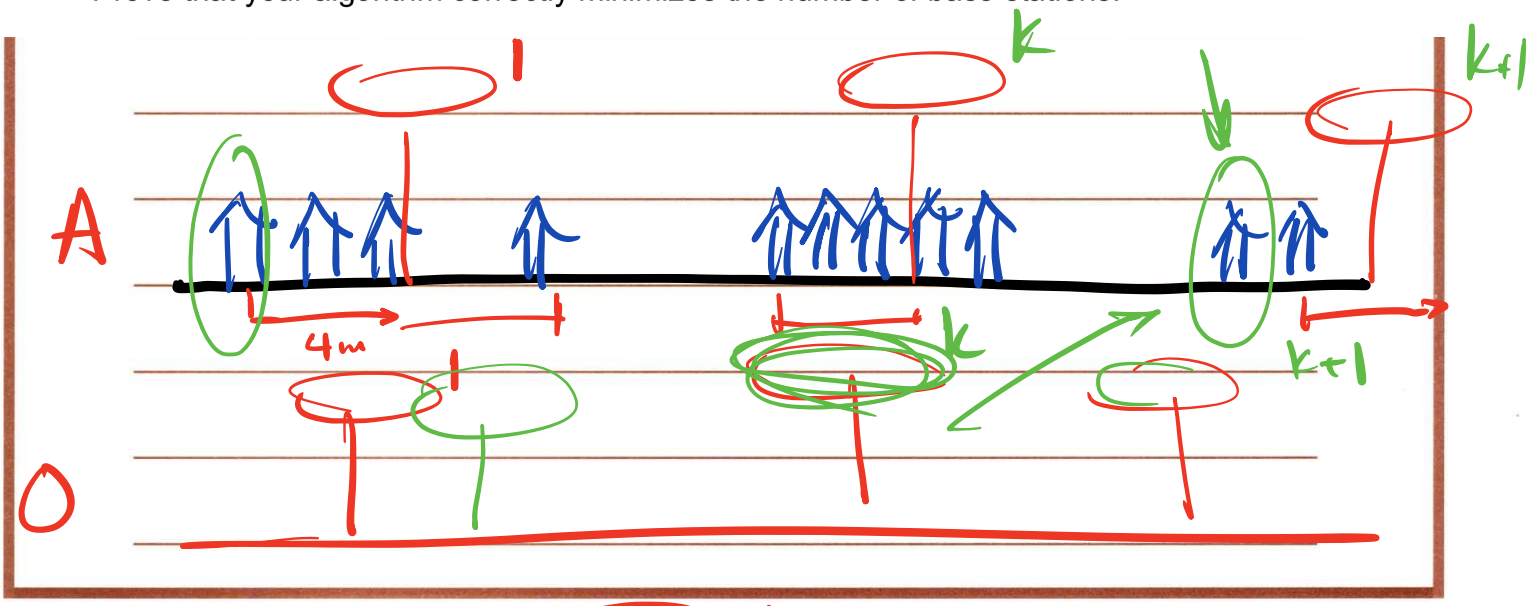
3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

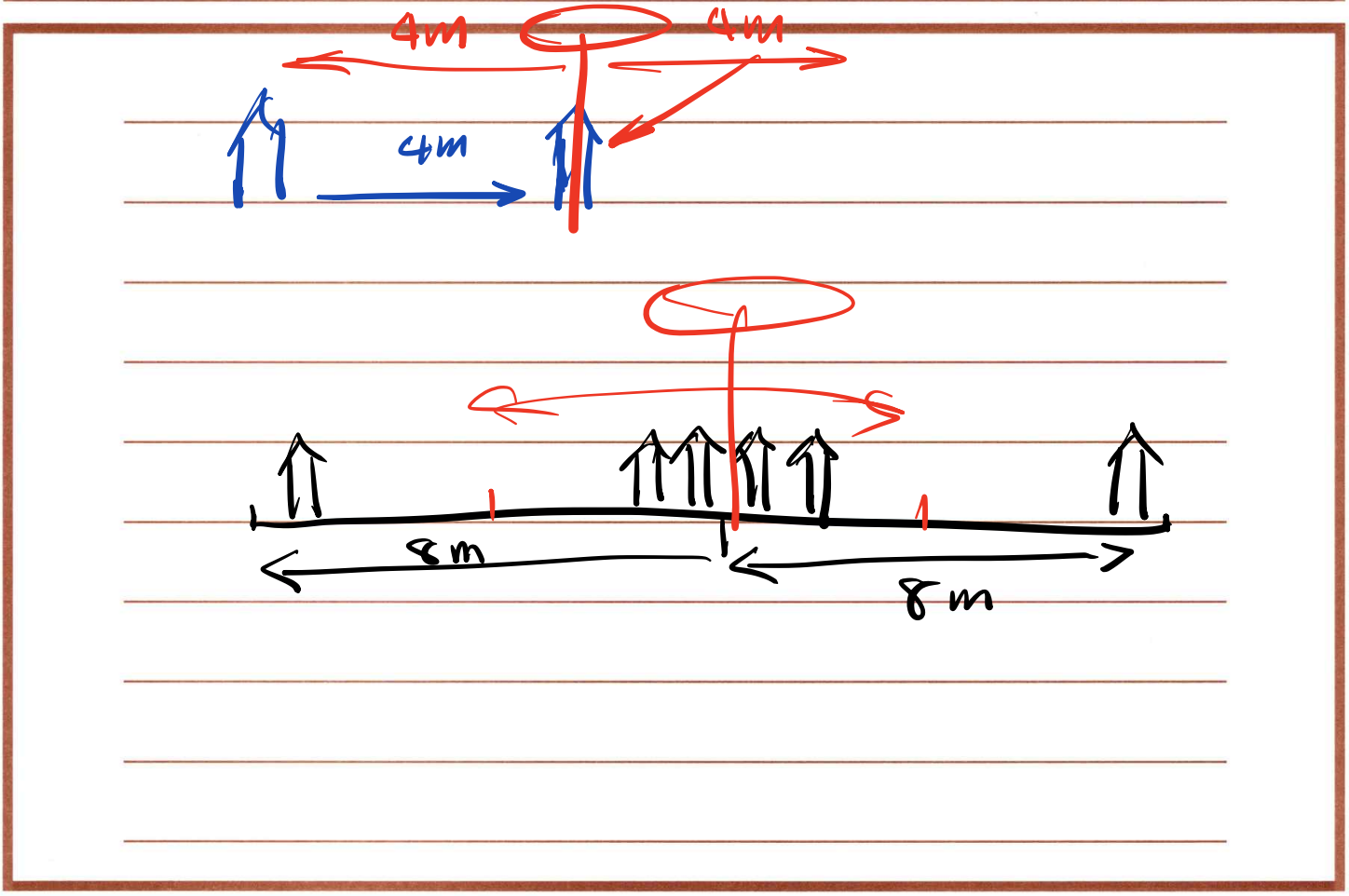
4. Given an unsorted array of size n. Devise a heap-based algorithm that finds the k-th largest element in the array. What is its runtime complexity?

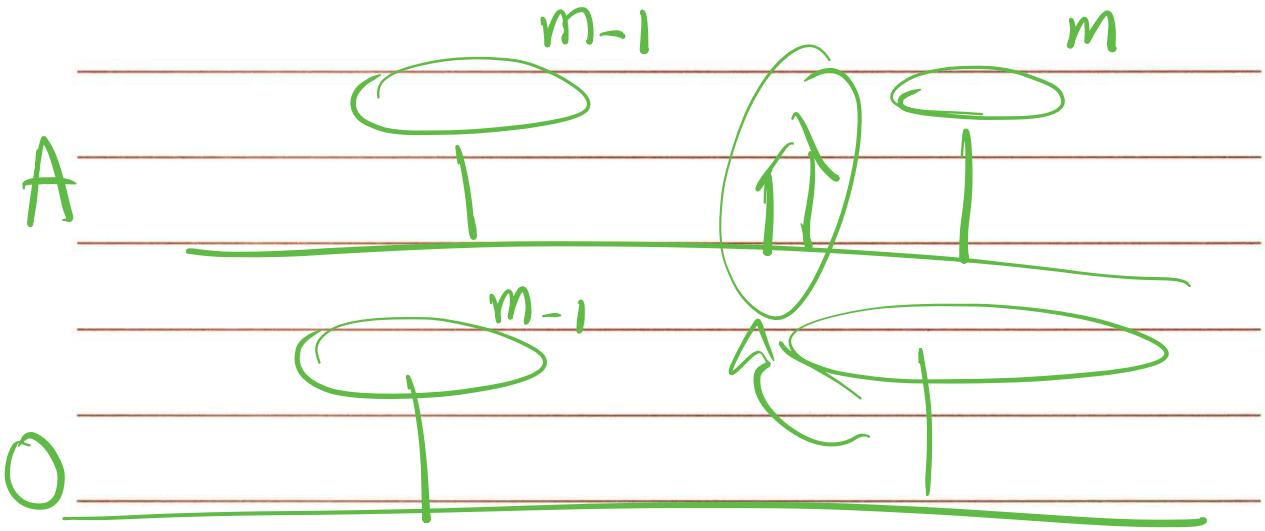
5. Suppose you have two min-heaps, A and B, with a total of n elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.



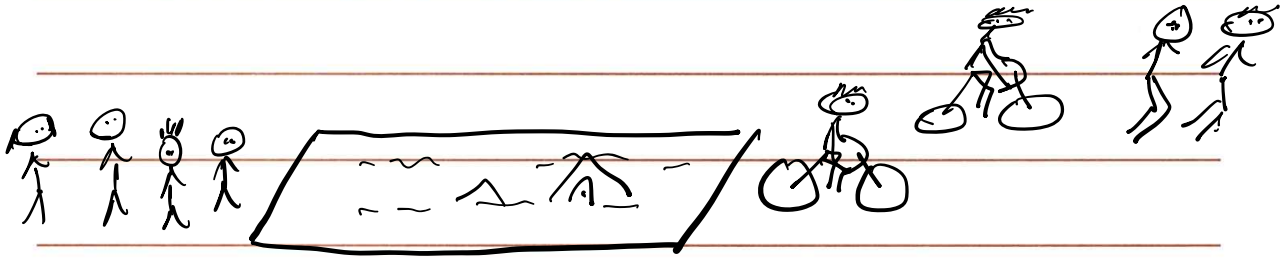




2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.

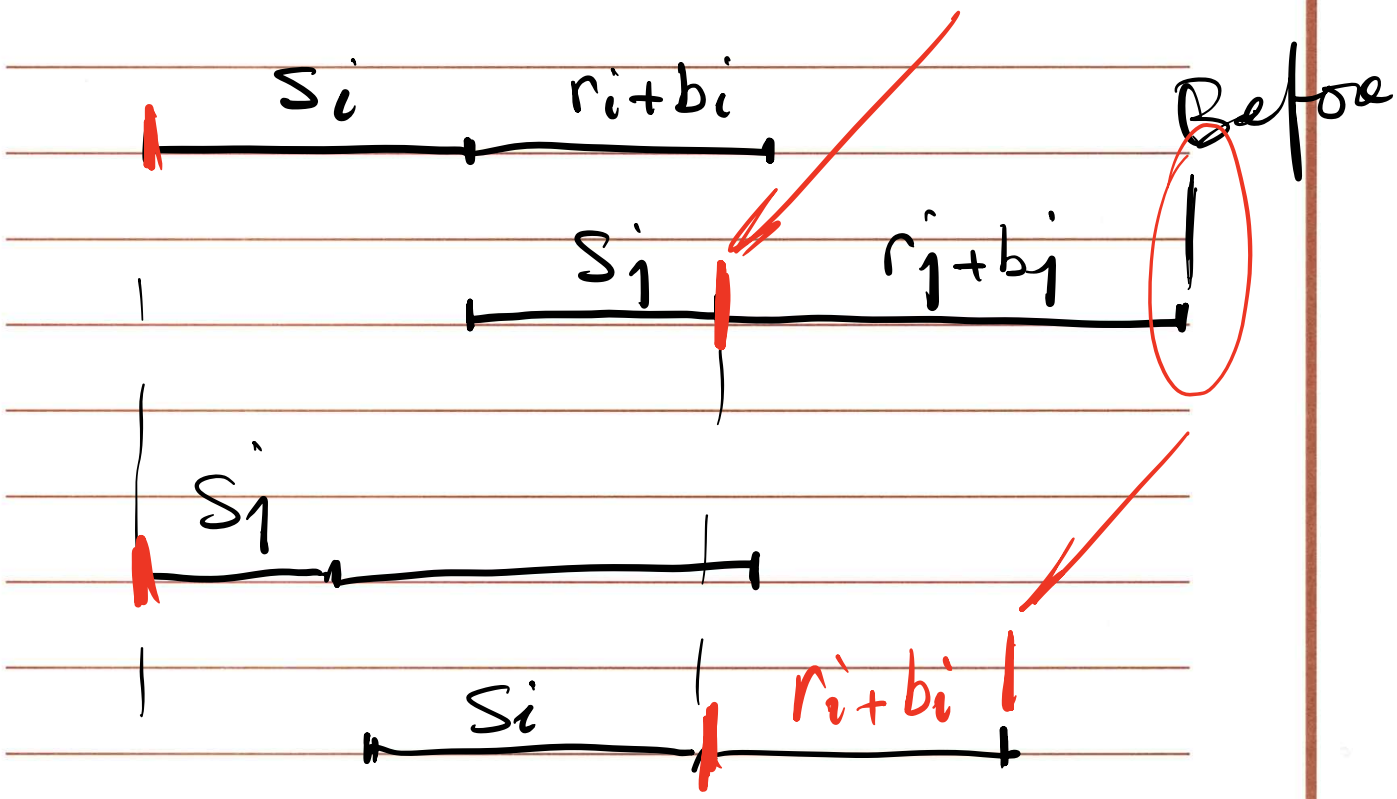
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.

What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

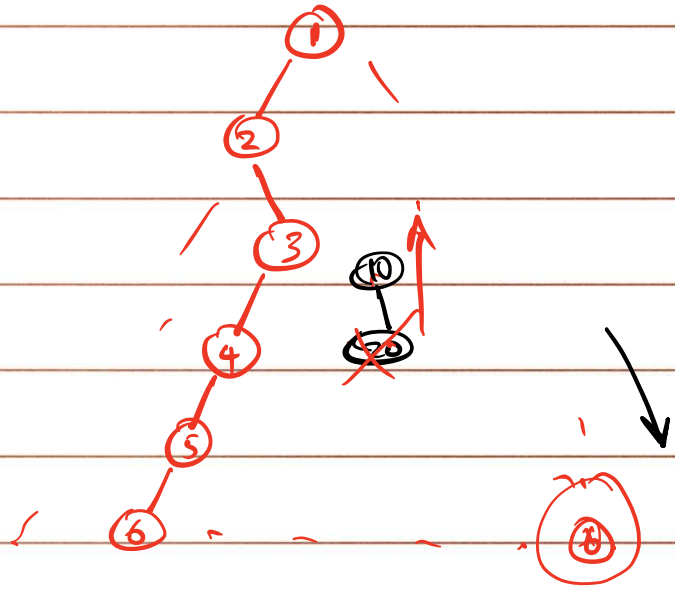


Swim segment

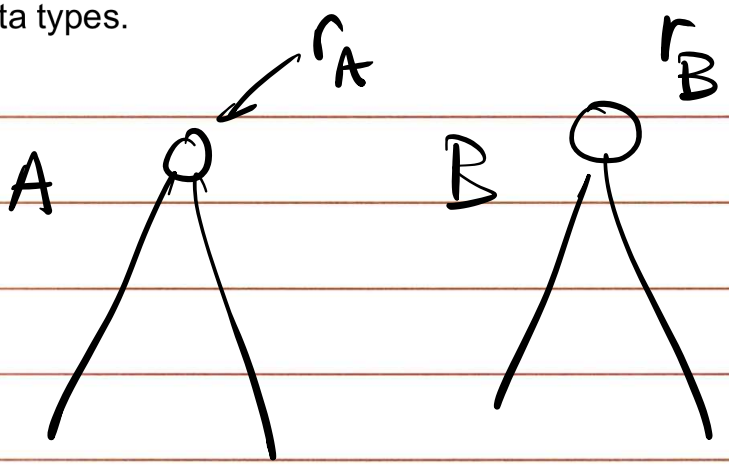
Inversion: Athlete i with $r_i + b_i$ less than that of athlete j w/ $r_j + b_j$ where i is scheduled before j.



3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?



5. Suppose you have two min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

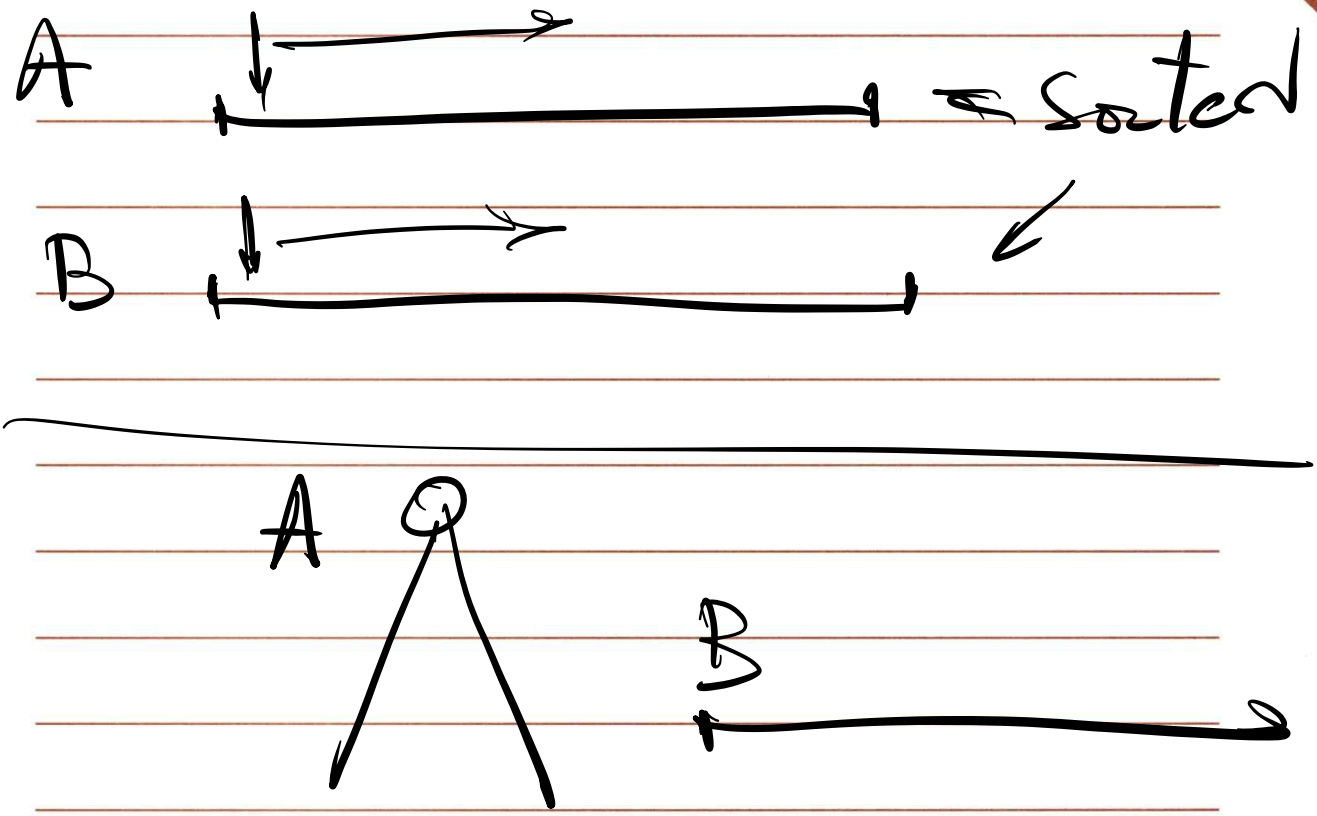


Takes $O(\log n)$

$r_A < r_B$ then Extract_Min(A)

$r_A > r_B$ " Extract_Min(B)

$r_A = r_B$ Then we're done!


A
Sorted
B
A
B

Discussion 3

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.
Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

Solution:
Find the first house from the left (western most house) and go four miles to its right and place a base station there. Eliminate all houses covered by that station and repeat the process until all houses are covered.

Proof:
The proof is similar to that for the interval scheduling solution we did in lecture. We first show (using mathematical induction) that our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal (using proof by contradiction).

1- Our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution:
   Base case: we place our first base station as far to the right of the leftmost house as possible. If the base station in the optimal solution is to its right then the first house on the left will not be covered.
   Inductive step: Assume our kth base station is to the right of the kth base station in the optimal solution. We can now show that our k+1st base station is also to the right of the k+1st base station in the optimal solution. To prove this we look at the leftmost house to the right of our kth base station which is not covered by our kth base station. We call this house H. If H is not covered by our kth base station, then it cannot be covered by the kth base station in the optimal solution since our base station is to the right of it. Our k+1st base station is placed 4 miles to the right of H. If the k+1st base station in the optimal solution is further to the right of our k+1st base station, then H is not going to be covered by neither the Kth base station nor the k+1st base station in the optimal solution so the k+1st base station in our solution must also be to the right of the k+1st base station in the optimal solution.
2- Now assume that our solution required n base stations and the optimal solution requires fewer base stations. We now look at our last based station. The reason we needed this base station in our solution was that there was a house to the right of our n-1st base station that was not covered by it. We call this house H. If H is not covered by our n-1st

base station, then H will not be covered by the n-1st base station in the optimal solution either (since our base stations are always to the right of the corresponding base stations in the optimal solution). So, the optimal solution will also need another base station to cover H.

Complexity of our solution will be O(n logn) since we need to sort the houses first from left to right by their x coordinates. The actual positioning of the base stations will require O(n) time.

2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.
What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

Solution:
Sort the athletes by decreasing biking + running time.

Proof:
The proof is similar to the proof we did for the scheduling problem to minimize maximum lateness. We first define an inversion as an athlete i with higher (bi+ri) being scheduled after athlete j with lower (bj+rj). We can then show that inversions can be removed without increasing the competition time. We then show that given an optimal solution with inversions, we can remove inversions one by one without affecting the optimality of the solution until the solution turns into our solution.

1- Inversions can be removed without increasing the competition time.
   Remember that if there is an inversion between two items a and b, we can always find two adjacent items somewhere between a and b so that they have an inversion between them. Now we focus on two adjacent athletes (scheduled one after the other) who have an inversion between them, e.g. athlete i with higher (bi+ri) is scheduled after athlete j with lower (bj+rj). Now we show that scheduling athlete i before athlete j is not going push out the completion time of the two athletes i and j. We do this one athlete at a time:
   - By moving athlete i to the left (starting earlier) we cannot increase the completion time of athlete i
   - By moving athlete j to the right (starting after athlete i) we will push out the completion time of athlete j but since the swimming portion is sequential and athlete j gets out of the pool at the same time that athlete i was getting out of the pool before removing the inversion, and since athlete j is faster than athlete i in the biking and

running sections, then the completion time for athlete j will not be worse than the completion time for athlete i prior to removing the inversion.

2- Since we know that removing inversions will not affect the completion time negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours, i.e. athletes sorted in descending order of biking+running time. So our solution is also optimal.

3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

Solution: since there are $2^6 - 1$ elements in the heap, the heap will consist of a full binary tree with 6 levels. So, the smallest possible value at a leaf node would be 6.

4. Given an unsorted array of size *n*. Devise a heap-based algorithm that finds the *k*-th largest element in the array. What is its runtime complexity?

Solution: build a max heap of size n in O(n) time. Do k extract_max operations to find the kth largest element. The overall complexity will be O(n + k logn)

5. Suppose you have two min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

Solution:
We can compare the element at the top of A (TA) with the element at the top of B (TB).

If TA < TB	Extract_Min (A)
Elseif TA > TB	Extract_Min (B)
Else	A mut be equal to B. We have found a common key

We repeat the above until we either find a common key or one of the heaps is empty.

The complexity of the solution is O(n log n) since at each iteration we do two extract_min operations at a cost of log n and there could be at most O(n) iterations.

CSCI 570 - Fall 2021 - HW 3

Due September 16, 2021

1 You have N ropes each with length L_1, L_2, ..., L_N, and we want to connect the ropes into one rope. Each time, we can connect 2 ropes, and the cost is the sum of the lengths of the 2 ropes. Develop an algorithm such that we minimize the cost of connecting all the ropes.

Rubric:

10 points for correct algorithm.

-1 point if it doesn't clarify that each step involves picking TWO smallest ropes

-2 points if it's not clearly mentioned that the resultant rope length after joining goes back into the set of candidates and re-sort (the sorting is ensured if heaps or appropriate data-structures are mentioned).

2 You have a bottle that can hold L liters of liquid. There are N different types of liquid with amount L_1, L_2, ..., L_N and with value V_1, V_2, ..., V_N. Assume that mixing liquids doesn't change their values. Find an algorithm to store the most value of liquid in your bottle.

Rubric:

10 points for correct algorithm -3 points if misinterpreted the "value" to mean value per unit and thus, went greedy on V instead of V/L.

3 Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let $d_1 < d_2$ ... $< d_n$ be the locations of all the gas stations along the route where d_i is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most p miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Give the time complexity of your algorithm as a function of n. (20 points)

Rubric:

10 points for correct algorithm that reaching the furthest gas stations each time.

7 points for proving that reaching the furthest gas stations is optimal using sta proof by contradiction.

3 points for pointing out that the running time is O(N).

4 (a) Consider the problem of making change for n cents using the fewest number of coins. Describe a greedy algorithm to make change consisting of quarters(25 cents), dimes(10 cents), nickels(5 cents) and pennies(1 cents). Prove that your algorithm yields an optimal solution. (Hints: consider how many pennies, nickels, dimes and dime plus nickels are taken by an optimal solution at most.) (20 points)

Rubric:

10 points for the greedy algorithm that always pick highest value of coins.

10 points for proving that the algorithm is optimal by supporting that this algorithm is optimal in each range of cents(1-5, 5-10, 10-25, 25-INF). -2 points for missing any range.

(b) For the previous problem, give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Assume that each coin's value is an integer. Your set should include a penny so that there is a solution for every value of n. (10 points)

Rubric:

10 points for any correct coin denominations that does not yield an optimal solution.

5 Solve Kleinberg and Tardos, Chapter 3, Exercise 3. (15 points)

Rubric:

10 points for correct algorithm. -2 points if the algorithm is slower than O(m + n).

5 points to indicating how this algorithm can output topological ordering or a cycle. -3 points for missing any of them.

6 Solve Kleinberg and Tardos, Chapter 4, Exercise 4. (20 points)

Rubric:

10 points for correct greedy algorithm that takes linear running time. -2 points for any slower algorithm.

7 points for proving the algorithm by induction.

3 points for correct time complexity.

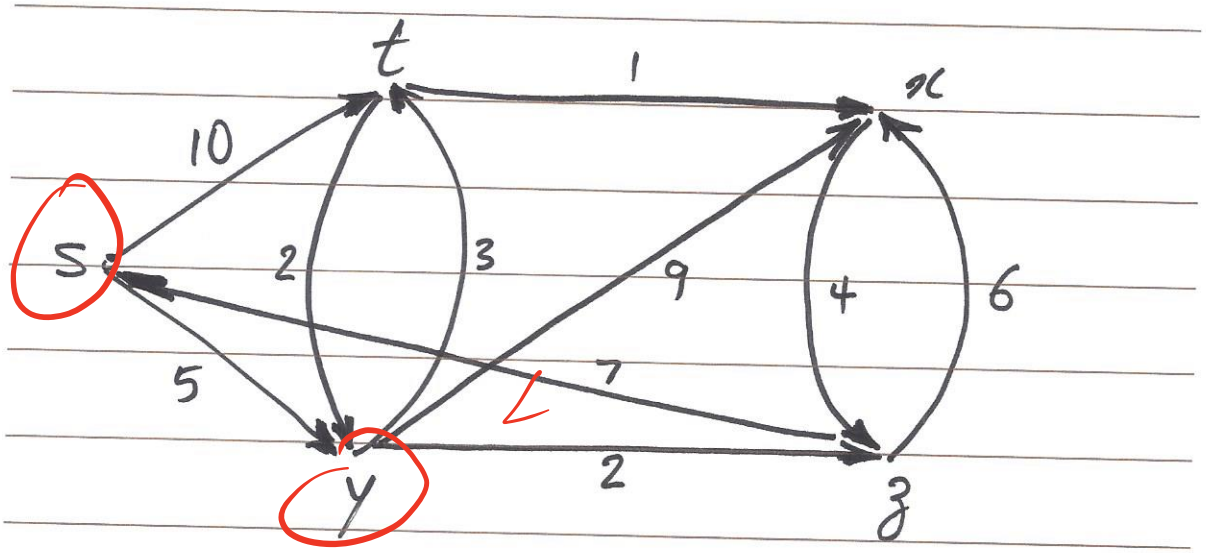
7 (Not Graded) There are N tasks that need to be completed by 2 computers A and B. Each task i has 2 parts that takes time a_i(first part) and b_i (second part) to be completed. The first part must be completed before starting the second part. Computer A does the first part of all the tasks while computer B does the second part of all the tasks. Computer A can only do one task at a time, while computer B can do any amount of tasks

at the same time. Find an O(nlogn) algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution is optimal.

Shortest Path

Problem Statement:

Given $G = (V, E)$ with $w(u,v) \geq 0$ for each edge $(u,v) \in E$, find the shortest path from $s \in V$ to $V - s$.

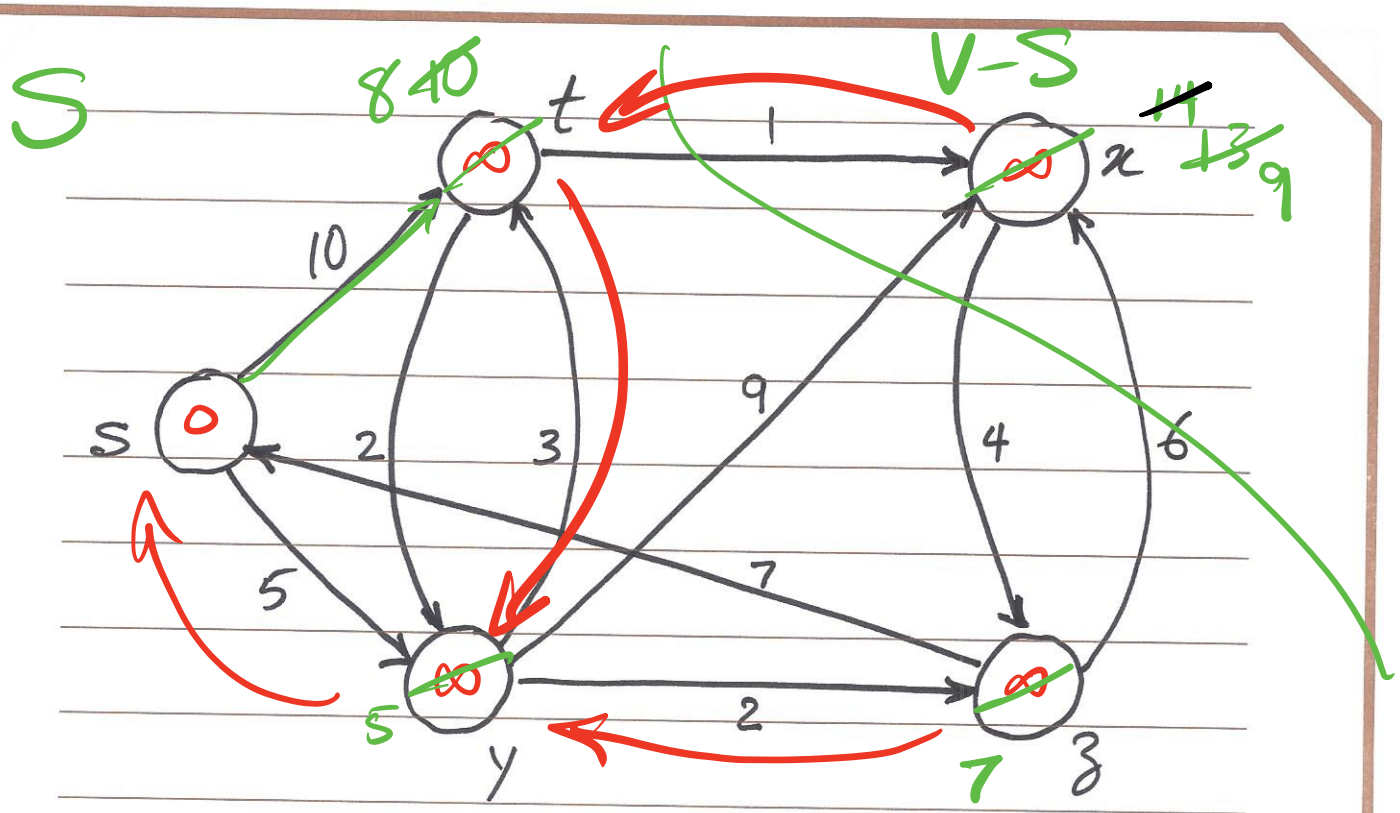


Solution

1. Start with a set S of vertices whose final shortest path we already know.

2. At each step, find a vertex $v \in V - S$ with shortest distance from s.

3. Add v to S, and repeat.



$S_1 = \{s\}$, $S_2 = \{s, x\}$, $S_3 = \{s, y, z\}$

$S_4 = \{s, y, z, t\}$, $S_5 = \{s, y, z, t, x\}$

Dijkstra's Alg.

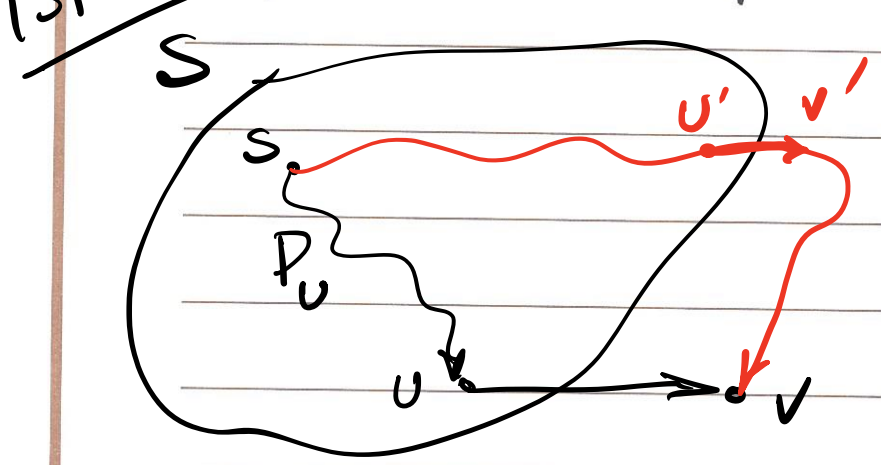
Proof of Correctness

We will prove that at each step, Dijkstra's algorithm finds the shortest path to a new node in the graph.

Proof by mathematical induction:

Base Case: $|S| = 1$, $S = \{s\}$ and $d(s) = 0$

Inductive Step: Suppose the claim holds when $|S| = k$ for some $k \geq 1$. We now grow S to size $k+1$ and prove that we have found the shortest path to the new node.



Implementation of Dijkstra's

Initially $S = \{s\}$ and $d(s) = 0$

for all other nodes $d(u) = \infty$

While $S \neq V$

Select a node $v \notin S$ with at least one edge from S for which

$$d(v) = \min_{e(u,v) : u \in S} (d(u) + l_e)$$

Add v to S

endwhile

More Detailed Implementation of Dijkstra's

S = Null

Initialize priority Queue Q with all nodes V where d(v) is the key value.
(All d(v)'s are $=\infty$, except for s where d(s)=0)

While $S \neq V$

v = Extract_Min (Q)

S = S ∪ {v}

for each vertex $u \in Adj(v)$

if $d(u) > d(v) + l_e$;

Decrease_key (Q, u, d(v)+l_e)

end for

endwhile

cost of edge from v to u

O(n)

O(n)

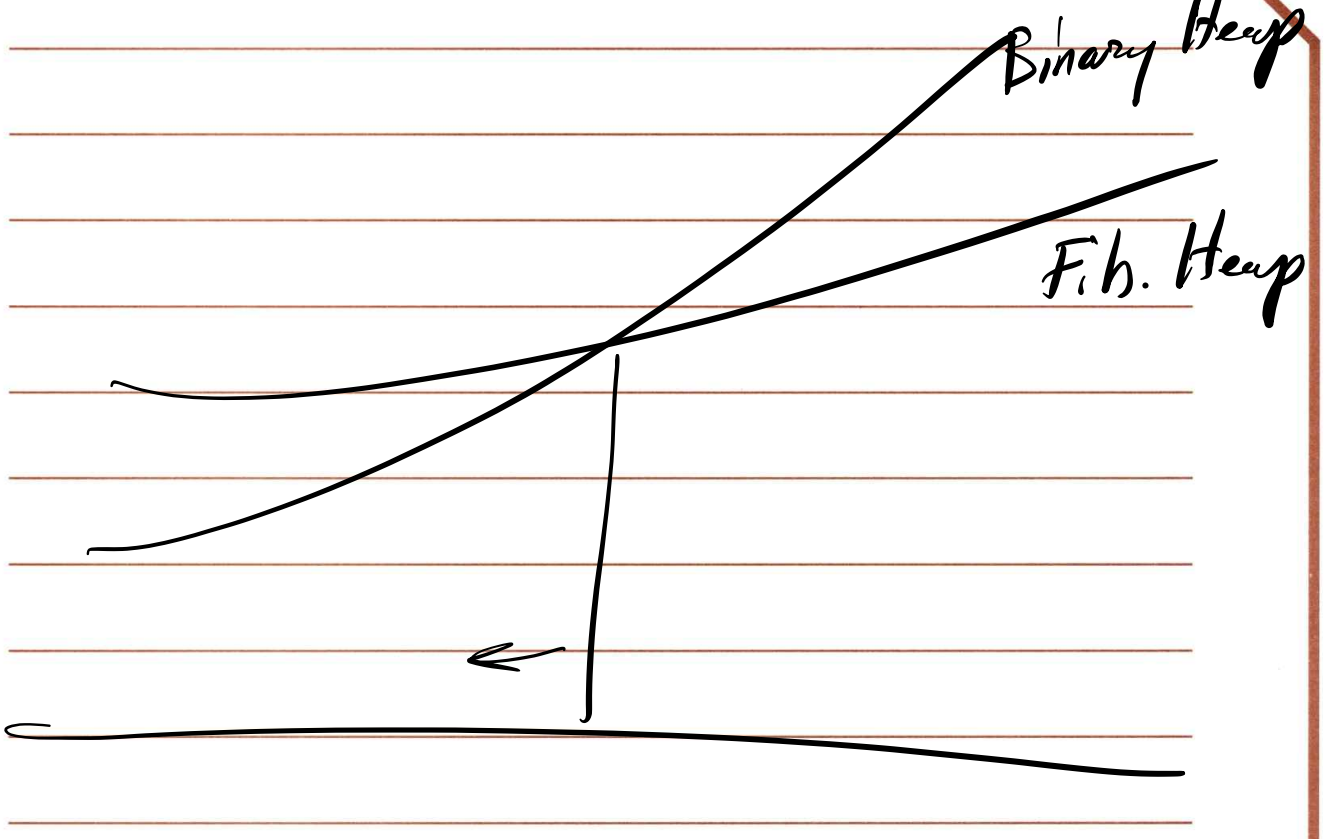
$O(n^2)$

O(n)

Complexity Analysis

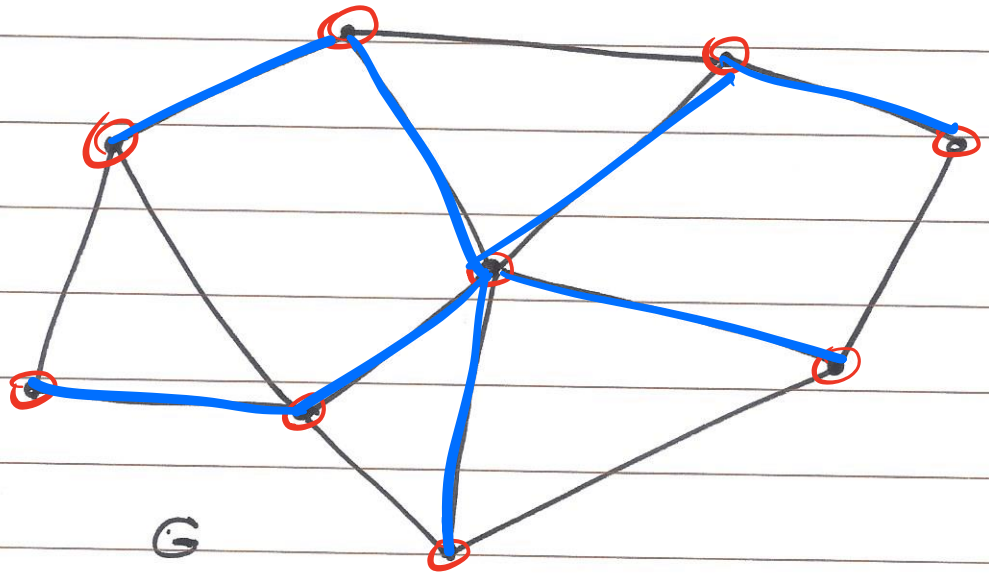
- Initialize Priority Queue $O(n)$
- Max. no. of Extract-Min op's : $O(n)$
- Max. no. of Decrease-key op's : $O(m)$

	Binary Heap	Binomial Heap	Fibonacci Heap
n * Extract-Mins	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
m * Decrease-key's	$O(m \lg n)$	$O(m \lg n)$	$O(m)$
Total	$O(m \lg n)$	$O(m \lg n)$	$O(m + n \lg n)$
Sparse Graphs	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Dense Graphs	$O(n^2 \lg n)$	$O(n^2 \lg n)$	$O(n^2)$


Binary Heap
F.h. Heap

Problem Statement

Find minimum cost network that connects all nodes in G.



G

Def. Any tree that covers all nodes of a graph is called a spanning tree.

Def. A spanning tree with minimum total edge cost is a minimum spanning tree. (MST)

Problem Statement

Find a MST in an undirected graph

Sol. 1: Sort all edges in increasing order of cost. Add edges to T in this order as long as it does not create a cycle. If it does, discard the edge.

Kruskal's Alg.

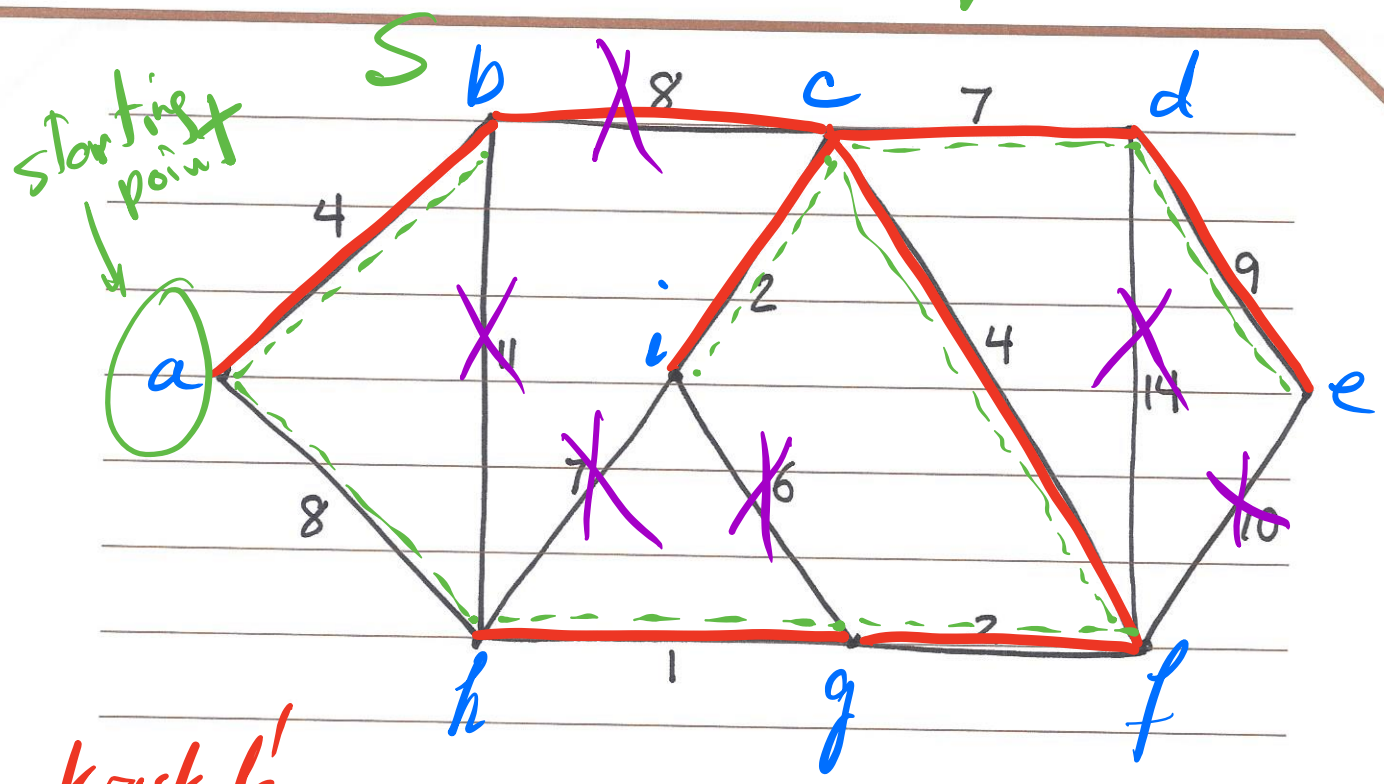
Sol. 2: Similar to Dijkstra's algorithm, start with a node set S (initially the root node) on which a minimum spanning tree has been constructed so far. At each step, grow S by one node, adding the node v that minimizes the attachment cost.

Prim's Alg.

Sol. 3: Backward version of Kruskal's.
Start with a full graph (V, E).
Begin deleting edges in order of decreasing cost as long as it does not disconnect the graph

Reverse-Delete

V-S

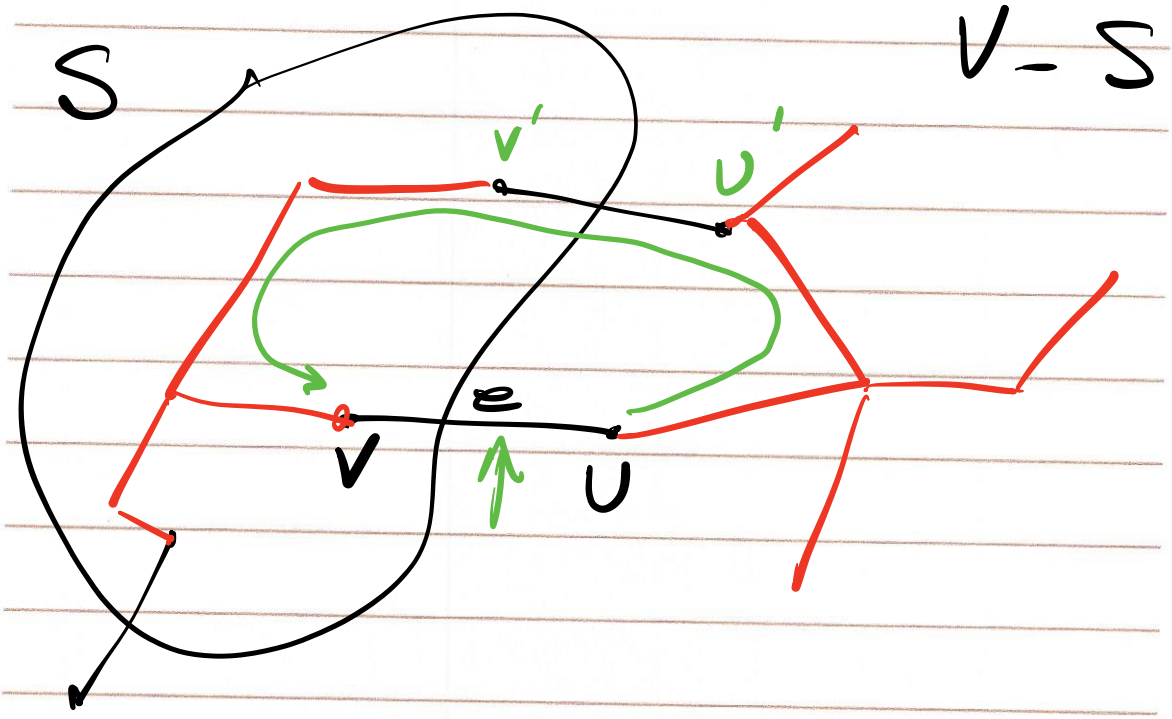


Kruskal's ——

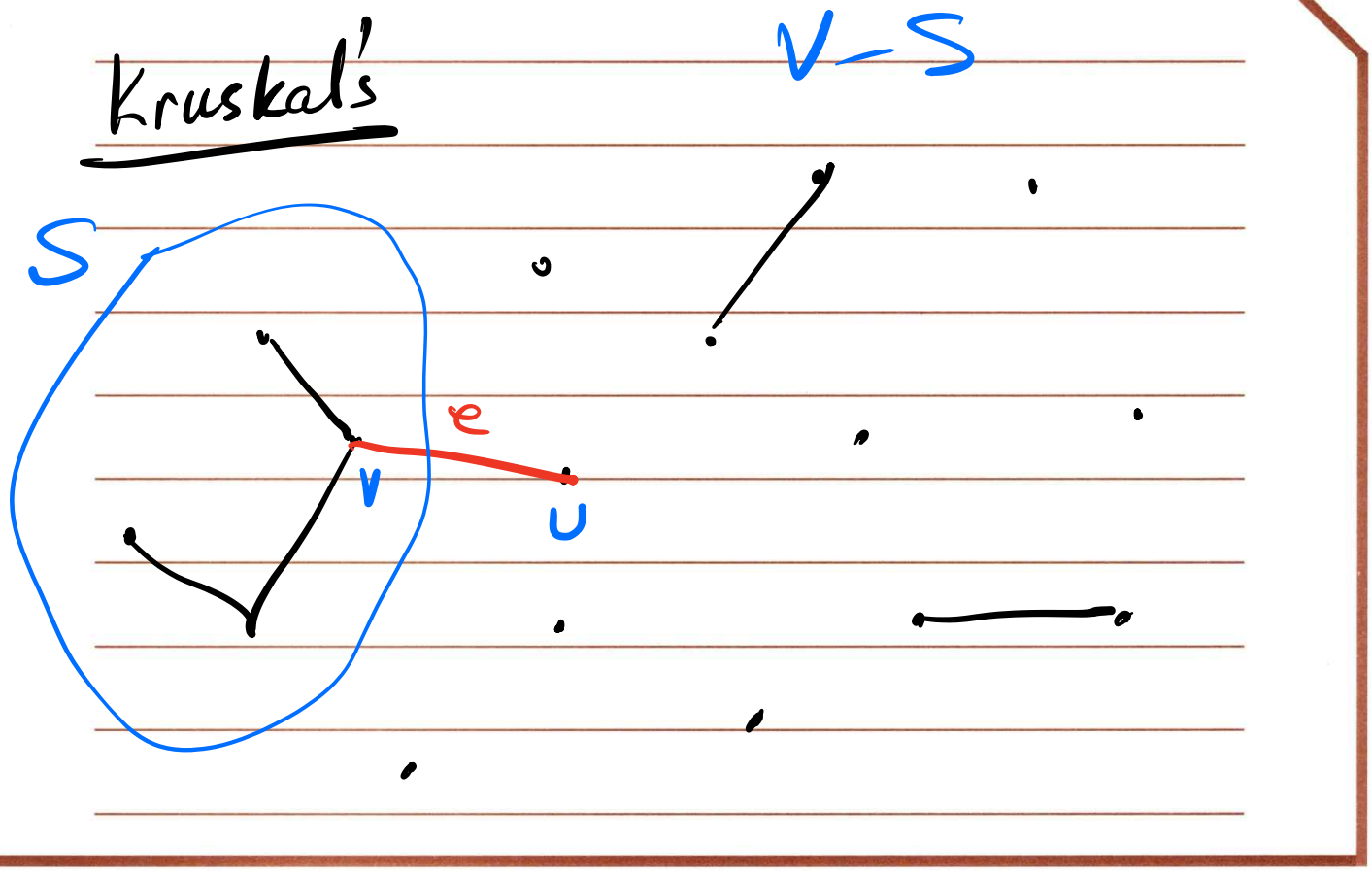
Prim's - - - - -

Reverse-Delete X

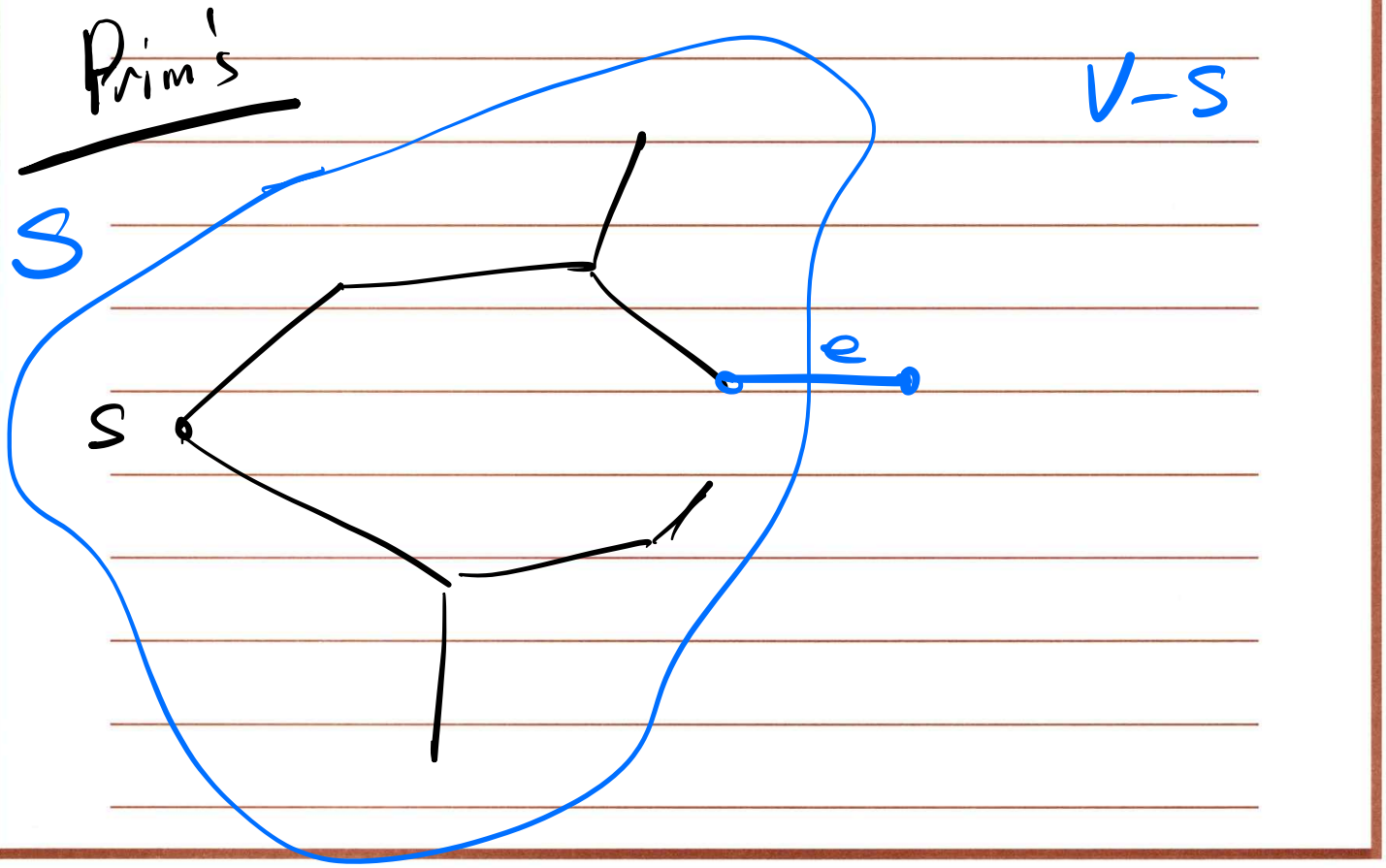
FACT: Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the min cost edge with one end in S and the other end in $V - S$. Then every MST contains the edge e



Kruskal's



Prim's



More Detailed Implementation of ~~Dijkstra's~~ Prim's

S = Null

Initialize priority Queue Q with all nodes V where d(v) is the key value.
(All d(v)'s are = ∞, except for s where d(s) = 0)

While $S \neq V$

v = Extract_Min (Q)

S = S ∪ {v}

for each vertex u ∈ Adj(v)

if d(u) > ~~d(v)~~ + le;

Decrease_key (Q, u, ~~d(v)~~ + le)

end for

end while

Kruskal's

Create an endp set for each node
A = Null
Sort edges in non-decreasing order of weight
for each edge $(u,v) \in E$, taken in this order, if u & v are NOT in the same set then
$A = A \cup \{(u,v)\}$
merge the two sets
end if
endfor

Union-Find data structure

- Make set — O(1) for set size = 1
- Find set — O(1) or O(lg n)
- Union — O(lg n) or O(1)

array impl. ptr impl.

Implementation of Kruskal's

```
A = Null
for each vertex v ∈ V
    Make-set(v)
end for
```

O(n)

Sort the edges of E into non-decreasing order of cost

O(m log m)

```
for each edge (u,v) ∈ E in this order.
    if Find-set(u) ≠ Find-set(v) then
        A = A ∪ {(u,v)}
        Union(u,v)
    endif
endfor
```

O(m)

overall complexity $= O(n) + O(m \log m) + O(m \log m)$

$= O(m \log m)$

Prim's

$O(m \log n)$

Kruskal's

$O(m \log m)$

$O(m \log m) = O(m \log n^2)$

$= O(m \log n)$

Your name!

Reverse-Delete

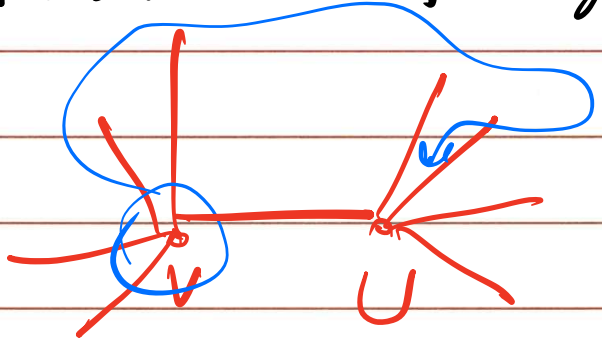
$O(m \log m)$ (Sort edges

loop over edges

$O(m+n)$ (for edge e check to see if removing it will disconnect the graph

$O(n)$

overall cost $= O(m^2)$



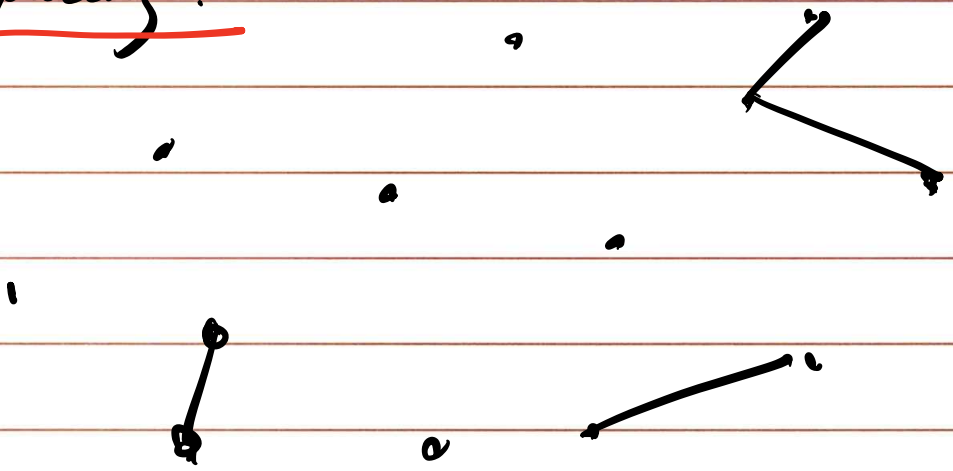
Clustering

Def. Given a set U of n objects $p_1 \dots p_n$ where $d(p_i, p_i) = 0$, $d(p_i, p_j) > 0$ for $p_i \neq p_j$, and $d(p_i, p_j) = d(p_j, p_i)$, a *k-clustering* of U is a partition of U into k nonempty sets $C_1 \dots C_k$

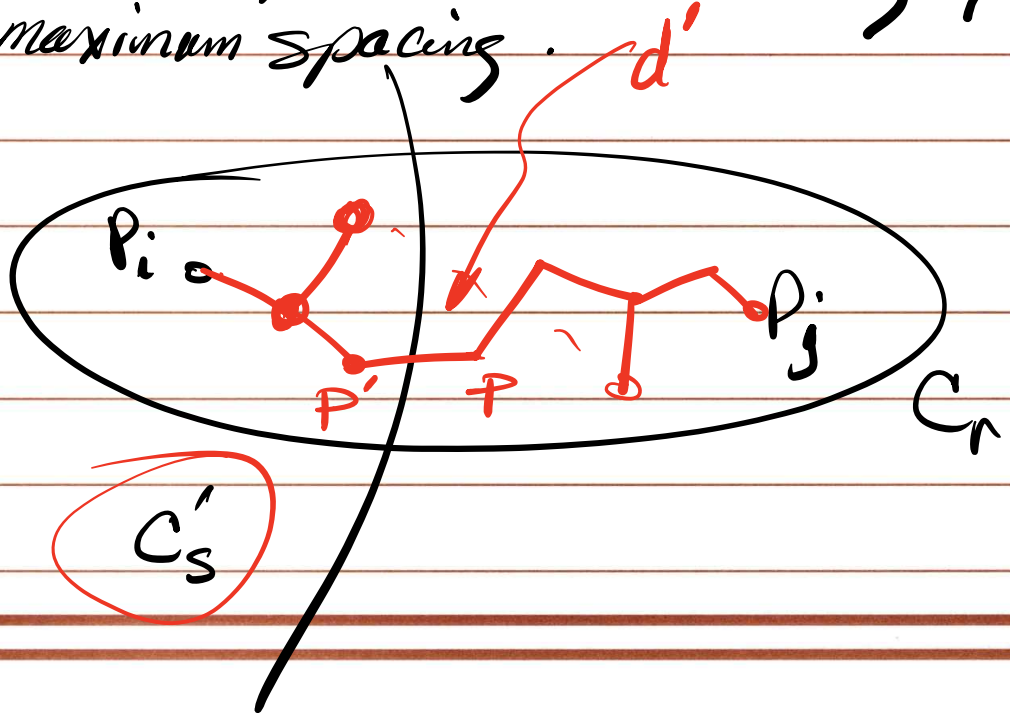
Def. The *spacing* of a k-clustering is the minimum distance between any pair of points lying in different clusters.

Problem Statement:

Given a set of n objects as described above, find a k-clustering with *maximum spacing*.



Claim: Clusters $C_1 .. C_k$ created by deleting the $k-1$ most expensive edges of the MST T give us a k-clustering of maximum spacing.



Say our spacing is d

Consider $pp' \leq d$

$d \leq d'$

Discussion 4

1. Hardy decides to start running to work in San Francisco city to get in shape. He prefers a route that goes entirely uphill and then entirely downhill so that he could work up a sweat uphill and get a nice, cool breeze at the end of his run as he runs faster downhill. He starts running from his home and ends at his workplace. To guide his run, he prints out a detailed map of the roads between home and work with k intersections and m road segments (any existing road between two intersections). The length of each road segment and the elevations at every intersection are given. Assuming that every road segment is either fully uphill or fully downhill, give an efficient algorithm to find the shortest path (route) that meets Hardy's specifications. If no such path meets Hardy's specifications, your algorithm should determine this. Justify your answer.

2. You are given a graph representing the several career paths available in industry. Each node represents a position and there is an edge from node *v* to node *u* if and only if *v* is a prerequisite for *u*. Top positions are the ones which are not prerequisites for any positions. The cost of an edge *(v, u)* is the effort required to go from one position *v* to position *u*. Ivan wants to start a career and achieve a top position with minimum effort. Using the given graph can you provide an algorithm with the same run time complexity as Dijkstra's? You may assume the graph is a DAG.

3. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations.

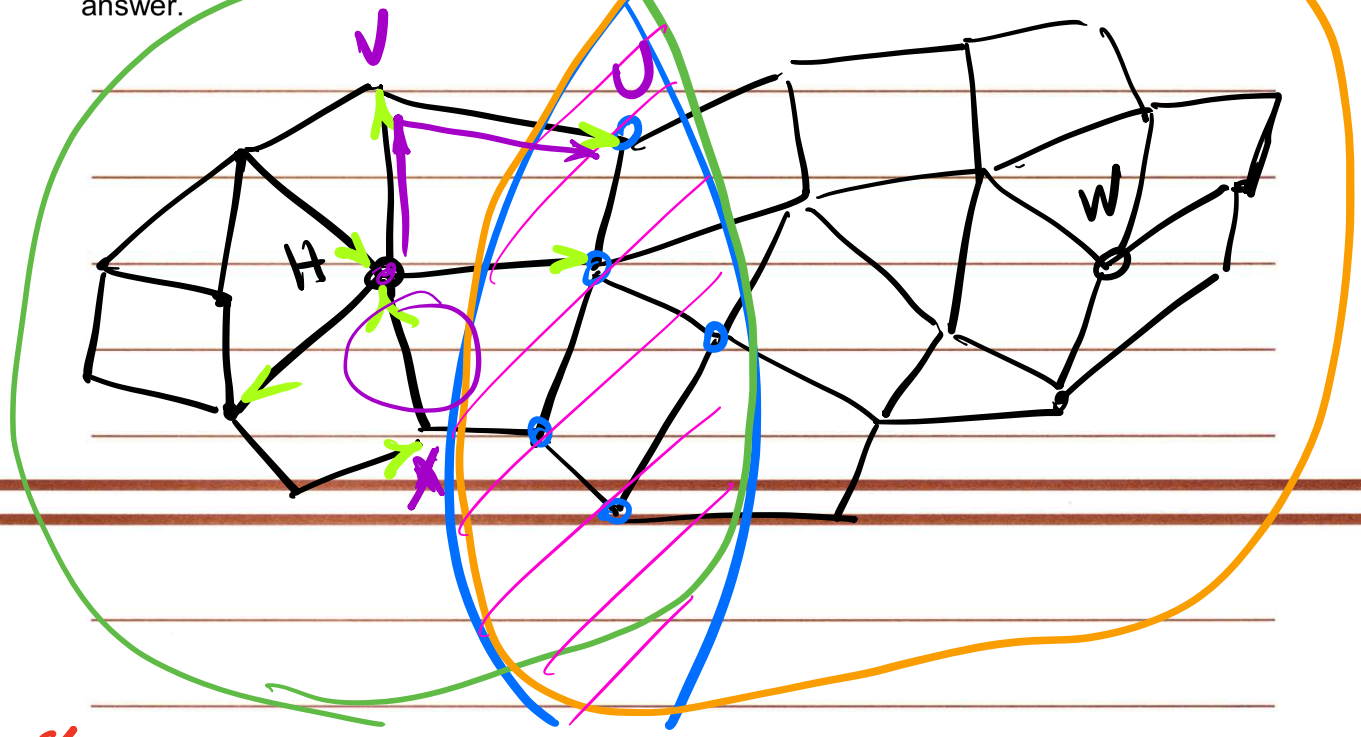
4. Given a sequence of numbers: 3, 5, 2, 8, 1, 5, 2, 7

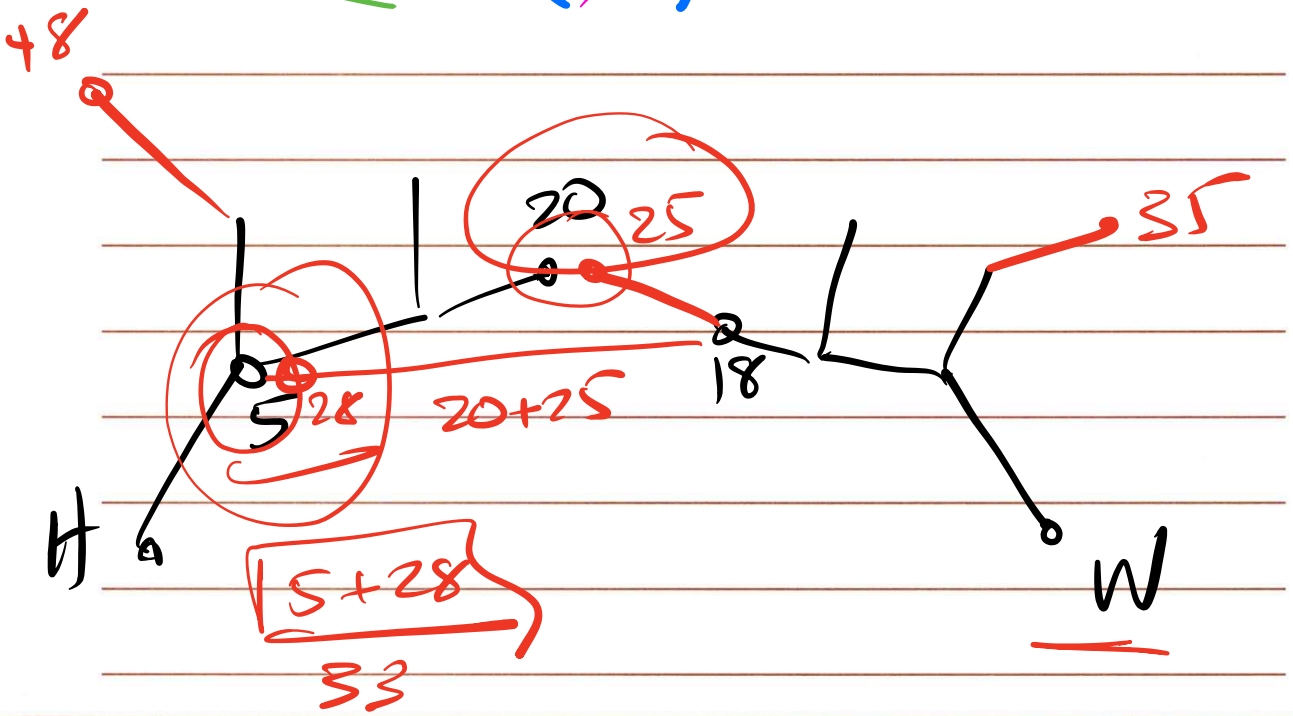
a. Draw a binomial heap by inserting the above numbers reading them from left to right
b. Show a heap that would be the result after the call to deleteMin() on this heap

5. (a): Suppose we are given an instance of the Minimum Spanning Tree problem on a graph G. Assume that all edges costs are distinct. Let T be a minimum spanning tree for this instance. Now suppose that we replace each edge cost c_e by its square, c_e^2 thereby creating a new instance of the problem with the same graph but different costs. Prove or disprove: T is still a MST for this new instance.

(b): Consider an undirected graph G = (V, E) with distinct nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G. Now suppose each edge weight is increased by 1: the new weights are $c'_e = c_e + 1$. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

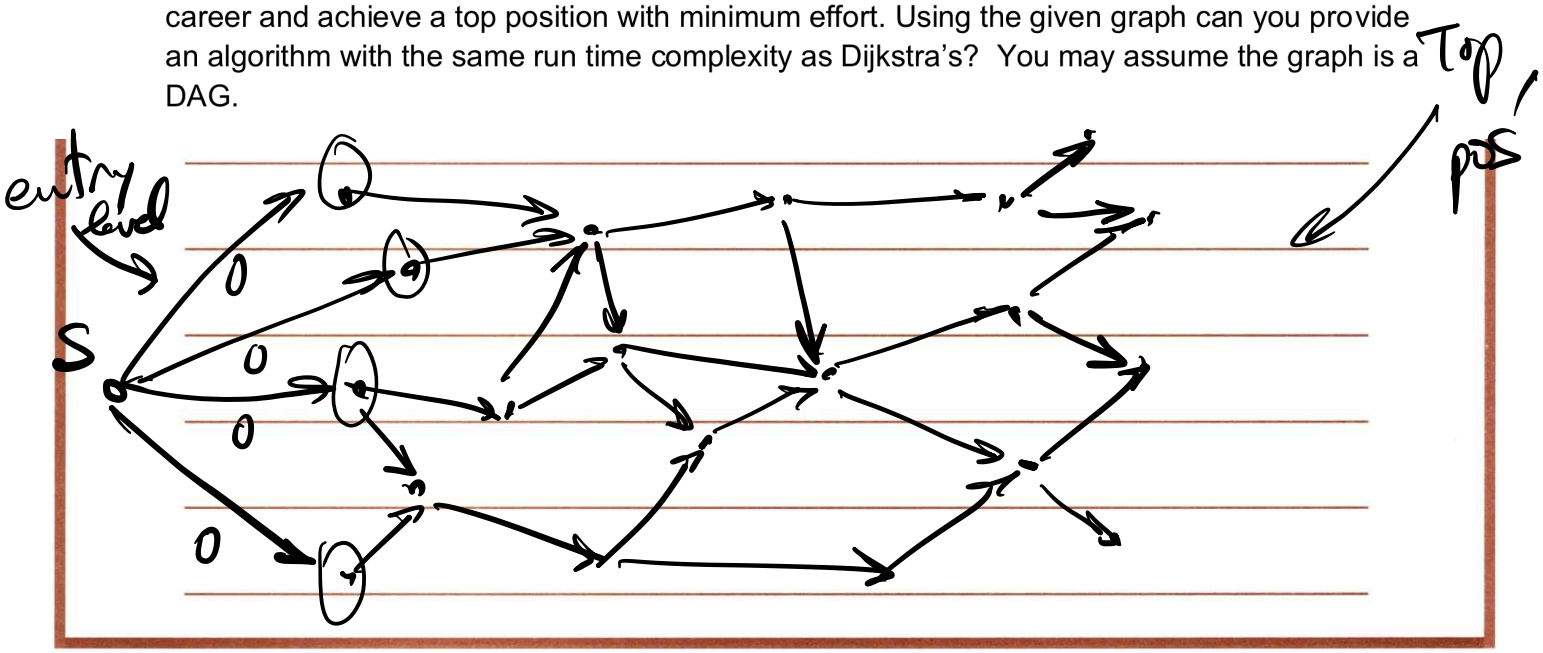
1. Hardy decides to start running to work in San Francisco city to get in shape. He prefers a route that goes entirely uphill and then entirely downhill so that he could work up a sweat uphill and get a nice, cool breeze at the end of his run as he runs faster downhill. He starts running from his home and ends at his workplace. To guide his run, he prints out a detailed map of the roads between home and work with k intersections and m road segments (any existing road between two intersections). The length of each road segment and the elevations at every intersection are given. Assuming that every road segment is either fully uphill or fully downhill, give an efficient algorithm to find the shortest path (route) that meets Hardy's specifications. If no such path meets Hardy's specifications, your algorithm should determine this. Justify your answer.





answer.

2. You are given a graph representing the several career paths available in industry. Each node represents a position and there is an edge from node *v* to node *u* if and only if *v* is a prerequisite for *u*. Top positions are the ones which are not prerequisites for any positions. The cost of an edge *(v, u)* is the effort required to go from one position *v* to position *u*. Ivan wants to start a career and achieve a top position with minimum effort. Using the given graph can you provide an algorithm with the same run time complexity as Dijkstra's? You may assume the graph is a DAG.



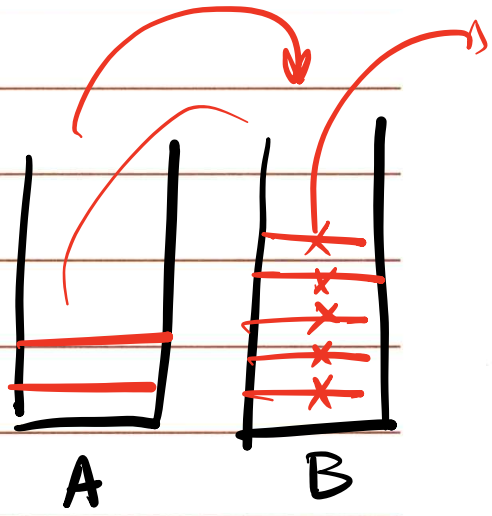
3. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations.

n enqueues → n pushes = n

1 dequeue → n + n + 1

total cost = 3n + 1

Amortized cost = $\frac{3n+1}{n} = O(1)$

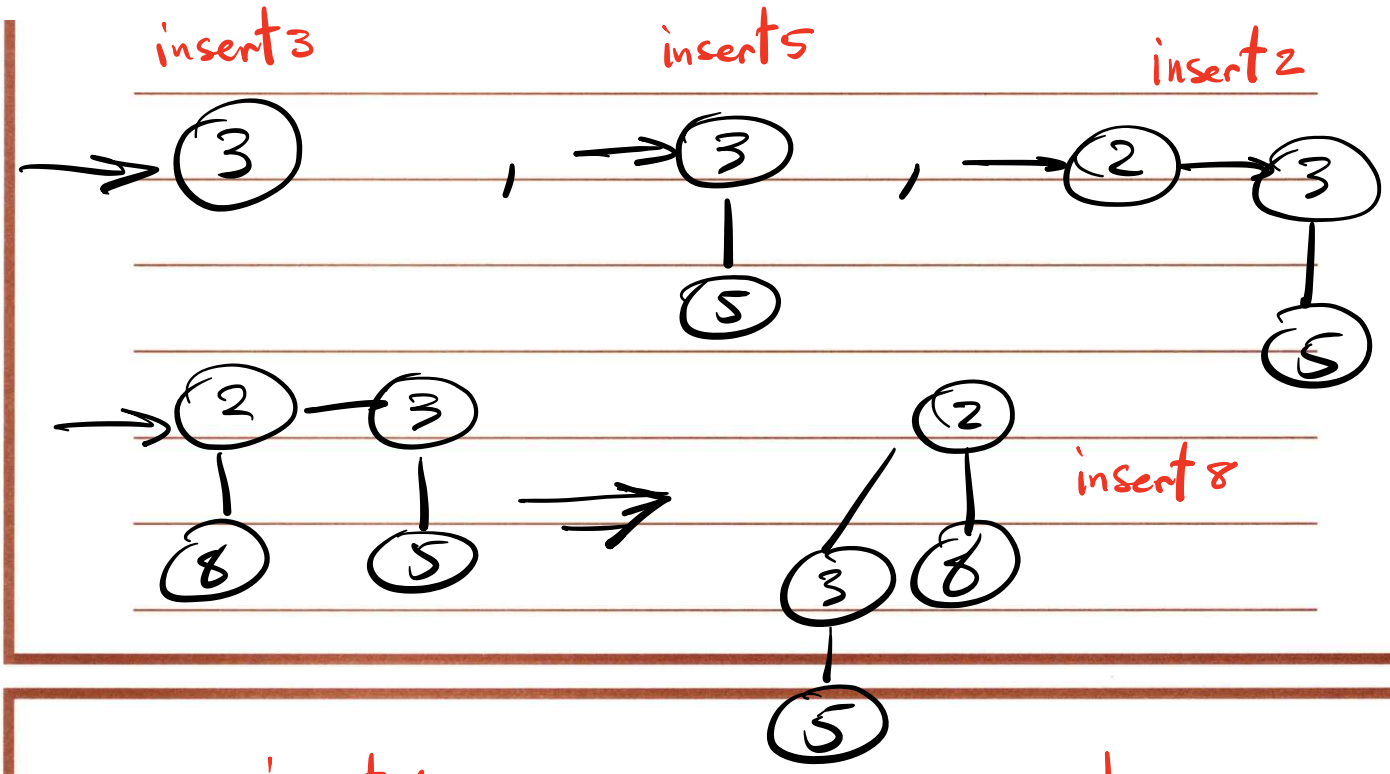


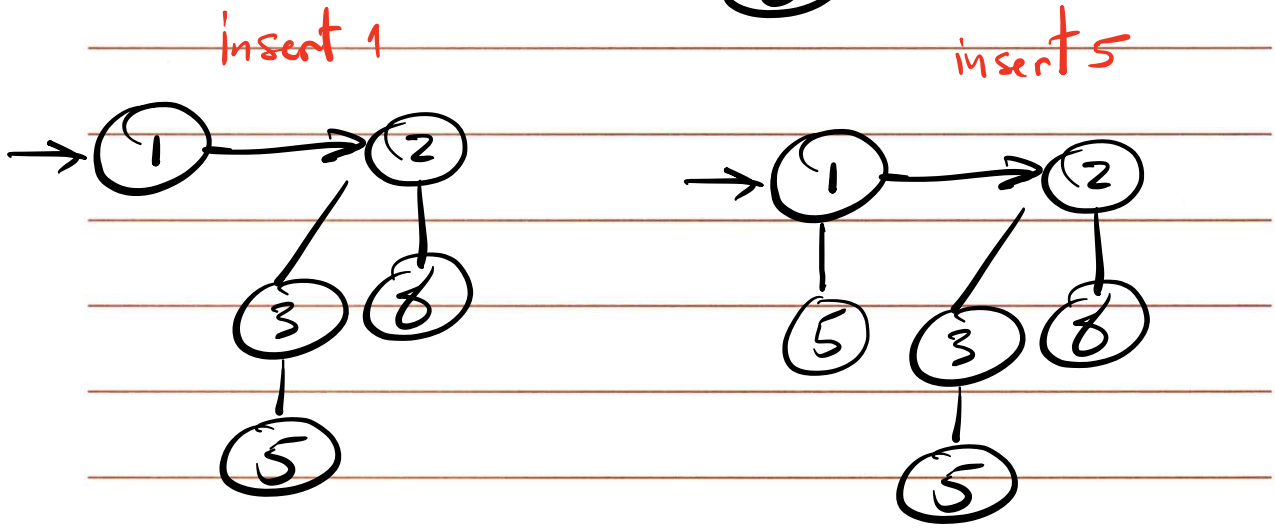
5. (a): Suppose we are given an instance of the Minimum Spanning Tree problem on a graph G. Assume that all edges costs are distinct. Let T be a minimum spanning tree for this instance. Now suppose that we replace each edge cost c_e by its square, c_e^2 thereby creating a new instance of the problem with the same graph but different costs. Prove or disprove: T is still a MST for this new instance.

(b): Consider an undirected graph G = (V, E) with distinct nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G. Now suppose each edge weight is increased by 1: the new weights are $c'_e = c_e + 1$. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

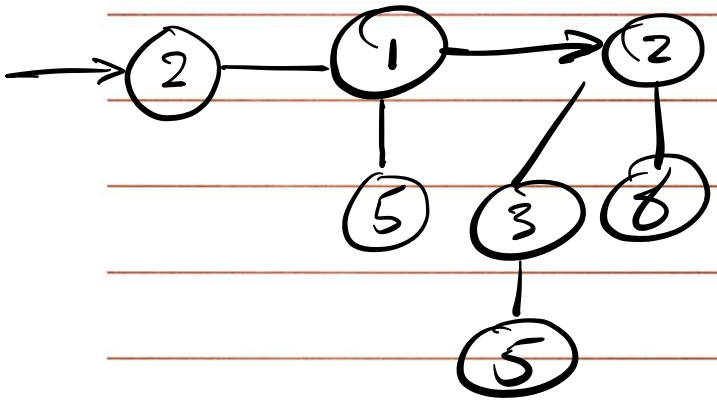
4. Given a sequence of numbers: 3, 5, 2, 8, 1, 5, 2, 7

a. Draw a binomial heap by inserting the above numbers reading them from left to right
b. Show a heap that would be the result after the call to deleteMin() on this heap

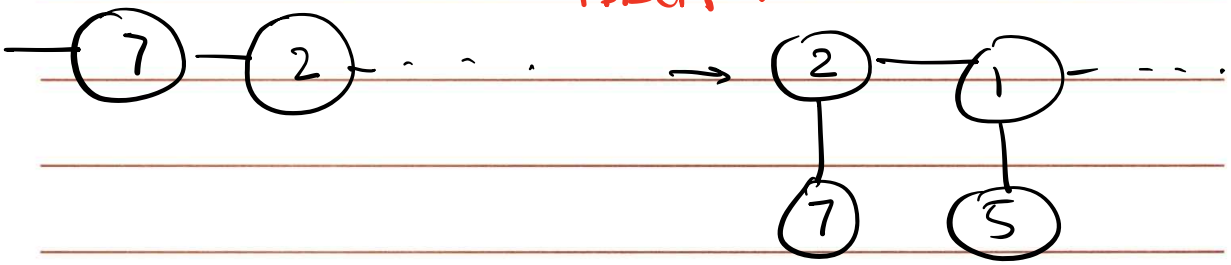


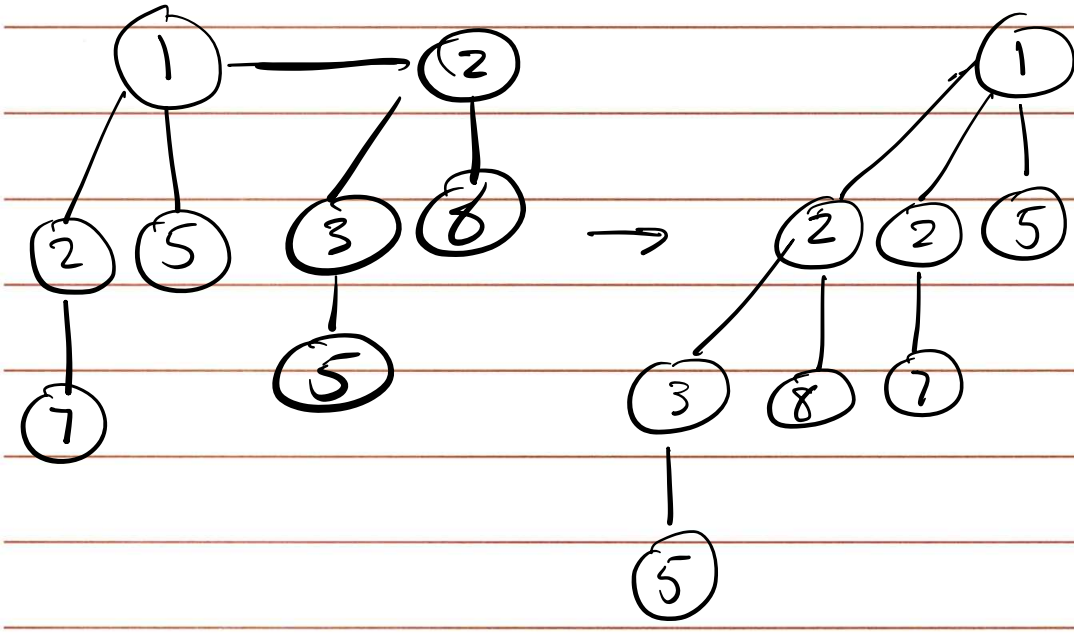


insert 2

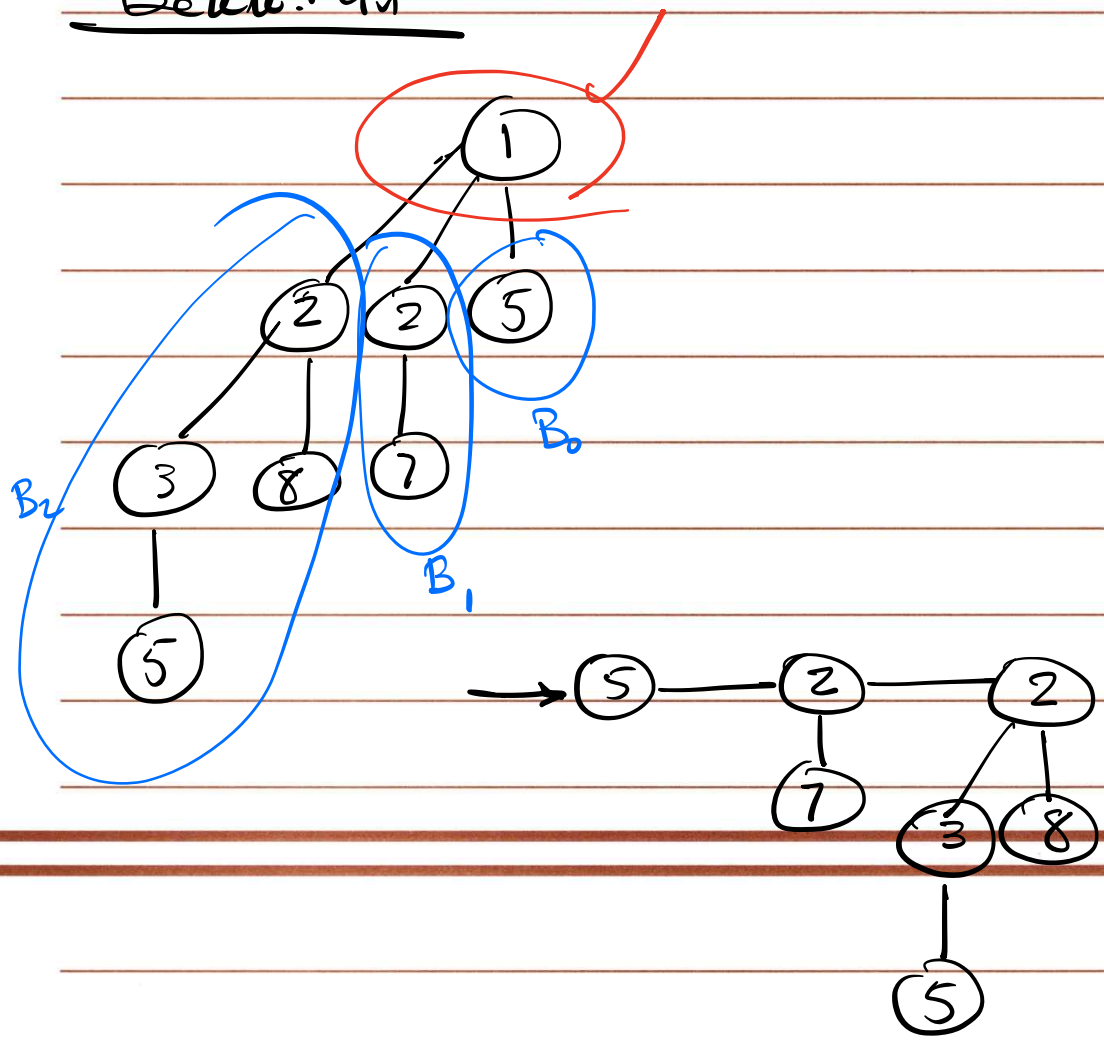


insert 7





Delete Min



Discussion 4

1. Hardy decides to start running to work in San Francisco city to get in shape. He prefers a route that goes entirely uphill and then entirely downhill so that he could work up a sweat uphill and get a nice, cool breeze at the end of his run as he runs faster downhill. He starts running from his home and ends at his workplace. To guide his run, he prints out a detailed map of the roads between home and work with k intersections and m road segments (any existing road between two intersections). The length of each road segment and the elevations at every intersection are given. Assuming that every road segment is either fully uphill or fully downhill, give an efficient algorithm to find the shortest path (route) that meets Hardy's specifications. If no such path meets Hardy's specifications, your algorithm should determine this. Justify your answer.

Solution: Use a directed graph to represent the map of the city. The edge direction will indicate the uphill direction of the road segment. Then

- Run Dijkstra's algorithm with Home as the starting point. This will find us the shortest paths and shortest distances from Home to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S_1.
- Run Dijkstra's algorithm with Work as the starting point. This will find us the shortest paths and shortest distances from Work to all nodes in the graph that can be reached using uphill edges only. The node set that can be reached using such uphill paths will be called S_2.
- $S = S_1 \cap S_2$
- If nodes corresponding to Work or Home fall into S remove them
- If S is null then there is no such solution, otherwise, for each node *i* in S, add up the shortest uphill distance from Home to *i* and shortest uphill distance from Work to *i*. Select the node *j* in S that gives us the smallest total distance. The path we are interested in will be the shortest uphill path from Home to *j* followed by the shortest downhill path from *j* to Work.

2. You are given a graph representing the several career paths available in industry. Each node represents a position and there is an edge from node *v* to node *u* if and only if *v* is a prerequisite for *u*. Top positions are the ones which are not prerequisites for any positions. The cost of an edge *(v, u)* is the effort required to go from one position *v* to position *u*. Ivan wants to start a career and achieve a top position with minimum effort. Using the given graph can you provide an algorithm with the same run time complexity as Dijkstra's? You may assume the graph is a DAG.

Solution: Add a new node S and connect S to all entry level positions with zero edges costs. Run Dijkstra's starting from S and find all shortest paths to all nodes in the graph. Find the top position *t* with lowest distance from S. The career path we are looking for will be on the shortest path from S to *t*.

3. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations.

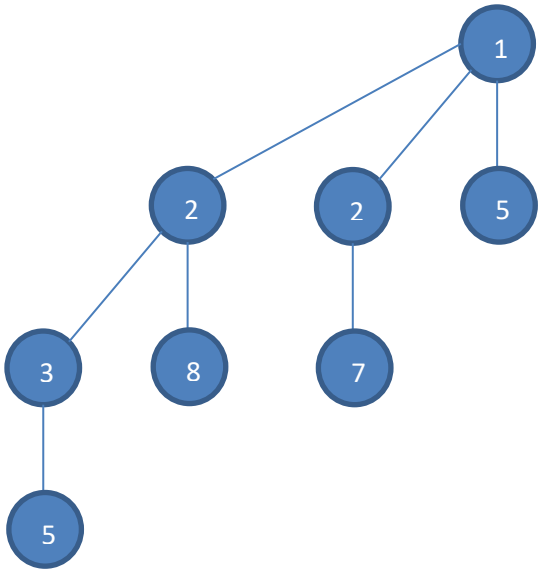
Implementation of the queue using two stacks:
Enqueue – Push element enqueued onto stack A
Dequeue – If stack B is not empty, Pop top element from B, otherwise, Pop all elements from A and Push them onto B one at a time. Then Pop the top element from B.

Solution 1 : Aggregate Method
A worst-case sequence of operations occurs when we do n Enqueues and one Dequeue. So there will be a total of n+1 operations involved.
T(n+1) = total time to Push n elements onto A + total time to Pop all elements from A + total time to Push all elements onto B + time to Pop top element from B
T(n+1) = n + n + n + 1 = 3n + 1
Average cost of the operations = T(n+1) / (n+1) = (3n+1) / (n+1)
When n -> ∞ , (3n+1) / (n+1) = 3 = O(1)
O(1) is the amortized cost of both Enqueue and Dequeue operations

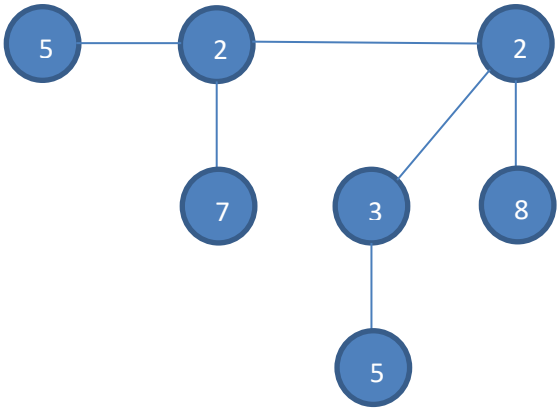
Solution 1 : Accounting Method
Charge Enqueue more than its actual cost so we can save credit for the expensive Dequeue operation. Since each element Pushed onto A needs to be Popped from A and then Pushed onto B, we can charge Enqueue 3 units instead of its actual cost (which is 1 unit). We can charge Dequeue 1 unit.
Check to see if there will be enough credit to pay for a worst case sequence of operations:
n Enqueue results in a total charge of 3n. When we subtract the actual cost of 1n, we are left with 2n units of credit. Dequeue is also charged 1 unit so we have a total to 2n+1 to pay for the Dequeue operation. Let's see if that is enough. The actual cost of Dequeue will be n + n + 1 which is exactly 2n+1. The charges are therefore sufficient.
Amortized cost of Enqueue = 3 = O(1)
Amortized cost of Dequeue = 1 = O(1)

4. Given a sequence of numbers: 3, 5, 2, 8, 1, 5, 2, 7

a. Draw a binomial heap by inserting the above numbers reading them from left to right



b. Show a heap that would be the result after the call to deleteMin() on this heap



5. (a): Suppose we are given an instance of the Minimum Spanning Tree problem on a graph G. Assume that all edges costs are distinct. Let T be a minimum spanning tree for this instance. Now suppose that we replace each edge cost c_e by its square, c_e^2 thereby creating a new instance of the problem with the same graph but different costs. Prove or disprove: T is still a MST for this new instance.

Solution: T may no longer be the MST since if there are any negative cost edges in G, then those edges will have positive costs after squaring the edge costs. So, the order of edges (based on their weight) could change.

(b): Consider an undirected graph G = (V, E) with distinct nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G. Now suppose each edge weight is increased by 1: the new weights are $c'_e = c_e + 1$. Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

Solution: Since we are adding a constant to all edge weights the order of edges (based on their weight) will NOT change. So, for example, Kruskal's algorithm will pick edges exactly in the same order since the graph topology has not changed.

CSCI 570 - Fall 2021 - HW 4

Due: Sep 23rd

Section 1: Heaps

Reading Assignment: Kleinberg and Tardos, Chapter 2.5.

Problem 1

[10 points] Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):

- Find median takes $\mathcal{O}(1)$ time
- Insert takes $\mathcal{O}(\log n)$ time

Do the following:

1. Describe how your data structure will work.
2. Give algorithms that implement the Find-Median() and Insert() functions.

(**Hint:** Read this only if you really need to. Your Data Structure should use a min-heap and a max-heap simultaneously where half of the elements are in the max-heap and the other half are in the min-heap.)

Solution. We use the dn/2e smallest elements to build a max-heap and use the remaining bn/2c elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time $\mathcal{O}(1)$ (we assume the case of even n, median is $n/2$-th element when elements are sorted in increasing order).

Insert() algorithm: For a new element x

- Initialize len of maxheap (maxlen) and len of minheap (minlen) to 0.
- Compare x to the current root of max-heap.
- If $x <$ median, we insert x into the max-heap.Maintain length of maxheap say maxlen, and every time you insert element into maxheap increase maxlen by 1. Otherwise, we insert x into the min-heap, and increase length of minheap say minlen by 1. This takes $\mathcal{O}(\log n)$ time in the worst case.

- If size(maxHeap) > size(minHeap)+1, then we call Extract-Max() on max-heap, and decrease maxlen by 1 of and insert the extracted value into the min-heap and increase minlen by 1. This takes $\mathcal{O}(\log n)$ time in the worst case.
- Also, if size(minHeap)>size(maxHeap), we call Extract-Min() on min-heap, decrease minlen by 1 and insert the extracted value into the max-heap and increase maxlen by 1. This takes $\mathcal{O}(\log n)$ time in the worst case.

Find-Median() algorithm:

- If (maxlen+minlen) is even: return (sum of roots of max heap and min heap)/2 as median
- Else if maxlen>minlen: return root of max heap as median
- Else: return root of min heap as median

■

Problem 2

[10 points] There is a stream of integers that comes continuously to a small server. The job of the server is to keep track of k largest numbers that it has seen so far. The server has the following restrictions:

- It can process only one number from the stream at a time, which means it takes a number from the stream, processes it, finishes with that number and takes the next number from the stream. It cannot take more than one number from the stream at a time due to memory restriction.
- It has enough memory to store up to k integers in a simple data structure (e.g. an array), and some extra memory for computation (like comparison, etc.).
- The time complexity for processing one number must be better than $\mathcal{O}(k)$. Anything that is $\mathcal{O}(k)$ or worse is not acceptable.

Design an algorithm on the server to perform its job with the requirements listed above.

Solution. Use a binary min-heap on the server.

1. Do not wait until k numbers have arrived at the server to build the heap, otherwise you would incur a time complexity of $\mathcal{O}(k)$. Instead, build the heap on-the-fly, i.e. as soon as a number arrives, if the heap is not full, insert the number into the heap and execute Heapify(). The first k numbers are obviously the k largest numbers that the server has seen.

2. When a new number x arrives and the heap is full, compare x to the minimum number r in the heap located at the root, which can be done in $\mathcal{O}(1)$ time. If $x \leq r$, ignore x. Otherwise, run Extract-min() and insert the new number x into the heap and call Heapify() to maintain the structure of the heap.

3. Both Extract-min() and Heapify() can be done in $\mathcal{O}(\log k)$ time. Hence, the overall complexity is $\mathcal{O}(\log k)$.

■

Section 2: MST

Reading Assignment: Kleinberg and Tardos, Chapter 4.5.

Problem 3

[10 points] Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution. Proof by contradiction:

1. If T_1, T_2 are Distinct, there must be at least one edge that's in T_1 but not in T_2 (and vice versa) since one MST cannot contain the other. let e_1 be the minimum weight edge among all such edges that are in exactly one but not both.

2. $T_2 + e_1$ contains a cycle, say C. C must contain e_1 since C wasn't present in T_2. Further, the edges in C other than e_1 cannot all be in T_1 since otherwise, T_1 would contain the cycle C. Hence, there's an edge in C, say e_2, which is in T_2, but not in T_1. However, since e_1 has the minimum weight among edges that are in exactly one, but not both, $w(e_1) < w(e_2)$. The strict inequality is because we have all edge weights distinct.

3. Note that $T = T_2 \cup \{e_1\} \backslash \{e_2\}$ is a spanning tree. The total weight of T is smaller than the total weight of T_2, but this is a contradiction, since we have supposed that T_2 is a minimum spanning tree.

■

Problem 4

[10 points] Let us say that a graph $G = (V, E)$ is a near tree if it is connected and has at most $n+8$ edges, where $n = |V|$. Give an algorithm with running time $\mathcal{O}(n)$ that takes a near tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all edge costs are distinct.

Solution. 1. To do this, we apply the Cycle Property nine times. That is, we perform BFS until we find a cycle in the graph G, and then we delete the heaviest edge on this cycle.

2. We have now reduced the number of edges in G by one while keeping G connected and (by the Cycle Property) not changing the identity of the minimum spanning tree.

3. If we do this a total of nine times, we will have a connected graph H with $n-1$ edges and the same minimum spanning tree as G. But H is a tree, and so in fact it is the minimum spanning tree.

4. The running time of each iteration is $\mathcal{O}(m+n)$ for the BFS and subsequent check of the cycle to find the heaviest edge; here $m \leq n+8$, so this is $\mathcal{O}(n)$.

■

Section 3: Shortest Path

Reading Assignment: Kleinberg and Tardos, Chapter 4.4.

Problem 5

[20 points] Given a connected graph $G = (V, E)$ with positive edge weights. In V, s and t are two nodes for shortest path computation, prove or disprove with explanations:

1. If all edge weights are unique, then there is a single shortest path between any two nodes in V.

2. If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k.

3. If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k.

4. If each edge's weight is replaced by its square, i.e., w to w^2, then the shortest path between s and t will be the same as before but with different costs.

Solution. 1. False. Counter example: (s, a) with weight 1, (a, t) with weight 2 and (s, t) with weight 3. There are two shortest path from s to t though the edge weights are unique.

2. False. Counter example: suppose the shortest path $s \to t$ consist of two edges, each with cost 1, and there is also an edge $e = (s, t)$ in G with cost(e)=3. If now we increase the cost of each edge by 2, e will become the shortest path (with the total cost of 5).

3. False.

   - Only true when we have the assumption that after decreasing all edge weights are still positive (however we don't have this in the problem). For any two nodes s, t, assume that $P_1, \ldots, P_k$ are all the paths from s to t. If e belongs to P_i then the path cost decrease by k, otherwise the path cost unchanged. Hence all paths from s to t will decrease by at most k. As shortest path is among them, then the shortest path cost will decrease by at most k.
   - When 1) there is cycle in the graph, 2) and there is a path from s to t that goes through that cycle, 3) and after decreasing, the sum of edge weights of that cycle becomes negative, then the shortest path from s to t will go over the cycle for infinite times, ending up with infinite path cost, hence not "decrease by at most k".

4. False. Counter example: 1) suppose the original graph G composed of $V = \{A, B, C, D\}$ and $E : (A \to B) = 100, (A \to C) = 51, (B \to D) = 1, (C \to D) = 51$, then the shortest path from A to D is $A \to B \to D$ with length 101. 2) After squaring this path length become $100^2 + 1^2 = 10001$. However, $A \to C \to D$ has path length $51^2 + 51^2 = 5202 < 10001$ Thus $A \to C \to D$ become the new shortest path from A to D.

■

Problem 6

[16 points] Consider a directed, weighted graph G where all edge weights are positive. You have one Star, which allows you to change the weight of any one edge to zero. In other words, you may change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find a lowest-cost path from node s to node t, given that you may set one edge weight to zero.

Note: you will receive 10 points if your algorithm is efficient. This means your method must do better than the naive solution, where the weight of each node is set to 0 per time and the Dijkstra's algorithm is applied every time for lowest-cost path searching. You will receive full points (16 points) if your algorithm has the same run time complexity as Dijkstra's algorithm.

Solution. Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta(u, v)$. Now, try setting each edge to zero. For each edge $(u, v) \in E$, consider the path $s \rightsquigarrow u \to v \rightsquigarrow t$. If we set $w(u, v)$ to zero, the path length is $\delta(s, u) + \delta(v, t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \rightsquigarrow u \to v \rightsquigarrow t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $\mathcal{O}(E)$ time to iterate through the edges and find the optimal edge to take for free. Thus the total running time is the same as that of Dijkstra:

- If we implement Dijkstra's algorithm with Fibonacci heap
  - time complexity of both the Dijkstra's algorithm and the proposed method is $\mathcal{O}(E + V \log V)$.
  - the corresponding time complexity of naive is $\mathcal{O}(E(E + V \log V))$
- If we implement Dijkstra's algorithm with a binary heap
  - the complexity of Dijkstra's algorithm is $\mathcal{O}((E + V) \log V)$, so that the proposed method is $\mathcal{O}(2(E+V) \log V+E)$, since $E = \mathcal{O}(E \log V)$, then it has same time complexity as Dijkstra's algorithm, i.e., $\mathcal{O}((E+V) \log V)$.
  - the corresponding time complexity of naive is $\mathcal{O}(E(E + V) \log V)$

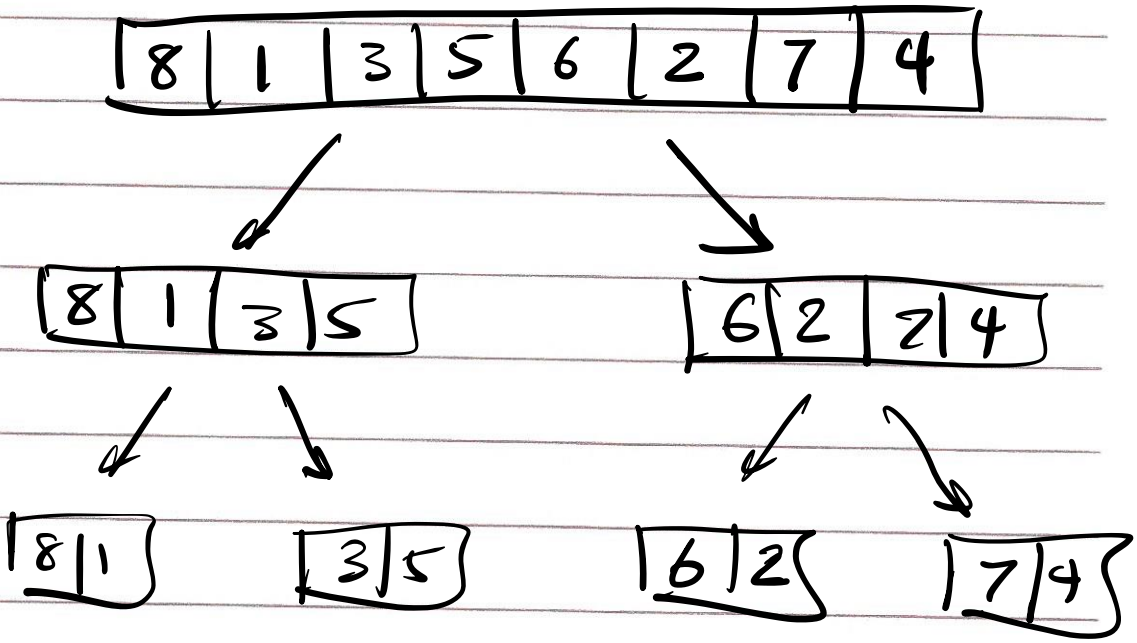
■

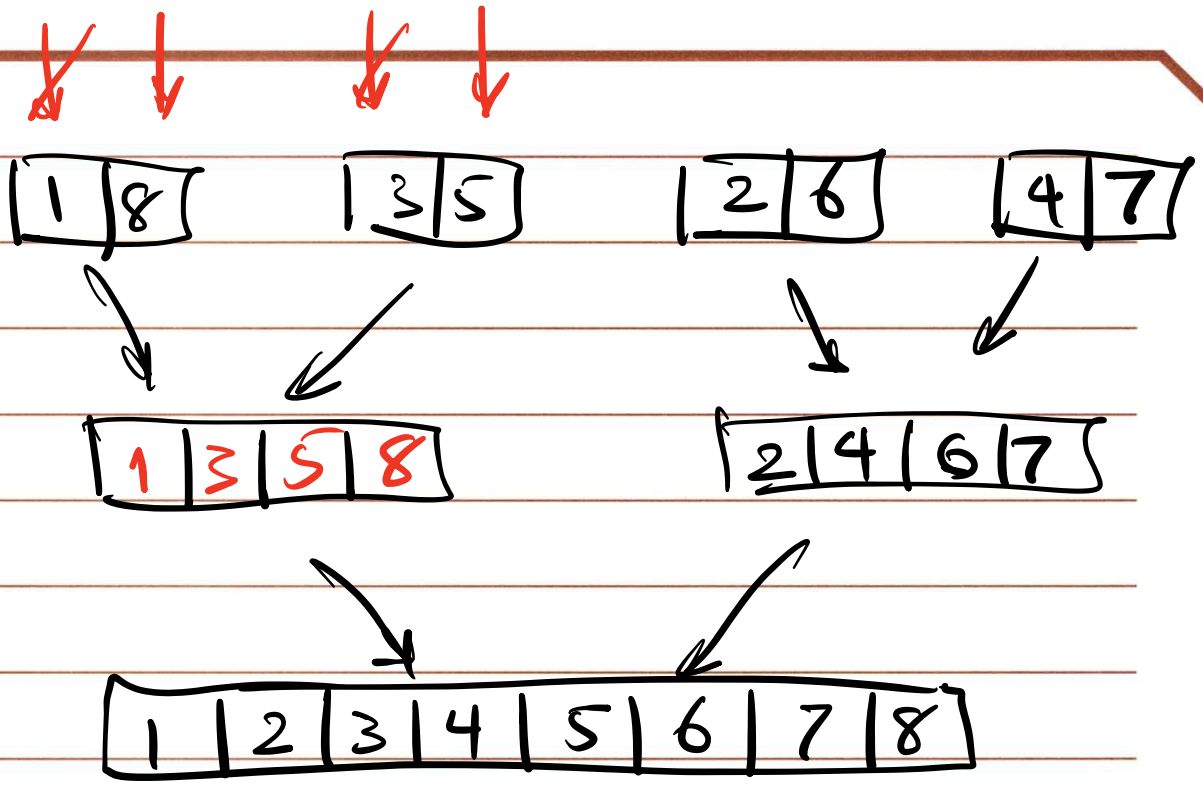
Divide & Conquer

1- Divide problem into n subproblems

2- Conquer: i.e. solve the subproblems recursively, or if trivial solve the problem itself

3- Combine the solution to the subproblems




1 8
3 5
2 6
4 7
1 3 5 8
2 4 6 7
1 2 3 4 5 6 7 8

```
1                MERGE-SORT(A, p, r)
2                if p < r then
3   Divide           q = ⌊(p+r)/2⌋
4   Conquer          MERGE-SORT(A, p, q)
5                    MERGE-SORT(A, q+1, r)
6   Combine          MERGE(A, p, q, r)
7                end if
```

Analyzing Merge-sort

Divide – Takes $O(1)$

Conquer – If the original problem takes $T(n)$ time, the two subproblems take $2 \cdot T(n/2)$

Combine – Takes $O(n)$ on array size of n.

Recurrence equation for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T(n/2) + O(n) + O(1) & \end{cases}$$

Conquer

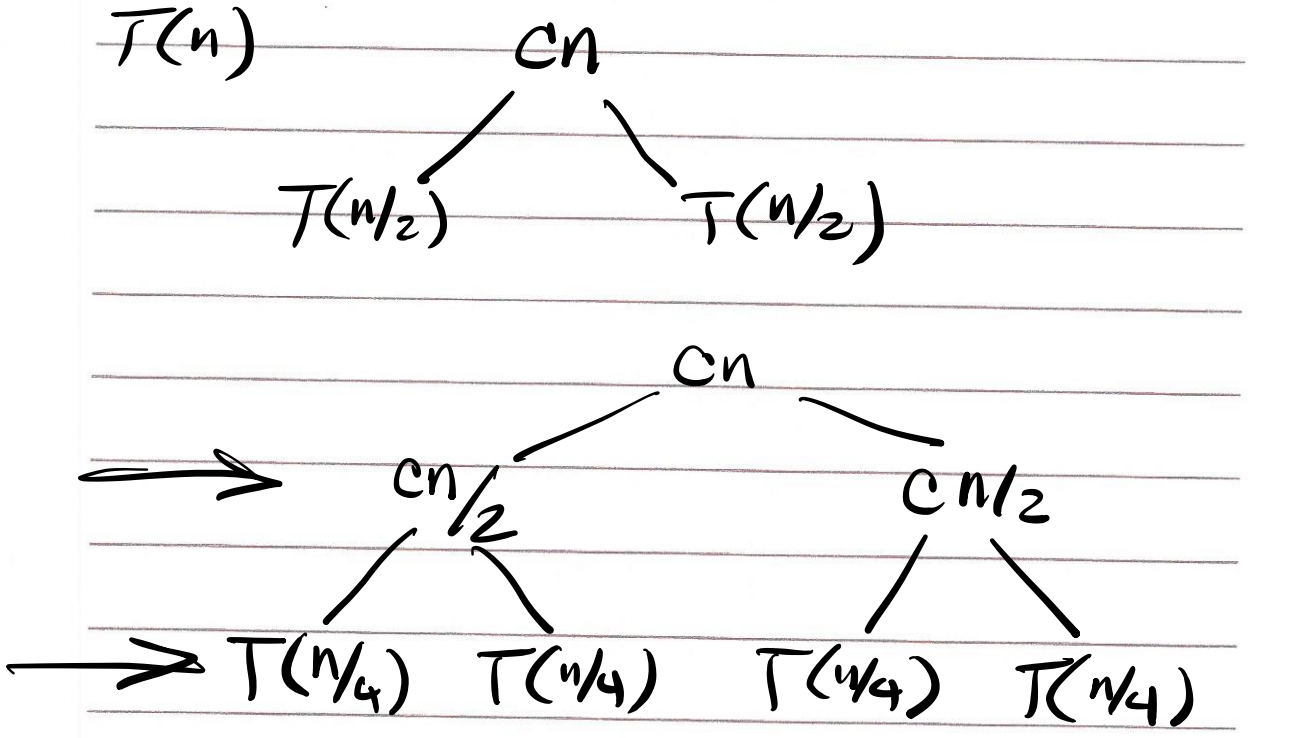
Combine

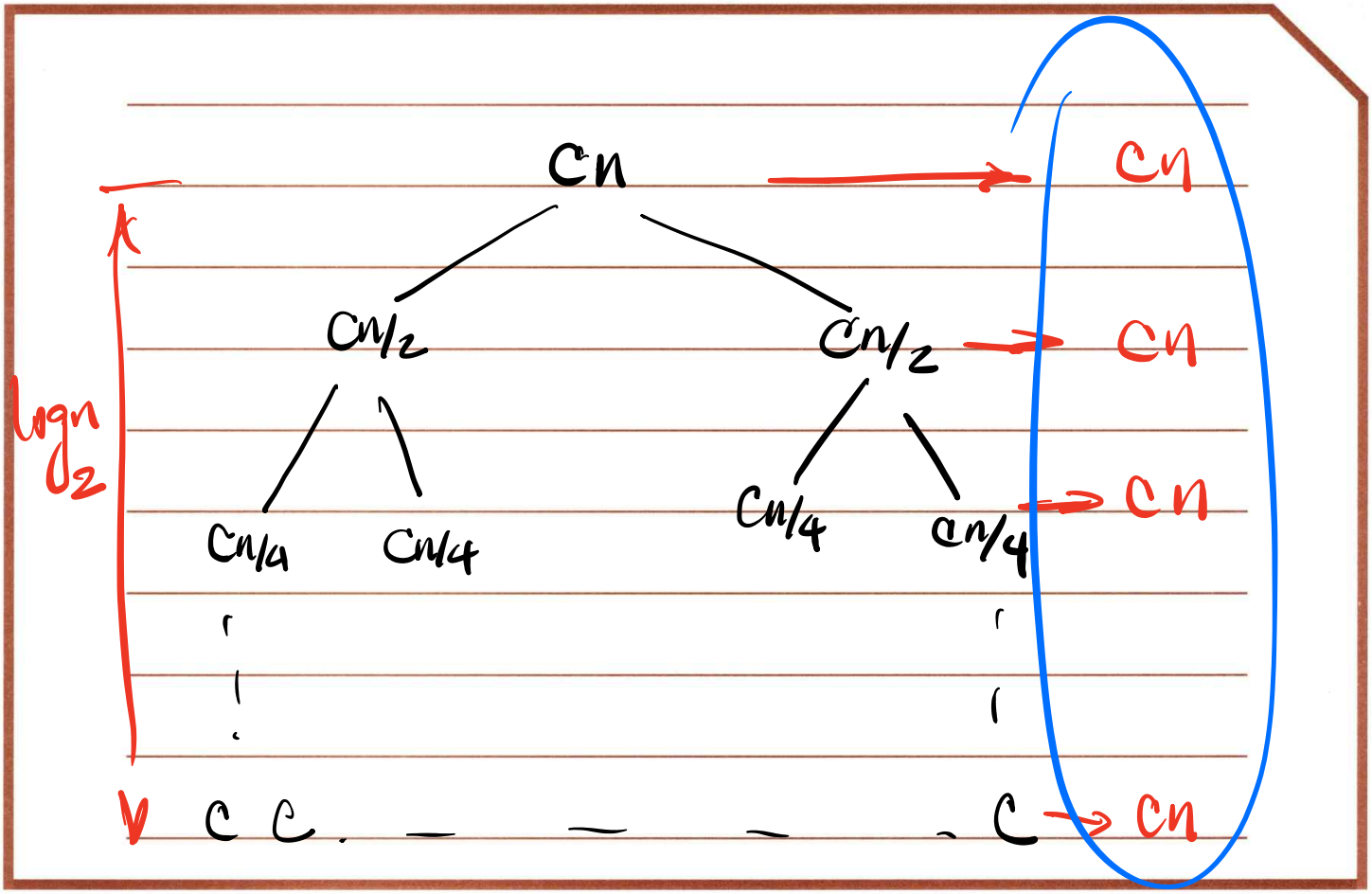
Divide

in general, our recurrence equation for a D&C solution will look like:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \end{cases}$$

- a: # of subproblems
- n/b: size of the subproblems
- $D(n)$: time to divide
- $C(n)$: time to combine





total cost = cn log n

Worst Case Complexity = O(n log n)

Master Method

It is a cookbook method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

- where $a \geq 1$, $b \geq 1$ are constants

- $f(n)$ is an asymptotically positive function.

Master Theorem

- Given the above definition of the recurrence relation, $T(n)$ can be bounded asymptotically as follows:

1- If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$,

then $T(n) = \Theta(n^{\log_b a})$

2- If $f(n) = \Theta(n^{\log_b a})$ then

$$T(n) = \Theta(n^{\log_b a} \lg n)$$

3- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then

$$T(n) = \Theta(f(n))$$

$f(n) \; ? \; n^{\log_b a}$

Case#3

$f(n) = n^{1.0001}$

$n^{\log_b a} = n$

$f(n) = n \lg n$

$n^{\log_b a} = n$

Diff is NOT polynomial

This is not a Case#3

in general if $f(n) = \Theta(n^{\log_b a} \lg^k n)$

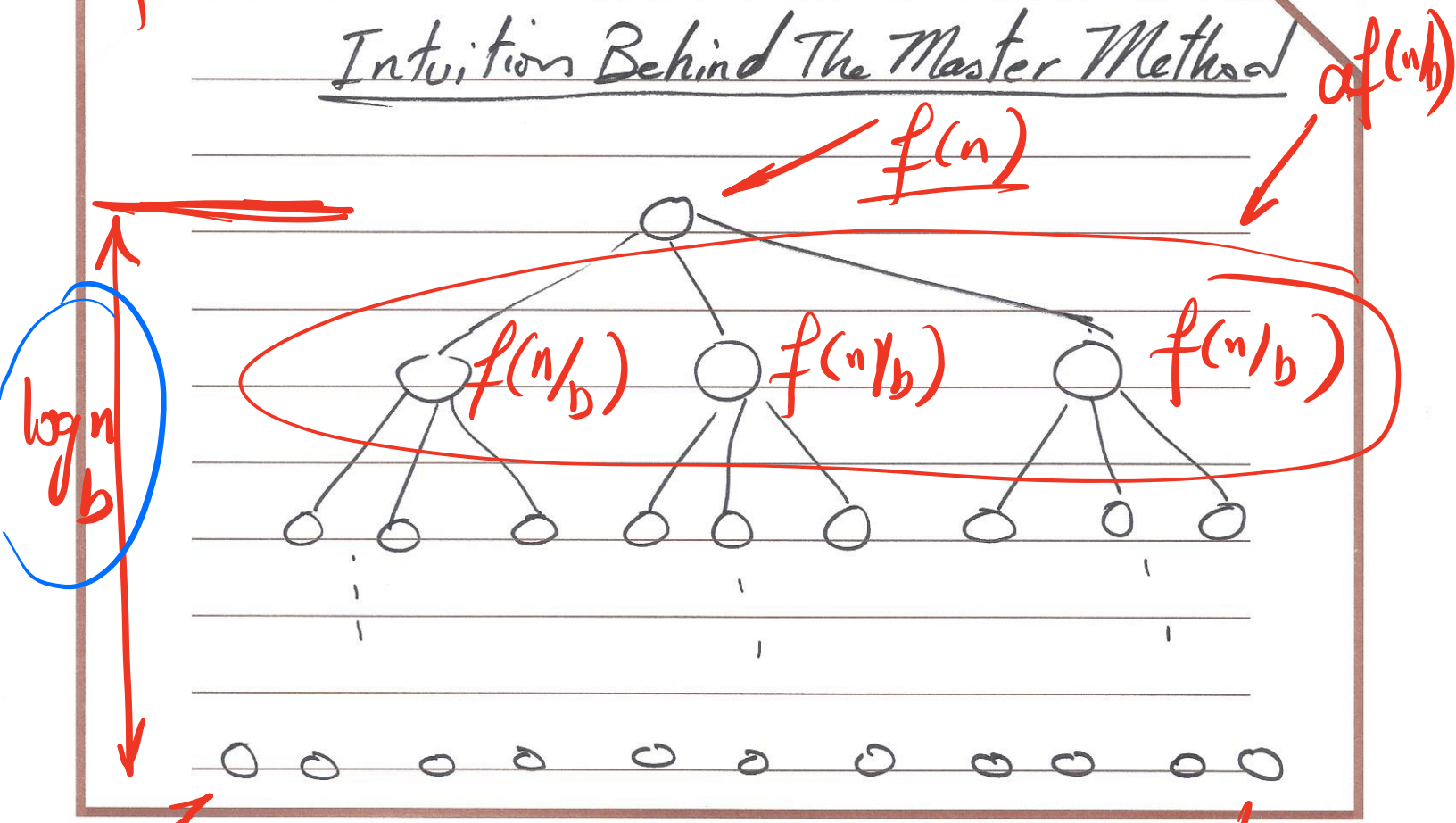
where $k \geq 0$

Then

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

$f(n) + \alpha f(n) + \alpha^2 f(n) \Rightarrow \Theta(f(n))$

Intuition Behind The Master Method



# of nodes at the bottom of tree $= a^{\log_b n}$

$$a^{\log_b n} = n^{\log_b a}$$

$$n^{\log_b q} + \frac{1}{a} n^{\log_b q} + \frac{1}{a^2} n^{\log_b q} + \cdots + 1$$

$$c\, n^{\log_b q} = \Theta(n^{\log_b q})$$

$$\sum x + \alpha x + \alpha^2 x + \cdots = cx$$

if $\alpha < 1$ $\Rightarrow$

Case 1: Complexity driven by the no. of leaf nodes in the recursion tree.

Case 3: Complexity driven by the cost of the root node in the recursion tree

Case 2: Cost of operations are the same at every level of the recursion tree.

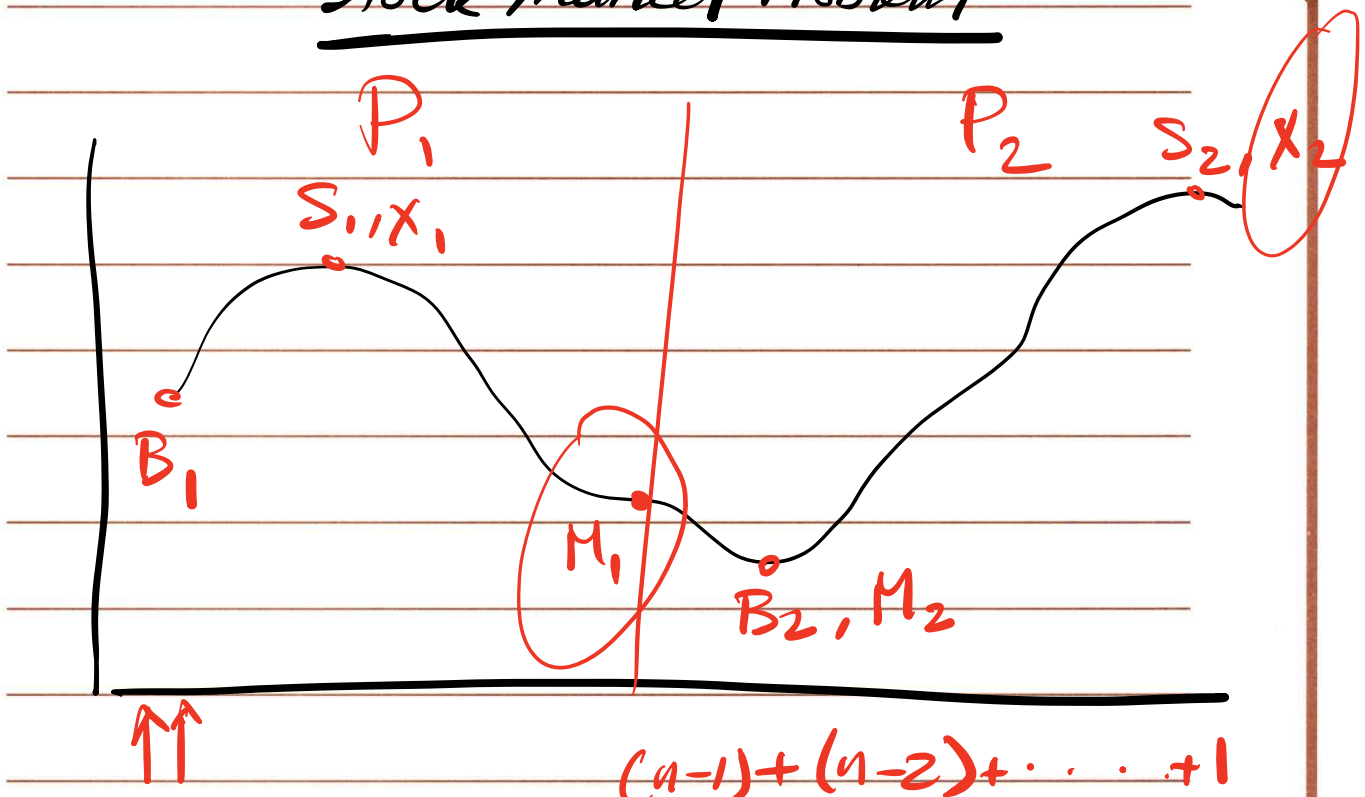
Complexity of Merge Sort

$f(n) = O(n^{\log_b a})$

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Case #2 $\rightarrow T(n) = \Theta(n \lg n)$

Stock Market Problem



Case #1

Buy in P_1 & Sell in P_1

$B = B_1$ $\quad M = Min(M_1, M_2)$

$S = S_1$ $\quad X = Max(X_1, X_2)$

Case #2

Buy in P_2 & Sell in P_2

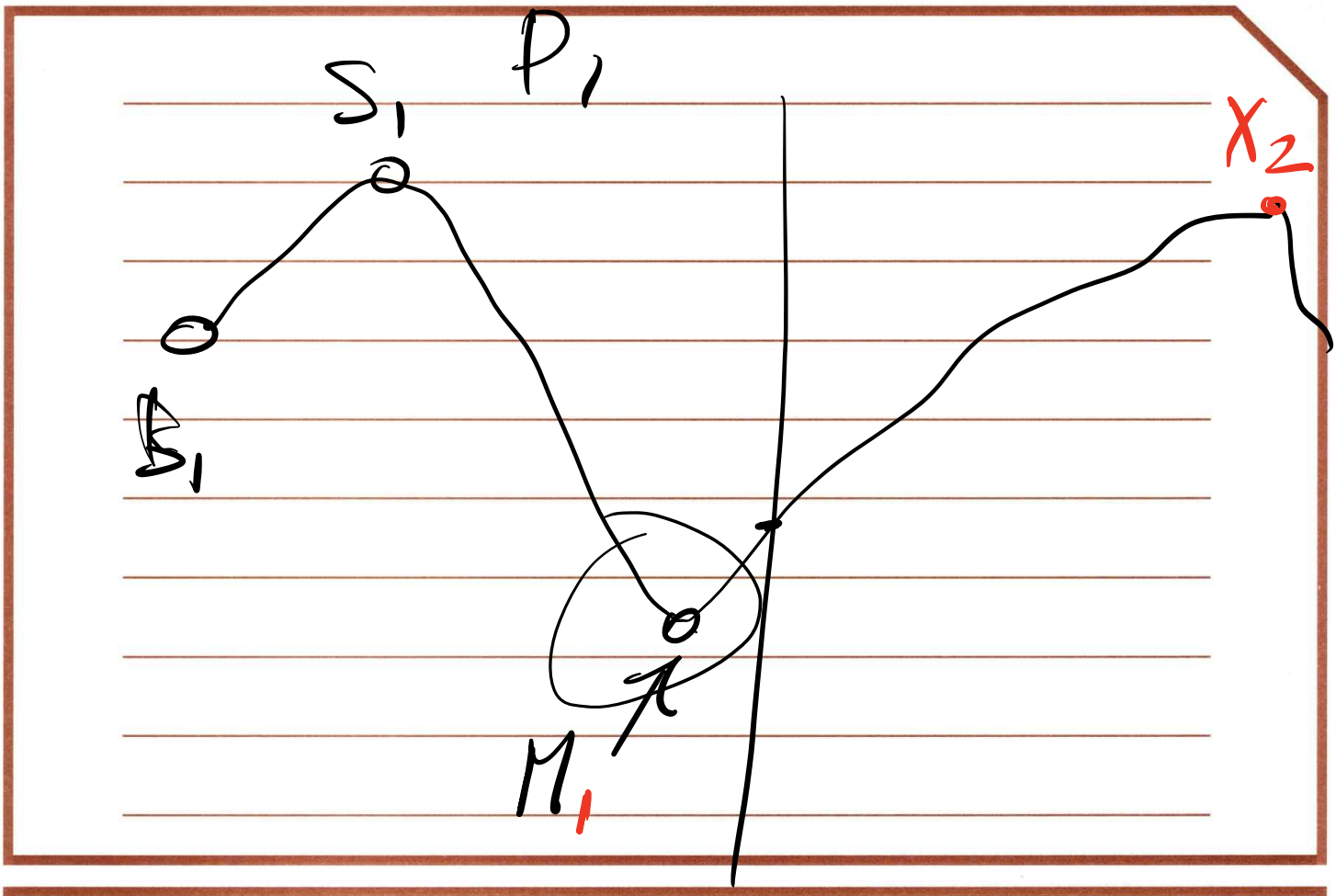
$B = B_2$ $\quad M = Min(M_1, M_2)$

$S = S_2$ $\quad X = Max(X_1, X_2)$

Case #3 Buy in P_1 & Sell in P_2

$B = B_1$ $\quad M = Min(M_1, M_2)$

$S = S_2$ $\quad X = Max(X_1, X_2)$



$\Theta(1)$ $\Theta(1)$

$f(n) = D(n) * C(n) = \Theta(1)$

$n^{\log_b a} = n^{\log_2 2} = n^1$

→ Case # 1 → $T(n) = \Theta(n)$

Dense Matrix Multiplication

$$\underbrace{\left[\underbrace{A}_{n}\right]\left[\underbrace{B}_{n}\right]}_{n} = \underbrace{\left[\underbrace{C}_{n}\right]}_{n}$$

Brute force $\longrightarrow \Theta(n^3)$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Combine step

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

$$f(n) = D(n) + C(n) = \Theta(1) + \Theta(n^2) = \Theta(n^2)$$

$$n^{\log_b q} = n^{\log_2 8} = n^3$$

case #1 $\longrightarrow T(n) = \Theta(n^3)$

Compute 7 $n/2 \times n/2$ intermediate matrices

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})\, B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})\, B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

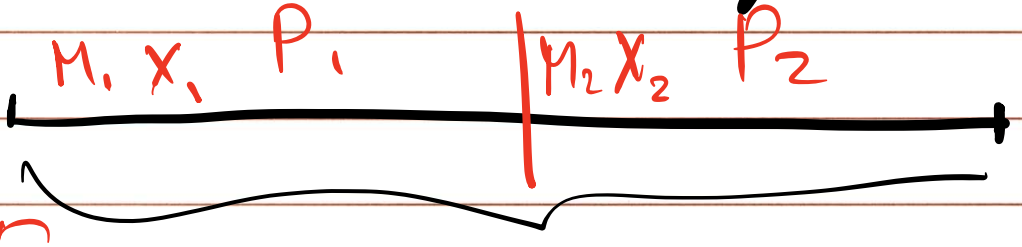
Strassen's Method

$a = 7,\ b = 2, \rightarrow n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

$f(n) = \Theta(n^2)$

Case #1 $\rightarrow T(n) = n^{2.81}$

Finding Min & Max in an unsorted array

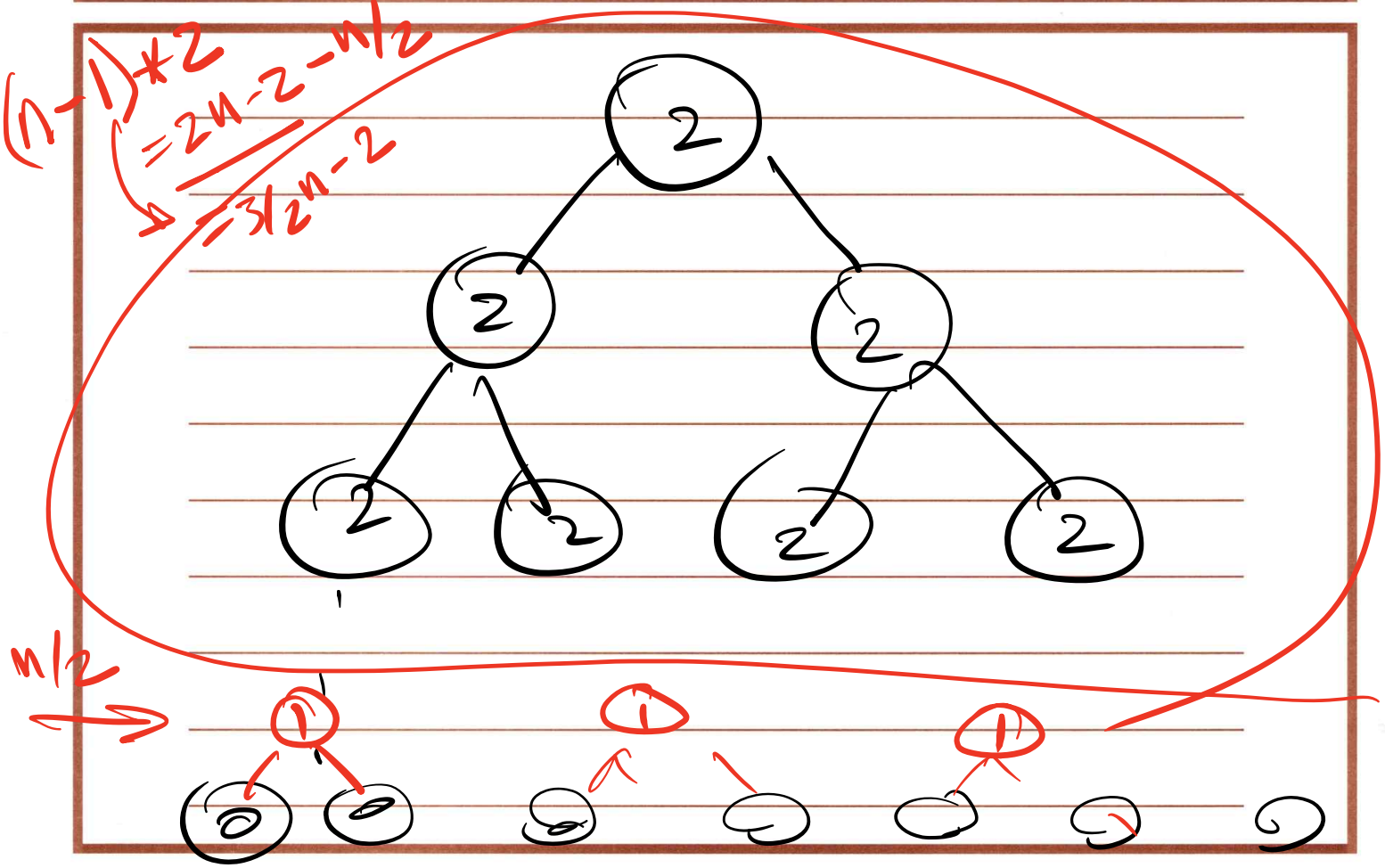


Brute Force:

# of comparisons to find Min = n-1

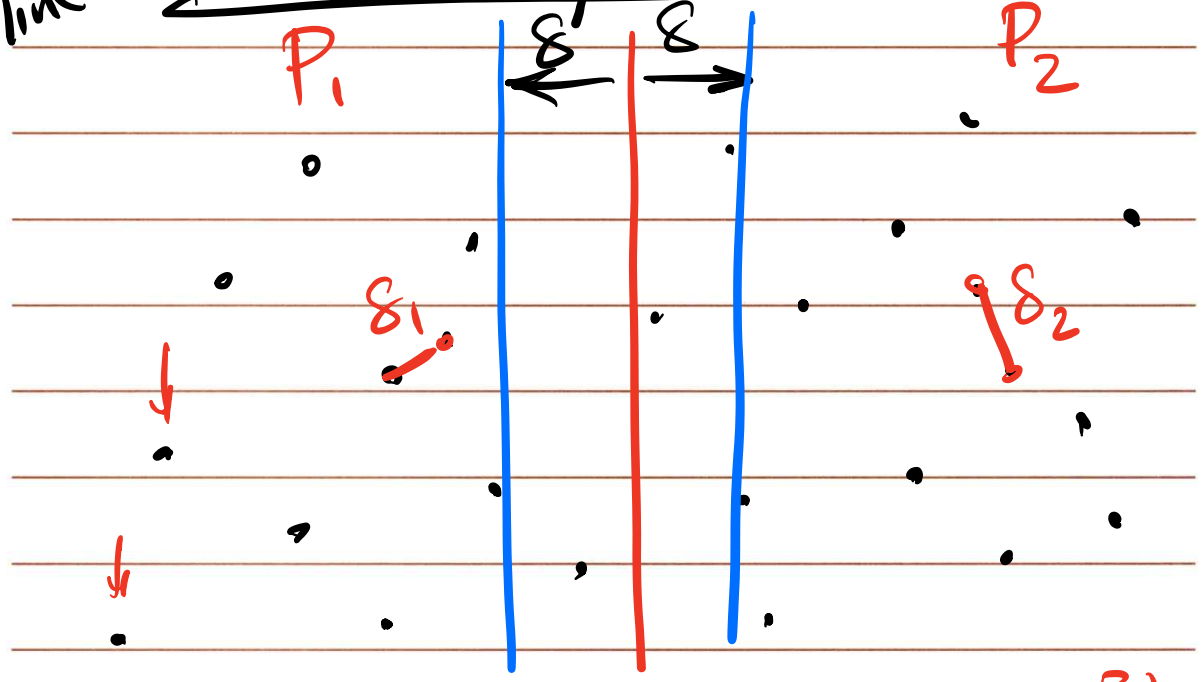
" " " " Max = n-1

$2n-2 = \Theta(n)$



Closest pair of points problem (2D)

$\delta = \min(\delta_1, \delta_2)$

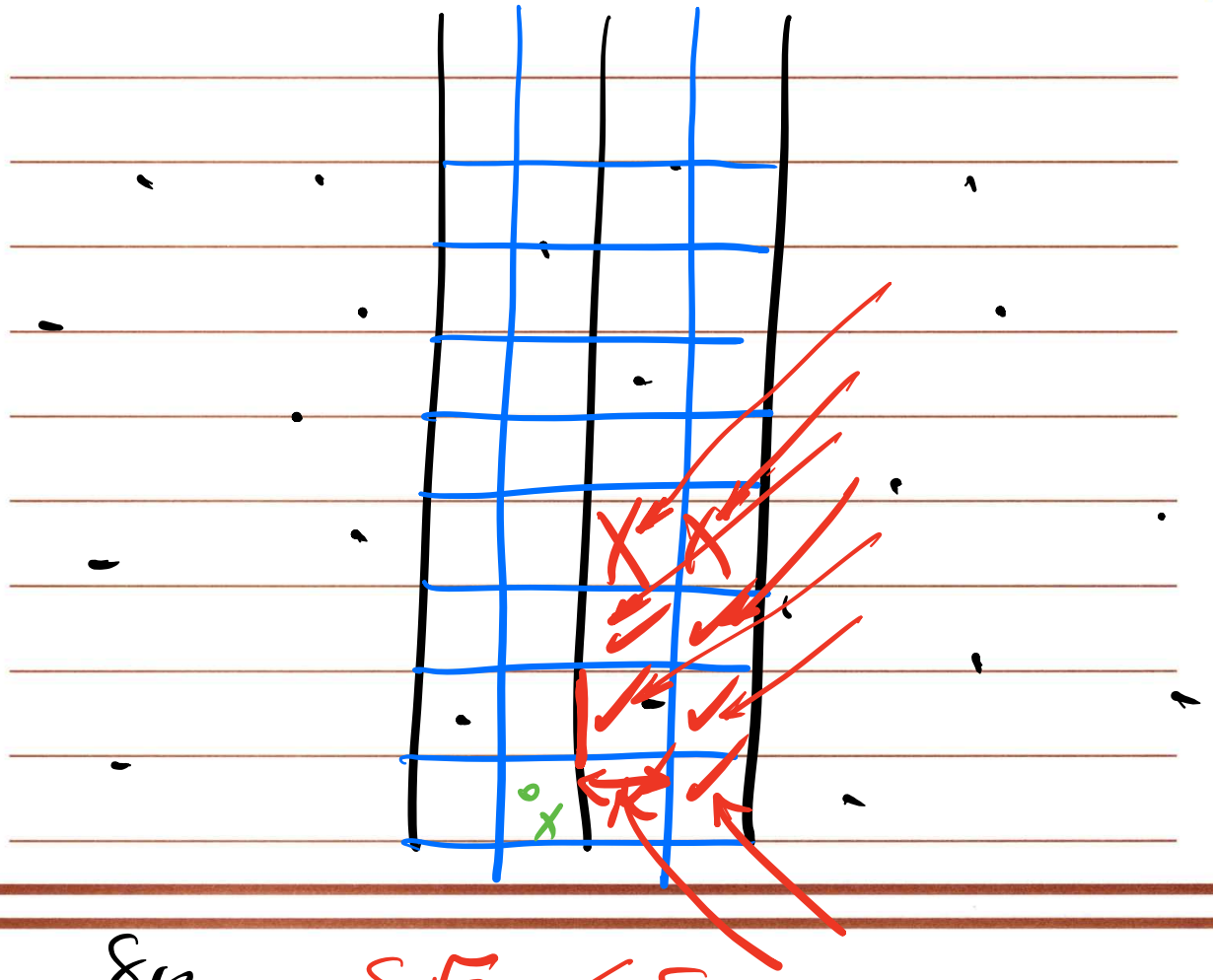


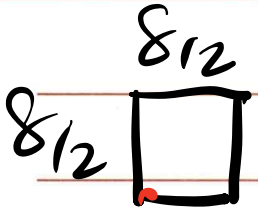
Brute force $\rightarrow (n-1) + (n-2) + \cdots + 1 = \Theta(n^2)$

Case #1 : Both pts are in P_1 ✓

Case #2 : " " " P_2 ✓

Case #3 : one pt is in P_1 & other in P_2





$$\frac{\delta\sqrt{2}}{2} < \delta$$

Implementation

$O(n \log n)$

closest-pair (P)

Construct P_x : list of points sorted by x-coord.

" P_y : list of points sorted by y-coord.

$(p_0, p_1) =$ closest-pair-Rec(P_x, P_y)

---

closest-pair-Rec (P_x, P_y)

if $|P| \leq 3$ then

solve it directly

else

$O(n)$

Divide

$O(1) \rightarrow$ Construct Q_x ... left half of P_x

$O(n) \rightarrow$ " Q_y ... list of points in Q_x sorted by y-coord.

$O(1) \rightarrow$ Construct R_x ... right half of P_x

$O(n) \rightarrow$ " R_y ... list of points in R_x sorted by y-coord.

(q_0, q_1) = closest_pair_Rec (Q_x, Q_y)

(r_0, r_1) = closest_pair_Rec (R_x, R_y)

$O(1)$ $\delta = \min(d(q_0, q_1), d(r_0, r_1))$

$O(n)$ S = set of points in P within distance of δ from L.

Construct S_y ... set of points in S sorted by y-coord.

$O(n)$ for each point $s \in S_y$, compute distance from s to each of next 11 points in S_y. Let (s, s') be pair with min. distance

$O(1)$ if $d(s, s') < \delta$ then
Return (s, s')
else if $d(q_0, q_1) < d(r_0, r_1)$ then
Return (q_0, q_1)
else
Return (r_0, r_1)
endif

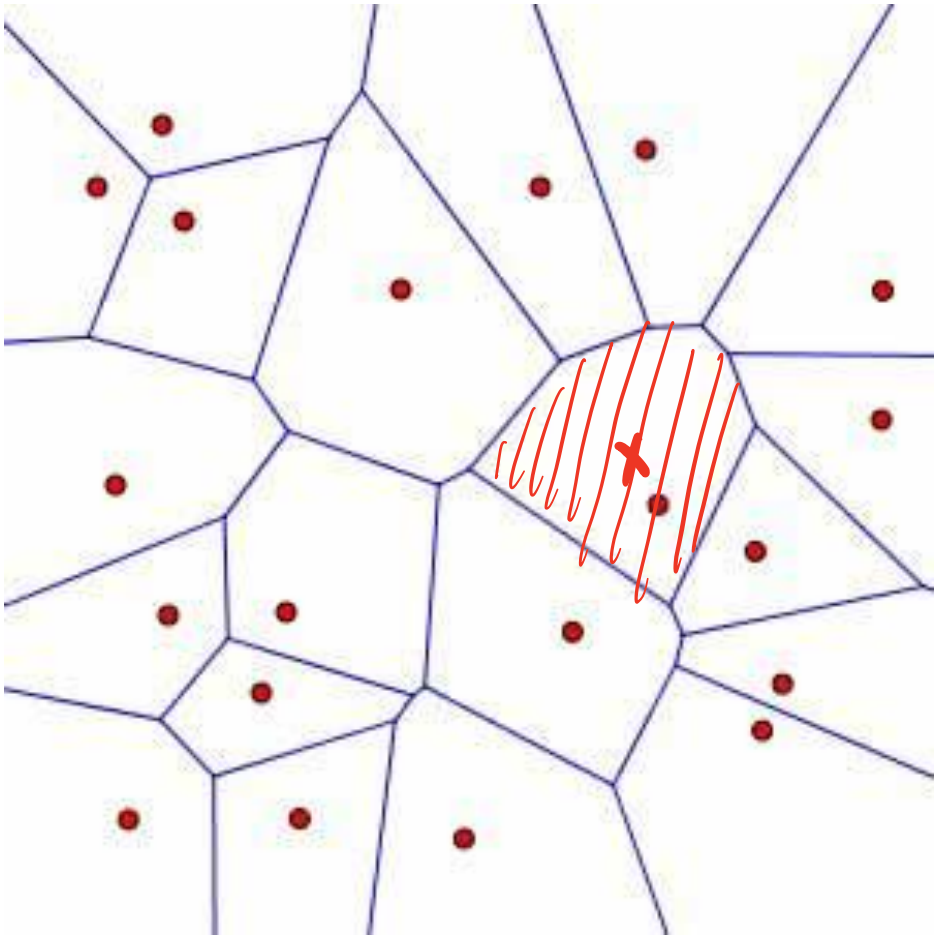
overall complexity:

$f(n) = O(n^{\log_2 2}) = n$

$\log_b n = \log_2 n$

case 2 $\rightarrow$ $T(n) = O(n \log n)$

All Nearest Neighbors Problem



Voronoi Diagram

Discussion 5

1. Suppose we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, along with T_1 which is a MST of G_1 and T_2 which is a MST of G_2. Now consider a new graph $G = (V, E)$ such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_3$ where E_3 is a new set of edges that all cross the cut (V_1, V_2).

Consider the following algorithm, which is intended to find a MST of G.

```
Maybe-MST(T1, T2, E3)
    emin = a minimum weight edge in E3
    T = T1 U T2 U { emin}
    return T
```

Does this algorithm correctly find a MST of G? Either prove it does or prove it does not.

2. Solve the following recurrences using the Master Method:

a. $A(n) = 3\,A(n/3) + 15$
b. $B(n) = 4\,B(n/2) + n^3$
c. $C(n) = 4\,C(n/2) + n^2$
d. $D(n) = 4\,D(n/2) + n$

3. There are 2 sorted arrays A and B of size n each. Design a D&C algorithm to find the median of the array obtained after merging the above 2 arrays (i.e. array of length 2n). Discuss its runtime complexity.

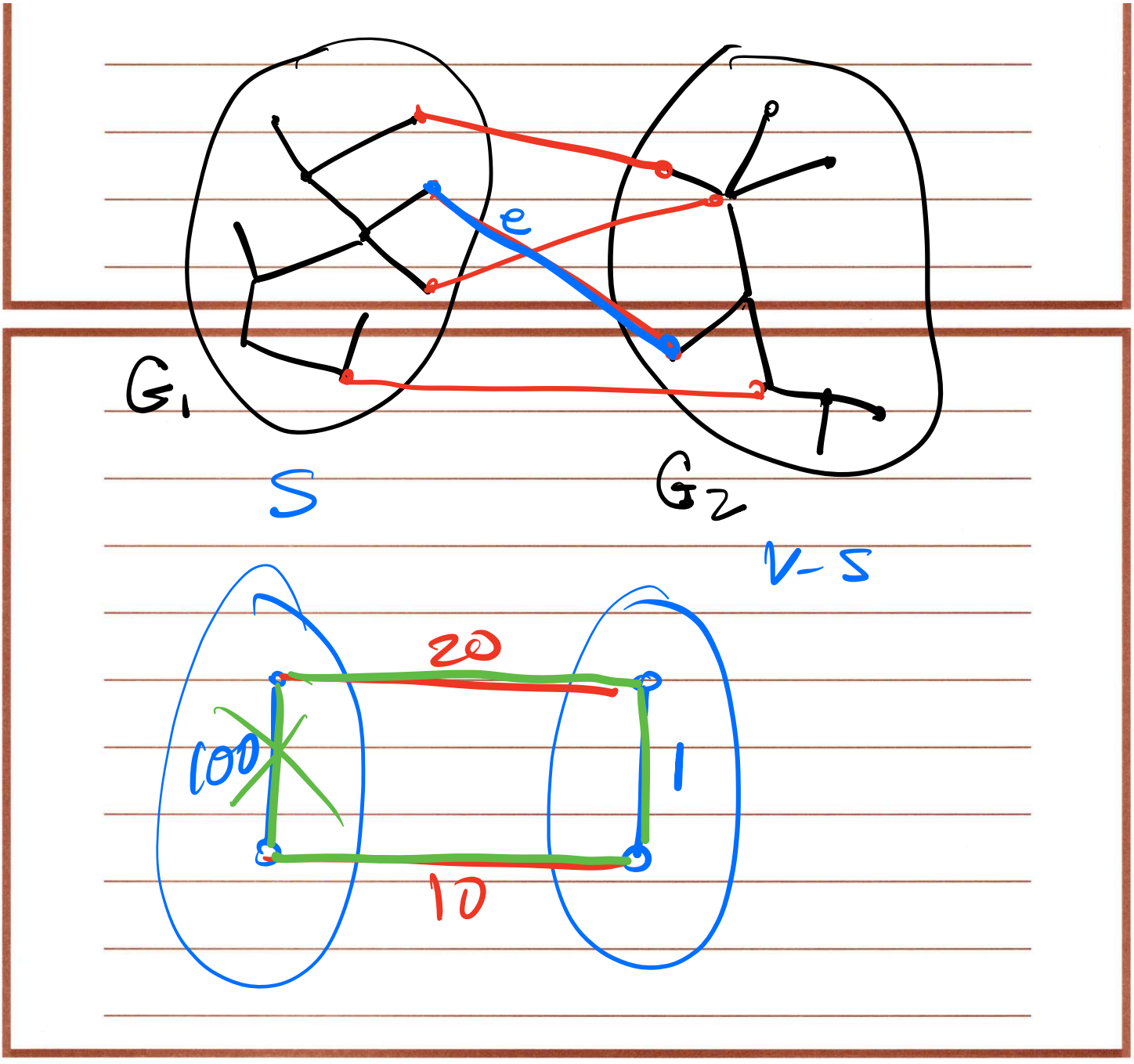
4. A tromino is a figure composed of three 1x1 squares in the shape of an L.
Given a $2^n x 2^n$ checkerboard with 1 missing square, tile it with trominoes.
Design a D&C algorithm and discuss its runtime complexity.

1. Suppose we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, along with T_1 which is a MST of G_1 and T_2 which is a MST of G_2. Now consider a new graph $G = (V, E)$ such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_3$ where E_3 is a new set of edges that all cross the cut (V_1, V_2).

Consider the following algorithm, which is intended to find a MST of G.

Maybe-MST(T_1, T_2, E_3)
    e_{min} = a minimum weight edge in E_3
    $T = T_1 \cup T_2 \cup \{ e_{min} \}$
    return T

Does this algorithm correctly find a MST of G? Either prove it does or prove it does not.



2. Solve the following recurrences using the Master Method:

a. $A(n) = 3\,A(n/3) + 15$
b. $B(n) = 4\,B(n/2) + n^3$
c. $C(n) = 4\,C(n/2) + n^2$
d. $D(n) = 4\,D(n/2) + n$

$A(n) = 3A(n/3) + 15$

$f(n) = 15 = \Theta(1)$

$n^{\log_b a} = n^{\log_3 3} = n^1 = n$ → Case #1

$A(n) = \Theta(n)$

$B(n) = 4B(n/2) + n^3$

$f(n) = n^3$

$n^{\log_b a} = n^{\log_2 4} = n^2$ → Case 3

$a f(n/b) < c f(n)$ for $c < 1$

$4 \cdot \left(\frac{n^3}{8}\right) < cn^3$

$n^3/2 < cn^3$

$.5 < c < 1$ ✓

$\rightarrow B(n) = \Theta(n^3)$

$$C(n) = 4C(n/2) + n^2$$

$f(n) = n^2$

$n^{\log_b a} = n^{\log_2 4} = n^2$ $\rightarrow$ Case #2

$C(n) = \Theta(n^2 \lg n)$

$$C'(n) = 4C'(n/2) + n^2 \lg^2 n$$

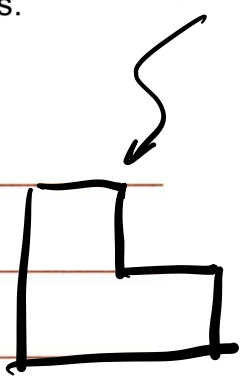
$f(n) = n^2 \lg^2 n$

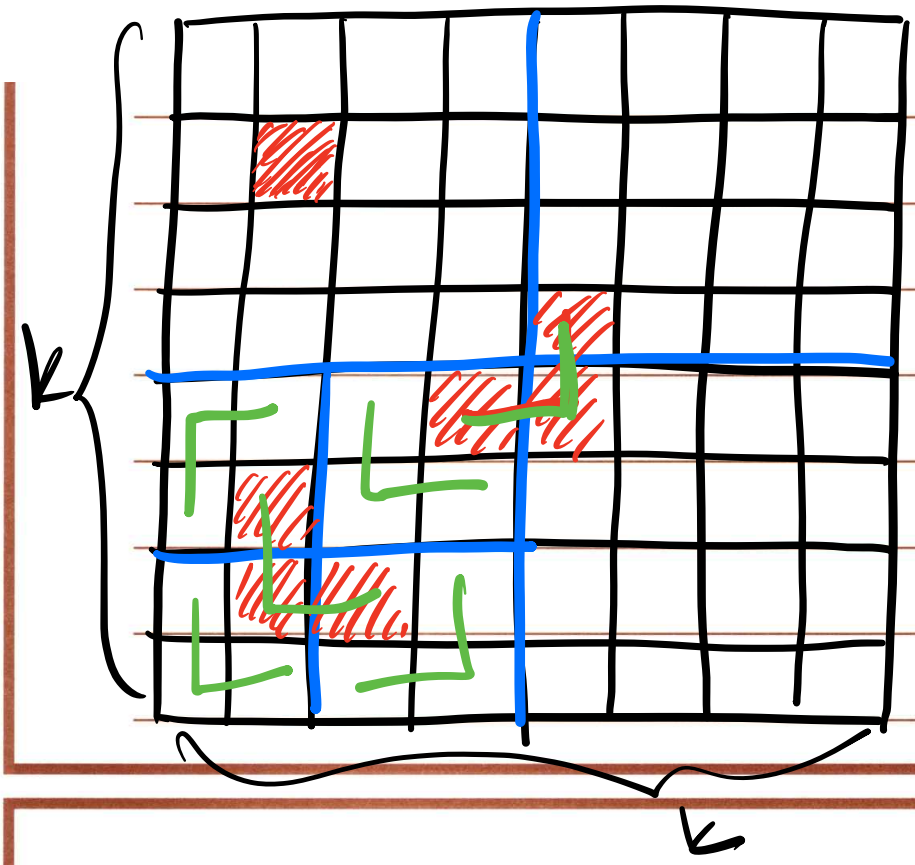
$n^{\log_b a} = n^2$

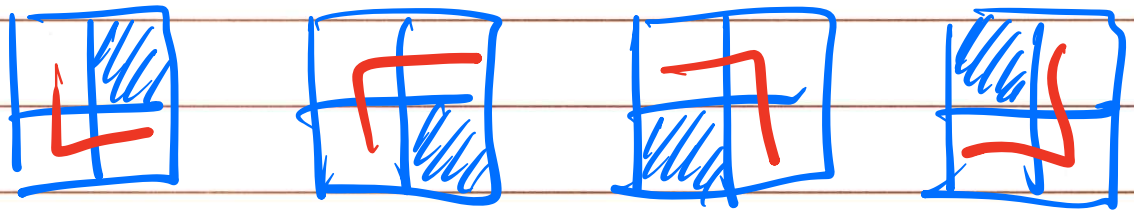
$\rightarrow$ General Case #2

$\rightarrow C'(n) = \Theta(n^2 \lg^3 n)$

4. A tromino is a figure composed of three 1x1 squares in the shape of an L. Given a $2^n x 2^n$ checkerboard with 1 missing square, tile it with trominoes. Design a D&C algorithm and discuss its runtime complexity.







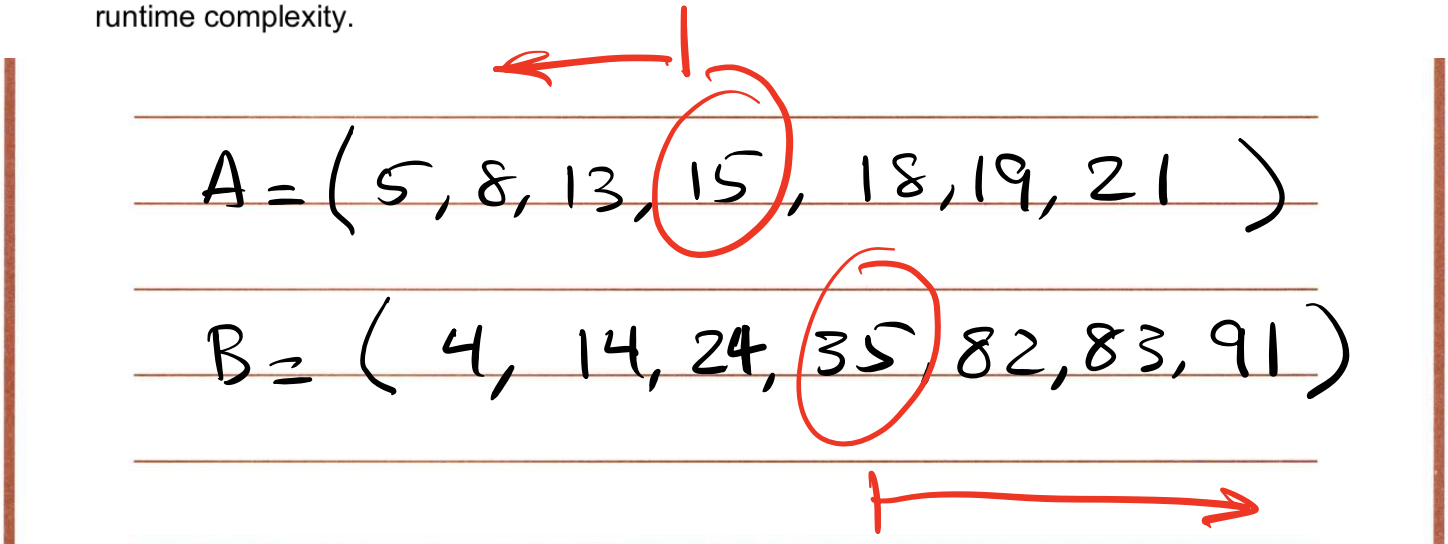
$$f(k) = \Theta(1)$$

$$k^{\log_b a} = k^{\log_2 4} = k^2$$

Case #1 $\Rightarrow$ $T(n) \in \Theta(k^2)$

d. D(n) = 4 D(n/2) + n

3. There are 2 sorted arrays A and B of size n each. Design a D&C algorithm to find the median of the array obtained after merging the above 2 arrays (i.e. array of length 2n). Discuss its runtime complexity.



$f(n) = \Theta(1)$

$n^{\log_b a} = n^{\log_2 1} = n^0 = \Theta(1)$

~~Case #1 $\Rightarrow T(n) = \Theta(n)$~~

Case #2 $\Rightarrow T(n) = \Theta(\lg n)$

Discussion 5

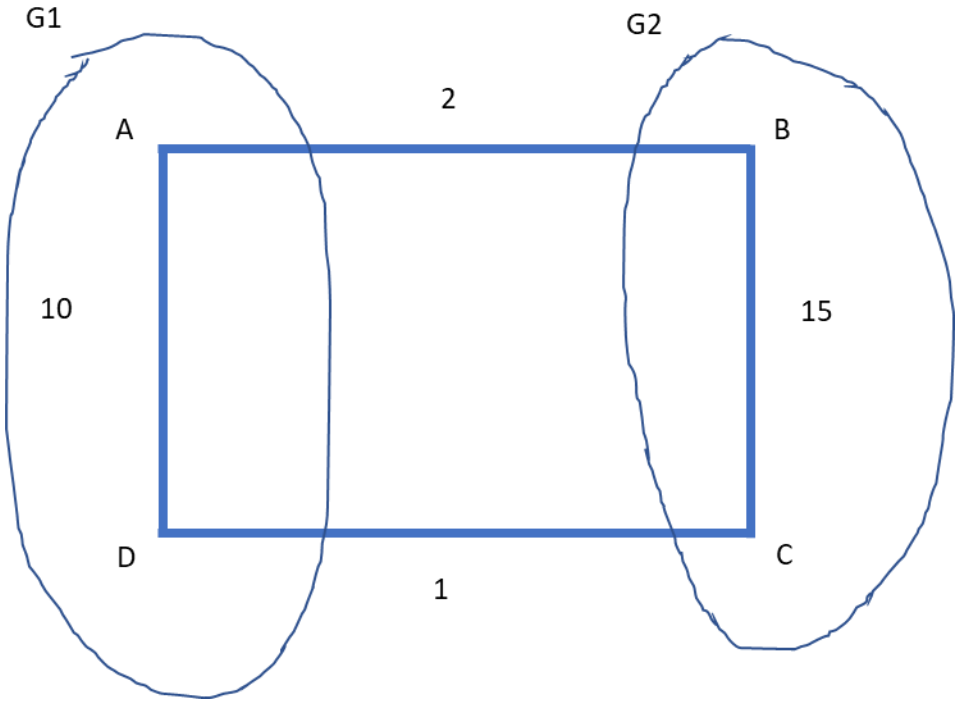
1. Suppose we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, along with T_1 which is a MST of G_1 and T_2 which is a MST of G_2. Now consider a new graph $G = (V, E)$ such that $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup E_3$ where E_3 is a new set of edges that all cross the cut (V_1, V_2).

Consider the following algorithm, which is intended to find a MST of G.

Maybe-MST(T_1, T_2, E_3)
    e_{min} = a minimum weight edge in E_3
    $T = T_1 \cup T_2 \cup \{ e_{min} \}$
    return T

Does this algorithm correctly find a MST of G? Either prove it does or prove it does not.

Solution:
No. Here is a counter example.



The correct solution will have edges AB, DC, and AD in the MST. The above solution will result in an incorrect MST with edges AD, BC, and DC.

2. Solve the following recurrences using the Master Method:

a. $A(n) = 3\,A(n/3) + 15$

f(n) = 15 = O(1), $n^{\log_b a} = n^{\log_3 3} = n^1$

This falls under case 1 → A(n) = θ(n)

b. B(n) = 4 B(n/2) + n^3

f(n) = n^3, $n^{\log_b a} = n^{\log_2 4} = n^2$

This can fall under case 3. Now we need to check that
a f(n/b) ≤ c f(n) for some c < 1 :
a f(n/b) = 4 * $(n/2)^3$ = 4 * $n^3/8$ = $n^3/2$ = .5 f(n) , so we have found c=.5 such that
a f(n/b) ≤ c f(n) and the inequality checks out, and case 3 applies → B(n) = θ(n^3)

c. C(n) = 4 C(n/2) + n^2

f(n) = n^2, $n^{\log_b a} = n^{\log_2 4} = n^2$

This falls under case 2 → C(n) = θ($n^2 \log n$)

d. D(n) = 4 D(n/2) + n

f(n) = n, $n^{\log_b a} = n^{\log_2 4} = n^2$

This falls under case 1 → D(n) = θ(n^2)

3. There are 2 sorted arrays A and B of size n each. Design a D&C algorithm to find the median of the array obtained after merging the above 2 arrays (i.e. array of length 2n). Discuss its runtime complexity.

Find the median of the two arrays. Say the medians are m_A and m_B. If m_A=m_B, then this is our median.
Otherwise, say m_A < m_B then throw away all terms lower than m_A in A, and all terms greater than m_B in B. Solve the resulting subproblem recursively.

Complexity analysis:

- Divide step takes O(1). This includes finding medians in A and B and throwing away half of A and B.
- There is no combine step
- Number of subproblems (a) at each step is 1. The size of the subproblem (n/b) is n/2

So we can apply the Master Method:

f(n) = O(1), $n^{\log_b a} = n^{\log_2 1} = n^0$ = O(1)

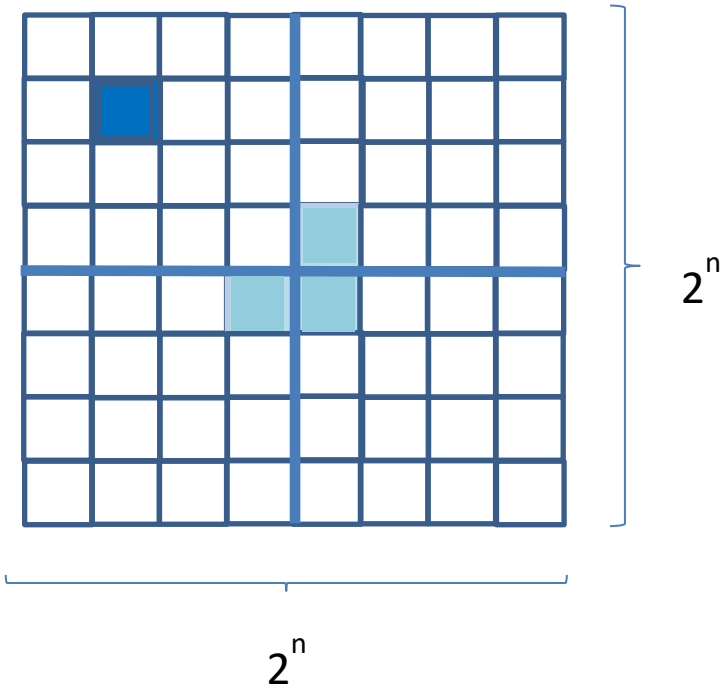
This falls under case 2 → T(n) = θ($\log n$)

4. A tromino is a figure composed of three 1x1 squares in the shape of an L.
Given a 2^nx2^n checkerboard with 1 missing square, tile it with trominoes.
Design a D&C algorithm and discuss its runtime complexity.

Solution: Here is a tromino:

The figure below shows a $2^n x 2^n$ checkerboard with a hole highlighted in dark blue. Here is a divide and conquer solution:

- Divide the grid into 4 equal grids of size $2^{n-1} x 2^{n-1}$
- Add holes to the three grids that do not have a hole in them. The position of the new holes should be at the center of the grid so that when each region is solved/tiled recursively we can cover the remaining three holes with one tromino.
- Solve all 4 subproblems recursively
- When combining the solutions, place a tile over the three holes in the center to complete the tiling.
- During recursion, when we reach 2x2 grids, we can solve the problem directly by placing a single tromino to tile the grid (that has a single hole).



Complexity analysis:

Divide steps takes O(1) time. So does the combine step.

Number of subproblems (a) = 4, size of each subproblem is half the size of the original problem, so b=2.

f(n) = O(1), $n^{\log_b a} = n^{\log_2 4} = n^2$

This falls under case 1 → T(n) = $\theta(n^2)$

CSCI 570 - Fall 2021 - HW 5 Solution

Due date : 30th September 2021

For all divide-and-conquer algorithms follow these steps:
1. Describe the steps of your algorithm in plain English.
2. Write a recurrence equation for the runtime complexity.
3. Solve the equation by the master theorem

1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T(·) represents the running time of an algorithm, *i.e.* T(n) is positive and non-decreasing function of n and for small constants c independent of n, T(c) is also a constant independent of n. Note that some of these recurrences might be a little challenging to think about at first.

   a) T(n) = 4T(n/2) + n^2logn
   b) T(n) = 8T(n/6) + nlogn
   c) T(n) = $\sqrt{6000}$ T(n/2) + $n^{\sqrt{6000}}$
   d) T(n) = 10T(n/2) + 2^n
   e) T(n) = 2T($\sqrt{n}$) + $\log_2 n$

Solution:

In some cases, we shall need to invoke the Master Theorem with one generalization as described next. If the recurrence T (n) = aT (n/b) + f (n) is satisfied with f(n) = Θ($n^{\log_b a}$ $\log^k n$) for some k ≥ 0, then T(n) = Θ($n^{\log_b a}$ $\log^{k+1} n$).

a) Observe that f(n) = n^2log n and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying the generalized Master's theorem, T (n) = Θ(n^2 $\log^2 n$).
b) Observe that $n^{\log_b a} = n^{\log_6 8}$ and f(n) = nlogn = O($n^{\log_6 8 - \varepsilon}$) for any 0 < ε < $\log_6 8 - 1$. Thus, invoking Master's Theorem gives T(n) = Θ($n^{\log_b a}$) = Θ($n^{\log_6 8}$).

c) We have $n^{\log_b a} = n^{\log_2 \sqrt{6000}} = n^{0.5 \log_2 6006} = O(n^{0.5 \log_2 8192}) = n^{13/2}$ and $f(n) = n^{\sqrt{6006}} = \Omega(n^{70}) = \Omega(n^{\frac{13}{2} + \varepsilon})$ for any $0 < \varepsilon < 63.5$. Thus, from Master's Theorem $T(n) = \Theta(f(n)) = \Theta(n^{\sqrt{6006}})$.

d) We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10 + \varepsilon})$ for any $\varepsilon > 0$. Therefore, Master's Theorem implies $T(n) = \Theta(f(n)) = \Theta(2^n)$.

e) Use the change of variables $n = 2^m$ to get $T(2^m) = 2T(2^{m/2}) + m$. Next, denoting $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$. Note that $S(\cdot)$ is a positive function due to the monotonicity of the increasing map $x \longmapsto 2^x$ and the positivity of $T(\cdot)$. All conditions for applicability of Master's Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $m^{\log_b a} = m$. We express the solution in terms of $T(n)$ by $T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log_2 n \log \log_2 n)$, for large enough n so that the growth expression above is positive.

2. Consider an array A of n numbers with the assurance that $n > 2$, $A[1] \geq A[2]$ and $A[n] \geq A[n-1]$. An index i is said to be a local minimum of the array A if it satisfies $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i+1] \geq A[i]$.

   (a) Prove that there always exists a local minimum for A.
   (b) Design an algorithm to compute a local minimum of A.

   Your algorithm is allowed to make at most O(log n) pairwise comparisons between elements of A. Please also provide proof of correctness for your algorithm.

   Solution:

   We prove the existence of a local minimum by induction.
   For $n = 3$, we have $A[1] \geq A[2]$ and $A[3] \geq A[2]$ by the premise of the question and therefore $A[2]$ is a local minimum. Let $A[1 : n]$ admit a local minimum for $n = k$. We shall show that $A[1 : k+1]$ also admits a local minimum. If we assume that $A[2] \leq A[3]$ then $A[2]$ is a local minimum for $A[1 : k+1]$ since $A[1] \geq A[2]$ by premise of the question. So let us assume $A[2] > A[3]$. In this case, the k length array $A[2 : k+1] = A'[1 : k]$ satisfies the induction hypothesis ($A'[1] = A[2] \geq A[3] = A'[2]$ and $A'[k-1] = A[k] \geq A[k+1] = A'[k]$)

and hence admits a local minimum. This is also a local minimum for A[1 : k + 1]. Hence, proof by induction is complete.

Consider the following algorithm with array A of length n as the input and return value as a local minimum element.
(a) If n = 3, return A[2].
(b) If n>3,

i. $k \leftarrow \lfloor n/2 \rfloor$.
ii. If $A[k] \leq A[k-1]$ and $A[k] \leq A[k+1]$ then return A[k].
iii. If $A[k] > A[k-1]$ then call the algorithm recursively on A[1 : k], else call the algorithm recursively on A[k : n].

Complexity: If the running time of the algorithm on an input of size n is T(n), then it involves a constant number of comparisons and assignments and a recursive function call on either $A[1 : \lfloor n/2 \rfloor]$ (size = $\lfloor n/2 \rfloor$) or $A[\lfloor n/2 \rfloor : n]$ (size = $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$). Therefore, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$. Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \Theta(1)$ and invoking Master's Theorem gives $T(n) = O(\log n)$.

Proof of Correctness: We employ induction. For n = 3 it is clear that step (a) returns a local minimum using the premise of the question that $A[1] \geq A[2]$ and $A[3] \geq A[2]$. Let us assume that the algorithm correctly finds a local minimum for all $n \leq m$ and consider an input of size m + 1. Then $k = \lfloor (m + 1)/2 \rfloor$. If step (b)(ii) returns, then a local minimum is found by definition, and the algorithm gives the correct output. Otherwise, step (b)(iii) is executed since one of $A[k] > A[k-1]$ or $A[k] > A[k+1]$ must be true for step (b)(ii) to not return. If $A[k] > A[k-1]$, then A[1 : k] must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $k \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $\lfloor (m + 1)/2 \rfloor \leq m$ which is true for all $m \geq 1$. Similarly, if $A[k] > A[k+1]$, then A[k:m+1] must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $m - k + 2 \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $k \geq 2$ or equivalently $\lfloor (m+1)/2 \rfloor \geq 2$ which holds for all $m \geq 3$. Therefore, the algorithm gives the correct output for inputs of size m + 1 and the proof is complete by induction.

3. The recurrence T(n) = 7T (n/2) + n^2 describes the running time of an algorithm ALG. A competing algorithm ALG′ has a running time of $T'(n)$ = a$T'\left(\frac{n}{4}\right)$ + $n^2 \log n$. What is the largest value of a such that ALG′ is asymptotically faster than ALG?

Solution:

We shall use Master Theorem to evaluate the running time of the algorithms.

(a) For T(n), setting $0 < \varepsilon < \log_2 7 - 2 \approx 0.81$ we have $n^2 = O(n^{\log_2 7 - \varepsilon})$. Hence $T(n) = \Theta(n^{\log_2 7})$.

(b) For $T'(n)$, if $\log_4 a > 2$ then setting $0 < \delta < \log_4 a - 2$ implies $n^2 \log n = O(n^{\log_4 a - \delta})$ and hence $T'(n) = \Theta(n^{\log_4 a})$.

To have $T'(n)$ asymptotically faster than T(n), we need $T'(n) = O(T(n))$, which implies $n^{\log_4 a} = O(n^{\log_2 7})$ and therefore $\log_4 a \leq \log_2 7 \Longrightarrow a \leq 49$. Since other expressions for run time of ALG′ require $\log_4 a \leq 2 \Longrightarrow a \leq 16 < 49$, the largest possible value of a such that ALG′ is asymptotically faster than ALG is a = 49.

4. Solve Kleinberg and Tardos, Chapter 5, Exercise 3.

Solution:

Let us call a card to be a *majority card* if it belongs to a set of equivalent cards encompassing at least half of the total number of cards to be tested, *i.e.* for a total of n cards, if there is a set of more than n/2 cards that are all equivalent to each other, then any one of these equivalent cards constitutes a majority card. To keep the conquer step as simple as possible, we shall require the algorithm to return a majority card if one exists and return nothing if no majority card exists (this can be implemented by indexing the cards from 1 to n and a return value of 0 could correspond to returning nothing). Then the algorithm consists of the following steps, with S denoting the set of cards being tested by the algorithm and |S| denoting its cardinality.

a) If $|S| = 1$ then return the card.
b) If $|S| = 2$ then test whether the cards are equivalent. If they are equivalent then return either card, else return nothing.
c) If $|S| > 2$ then arbitrarily divide the set S into two disjoint subsets S1 and S2 such that $|S1| = \lfloor |S|/2 \rfloor$ and $|S2| = \lceil |S|/2 \rceil$.
d) Call the algorithm recursively on the set of cards S1.
e) If a majority card is returned (call it card m1), then test it for equivalence against all cards in $S \setminus \{m1\}$. If m1 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m1.
f) If m1 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S1, then call the algorithm recursively on the set of cards S2.
g) If a majority card is returned (call it card m2), then test it for equivalence against all cards in $S \setminus \{m2\}$. If m2 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m2.
h) If m2 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S2, then return nothing.

Complexity: Let T(n) be the number of equivalence tests performed by the algorithm on a set of n cards. We have T(2) = 1 from step (b). Steps (d) and (f) can incur a total of $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ equivalence tests whereas steps (e) and (g) could require up to $2(n - 1)$ equivalence tests. Therefore, we have the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2n - 2$, which can be solved from first principles like in (6). Alternatively, the recurrence simplifies to $T(n) \leq 2T(n/2) + 2n - 2$ on assuming n to be a power of 2. The simplified recurrence can be solved using Master's Theorem to yield $T(n) = O(n \log n)$.

Proof of Correctness: We need to show that the algorithm returns the correct answer in both cases, *viz.* majority card present and majority card absent. If $|S| = 1$ then this single card forms a majority by our definition and is correctly returned by step (a). If $|S| = 2$ then either card is a majority card if both cards are equivalent and neither card is a majority card if they are not equivalent. Clearly, step (b) implements exactly this criterion. For steps (d)-(h) we consider what happens when a majority card is present or absent.

(a) *Majority card present:* If a majority card is present (say m), then at least one of the subsets S1 or S2 must have m as a majority card. It is simple to see this by contradiction since if neither S1 nor S2 have a majority card (or if m is not a mojority card in either subset) then

necessarily the number of equivalent cards (or cards equivalent to m, including m) is upper bounded by $0.5\lfloor|S|/2\rfloor + 0.5\lceil|S|/2\rceil = 0.5|S|$ which immediately implies the absence of any majority card in S. Thus, at least one of m1 (in step (e)) or m2 (in step (g)) will be returned and at least one of these will be equivalent to m. Since steps (e) and (g) respectively test m1 and m2 against all other cards, the algorithm will necessarily find one of them to be equivalent to m and hence also as a majority card (since the set of majority cards is necessarily unique) and would return the same.

(b) *Majority card absent:* There are multiple possibilities in this scenario. If neither S1 nor S2 returns a majority card then clearly there is no majority card, and the algorithm terminates at step (h) returning nothing. If m1 or m2 or both are returned as valid majority cards then they would be respectively discarded as candidates in steps (e) and (g) when checked against all other cards in S as they are not truly in majority on the set S. Thus the algorithm correctly terminates in step (h) returning nothing.

Note: There are a lot of repeated equivalence tests in the algorithm described above, primarily because we are solving the problem in a top-down manner. Can you think of a bottom-up approach to solving this problem with potentially fewer equivalence tests? Hint: It is possible to achieve O(n) complexity in terms of the number of equivalence tests required.

5. Given a binary search tree T, its root node *r*, and two random nodes *x* and *y* in the tree. Find the lowest common ancestry of the two nodes *x* and *y*. Note that a parent node *p* has pointers to its children *p.leftChild()* and *p.rightChild()*, a child node does **not** have a pointer to its parent node. The complexity must be $O(\log n)$ or better, where *n* is the number of nodes in the tree.
   Recall that in a binary search tree, for a given node *p*, its right child *r*, and its left child *l*, $r.value() \geq p.value() \geq l.value()$. Hint: use divide and conquer

   Solution: Use divide and conquer:

```
Node  LCA(r, x, y){
        if (x.value() < r.value && y.value() > r.value())
                Return r;
        else If(x.value() < r.value && y.value() < r.value())
                Return LCA(r.leftChild() x, y);
        else
```

Return LCA(r.rightChild() x, y);
}

Recurrence relation: T(n/2)+c
Complexity O(log n)

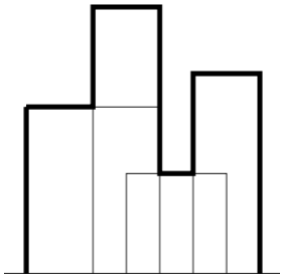
Note: Using the property of the binary search tree to decide if two nodes belong to the same branch or different branches of a parent node is crucial for maintaining a complexity of O(log n). Simply using the binary search to answer this question will make the overall complexity of the algorithm to be $O(\log^2 n)$.

Note that the property of a classic binary search tree is that no duplicates are allowed, so students need not care about the fact that the elements in the tree can be equal. Some few modern implementations allow duplicates in the binary search tree, but that is specifically implemented according to the requirements of each application and is not considered as a pure binary search tree.

6. Suppose that you are given the exact locations and shapes of several rectangular buildings in a city, and you wish to draw the skyline (in two dimensions) of these buildings, eliminating hidden lines. Assume that the bottoms of all the buildings lie on the x-axis. Building B_i is represented by atriple (L_i, H_i, R_i), where L_i and R_i denote the left and right x coordinates of the building, respectively, and H_i denotes the building's height. A skyline isa list of x coordinates and the heights connecting them arranged in order from left to right.
For example, the buildings in the figure below correspond to the following input

(1, 5, 5), (4, 3, 7), (3, 8, 5), (6, 6,8).

The skyline is represented as follows: (1, **5**, 3, **8**, 5, **3**, 6, **6**, 8). Notice that the x-coordinates in the skyline are in sorted order.



Solution:

a) Given a skyline of n buildings and another skyline of m buildings in the form (x1, **h1**, x2, **h2**, ….xn) and (x'1, **h'1**, x'2, **h'2**, ….x'm) , show how to compute the combined skyline for the m + n buildings in O(m + n) step.

Merge the "left" list and the "right" list iteratively by keeping track ofthe current left height (initially 0), the current right height (initially 0),finding the lowest next x-coordinate in either list; we assume it is the left list. We remove the first two elements, x and h, and set the currentleft height to h, and output x and the maximum of the current left and right heights.

b) Assuming that we have correctly built a solution to part a, give a divide and conquer algorithm to compute the skyline of a given set of n buildings. Your algorithm should run in O(n log n) steps.

If there is one building, output it.
Otherwise, split the buildings into two equal groups, recursively compute skylines, output the result of merging them using part (a).

Algorithm :

getSkyline for n buildings :

- If n == 0 : return an empty list.
- If n == 1 : return the skyline for one building (it's straightforward).
- leftSkyline = getSkyline for the first n/2 buildings.

- rightSkyline = getSkyline for the last n/2 buildings.
- Merge leftSkyline and rightSkyline.

Recurrence Relation: $T(n) \leq 2T(n/2) + O(n)$

Time Complexity :
The runtime is bounded by the recurrence$T(n)$
$\leq 2T(n/2) + O(n)$,
$a = 2, b = 2, k = 1$
which implies that $T(n) = O(n \log n)$ by masters theorem.