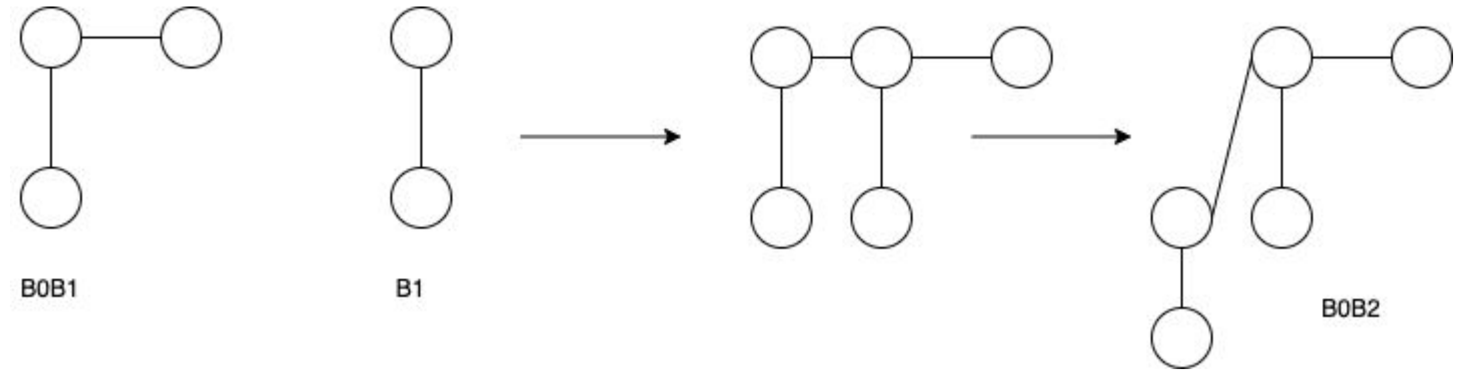
CSCI 570 Exam 1 Review

Heaps and Amortized Cost

Garima Rawat

Merge in Binomial heaps



11 + 10 = 101 → B2B0

Question 1

Merge two binomial heaps

B0B1B2B4 and B1B4

What would be the sizes of individual component trees?

B0B1B2B4 → 10111

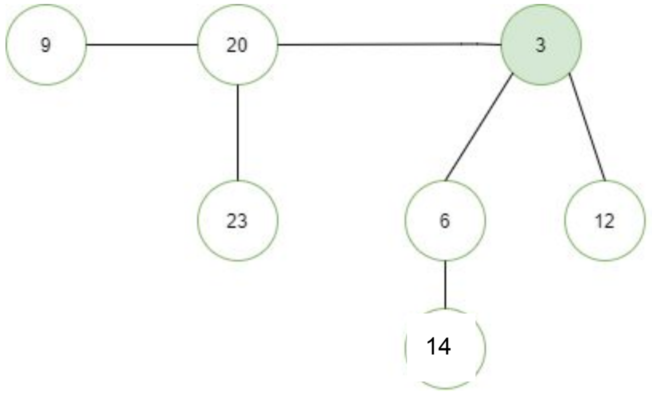
B1B4 → 10010

Perform Binary addition

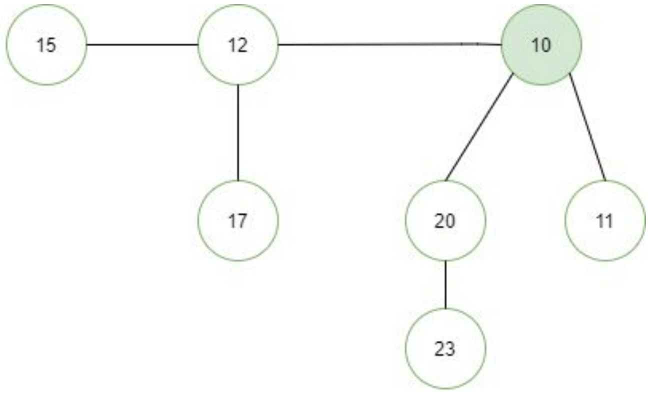
Result → 101001

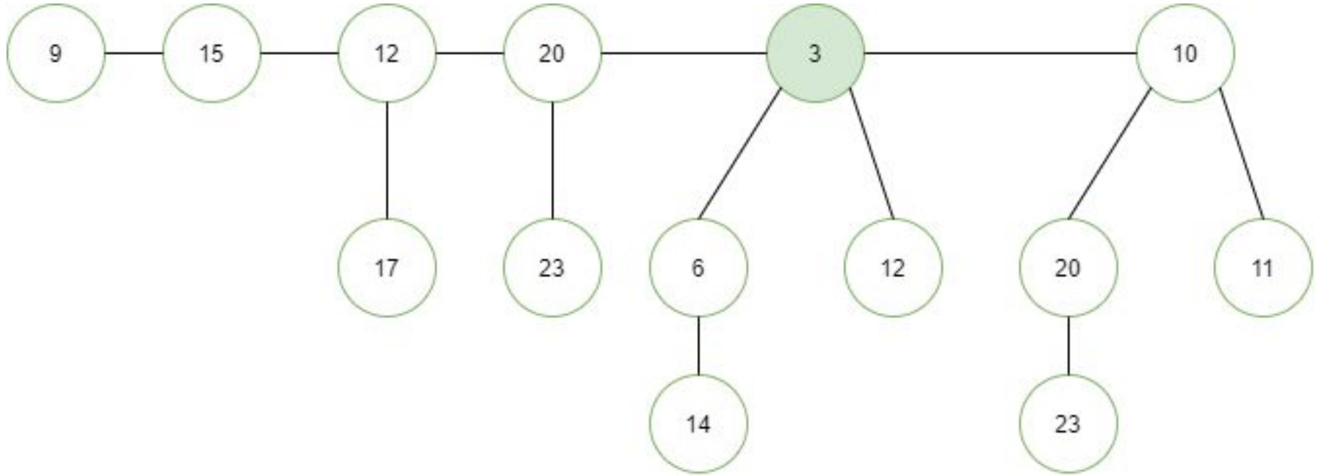
Heap after merging → B5B3B0

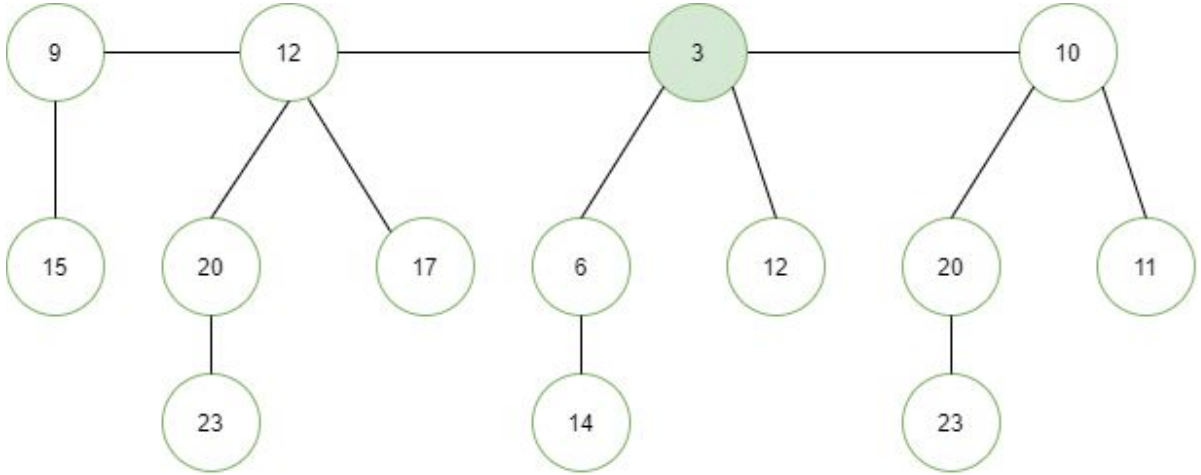
Question 2

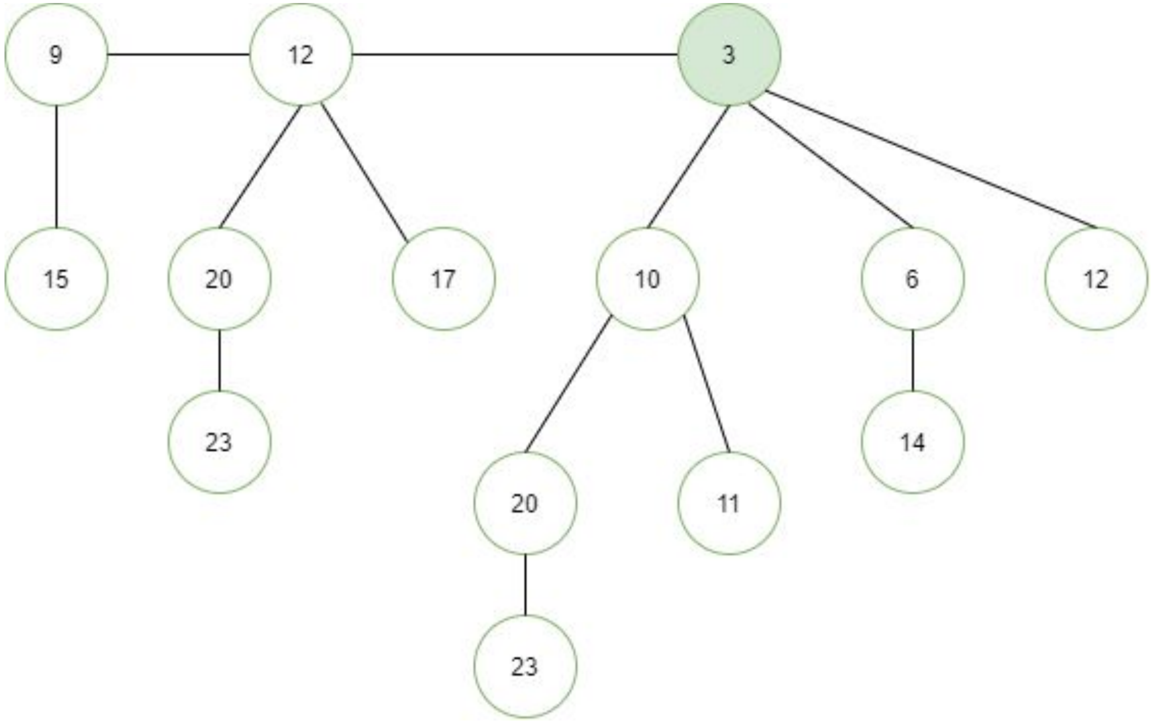


Merge with

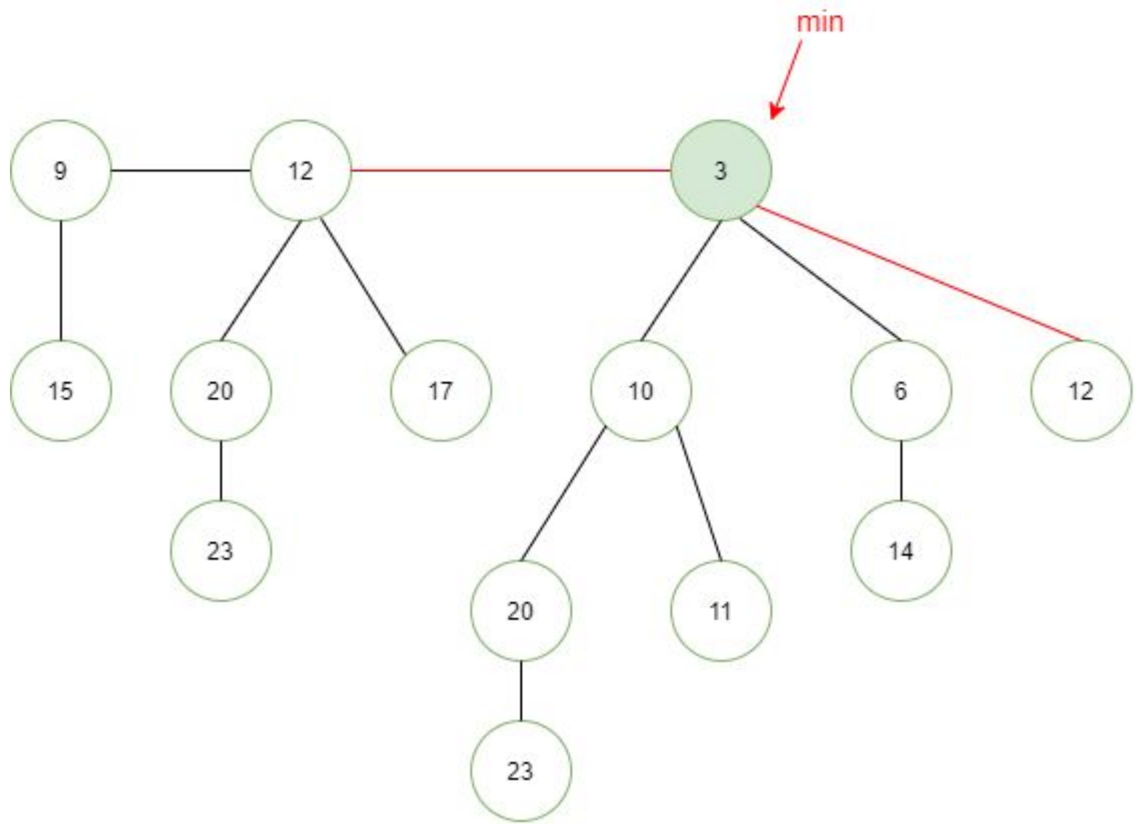


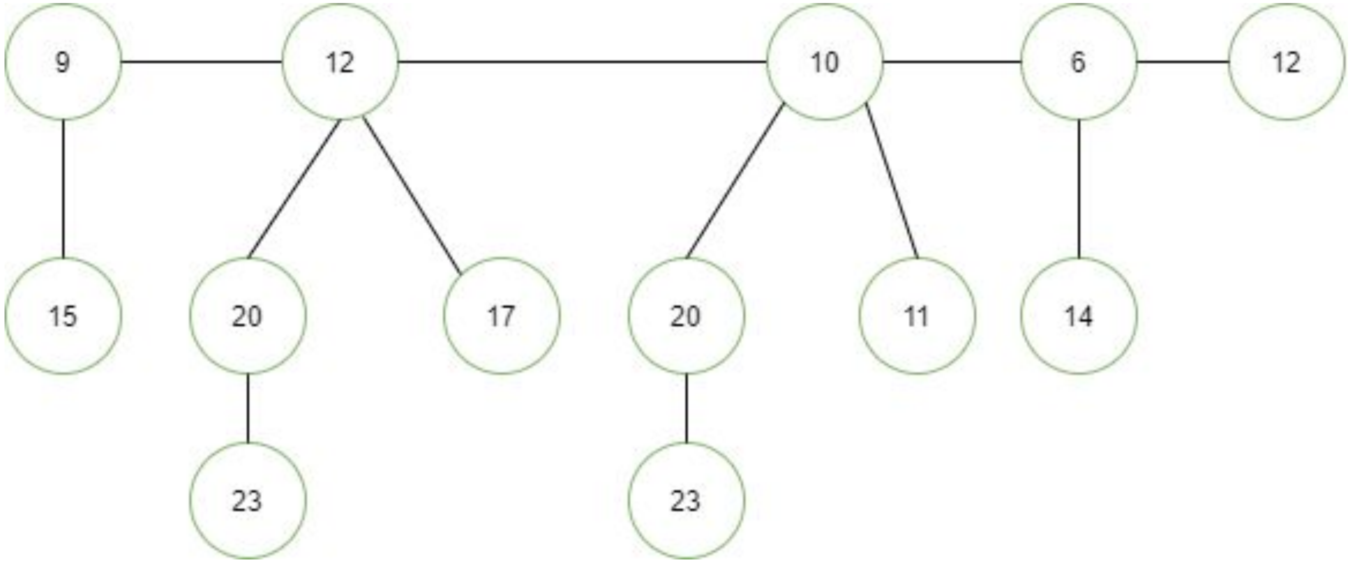

9
15
12
20
3
10
17
23
6
12
20
11
14
23

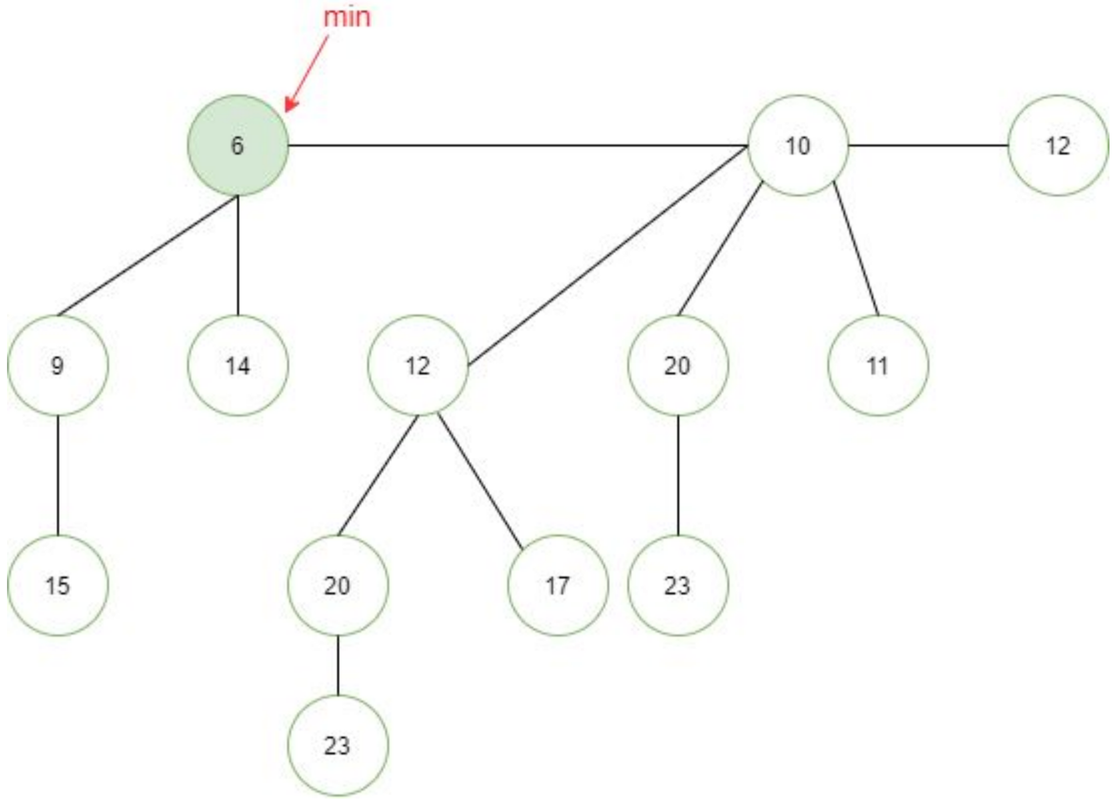

9
12
3
10
15
20
17
6
12
20
11
23
14
23


9
12
3
15
20
17
10
6
12
23
20
11
14
23

deleteMin()




9
12
10
6
12
15
20
17
20
11
14
23
23


min
6
10
12
9
14
12
20
11
15
20
17
23
23

Meeting Scheduler

There are N meetings to be scheduled. A company wants to make sure the first K meetings get scheduled on-time. Rest of the meetings can get delayed. Find the least number of rooms required to schedule the first K meetings exactly at their starting time. You are given start time S_i and end time E_i of each meeting.

Can you do it in O(NlogN) complexity?

Can you do better? O(N + KlogN)?

Which data structure do you think would be the best?

```
Step 1 :Create an array of pairs of start_time and end_time (s_i, t_i) for all meetings.
Step 2 : Create a min heap of the above array. O(n)
Step 3 : For j in 1 to k: O(k)
            Perform extract_min operation O(log n)
            Add the extracted pair in an array selected_meetings
Step 4 : selected_meetings is sorted in ascending order of start_time and end_time. Find max overlap.
```

One way to find overlap:

1. Initialize curr_room, max_room as 0.
2. Sort each time as a pair (time, Start or end) O(k log k)
3. Traverse the array, for each start, increment curr_room by 1, for each end, decrement curr_room by 1. Max_room = max(max_room, curr_room)
4. Return max_room

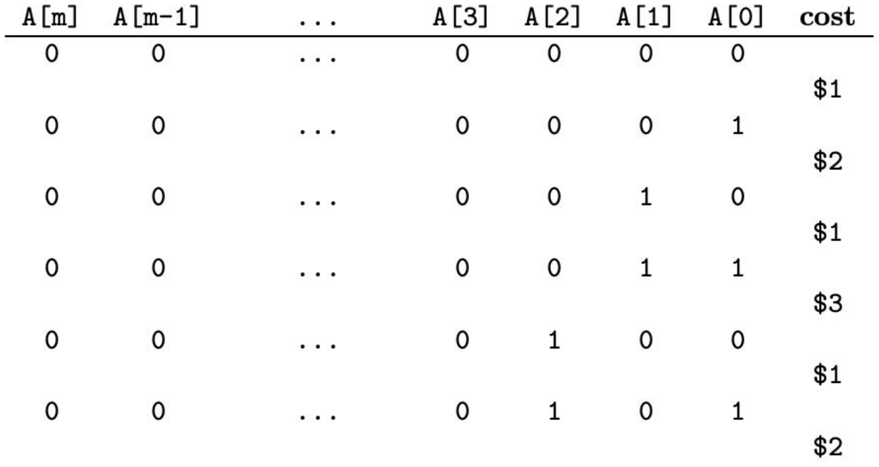
[T/F] The smallest element in a binary max-heap of size n can be found with at most n/2 comparisons.

[T/F] The smallest element in a binary max-heap of size n can be found with at most n/2 comparisons.

True. In a max heap, the smallest element is always present at a leaf node. So we need to check for all leaf nodes for the minimum value → how many leaf nodes are there?
ceil(n/2) leaf nodes. → Therefore, we will only have to do at most n/2 comparisons.

Binary Counter

Imagine we want to store a big binary counter in an array A. Each element of the array holds a bit which is either marked 0 or 1. All the entries start at 0 and at each step we will be simply incrementing the counter. Let’s say our cost model is: whenever we increment the counter, we pay $1 for every bit we need to flip.
For instance, here is a trace of the first few operations and their cost:
What is our amortized cost per increment?



A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	cost
0	0	...	0	0	0	0	
							$1
0	0	...	0	0	0	1	
							$2
0	0	...	0	0	1	0	
							$1
0	0	...	0	0	1	1	
							$3
0	0	...	0	1	0	0	
							$1
0	0	...	0	1	0	1	
							$2

In a sequence of n increments, the worst-case cost per increment is O(log n), since at worst we flip lg(n) + 1 bits

Aggregate analysis

First, how often do we flip A[0]?
Answer: every time.

How often do we flip A[1]?
Answer: every other time.

How often do we flip A[2]? Answer: every 4th time, and so on.

So, the total cost spent on flipping A[0] is n,
the total cost spent flipping A[1] is at most n/2,
the total cost flipping A[2] is at most n/4, etc.

For n increments, we'll flip till log(n) bits

n + n/2 + n/4 + …. + 1 = O(2n)

Summing these up, the total cost spent flipping all the positions in our n increments is at most 2n for n increments.

Amortized cost = 2n/n = O(2) = O(1)

Accounting method

Every time you flip 0 → 1, pay the actual cost of \$1, plus put \$1 into a piggy bank.
So the total amount spent is \$2.

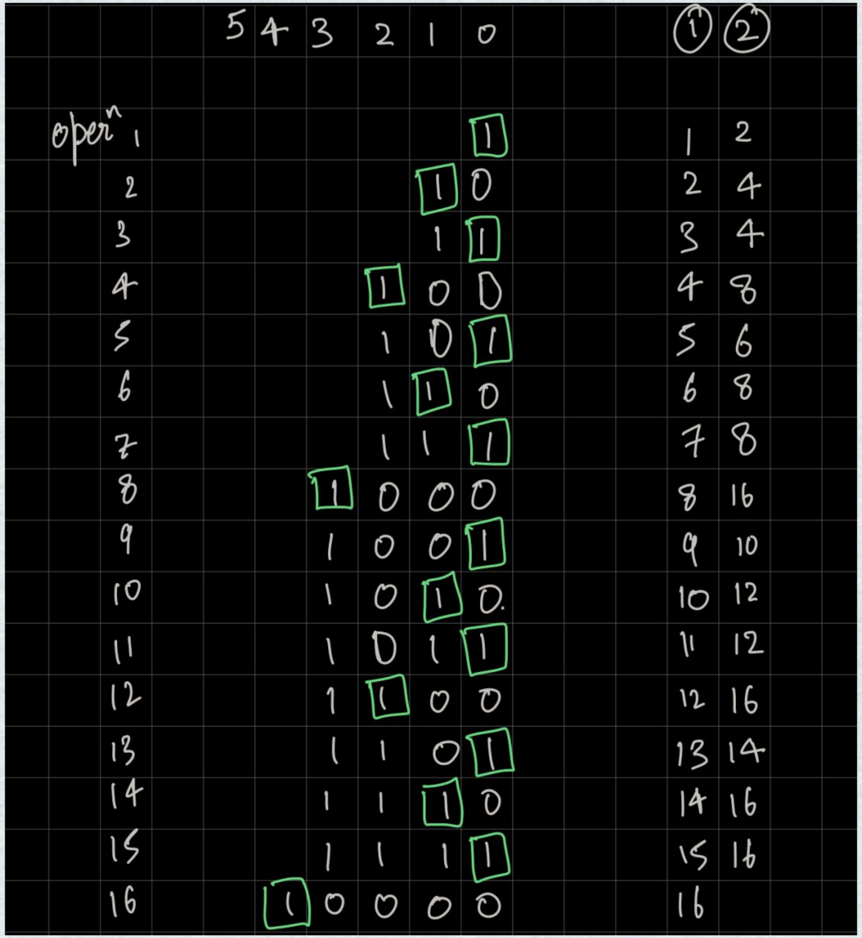
Now, every time you flip a 1 → 0, use the money in the bank to pay for the flip.

Clearly, by design, our bank account cannot go negative.

The key point now is that even though different increments can have different numbers of 1 → 0 flips, each increment has exactly one 0 → 1 flip.

So, we just pay \$2 (amortized) per increment

operation #
where coin is used



	5	4	3	2	1	0	①	②
oper^n 1						$\boxed{1}$	1	2
2					$\boxed{1}$	0	2	4
3					1	$\boxed{1}$	3	4
4				$\boxed{1}$	0	0	4	8
5				1	0	$\boxed{1}$	5	6
6				1	$\boxed{1}$	0	6	8
7				1	1	$\boxed{1}$	7	8
8			$\boxed{1}$	0	0	0	8	16
9			1	0	0	$\boxed{1}$	9	10
10			1	0	$\boxed{1}$	0.	10	12
11			1	0	1	$\boxed{1}$	11	12
12			1	$\boxed{1}$	0	0	12	16
13			1	1	0	$\boxed{1}$	13	14
14			1	1	$\boxed{1}$	0	14	16
15			1	1	1	$\boxed{1}$	15	16
16		$\boxed{1}$	0	0	0	0	16	

USC University of Southern California

CSCI 570 Exam 1 Review

BFS & DFS
Chris Denniston

Search Algorithms

- Search algorithms systematically look at each **node** in a graph
- Search Algorithms can be used for finding things in graphs at a specific node
- Search algorithms can be used to make a path
- Search algorithms can be used to determine if a graph is connected
- Use DFS when you: can’t jump from non connected node to non connected node, don’t care about path length, can easily backtrack
- Use BFS when you: want to know the # of edges between two nodes, want to explore locally first, can’t easily backtrack

Breadth First Search (BFS)

- BFS is a *graph search algorithm*, an algorithm for exploring a graph $G=(V,E)$.
- BFS starts at a vertex s and explores outwards, visiting all nodes distance 1 from s, then all nodes distance 2 from s, and so on.
- BFS can find a path between two vertices s and t, if such a path exists.
- BFS finds the shortest edge path between s and t if the graph is unweighted.
- BFS is implemented using a *queue* (FIFO).
- The worst-case runtime of BFS is $O(|V| + |E|)$.

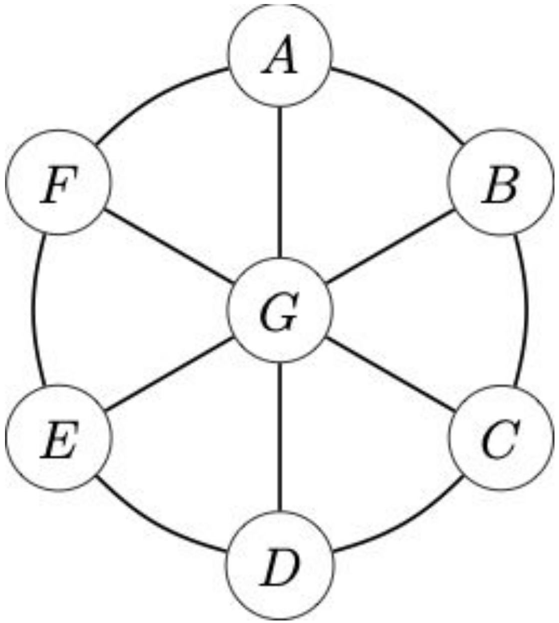
BFS Tree

- A graph search algorithm induces a *search tree* $T \subseteq G$, with root s and edges $u \rightarrow v$, where the first time the algorithm explored v was by traversing the edge (u,v) in G.

  (3.3) *For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.*

Problem 1

List the order in which vertices are visited when executing BFS starting from vertex E and breaking ties alphabetically.



Answer:

E, D, F, G, C, A, B

Depth First Search (DFS)

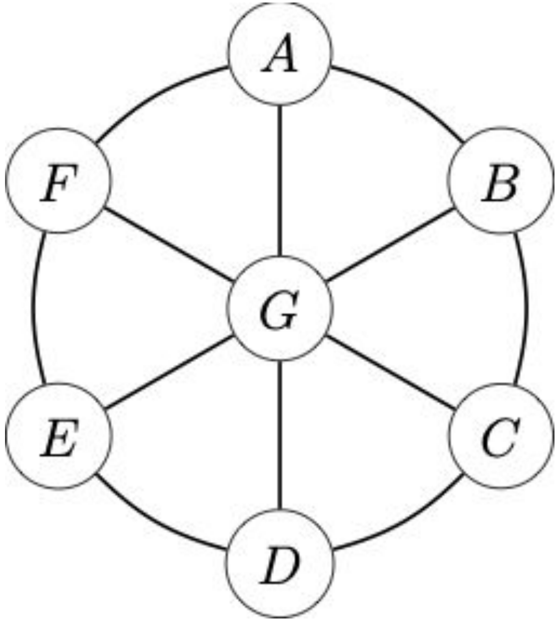
- DFS is also a graph search algorithm.
- DFS starts at a vertex *s* and explores by walking around the graph, backtracking when it hits a dead-end.
- DFS can find a path between two vertices *s* and *t*, if such a path exists.
- DFS is implemented using a *stack* (LIFO).
- The worst-case runtime of DFS is also $O(|V| + |E|)$.

DFS Tree

- **(3.7)** *Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.*

Problem 2

List the order in which vertices are visited when executing DFS starting from vertex E and breaking ties alphabetically.



Answer:
E, D, C, B, A, F, G

BFS v. DFS

- Both BFS and DFS can find a path from *s* to *t*.
- Both BFS and DFS can find the number of connected components in *G*.
- Both run in time $O(|V| + |E|)$.
- BFS is implemented using a queue (FIFO). DFS is implemented using a stack (LIFO).
- BFS can find shortest paths in unweighted graphs.

Problem 3

Detect if a graph is bipartite using BFS and DFS
A graph is bipartite if it is 2-colorable

Answer: Run a graph search, coloring each node the opposite of its parent. If a node is ever already colored the same as its parent, the graph is not bipartite

Prove this is true.

USC University of Southern California

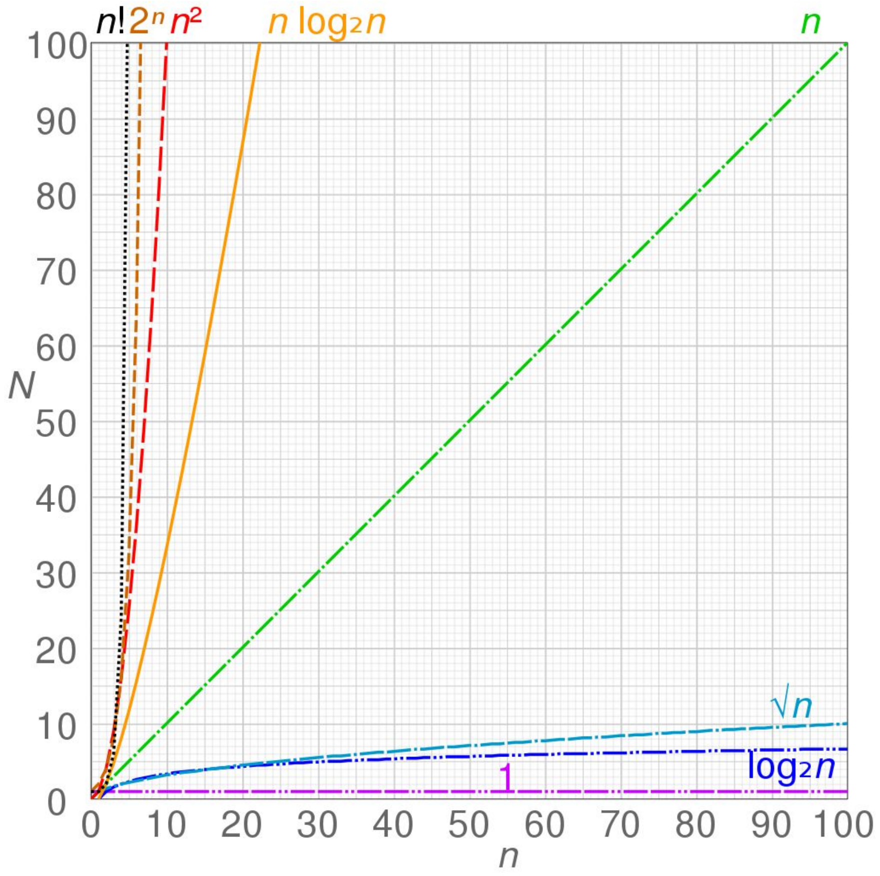
CSCI 570 Exam 1 Review

Big-O
Chris Denniston

Things to remember about Big O

- Big O ignores constants. If you have to do a fixed* amount of work (say 1000) but the actual algorithm is O(n^2) it's still n^2, even though it might run slower than an n^3 algorithm on a real computer)
- Big O only counts certain operations - be careful with which operations you care about
- Generally, want the tightest bound but e.g if something is O(n^2) it is also O(n^3) and O(n!) and O(2^n)

Common Big O scaling



Problem 1

What is the big o of this function?

$$f(n) = 9\log n + 5(\log n)^4 + 3n^2 + 2n^3$$

Answer: O(n^3) only the “worst” one matters

Problem 2

- If f1 = O(g1) and f2=O(g1) what is the O of f1*f2 ?

  O(g1g2)

- If f1 = O(g1) and f2=O(g1) what is the O of f1+f2 ?

  O(max(g1,g2)) or O(g1+g2) <- this fact is really useful if your algorithm needs to do “pre-work”, can do one “for free”

- If f1 = O(g1) and r is a constant, what is f1*r

  O(g1)

Problem 3
What is the time complexity of this algorithm? (assume only 1 node has the id 1)

```
def search(v,visited):
    If v in visited:
        Return nothing
    For v' in neighbors(v):
        If id(v) == 1:
            Return mergesort(visited)
        Q = search(v',visited + {v})
        If Q is not nothing:
            Return Q
search(random_node(), [])
```

Answer: O(nlogn)

Even though we only visit each node once (making it O(n)), at exactly one node we do an nlogn operation

USC University of Southern California

CSCI 570 Exam 1 Review

Stable Matching
Ali Omrani

Stable Matching Recap

Input: A set of n hospitals H and a set of n students S.

- Each hospital $h \in H$ ranks students.
- Each student $s \in S$ ranks hospitals.

Def. A matching M is a set of ordered pairs $h–s$ with $h \in H$ and $s \in S$ s.t.

- Each hospital $h \in H$ appears in at most one pair of M.
- Each student $s \in S$ appears in at most one pair of M.

Def. Given a perfect matching M, hospital h and student s form an unstable pair if both:

- h prefers s to matched student.
- s prefers h to matched hospital.

A stable matching is a perfect matching with no unstable pairs.

GALE–SHAPLEY (*preference lists for hospitals and students*)

```
INITIALIZE $M$ to empty matching.
WHILE (some hospital $h$ is unmatched and hasn't proposed to every student)
    $s \leftarrow$ first student on $h$'s list to whom $h$ has not yet proposed.
    IF ($s$ is unmatched)
        Add $h$–$s$ to matching $M$.
    ELSE IF ($s$ prefers $h$ to current partner $h'$)
        Replace $h'$–$s$ with $h$–$s$ in matching $M$.
    ELSE
        $s$ rejects $h$.

RETURN stable matching $M$.
```

$O(n^2)$

Indifferent Stable Matching

The Stable Matching Problem, as discussed in the text, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be indifferent between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with n = 4), a woman could say that m1 is ranked in first place; second place is a tie between m2 and m3 (she has no preference between them); and m4 is in last place. We will say that w prefers m to m′ if m is ranked higher than m′ on her preference list (they are not tied). With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

Problem - Strong Instability with Indifference

A strong instability in a perfect matching S consists of a man m and a woman w, such that each of m and w prefers the other to their partner in S. Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

Solution to Strong Instability

The answer is Yes. A simple way to think about it is to break the ties in some fashion and then run the stable matching algorithm on the resulting preference lists. We can for example break the ties lexicographically - that is if a man m is indifferent between two women wi and wj then wj appears on m's preference list. before w; if i < j and if j < i wj appears before wi- Similarly if w is indifferent between two men mi and mj; then mi appears on w's preference list before m; if i < j and if j < i mj appears before mi. Now that we have concrete preference lists, we run the stable matching algorithm. We claim that the matching produced would have no strong instability. But this latter claim is true because any strong instability would be an instability for the match produced by the algorithm, yet we know that the algorithm produced a stable matching a matching with no instabilities.

Problem - Weak Instability

A weak instability in a perfect matching S consists of a man m and a woman w, such that their partners in S are w' and m', respectively, and one of the following holds: m prefers w to w', and w either prefers m to m' or is indifferent between these two choices; or w prefers m to m', and m either prefers w to w' or is indifferent between these two choices. In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

Solution to Weak Instability

The answer is No. The following is a simple counterexample. Let n = 2 and m1, m2 be the two men, and w1, w2 the two women. Let m1 be indifferent between w1 and w2 and let both of the women prefer m1 to m2. The choices of m2 are insignificant. There is no matching without weak stability in this example, since regardless of who was matched with m1, the other woman together with m1 would form a weak instability.

Problem - Shipping Lines

Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a schedule that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some m > n.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) No two ships can be in the same port on the same day.

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to truncate each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the truncation of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P. Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.Show that such a set of truncations can always be found, and give an algorithm to find them.

Solution to Shipping Lines

For each schedule, we have to choose a stopping port: the port in which the ship will spend the rest of the month. Implicitly, these stopping ports will define truncations of the schedules. We will say that an assignment of ships to stopping ports is acceptable if the resulting truncations satisfy the conditions of the problem - specifically, condition (†).

(Note that because of condition (†), each ship must have a distinct stopping port in any acceptable assignment.)

We set up a stable marriage problem involving ships and ports. Each ship ranks each port in chronological order of its visits to them. Each port ranks each ship in reverse chronological order of their visits to it. Now we simply have to show:

(1) A stable matching between ships and ports defines an acceptable assignment of stopping ports.

Proof. If the assignment is not acceptable, then it violates condition (†). That is, some ship S; passes through port Pk after ship S; has already stopped there. But in this case, under our preference relation above, ship S; "prefers" Pr to its actual stopping port, and port Pk "prefers" ship S; to ship S;. This contradicts the assumption that we chose a stable matching between ships and ports.

USC University of Southern California

CSCI 570 Exam 1 Review

Greedy
Komal Kadam

T/F questions

1. A greedy algorithm always makes the choice that looks best at the moment.

T/F questions

1. A greedy algorithm always makes the choice that looks best at the moment.

   Answer: True

T/F questions

2 . In the interval scheduling problem, if all intervals are of equal size, a greedy algorithm based on the earliest start time will always select the maximum number of compatible intervals.

T/F questions

2 . In the interval scheduling problem, if all intervals are of equal size, a greedy algorithm based on the earliest start time will always select the maximum number of compatible intervals.

True

You can prove it by induction

Greedy: 2 Common Methods of Proof of Optimality

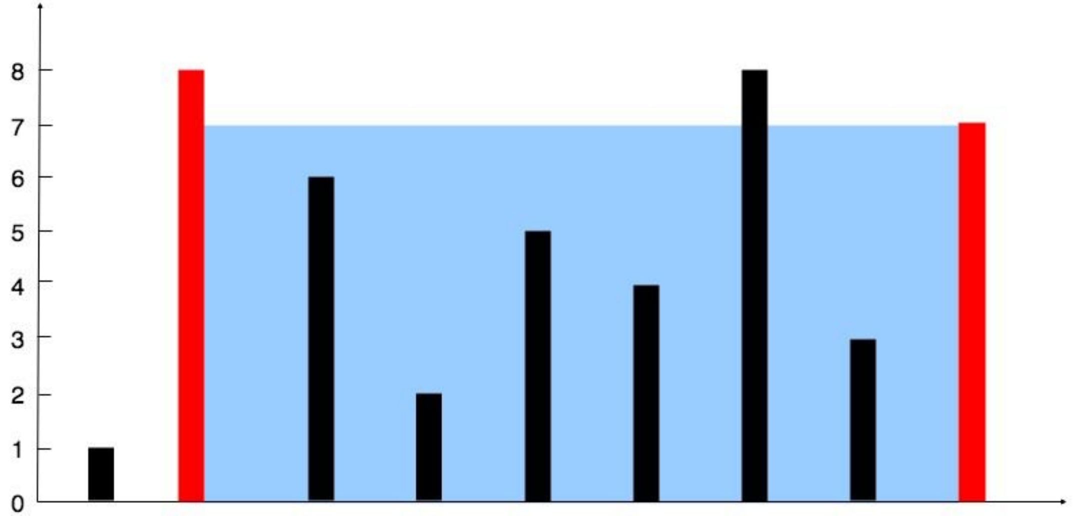
- **stay-ahead:** an inductive method to show that for each step you take, you are always ahead of any other solution according to some metric

- **exchange:** show that for your greedy solution, any swaps between steps/elements cannot achieve a better result

Question 2

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).
Find two lines that together with the x-axis form a container, such that the container contains the most water.

- Calculate *the maximum amount of water a container can store*.



Question 2

Intuition

- Be greedy
- Start with the two endpoints. As larger the distance between two lines, more water it can hold.
- At each moment, we either move left or right depending on which line is smaller in height.

Question 2
Proof

We can prove this using an exchange argument. Let us consider that there exists an optimal solution O that does NOT choose the smaller line at every point.

Let us consider left and right to be two lines. Let W be the water capacity of these two lines.
Therefore W(left,right) = (right-left) * min(height[left] , height[right])

There are two cases
Case 1: height[left] < height[right]
So W (left,right) = (right-left) * height[left]

In this case, there are two possibilities, we can go either left+1 or right-1. Optimal solution O may choose left+1. Then it is same as our solution. But optimal solution O may choose right-1

W'(left,right-1) = (right-1-left)* min(height[left],height[right-1])

<= (right-1-left) * height[left] (as height[left] < height[right])

Therefore
W' <=(right - left) * height[left]
i.e. W' <= W

Same is true for W'(left,right-2), W'(left,right-3)....

So there is no need of calculating W'(left,right-1). So this optimal solution is worse than our solution. So our solution is optimal. Case 2 can be proven similarly.

Question 2

Algorithm:

Two pointer approach.

```
left = 0
right = n

While left<right
    water = max(water,min(height[left],height[right]) * (right-left))
    if(height[left]<height[right])
        left++
    else
        right—
```

Maximize the value of water

Time complexity ?

Question 2

Algorithm:

Two pointer approach.

```
left = 0
right = n

While left<right
    water = max(water,min(height[left],height[right]) * (right-left))
    if(height[left]<height[right])
        left++
    else
        right—
```

Maximize the value of water

Time complexity ?

O(n)

Question 3

For a graph G = (V, E), a vertex cover of G is a set of vertices that includes at least one endpoint of each edge of G. Assuming our graph G is a tree, find an algorithm that finds a minimum vertex cover of G.

Question 3

Let S be the set of vertices that will be the vertices we return. If our graph G has no vertices or only 1 vertex, return S. Otherwise, add all vertices 1 layer above all our leaf vertices in the tree to S, and remove the vertices and its adjacent edges from G, as well as the leaf vertices. Recursively apply this step until we end up with 1 or no vertices in G. Return S.

Proof:

This is optimal because we have to cover the leaf edges regardless, so our cover will always either include the leaf node itself or its parent. If we include its parent, we cover just as much if not more than if we select the leaf node. The problem then reduces to a vertex cover for a subset of the original tree. This subset is smaller than the one we would have gotten from choosing the leaf node, and a vertex cover of tree is at least as large as the vertex cover of a subset of it.

Divide & Conquer, Master theorem

Pegah Jandaghi

Question 1/3

Emily has received a set of marbles as her birthday gift. She is trying to create a staircase shape with her marbles. A staircase shape contains k marbles in the k th row. Given n as the number of marbles help her to figure out the number of rows of the largest staircase she can make.

A staircase of size 4:

```
*
**
***
****
```

Solution:

The number of marbles in a staircase of size m is: $m(m+1) / 2$

We are trying to find the m such that $m(m+1)/2 < n$ and $(m+1)(m+2) > n$

We perform a binary search to find such m.

We start the process of searching with $k=n/2$ and at every step:

- If $k*(k+1) = n$ then $k=n/2$ is the solution
- If $k*(k+1) > n$ then we perform binary search in the 1…k array
- If $k*(k+1) < n$ then we perform binary search in the k, ….n array

$T(n) = T(n/2) + O(1) \rightarrow T(n) = O(\log n)$

Question 2/3

Suppose you are given a string S of length n which consists of lower case letters. Given k as the input, find the length of a substring T of S such that every character in T appears at least k times in T. T is a contiguous substring in S and can start from any index in S.

Solution:

Iterate through the string and create C the array of occurrences of the letters:

C[i, ‘b’] = number of occurrences of ‘b’’ in A[1, .., i] → O(n)

If a letter appears less than k times it should not be in T.

Suppose a_i is the letter that appeared less than k times in S. Split the string with a_i and repeat the same process for substrings. Using C computing the number of letters in each subarray takes O(1)

$T(n) = T(n-i) + T(i-1) + O(1) \rightarrow T(n) = O(n)$

Question 3/3

Find the complexity of the following recurrences using master theorem

- $T(n) = 2\,T([\sqrt{n}]) + 1$

- $T(n) = 2\,T(n/2) + \sqrt{n}$

- $T(n) = 2T(n/2)+n/\log n$

Solution

- $T(n) = 2\,T([\sqrt{n}]) + 1$

  $n = 2^m \rightarrow T(2^m) = 2\,T(2^{(m/2)}) + O(1) \rightarrow g(m) = 2g(m/2) + O(1)$

  Case 1 $\rightarrow T(n) = O(\log n)$

- $T(n) = 2\,T(n/2) + \sqrt{n}$

- $T(n) = 2T(n/2)+n/\log n$

Solution

- $T(n) = 2\, T([\sqrt{n}]) + 1$

  $n = 2^m \rightarrow T(2^m) = 2\, T(2^{(m/2)}) + O(1) \rightarrow g(m) = 2g(m/2) + O(1)$

  Case 1 $\rightarrow T(n) = O(\log n)$

- $T(n) = 2\, T(n/2) + \sqrt{n}$

  $f(n) = \sqrt{n} = n^{0.5} = O(n^{(1-e)}) \rightarrow$ Case 1 $\rightarrow T(n) = O(n)$

- $T(n) = 2T(n/2)+n/\log n$

Solution

- $T(n) = 2\ T([\sqrt{n}]) + 1$

  $n = 2^m \rightarrow T(2^m) = 2\ T(2^{(m/2)}) + O(1) \rightarrow g(m) = 2g(m/2) + O(1)$

  Case 1 → $T(n) = O(\log n)$

- $T(n) = 2\ T(n/2) + \sqrt{n}$

  $f(n) = \sqrt{n} = n^{0.5} = O(n^{(1-e)}) \rightarrow$ Case 1 → $T(n) = O(n)$

- $T(n) = 2T(n/2)+n/\log n$

  Case 1 → n, n/logn are not polynomial different and we can not find e such that $n/\log n = O(n^{(1-e)})$.
  Master theorem not applicable

Find the complexity of the following recurrences using master theorem

- $T(n) = T(n/2) + n(2 - \cos n)$

- $T(n) = 2T(n/2) + n\log n$

Find the complexity of the following recurrences using master theorem

- $T(n) = T(n/2) + n(2 - \cos n)$

  Case 3, Regularity condition: $f(n/b) < c\, f(n)$, for $c < 1$

  $f(n) = n(2\text{-}\cos n) \rightarrow$

  $n = 2\pi k \rightarrow f(n/2) = \pi k * (2\text{-}\cos \pi k)$, $f(n) = 2\pi k$

  When k is odd $\rightarrow f(n/2) = 3/2\, f(n) \rightarrow$ regularity condition not satisfied

Find the complexity of the following recurrences using master theorem

- $T(n) = T(n/2) + n(2 - \cos n)$

  Case 3, Regularity condition: $f(n/b) < c\, f(n)$, for $c < 1$

  $f(n) = n(2-\cos n) \rightarrow$

  $n = 2\pi k \rightarrow f(n/2) = \pi k * (2-\cos \pi k)$, $f(n) = 2\pi k$

  When k is odd $\rightarrow f(n/2) = 3/2\, f(n) \rightarrow$ regularity condition not satisfied

- $T(n) = 2T(n/2)+n\log n$

Find the complexity of the following recurrences using master theorem

- $T(n) = T(n/2) + n(2 - \cos n)$

  Case 3, Regularity condition: $f(n/b) < c\, f(n)$, for $c < 1$

  $f(n) = n(2 - \cos n) \rightarrow$

  $n = 2\pi k \rightarrow f(n/2) = \pi k * (2 - \cos \pi k)$, $f(n) = 2\pi k$

  When k is odd $\rightarrow f(n/2) = 3/2\, f(n) \rightarrow$ regularity condition not satisfied

- $T(n) = 2T(n/2) + n\log n$

  Case 2 $\rightarrow T(n) = O(n(\log n)^2)$

MST & Shortest Path

Mehrnoosh Mirtaheri

Question 1/4

True or False

If all edge weights of a given graph are the same, then every spanning tree of that graph is minimum.

True or False

If all edge weights of a given graph is the same, then every spanning tree of that graph is minimum.

Solution. True.
Spanning tree: Any tree that covers all nodes of a graph is called a spanning tree.

Question 2/4

MST

There are n cities and existing roads R_u. The government plans to build more roads such that any pairs of cities are connected by roads. The candidate roads are R_p and building each road has a cost c_p. Design an algorithm to decide the roads to be built such that all cities are connected by road(s) and the cost is minimized.

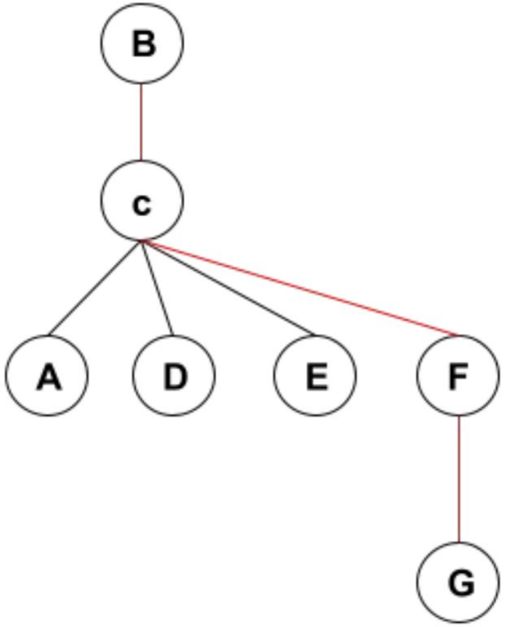
MST - Solution

1. Construct a graph with n nodes.
2. Add R_u to the graph and set edge weight to be 0.
3. Add R_p to the graph and set edge weights to be the cost.
4. Run MST algorithm, return roads that are in both R_p and MST.

Question 3/4

Shortest Path

The diameter of a graph is the maximum of the shortest paths' lengths between all pairs of nodes in graph G. Design an algorithm which computes the diameter of a connected, undirected, unweighted graph in O(mn) time, and explain why it has that runtime.



Diameter of this graph is: 3

Shortest Path - Solution

- Unweighted graph => BFS can be used for shortest path search for each source node in O(m+n)
  - For one source node s, report the maximum layer reached
- Repeat BFS for each node in O(n)
- Total time complexity: O(n(m+n))
  - Connected graph, the number of edges are at least n-1, at most n(n-1)/2, hence n=O(m)
  - => O(m+n) is O(m)
  - total time complexity: O(nm)

Question 4/4

Shortest Path

Suppose that you want to get from vertex s to vertex t in a connected undirected graph G = (V; E) with positive edge costs, but you would like to stop by vertex u (imagine that there are free burgers at u) if it is possible to do so without increasing the length of your path by more than a factor of **α.**

Describe an efficient algorithm in O(|E| log |V|) time that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (In other words, your algorithm should either return the shortest path from s to t, or the shortest path from s to t containing u, depending on the situation.)

Shortest Path - Solution

- Positive edges => Dijkstra's algorithm for shortest path from source node s, and from source node u.
    - Shortest path from s => we know ds(t) and ds(u)
    - Shortest path from u => we know du(t)
- Compare ds(u) + du(t) and α*ds(t)
    - If ds(u) + du(t) <= α*ds(t), stop by u for burger!
    - If ds(u) + du(t) > α*ds(t), go directly from s to t
- Time complexity
    - Connected graph: |V|=O(|E|)
    - Running Dijkstra's algorithm twice:
        - Binary heap: O((|V| +|E|) log |V|) => O(|E| log |V|)
        - Fibonacci heap: O(|E|+|V|log|V|) => O(|E| log |V|)