CSCI 570
Exam-1 Review

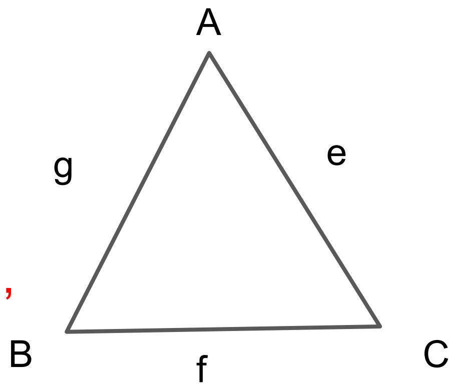
July 20, 2023

MST and Shortest Path

Junying Wang

Q2

[T/F] If the weight of each edge in a connected graph is distinct, then the graph contains exactly one unique minimum spanning tree.



True.

Proof: Suppose two MSTs T1 {AB, AC} and T2 {AB, BC} of G,
E = set of edges that is in either T1 or T2 but not both.
e is the min edge in E.
Suppose e is not in T2. Adding e to T2 creates a cycle. Then in this cycle, at least one edge, say f, is not in T1. Since e is te min edge in E, then w(e) <= w(f). And all edges have distinct weights, w(e) < w(f). Replacing f with e results in a new spanning tree with less weight than T1 and T2. Therefore, contradiction.

Q1

[T/F] If all edge weights of a given graph is the same, then every spanning tree of that graph is minimum.

True.

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

Minimum spanning tree: A spanning tree that minimizes the total weights

Q3

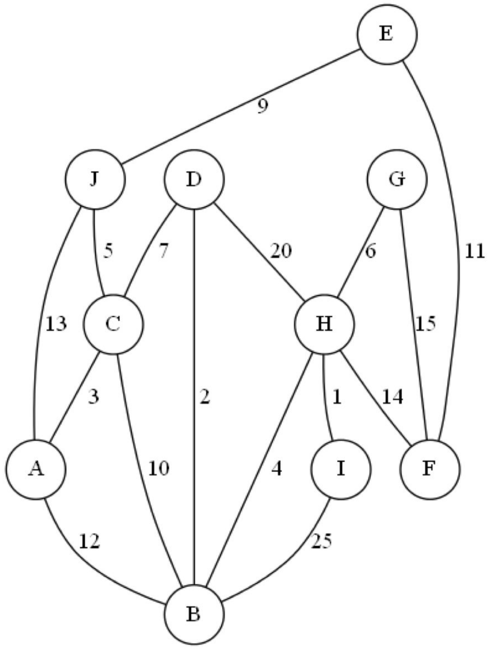
Breadth-First Search (BFS) can find the shortest path in an unweighted graph.

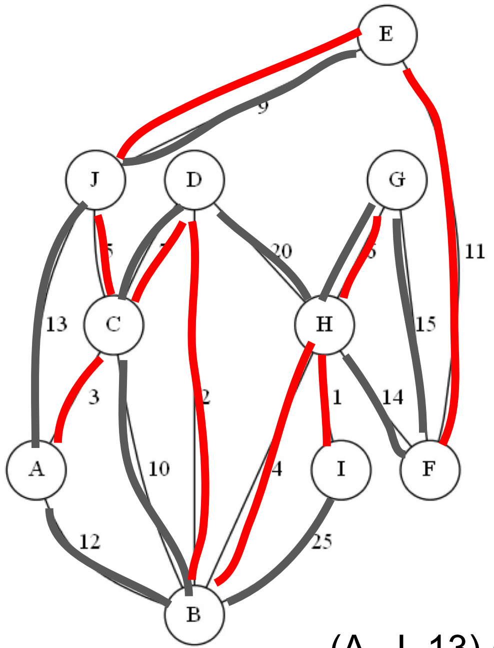
True.

In an unweighted graph, all edges have the same weight (usually considered to be 1), which means that the shortest path between two nodes is the one with the fewest number of edges.

Q4

1. Use Prim's algorithm starting at node A to compute the Minimum Spanning Tree (MST) of the following graph. In particular, write down the edges of the MST in the order in which Prim's algorithm adds them to the MST. Use the format (node1, node2) to denote an edge.





The strategy of Prim's algorithm is a greedy approach
It starts from an arbitrary node and keeps expanding the MST by adding the edge with the minimum weight, it connects a vertex in the MST to a vertex outside the MS
The process continues until all vertices are included in the MST.

(A, J, 13) (A, C, 3) (A, B, 12) → **(A, C)**

(A, J, 13) (A, B, 12) (C, J, 5) (C, D, 7) (C, B, 10) →**(C, J)**

(A, J, 13) (A, B, 12) (C, D, 7) (C, B, 10) (J, E, 9) → **(C, D)**

(A, J, 13) (A, B, 12) (C, B, 10) (J, E, 9) (B, D, 2) (D, H, 20) → **(B, D)**

(A, J, 13) (A, B, 12) (C, B, 10) (J, E, 9) (D, H, 20) (B, H, 4) (B, I, 25) → **(B, H)**

(A, J, 13) (A, B, 12) (C, B, 10) (E, J, 9) (D, H, 20) (B, I, 25) (H, I, 1) (H, G, 6) (H, F, 14)→ **(H, I)**

(A, J, 13) (A, B, 12) (C, B, 10) (E, J, 9) (D, H, 20) (B, I, 25) (H, G, 6) (H, F, 14)→ **(H, G)**

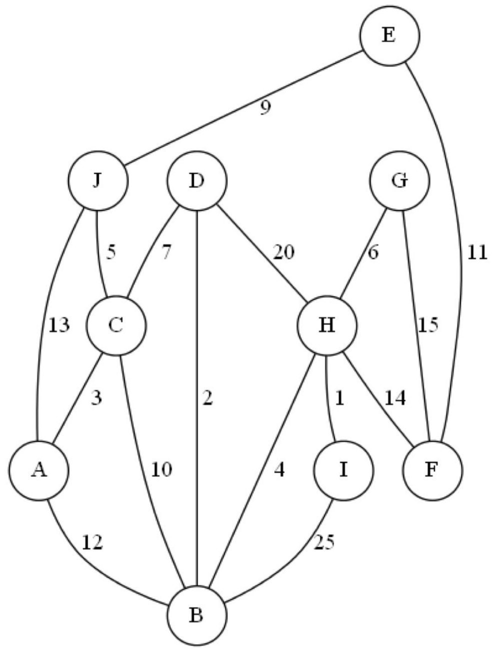
(A, J, 13) (A, B, 12) (C, B, 10) (E, J, 9) (D, H, 20) (B, I, 25) (H, F, 14), (G, F, 15)→ **(E, J)**

(A, J, 13) (A, B, 12) (C, B, 10) (D, H, 20) (B, I, 25) (H, F, 14), (G, F, 15) (E, F, 11)→ **(E, F)**

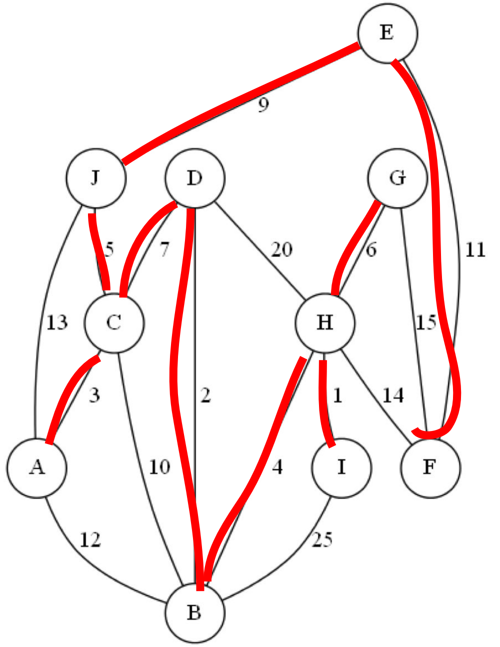
Ordering: (A,C), (C,J), (C,D), (B,D), (B,H), (H,I), (H,G), (E,J), (E,F)

Q4

2. For the same graph as above,write down the edges of the MST in the order in which Kruskal's algorithm adds them to the MST.



The strategy of Kruskal's algorithm is also a greedy approach, but it focuses on selecting edges rather than nodes. Kruskal's algorithm selects edges in ascending order of their weights, as long as adding the edge does not create a cycle in the MST.



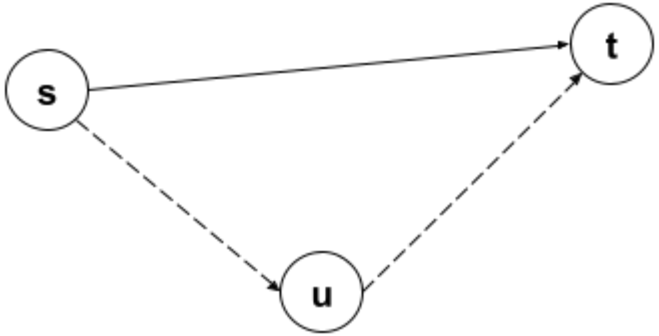
(H, I)
(B, D)
(A, C)
(B, H)
(C, J)
(G, H)
(C, D)
(E, J)
(E, F)

Ordering: (H,I), (B,D), (A,C), (B,H), (C,J), (G,H), (C,D), (E,J), (E,F)

Q5

- 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**

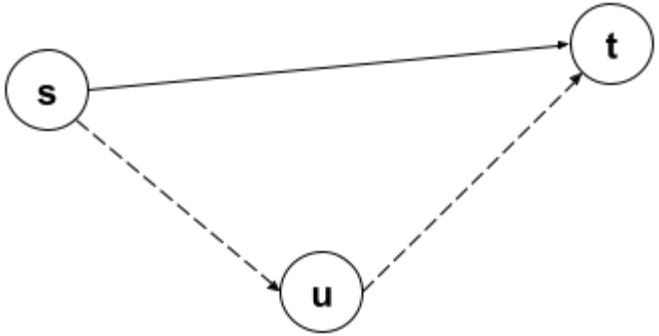
- Describe an efficient algorithm in O(|E| log |V|) time that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (In other words, your algorithm should either return the shortest path from s to t, or the shortest path from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



Q5

- Positive edges => Dijkstra's algorithm for shortest path from source node s, and from source node u
  - shortest path from s => we know $d_s(t)$ and $d_s(u)$
  - Shortest path from u => we know $d_u(t)$
- Compare $d_s(u) + d_u(t)$ and $\alpha^* d_s(t)$
  - If $d_s(u) + d_u(t) <= \alpha^* d_s(t)$, stop by u for burger!
  - If $d_s(u) + d_u(t) > \alpha^* d_s(t)$, go directly from s to t
- Time complexity

  Running Dijkstra's algorithm twice:
  - Binary heap: O((|V| +|E|) log |V|)
  - Fibonacci heap: O(|E|+|V|log|V|)



Greedy Algorithms

KR Zentner

What are greedy algorithms?

Greedy algorithms “just take the best option” each step.

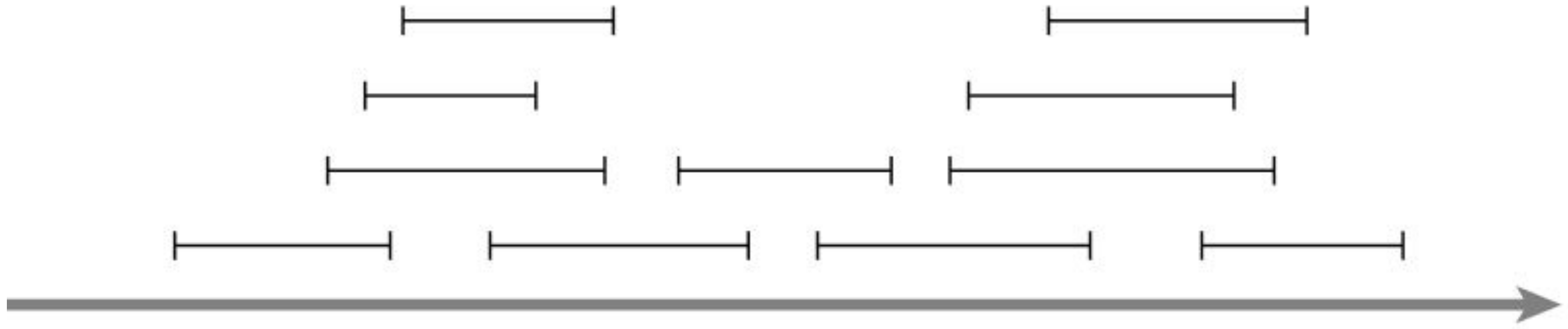
Are greedy algorithms always linear time (O(n))?

- No! Often use a heap or sort (O(n log n)) time.

Do greedy algorithms compute sub-optimal solutions?

- Not always! Some (not all) greedy algorithms are optimal.

The Interval Scheduling Problem



Want to find the largest set of requests / intervals that don’t overlap.

How to find optimal greedy algorithms (when possible)?

Find a rule for choosing next “option.”

For interval scheduling: Accept the request that ends first (and doesn’t overlap with any previous requests).

Prove that the rule is optimal. Usually, this is done by assuming there were a better option than the one the rule chose, and showing a contradiction.

Interval Scheduling: Bad Solutions

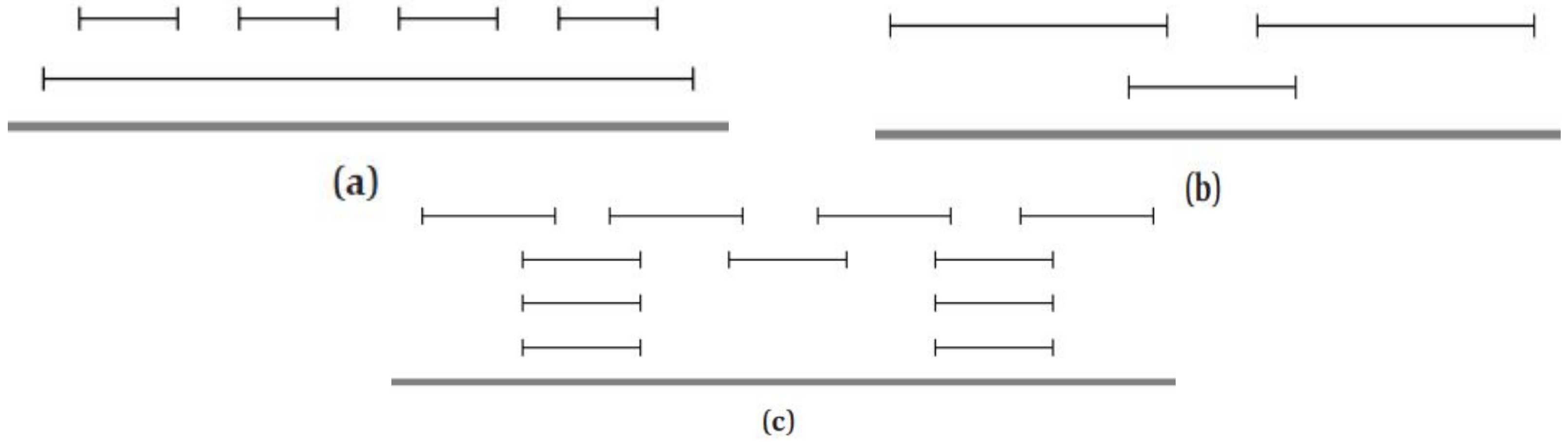
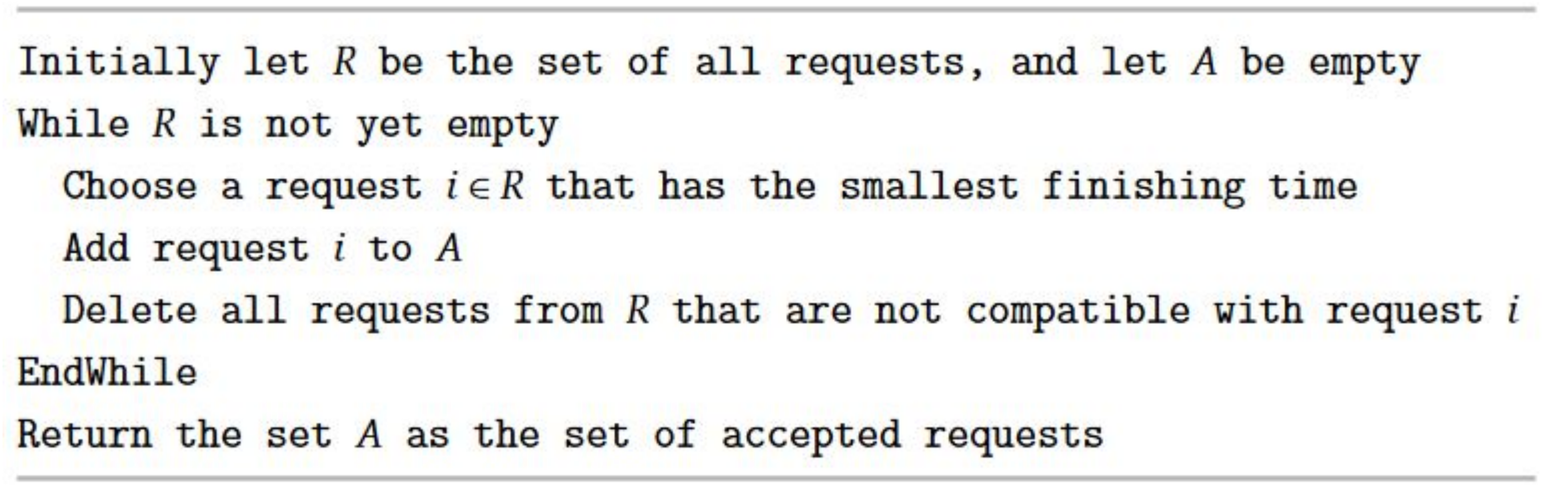


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Interval Scheduling: Optimal Solution

Accept the request that ends first (and doesn't overlap with any previous requests).



```
Initially let $R$ be the set of all requests, and let $A$ be empty
While $R$ is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request $i$ to $A$
  Delete all requests from $R$ that are not compatible with request $i$
EndWhile
Return the set $A$ as the set of accepted requests
```

Interval Scheduling: Optimality Proof Sketch

Suppose there were a sequence of requests O that is bigger than the sequence A returned by the greedy algorithm.

But then there must be some element O[i] not in A where O[i] finishes before A[i] (otherwise, A would not have “run out of space” before O).

But our algorithm would have chosen O[i] instead of A[i], so there is a contradiction and thus no O larger than A.

Interval Scheduling: Runtime

Need to sort requests by first to finish O(n log n).

Need to eliminate all requests that overlap with a chosen request. But, we will check each request to eliminate it at most once (either we eliminate it or it’s next in our sequence), taking O(n) time in total.

So this greedy algorithm is O(n log n).

Interval Scheduling: Runtime

Need to sort requests by first to finish O(n log n).

Need to eliminate all requests that overlap with a chosen request. But, we will check each request to eliminate it at most once (either we eliminate it or it’s next in our sequence), taking O(n) time in total.

So this greedy algorithm is O(n log n).

Important Example: Huffman Codes

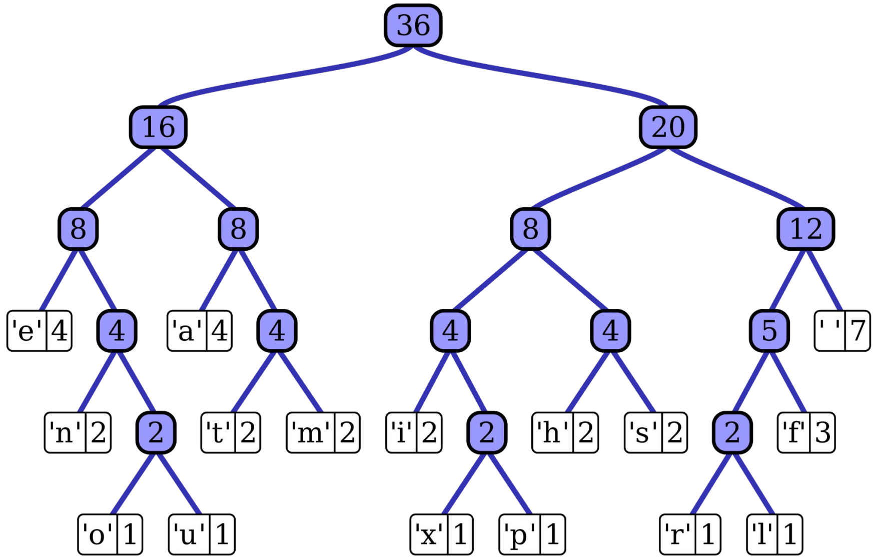
Input: a set of symbols (s[i]) and the frequency of each (f[i]).

Output: a binary code sequence (c[i]) for each symbol that minimizes the average code length under the symbol frequencies (i.e. minimize sum(f[i] * len(c[i])).

Need to make sure no code sequence c[i] is a prefix of another code c[j].

Huffman Code Generation: Overview

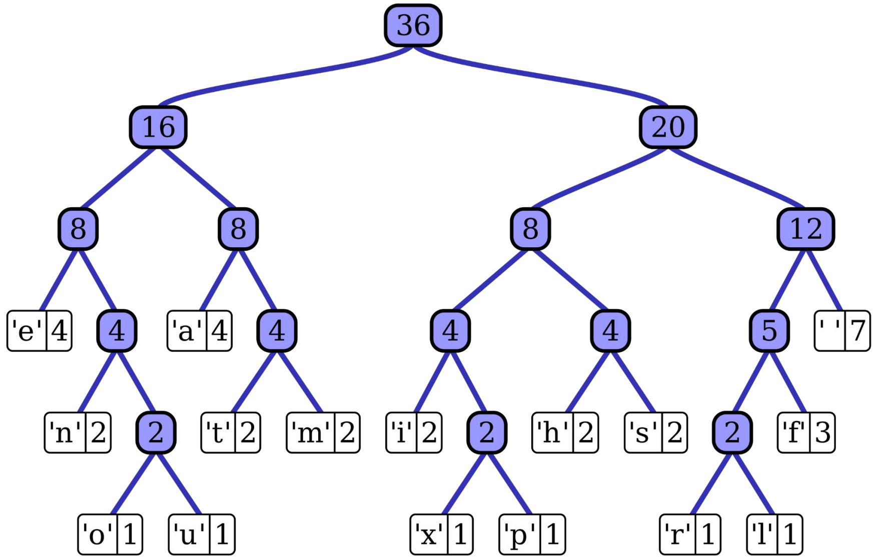
To avoid prefix overlap, create a binary tree with symbols only at the leaves where the path down the tree is the binary code.



By Meteficha - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2875155

Huffman Code Generation: Overview

Key Idea: Build tree by using a min-heap of tree nodes ordered by total frequency in subtree.



By Meteficha - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2875155

Huffman Code Generation: Algorithm

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   a. Remove the two nodes of highest priority (lowest frequency) from the queue
   b. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes’ frequencies.
   c. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Huffman Code Generation: Runtime

1. Create heap in O(n) time.
2. Remove two elements O(log n), add one element back into heap O(log n).
3. Repeat step 2 until heap has only one element, taking O(n log n).

Therefore overall runtime is O(n log n).

Heaps and Amortized analysis

Leili Tavabi

Q1

Imagine you are in a car, and you want to travel on a road. You need to go up and down several hills on this road. Fortunately, your car is especially equipped with a nitrogen booster in addition to a regular gasoline burning engine. You can only use the booster for a limited number of times (k), and you have a certain amount of gas (m gallons) in your tank at the start of your journey.

The car does not need to burn any gas or use the booster when you are coming down the hills or going straight on a flat road. But for each hill (with elevation gain of "h") that you want to go up, you need to use one of these two options:

1) You can use the engine and burn 'h' gallons of gas, or

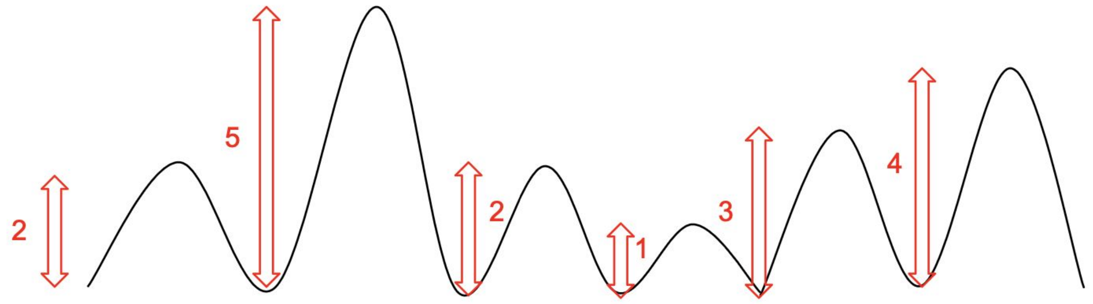
2) You can spend one of your k special boosters (if you have any left). The booster can take the car from any height to any height.

Q1

You are given an array of heights for the hills you need to pass hills [0, n-1], in the order in which you encounter the hills on your journey. Your job is to find the maximum number of hills that you can pass with the given amount of gas (m gallons) and the given number of boosters (k). In other words, you need to design an algorithm so that you can reach the furthest on this road by optimally using either your boosters or gas to climb up each hill.

Your algorithm should run in O(n logk). Explain your algorithm. How much space does it take?

Q1 - Example



boosters=2
gasoline=8

Q1 - Solution

- It’s efficient to save the boosters for the taller hills
- In the beginning, you can assign the k boosters to the first k hills
- For every following hill after k, you compare the height h[i] with the min height of a booster-assigned hill $\min(h_{booster})$
    - If $h[i] <= \min(h_{booster})$ then the booster is already efficiently assigned. Then use gasoline for the current hill
    - If $h[i] > \min(h_{booster})$, then reassign the booster to the current hill
- Continue until you run out of gasoline or reach the end of the hills

We always need to access the min height of the booster-assigned hills.

Therefore we assign the booster-assigned hills to a min-heap

Q1 - Code (python)

```
import heapq            #Python library for using heaps

def furthestHill(heights: list[int], k: int, gas: int) -> int:

        min_heap = []
        n = len(heights)

        for i in range(n):

                #assign boosters for the first k hills
                if len(min_heap) < k:
                        heapq.heappush(min_heap, heights[i])

                #if the current hill is shorter that the shortest booster-assigned hill, use gas
                elif heights[i] <= min_heap[0]:
                        gas -= heights[i]

                #if the current hill is taller than the shortest booster-assigned hill, reassign the booster to the current hill
                else:
                        reassigned_hill = heapq.heappop(min_heap)
                        gas -= reassigned_hill
                        heapq.heappush(min_heap, heights[i])

                #If the gas tank is negative, you can’t reach the current hill
                if gas < 0:
                        return i-1
                #If the gas tank is empty, return the current hill
                elif gas == 0:
                        return i

        #If you complete the for loop withouting emptying the gas tank, then you have reached the final hill
        return n-1
```

Q1-Complexity Analysis

Maintaining a min-heap of size k

Space complexity O(k)

In the worst case, for every hill you encounter you perform a push or pop+push on to your stack, each with O(logk) complexity

Time complexity O(nlogk)

Q2

Consider the problem of storing a very large binary counter. Say we decide to use an array, where each entry A[i] stores the i-th bit.

We will analyze the running time of the operation of counting using this representation, so the sequence of operations is a sequence of increments of the counter.

We will use the standard way of incrementing the counter, which is to toggle the lowest order bit. If that bit switches to a 0 we toggle the next higher order bit, and so forth until the bit that we toggle switches to a 1 at which point we stop.

Assume we start from counter 0 and perform n increment operations until we reach n. What is the amortized cost of these sequence of n increments?

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

   0     0             0     0     0     1         1

   0     0             0     0     1     0         2

   0     0             0     0     1     1         1

   0     0             0     1     0     0         3

   0     0             0     1     0     1         1

   0     0             0     1     1     0         2
```

Q2

- When the result of the increment operation is n, the number of changed bits is at most 1+floor(logn) – since number of bits for n is logn
- So in a traditional worst case analysis, the cost of counting up to n (a sequence of n increments), is O(nlogn)
- But does each increment really cost O(logn)?

Q2 - Amortized analysis

Aggregate method

Instead of obtaining a worst case bound for all bits, let's consider the bit changes for each individual bit, and sum those up to bound the total

How often is A[0] toggled? Every time per increment – n

How often is A[1] toggled? Every other time per increment – floor(n/2)

How often is A[2] toggled? Every 4th time per increment – floor(n/4)

n + floor(n/2) + floor(n/4) + … <= n + n/2 + n/4 <= 2n

So the amortized cost of an increment is 2n/n=2, and the time to count from zero to n (n increments) is 2n=O(n)

Q2 - Amortized analysis

Banker’s/Accounting method

- Each operation pays a certain amount to a central pot, but some operations that require more resources may pay for it with money from the central pot
- For each increment, how much money is needed to allocate to each increment to have banked up enough money to pay for all the toggles as they occur?
- Having used the aggregate method, $2 per increment would be a good guess, but let’s see if it covers the cost of all operations within a sequence of increments

Q2 - Amortized analysis

What happens during an increment?

- The m-th bit changes from 0 to 1
- Some number of lower order bits (say m) change from 1 to 0

How do we charge the operations?

- We charge \$2 to each increment, we pay \$1 for flipping the m-th bit from 0 to 1 and save the remaining \$1 at the m-th bit for later use
- For the m bits we had to change from 1 to 0, we pay for those with \$1 left at those bits by previous operations

Q2 - Amortized analysis

- In other words, every 1 bit has \$1 associated with it that can be used the next time it has to flip back to 0
- Every increment would have only one bit flipping from 0 to 1, which is charged \$2 (\$1 is immediately used and \$1 is saved at that bit for future use)
- All the remaining bits flipping from 1 to 0, use the stored \$1 from previous operations
- This shows that amortized cost of the increment is \$2, and the overall time is O(n) for n increments

Divide and Conquer

Sarik Ghazarian

Q1: Local Minimum

Consider an array A containing n **distinct** integers. We define a local minimum of A to be an x such that x=$A[i]$, for some $0 \leq i < n$, with $A[i-1] > A[i]$ and $A[i] < A[i+1]$. In other words, a local minimum x is less than its neighbors in A (for boundary elements, there is **only one neighbor** to consider). As an example, suppose A = [10, **3**, 6, 13, 15, 19, **18**]. Then A has two local minima: 3 and 18.

Part a: Describe an algorithm using the divide and conquer technique to find **a** local minimum. Note that A might have multiple local minima, but you only have to locate and return one.

Part b: Express a recurrence equation for the running time of your algorithm and solve the recurrence using Master Theorem.

Q1: Local Minimum

Part a:

Let m=n/2 and examine if A[m] is local minimum:

1. A[m] is local minimum or A has only one member -> return it.
2. A[m] > A[m-1] -> the left half the array must contain a local minimum so do the same thing on the left half.
3. A[m] > A[m+1] -> the right half of the array must contain a local minimum so do the same thing on the right half.

$A = [10, 3, 6, \underline{13}, 15, 19, 18]$

$A = [10, \underline{3}, 6]$

Q1: Local Minimum

Part b:

T(n) = T(n/2) + θ(1)

f(n) = 1

a = 1, b =2, $log_b^a = 0 \Rightarrow n^{log_b^a} = 1$

Case 2 of Master Theorem

T(n) = θ(logn)

Q2: find *x* in array

- Consider a two-dimensional array A[1:n,1:n] of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

Design a divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm.

Your algorithm should take strictly less than O(n^2) time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.

Q2: find *x* in array

Let m be the middle element of the full matrix.

If x == m return True

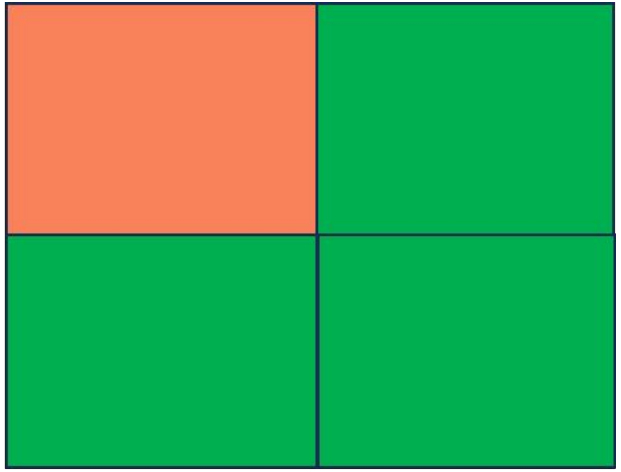
1. If x < m:

   Eliminate A$[\frac{n}{2}..n, \frac{n}{2}..n]$

2. If x > m:

   Eliminate A$[1..\frac{n}{2}, 1..\frac{n}{2}]$

recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$



x>m

Q2: find *x* in array

Master Theorem

T(n) = $3T(\frac{n}{2})$+O(1)

f(n) = 1

a = 3, b =2, $log_b^a = log_2^3 \Rightarrow n^{log_b^a} = n^{log_2^3}$

Case 1 of Master Theorem → f(n) = $O(n^{log_2^3})$

T(n) = $\theta(n^{\log_2 3})$

Q2: find *x* in array

Any linear time Divide and Conquer solution?

Let m be the top rightmost element of the full matrix.

If x == m return True

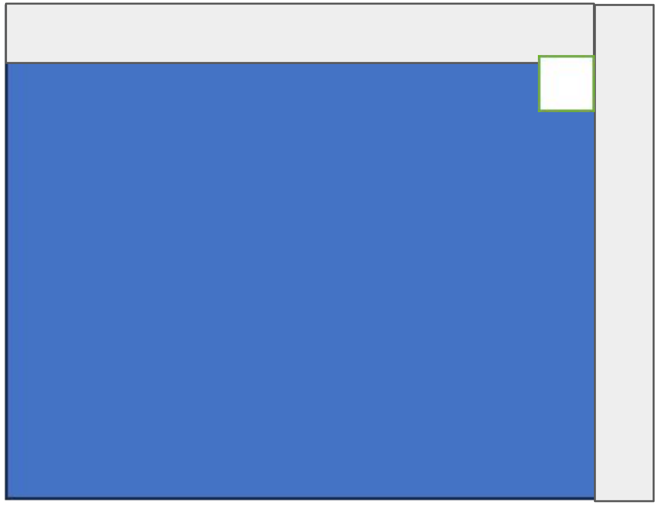
1. If x < m:

    Eliminate A[1..n, n]

2. If x > m:

    Eliminate A[1, 1..n]

recursively search in the remaining A



Q3: Find maximum element of array

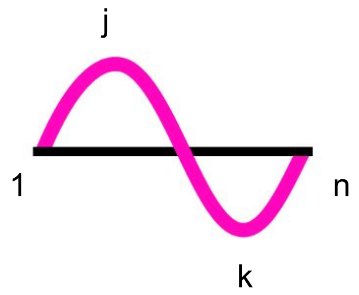
The numbers stored in array A[1..n] represent the values of a function at different points in time. We know that the function behaves the following way:

$A[1] < A[2] < \ldots . < A[j] \quad 1 < j < n$

$A[j] > A[j+1] > \ldots . > A[k] \; j < k < n$

$A[k] < A[k+1] < \ldots . < A[n]$

$A[n] < A[1]$



Your task is to design a divide-and-conquer algorithm to find the maximum element of the array. Your algorithm must run in better than linear time. You need to provide complexity analysis of your algorithm.

Note: we don’t know the exact values of j and k, we only know their range as given above.

Q3: Find maximum element of array

Consider an algorithm $ALG(A, n)$ that takes as input an array A of size n, where array A either satisfies all four conditions above or it satisfies the first two conditions with $k = n$. Also, let $ALG(A, n)$ output the maximum element in A.

Store $l = A[0]$ and $r = A[n]$. $ALG(A, n)$ consists of the following steps

a. If $n \leq 4$ output the maximum element.

b. If $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} + 1\right]$ and $A\left[\frac{n}{2}\right] > A\left[\frac{n}{2} - 1\right]$ then return $A\left[\frac{n}{2}\right]$.

c. If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} + 1\right]$

  i. If $A\left[\frac{n}{2}\right] > l$ then return $ALG\left(A\left[\frac{n}{2} + 1 : n\right], \frac{n}{2}\right)$

  ii. If $A\left[\frac{n}{2}\right] < r$ then return $ALG\left(A\left[1 : \frac{n}{2}\right], \frac{n}{2}\right)$.

d. If $A\left[\frac{n}{2}\right] < A\left[\frac{n}{2} - 1\right]$ then return $ALG\left(A\left[1 : \frac{n}{2}\right], \frac{n}{2}\right)$.

Q3: Find maximum element of array

To see why this is correct, observe that step (a) covers the base case for termination of the algorithm, step (b) covers the case when $\frac{n}{2} = j$, step (d) covers the case when $j < \frac{n}{2} \leq k$, step (c)(i) covers the case when $1 < \frac{n}{2} < j$, and finally step (c)(ii) covers the case of $k < \frac{n}{2} < n$.

For complexity analysis, notice that at any stage that is not the final stage of the algorithm, exactly one of step (c)(i), (c)(ii), or (d) is executed so that the associated recurrence is $T(n) = T\left(\frac{n}{2}\right) + O(1)$, implying that $T(n) = O(\log n)$ by Master's Theorem.

Dynamic Programming

Kiran Lekkala

DP Example-1

Given a value N, if we want to make change for N cents, and we have the denominations given by *coins[0:n-1]* and we have an infinite supply of each, how many ways can we make the change? The order of coins doesn’t matter.

DP Example-1

Given a value N, if we want to make change for N cents, and we have the denominations given by *coins[0:n-1]* and we have an infinite supply of each, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and *coins* = *{1,2,3}*, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

DP Example-1

Let OPT(i, j) denote number of ways to make up a value j using the first i coins.

Following is the recurrence relation:

$$OPT(i,j) = OPT(i-1,j) + OPT(i, j - coins(i-1))$$

Base cases:

1. $OPT(0,j) = 0$
2. $OPT(i,0) = 1$
3. $OPT(0,0) = 1$

DP Example-1

	0	1	2	3	4	5
{}	1	0	0	0	0	0
{1}	1	1	1	1	1	1
{1,2}	1	1	2	2	3	3
{1, 2, 5}	1	1	2	2	3	4

DP Example-2: Egg dropping puzzle

Suppose you are in an n story building and you have k eggs in your bag. You want to find out the lowest floor from which dropping an egg will break it.

DP Example-2: Egg dropping puzzle

Following are the constraints:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks from a fall, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.
- All eggs break when they are dropped from the n-th floor.

Remember, We are finding the least amount of drops to find the threshold floor and not the threshold floor itself!

DP Example-2: Egg dropping puzzle

Suppose you are in an n story building and you have k eggs in your bag. You want to find out the lowest floor from which dropping an egg will break it. What is the minimum number of egg-dropping trials that is sufficient for finding out the answer in all possible cases?

For example, if we the building has 100 floors and we have 2 eggs, then we can find the answer in 14 trials!

DP Example-2: Base Cases

- If we there are 5 floors and 1 egg, we need to do a trial every floor and so worst case k=n=5
- If there are k eggs and 0 floors, we would need to do 0 trials
- If there are k eggs and 1 floor, we need to do 1 trial

DP Example-2: Solution

Let $O(n, k)$ denote the number of trials required for an `n` story building and `k` eggs, knowing that dropping an egg from the last floor will break it.

Suppose the first move we do is to drop an egg from floor $i < n$.

1. If the egg breaks, then we learn that ans <= i, but we are left with k-1 eggs. To find out the answer from this state we would need extra $O(i - 1, k - 1)$steps.

2. If the egg doesn't break, then we learn that ans > i and we still have k eggs. All the above (n - i) floors are the possible candidates and we have k eggs. Therefore, we would need to do $O(n - i, k)$ extra steps.

DP Example-2: Solution

$$O(n,k) = \min_{1 \le i < n} \left[1 + \max\{O(i-1,k-1), O(n-i,k)\}\right]$$

Base cases:

1. $O(n,k) = 0 \quad \text{if } n = 0$
2. $O(n,k) = 1 \quad \text{if } n = 1$
3. $O(n,k) = n \quad \text{if } k = 1$

Complexity: $O(n^2k)$

DP Example-2: Table

Eggs\Floors	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1	2	2	3	3	3
3	0	1	2	2	3	3	3