Exam 1 Review

Greedy

Ideas

When given a problem, your “first instinct” on how to solve it is often the greedy approach. (e.g. “just sort the array”, “just pick the largest number”)

Runtime is usually the best: typically O(n), or if sorting required, O(n log n).

For most problems, it usually takes a bit/a lot more thinking to get an optimal solution, but for some interesting problems, the greedy approach works right off the bat. The hard part with these problems is usually not creating the algorithm but rather proving that the algorithm is optimal.

Two Ways to Prove

Greedy Stays Ahead

- Think of a solution to the problem as a sequence of instructions to be carried out over a time process.
- Prove that at each time step, the greedy solution gives you the “best” instruction.
  - Let *x* be any other sequence of instructions. Show that greedy beats *x*.
- Explicitly use mathematical induction.
  - At time step *i*, the greedy instruction *g[i]* is better than *x[i]*.

Exchange Argument

- If it doesn’t make sense to think of the problem as a time process, see if you can think of the solution as an array.
- Define a “greedy rule” that all positions in your array fulfill (e.g. *g[i] < g[i+1]*).
- Let *x* be any other array, i.e. one that at some point does not satisfy the greedy rule. Show that *exchanging* two adjacent positions in the array to satisfy the greedy rule does not make the array any worse.
- ⇔ Implicitly, the greedy array is no worse than any other array. ⇔ It is optimal.

Problem

Suppose you are working at a help desk. The moment you start your shift, you are greeted by *n* customers seeking help. Each customer needs 1 minute of your time, and you can only help one customer at a time, so you help your first customer at time 1, your second at time 2, etc. You decide what order to help customers in.

Each customer *i* has an associated “annoyingness” $a(i) > 0$. Each minute, *i* generates $a(i)$ units of annoyingness until you have helped him or her.

Give an algorithm to find an order in which you should serve customers to minimize *total annoyingness*.

Solution

Simply serve the most annoying customers first. Sort them in descending order of annoyingness and let that be the output. Runtime is $O(n \log n)$.

To prove correctness, we may try either “greedy stays ahead” or an “exchange argument.”

Solution: GSA

Let *g* be the greedy ordering; we will show by induction that at each minute, *g* outperforms any other ordering *u*.

- At minute 1, *g* picks the most annoying customer *c*, who afterwards does not contribute *a(c)* to total annoyingness. Let *u* be some other ordering that picks a different first customer *k*. After minute 1, *u* no longer has *a(k)* contributing to total annoyingness, but it will still have *a(c)* contributing. Since *a(c) > a(k)* by construction, the total annoyingness of *u* must be greater than the total annoyingness of *g*.
- Suppose we know *g* beats all other orderings up to some time *i*. We wish to show that it will also beat all other orderings up to time *i+1*. The proof is extremely similar to that above; let *u* be an ordering that is identical to *g* for the first *i* customers and then picks a different *i+1*th customer, and show that thereafter, the annoyance of *u* is greater than the annoyance of *g*.

Solution: EA

Let *g* be the greedy ordering. Let *u* be any other ordering. We will show that *g* is strictly better than *u* by showing *u* is improved by making it slightly more like *g*.

- To formalize this, we define a “greedy rule” that every entry in *g* can be said to follow.
  - “For each index *i*, we have that the annoyance of customer *g(i)* is greater than that of customer *g(i+1)*.”
- Since *g != u*, we know that at at least one index *i* this is false, i.e. *u(i)* is less annoying than *u(i+1)*.
- We will show that simply swapping these two customers reduces the total annoyingness of *u*.
- Let *S* be the sum of all customer annoyingnesses under the ordering *u* besides customers *i* and *i+1*. The total annoyingness of *u* is $S + i * a_{u(i)} + (i+1) * a_{u(i+1)}$.
- If we swap the order in which these two customers are served, we have a new annoyingness of $S + ia_{u(i+1)} + (i+1)a_{u(i)}$.
- Since $a_{u(i+1)} > a_{u(i)}$ by assumption, it is clear the new annoyingness is less than the old.
- We have shown that for *any* ordering that is not greedy, we can conduct a simple “greedy adjacent swap” to make it more optimal than before.
- Any array can be transformed into the greedy array with a finite number of greedy adjacent swaps.
- Thus, any array can be made more optimal by turning it into the greedy array. Thus, the greedy array is the most optimal.

Stable Matching

Gale-Shapley T/F

- GS will always correctly return a stable matching.
  - True!
- The proposing side receives their worst valid partners.
  - False! The proposing side receives their best valid partners. The accept/reject side receives their worst valid partners.
- The order in which men propose will not affect the outcome of GS.
  - True!

Indifferent Stable Matching

We will consider a version of the Stable Matching problem in which men and women can be indifferent between certain options. We have a set M of n men and a set W of n women. Each man and woman ranks the members of the opposite gender, but now we allow ties in the ranking.

E.g.: $n = 4$. $w = \{m_1 > m_2 = m_3 > m_4\}$. We 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

Strong Instability: In a perfect matching S, there exists an unmatched pair *m*-*w* that prefer each other to their current partner (same as standard GS!)

Problem: 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

Yes!

→ Break the ties in some fashion and then run the stable matching algorithm on the resulting preference lists. For example, we can break the ties lexicographically – that is, if a man m is indifferent between two women w_i and w_j then w_i appears before w_j, since $i < j$ (similar reasoning for w).

With concrete preference lists, we run the GS stable matching algorithm. We claim that the matching produced would have no strong instability. This claim is true because GS does not produce any strong instabilities in its final matching.

Problem – Weak Instability with Indifference

Weak Instability: In a matching S, there consists of a man *m* and a woman *w*, such that their partners in S are *w′* and *m′*, respectively. The pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Formally, one of the following holds:

1) *m* prefers *w* to *w′*, and *w* either prefers *m* to *m′* or is indifferent between them
2) *w* prefers *m* to *m′*, and *m* either prefers *w* to *w′* or is indifferent between them

Problem: 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

No :(

Counterexample: Let $n = 2$ and m_1, m_2 be the two men, and w_1, w_2 the two women.

Let $m_1 = \{w_1 = w_2\}$

Let $w_1 = \{ m_1 > m_2\}$, and $w_2 = \{m_1 > m_2\}$

The choices of m_2 are insignificant.

There is no matching without weak stability in this example, since regardless of who was matched with m_1, the other woman and m_1 would form a weak instability.

Example Problem #2

We have a set M of *n* men and a set W of *n* women. Suppose all women have identical rankings. Propose a simpler variant of GS that does not have to break any engagements and computes a stable matching.

Hint: Consider the order of proposals.

Problem #2 Solution

Algorithm: Similar to GS, but the men will propose in the order of the women's rankings (top-ranked man proposes first, bottom-ranked man proposes last).

Proof (no engagements broken): by Contradiction → There is a woman, w, who breaks an engagement by choosing a man, m', over her previous engagement, m. This means that w must prefer m' over m. However, since men propose in the order of the women's rankings and m proposed before m', m must be ranked higher than m'. Contradiction.

Proof (stable matching): by Contradiction → There exists an unmatched pair, (m, w') such that m prefers w' to w, his assigned partner, and w' prefers m to m', her assigned partner. m must have proposed to w' before w. If w' was single: she would have matched with m, and since she will not break the engagement, she cannot end up with m'. Else, w' would have already been engaged with an m'' > m > m', and will not break the engagement with m''. Contradiction.

Complexity Analysis + BFS/DFS

True/False

If $f=O(g)$ and $g=O(h)$, then $f=O(h)$.

True/False

If $f=O(g)$ and $g=O(h)$, then $f=O(h)$

True

Problem 1

What is the big-O of this function?

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

Problem 1

What is the big-O of this function?

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

Solution: O(n^3) only the worst one matters

Problem 2

Give upper bound for the following code.

```
k = 1
for i = 1 to n
    j = 1
    while j <= i
        k = k + 1
        j = j * 2
```

Solution 2

O(nlog(n))

Give upper bound for the following code.

```
k = 1
for i = 1 to n
    j = 1
    while j <= i
        k = k + 1
        j = j * 2
```

True/False

BFS can be used to find the shortest path between two nodes in any graph as long as the edge costs are all positive.

True/False

BFS can be used to find the shortest path between two nodes in any graph as long as the edge costs are all positive.

False

True/False

For some graphs BFS and DFS trees can be the same.

True/False

For some graphs BFS and DFS trees can be the same.

True

Problem 1 (Number of islands)

Given an m x n 2D binary grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Eg: [["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]]

Solution 1 (Number of islands)

1. Iterate through all the cells of the matrix and start DFS/BFS search if the cell has 1 as it’s value and is not visited yet.
2. Increase the count of islands by one at the end of each DFS/BFS search from a cell.

Heap + Amortized Analysis

Problem: Find the smallest range

- Input: M sorted lists $A_1, A_2, \ldots, A_M$ with N elements in total
- Output: tuple (l, h) s.t.
  - At least an element in A_i is in range $[l, h]$
  - Minimum $h - l$

[3, 6, 8, 10, 15]
[1, 5, 12]
[4, 8, 15, 16]
[2, 6]

[4, 6]

Solution

Key idea: Maintain a priority queue that contains one element from each list

1. Collect the first element from each list
2. Form a min-heap *H* and maintain a variable *K*, which is the maximum in *H*
3. At each iteration, pop *H.top()*
4. Add the next element *x* that stays in the same list as *H.top()*
5. If *x* > *K,* then update *K* to *x*

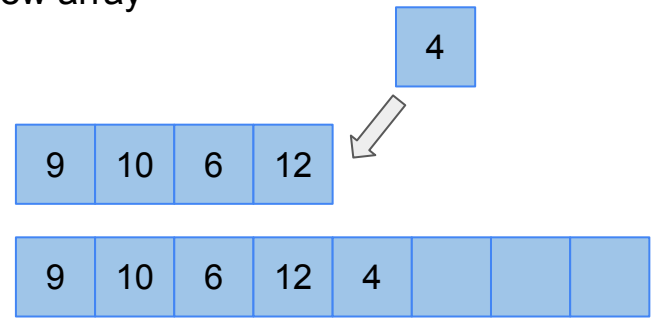
Time Complexity: O(N log M)

Traverse all the N elements, each requiring a heapify operation: O(log M)

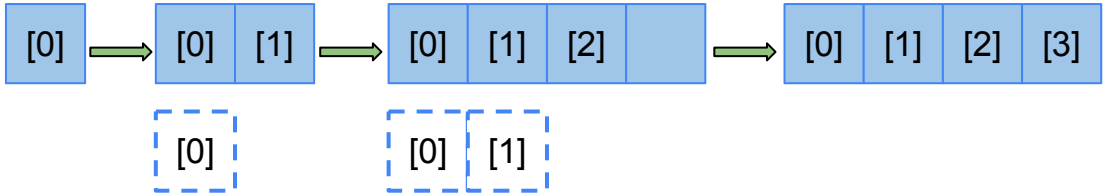
Problem: Dynamic Array (table doubling)

Definition

- Initialized of size *m*=1
- When inserting a new element, if the total number of elements *n* > size *m*
  - Create (**Allocate**) a new array of size *2m*
  - **Copy** all existing elements to the new array
  - **Append** the new element



Aggregation Method



Assume that allocating memory is **free**, the cost sequence of inserting each element will be:

Array	[0]	**[1]**	**[2]**	[3]	**[4]**	[5]	[6]	[7]	**[8]**	[9]	[10]	[11]	[12]
Append cost	1	**1**	**1**	1	**1**	1	1	1	**1**	1	1	1	1
Copy cost	0	**1**	**2**	0	**4**	0	0	0	**8**	0	0	0	0
Sum	1	**2**	**3**	1	**5**	1	1	1	**9**	1	1	1	1

- Sometimes we simply do insertion; sometimes we do insertion with copying
- It's not fair to take the worst case of a single operation → **average cost**

Array	[0]	**[1]**	**[2]**	[3]	**[4]**	[5]	[6]	[7]	**[8]**	[9]	[10]	[11]	[12]
Append cost	1	**1**	**1**	1	**1**	1	1	1	**1**	1	1	1	1
Copy cost	0	**1**	**2**	0	**4**	0	0	0	**8**	0	0	0	0
Sum	1	**2**	**3**	1	**5**	1	1	1	**9**	1	1	1	1

Cost of inserting element i = $\begin{cases} i + 1, & \text{if } i = 2^k \\ 1, & \text{otherwise} \end{cases}$

In the worst case, the total number of elements n can be 2^K
(double the size of array but never use them)

Total insertion cost = n

Total copy cost = $1 + 2 + 4 + \ldots + 2^K = 2n - 1 = O(n)$

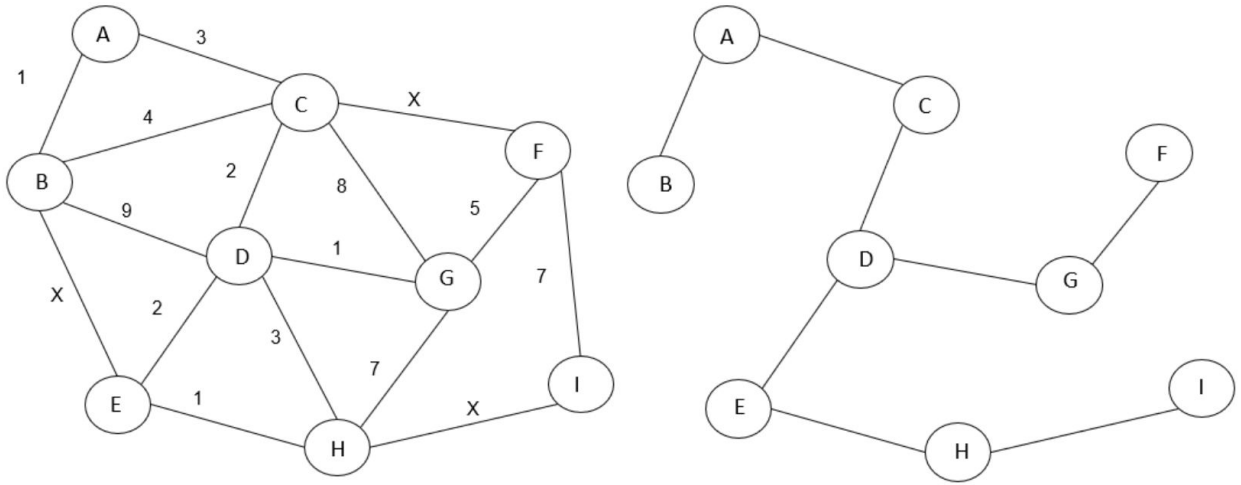
Amortized cost = O(n) / n = O(1)

Total cost = O(n)

MST + Shortest Path

Problem 1

Consider the weighted undirected graph *G* on the left (see below graphs). Suppose the graph *T* on the right is the unique MST of *G.* Find the value of X, assuming X is an integer. You must provide the reasoning for your answer.



Solution 1

1. If $X < 5$ then FG will not be the lowest connection cost between F and the rest of the MST. So, $X \geq 5$. But if X=5 then the MST will not be unique since FC can replace FG without affecting the cost of the MST. So it must be that $X > 5$.

2. If $X > 7$ then FI will be the lowest connection cost between I and the rest of the MST, the MST will include the edge FI instead of HI. So, $X \leq 7$. But if X=7 then the MST will not be unique since FI can replace HI without affecting the cost of the MST. So it must be that $X < 7$.

3. Since edge costs are integer and $X < 7$ and $X > 5$ then X=6.

4. Arguments that use a specific MST algorithm are also acceptable.

Problem 2

There are n houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house i, we can either build a well inside it directly with cost wells[i], or pipe in water from another well to it. The costs to lay pipes between houses are given by the array pipes where each pipes[j] = [house1j, house2j, costj] represents the cost to connect house1j and house2j together using a pipe. Connections are bidirectional, and there could be multiple valid connections between the same two houses with different costs.

Return *the minimum total cost to supply water to all houses*.

Solution 2

Create a graph where each house is represented by a node. For every pipes[j], we create an edge between house1j and house2j with edge weight as costj.

We also create a node representing the well and connect to each house such that the edge weight of the edge between the new node and the ith house node is wells[i]

Find the MST in this graph and return the cost of the tree.

Problem 3

Because of a bad update, you (the CTO of SpaceX) are in a hurry to fix the ongoing issue with the Starlink satellite. Your talented engineers have prepared the urgent fix, which needs to be sent to all the satellites as soon as possible.

The fix is first sent from the base station situated in California to satellite *S1*. From there, you can send the fix from satellite *Si* to a satellite *Sj* if there is a link (edge) between *Si* and *Sj*. You know the time required to send the fix over this link from satellite *Si* to satellite *Sj* is *Tij*. We can assume that if a satellite *Si* needs to send the fix to *k* of its neighbors the *k* messages can be sent out at the same time. In other words, messages from a satellite to its neighbors go out in parallel not sequentially. However, a satellite can send the fix to its neighbors only AFTER it has fully received it from a neighbor.

a) Provide an algorithm to determine the minimum time required to propagate the update to all satellites. In other words, we need to minimize the time between the first message leaving the base station and when the last satellite completes receiving the update

b) Assuming that each satellite requires a single processing time *Di* before broadcasting the fix to any nearby satellites that need to receive the fix, how would you modify the solution in part a to determine the minimum time required to propagate the fix to all satellites.

Solution 3

a) Run Dijkstra's on the undirected graph and find the shortest distance to the furthest node from the California base station.

b) Turn the satellite network into a directed graph G (one undirected edge -> two directed edges in opposite directions). For every (S_i, S_j) in E, let the edge weight be E_(S_i, S_j) = T_ij + D_i. Run Dijkstra's on this directed graph and find the shortest distance to the furthest node from the California base station.