Chapter 1:

Stable Matching Problem:

Problem: Can we design a college admissions process or a job recruiting process that is self enforcing?

- Fact: Process of employers hiring students for internships is not self enforcing.
- Assume: An employer has hired 5 applicants for internship and can't take any more
  An applicant reaches out to an employer saying he/she would love to work with them
  Employer wants to hire the new applicant instead of existing one
  Emp. replaces the applicant. The replaced applicant then reaches out to another employer saying they would like to work with them. That employer then replaces of their existing applicant with this one and so on. The cycle continues.
- Best Solution: Either employer keeps the hired applicants, new applicant doesn't reach out to any employer. Anything is a solution that doesn't lead to Assumption scenario, a solution that will lead to a stable outcome.
- But best solution might not be possible in every scenario

EXAM TIPS:

- Remember the name of the algorithms

Lecture 1:

Stable Matching:

Problem: Match n men with n women.

Each man ranks all women in order: His preference list
Each woman ranks all men in order: Her preference list

Matching: A matching set S is a set of ordered pairs. (Can use values more than one, or some none at all)

Perfect Matching: A perfect matching set S is a matching with the property that each member of the M and each member of W appear in exactly one pair in S.

Instability: When a woman w1 wants m1, but m1 wants w2, w2 wants m2, and m2 wants w1

```
w1 -> m1
 |      |
m2  <- w2
```

Stable Matching: Perfect matching and no Instability

Step 1: Come up with a concise problem statement

We have a set of n men, M = { m1, m2, m3, …, mn}
We have a set of n women, W = {w1, w2, w3, …, wn}

Input: Preference lists for n men and n women
Output: Set of n marriages with no instabilities

Step 2: Present a Solution

Gale_Shapley Algorithm

1. Loop: Select a guy that is single
2. Based on his preference list he will propose to a woman
3. Woman: if the woman is already engaged but this guy is higher on her preference list compared to the guy she is with, she will leave that guy and engage this guy
4. That guy is now single
5. But if the woman is single, she will engage this guy and wait for a better rank guy to propose to her, based on her preference list

Step 3: Prove Correction

Tip: Start writing down observation/facts

1. Once a woman gets engaged, she remains engaged
2. Once a man gets engaged, he can go back to single any time
3. In this algorithm: man goes from top to bottom with proposals, woman goes bottom to top with acceptances.
4. Mans choices get worth with each proposal, womans choices get better with each acceptance
5. This algorithm favors man
6. Solution is a perfect matching
7. Solution is stable
8. At most n^2 iterations

Now that we have noted the observations: lets see what we can use to perform the actual proof.
Look at the above observations and see what you can prove

We can prove if a solution is stable or not
Use ‘Proof by Contradiction’: Try to prove that it is not stable, and when that is not successful state that this is why it is stable

“Proof by Contradiction”:
Assume that instability exists in our solution involving two pairs (m,w), (m’,w’)
(what makes a solution unstable: instability: m is paired with w but wants w’)
Two situations possible in this case: m proposed to w’ and m didn’t propose to w’.

1. If m did not propose to w': meaning w was higher in m's list as compared to w'. -> Contradiction
2. If m did propose to w', he was rejected at one point when another m' who is ranked higher in w' 's list than m proposed to w'. -> Contradiction.

Proof by contradiction: When men propose women, they end up with worst valid partners.
Suppose we end up with a matching set 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.

Step 4: Perform Complexity Analysis

Tip: Break down what the code is doing in steps and add complexity

1. Identify/Get a free man
   O(1) -> getting a free man from an array of all men

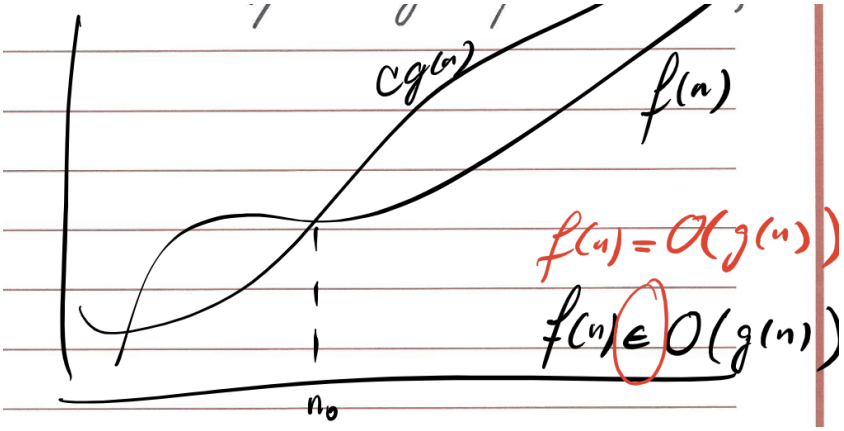
	Get()
Array	O(1)
Linked List	O(1)
Queue	O(1)
Stack	O(1)

2. Identify the highest ranked woman who m didn't propose to next
   O(1)
3. Determine if the next woman in man's preference list is proposed or not
   O(1)
4. If a woman is proposed determine which man is preferred by the woman (arrays of woman preference and mens preference are already sorted)
   O(1)
5. Put a man back in the list O(1)

Lecture 2:

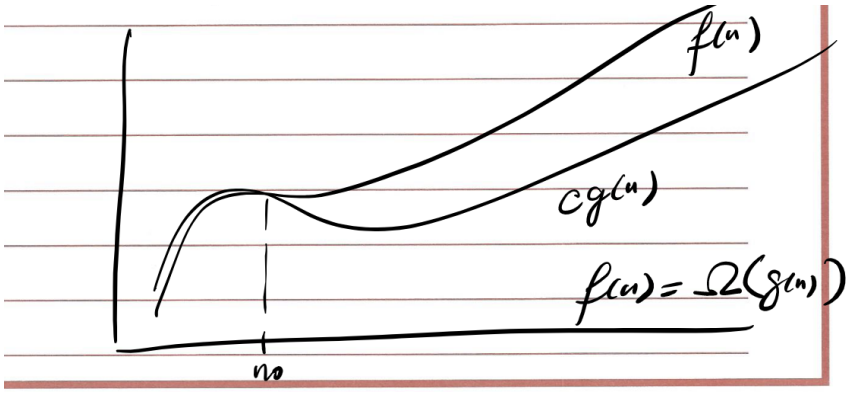
Asymptotic Notation:

- O(g(n) = 0 <= f(n) <= c g(n) for all n >= n0



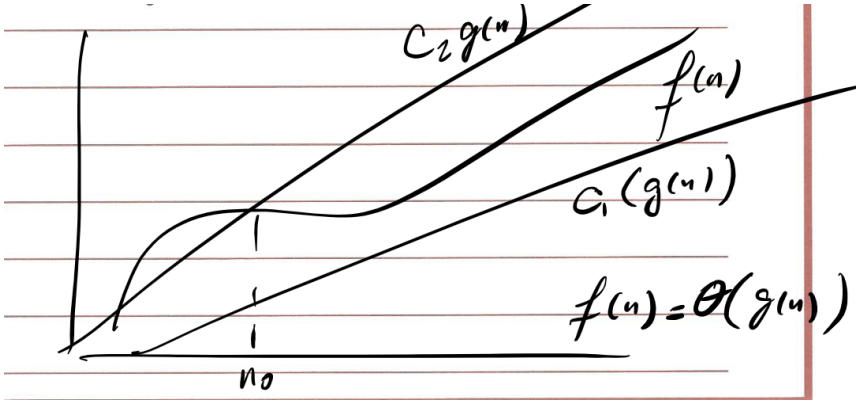
Quadratic function - O(n^2)
Linear Function = O(n^2)
Cubic function != O(n^2)

- Omega(g(n)) = 0 < c g(n) <= f(n), for all n >= n0
  f(n) = Omega(g(n))



Quadratic function - Omega(n^2)
Linear Function != Omega(n^2)
Cubic function = Omega(n^2)

- Theta(g(n) = f(n) 0 <= c1 g(n) <= f(n) <= c2 g(n), for all n >= n0



Quadratic function = Theta(n^2)
Linear Function != Theta(n^2)
Cubic function != Theta(n^2)

	Worse Performance	Best Performance
Linear Search	O(n) Theta(n) Omega(n), Omega (1)	O(1) Theta(1) Omega(1)
Binary Search	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)

Faster Performance:

Exponential > Polynomial > Logarithmic

Example: Which one is faster:

A: O(4^n n^3 lg n)
B: O(3^n n^8 (lg n)^2)
Answer: Can't tell

Example: Which one is faster:

A: Theta(4^n n^3 lg n)
B: Theta(3^n n^8 (lg n)^2)
Answer: Can't tell

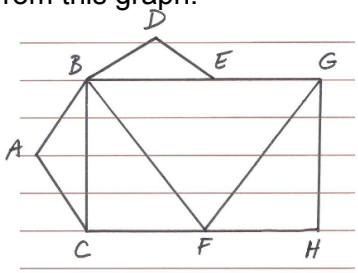
BFS: Breadth First Search: Directed or Undirected

e = total edges

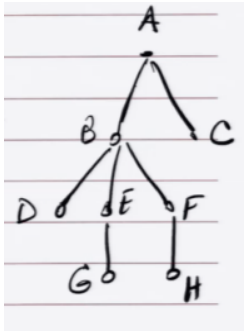
n = total nodes

O(n+e)

From this graph:



Create graph starting at position A (go through every node only once)



BFS tree is basically go through as much nodes as possible for the first node that the root is connected to , then do the same for remaining nodes connected to root, but the goal is to go to each node exactly once.

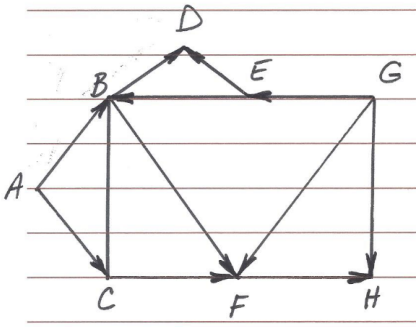
DFS: Depth First Search: (Bipartite Graph) Directed or Undirected

e = total edges

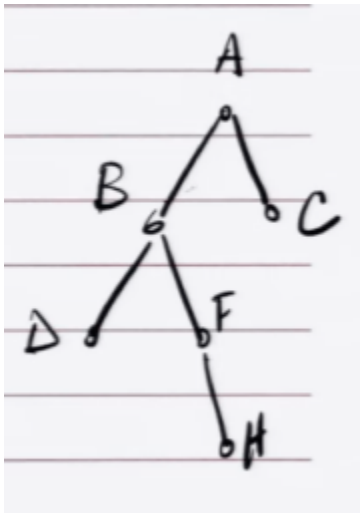
n = total nodes

O(n+e) -> Worst Case, can say Theta(n+e)

From this graph:



Create graph starting at position A (go through every node only once)

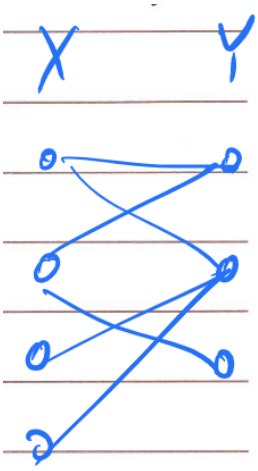


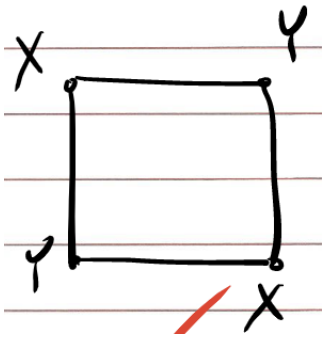
TODO: Difference between BFS and DFS, how their graphs vary?

Bipartite Graph: Two graphs connected by edges, no node is left behind, no node is being connected twice. MEANING: A graph with each node connected and being used only once can be broken down into two graphs

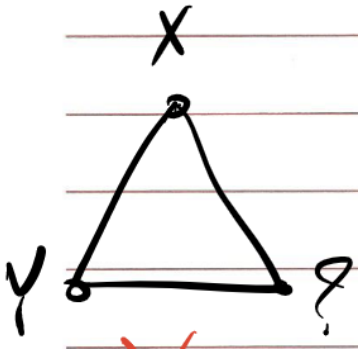
e = total edges
n = total nodes
O(n+e)

Bipartite Graph: No odd cycle. If bipartite the last node will be (2k+1) - use this for proof if a graph is bipartite or not.

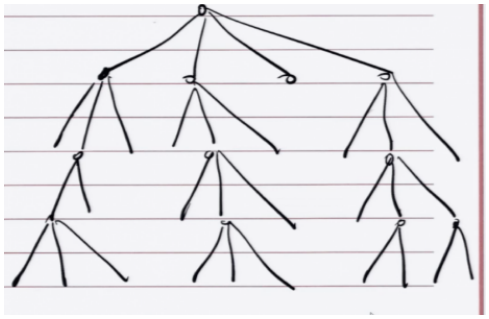




Not Bipartite:



Tip: To check if a graph is bipartite, run BFS on it and it should look something like this.



Where the first row can be x, the second y, the third x, the forth y, and so on.. After creating the graph compare it with original bipartite and confirm two ends of the node

match the original. If x is connected to x, or y is connected toy, they must be at the same row (only same row connection is allowed)

Steps to check if a graph is bipartite or not:

1. Run BFS starting at any node
2. Label each node x or y depending on whether they appear at odd or even level of BFS tree
3. Go through each node/edge to see if two nodes are connected properly
4. If all edges have one y end and one x end - then nipartite
5. Otherwise not bipartite

Directed Graph:

- **Strongly Connected**: only if all nodes are connected. (path from any point to any other point in graph).

- **Brute Force**:
  To check if a directed graph is strongly connected:
  Run BFS/DFS on it
  O(n^2 + en)

-

Lecture 3: Scheduling to Minimize Lateness

Given a number of requests
Each request has a deadline
The length of request is also given
Requests can be scheduled at any time

Notation: L(i) = f(i) - d(i)
Where L(i) = lateness for request i

Goal: Minimize the maximum lateness : L = Max(i)L(i)

Example:
Case 1:

Job1 is late by 5 hours
Job 2 is late by 6 hours

Case 2:
Job 1 is late by 0 hours
Job 2 is late by 7 hours

Compare: Max lateness in case 1: 6 hours
Max lateness in case 2: 7 hours
So Case 1 is better

Examples that will not work:

Example 1: Schedule shortest job first

```
-----1-----                                  deadline1
--------------2---------------deadline2
-----1-----|--------------2----|-----------      Late2 > 0
```

Because we have to schedule the shortest job 1st, we couldn't complete job 2 on time, it was done past the deadline.

If it was longest job 1st:

```
-----1-----                                  deadline1
--------------2---------------deadline2
--------------2---------------|-----1-----      Jobs could be completed on time
```

Example 2: Schedule shortest slack time first

```
-----1-----        deadline1
--------------2---------------deadline2
```

If we schedule job 2 first:

```
                   d1
--------------2---------------|-----1-----      Late1
```

If we schedule job 1 first:

```
                                d2
-----1-----|--------------2----------          Late2
```

Solution: Schedule jobs in order of their deadline without any gaps between jobs

Proof of Correction:

1. There is an optimal solution with no gaps
   An optimal solution with gaps can be turned into an optimal solution without gap

2. Jobs with identical/same deadlines can be scheduled in any order. The order of the jobs will not affect maximum lateness

3. Definition: Schedule A has an inversion if a job i with deadline d(i) is scheduled before job j with earlier deadline

   ----------i---------- deadline i
   ----------j---------- deadline 2

   Not Inversion:
   ----------j----------|----------i---------- done

   Inversion:
   ----------i----------|----------j----------

4. All schedules with no inversions and no idle time have the same maximum lateness
5. There is an optional schedule that has no inversion and no idle time

   So if there is an optional solution that has inversions, we can eliminate the inversions one by one until there are no more inversions. Optimal Solution.

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 solution

Once again, solution is to Schedule jobs in order of their deadline without any gaps between jobs
O (n log n)

Priority Queues

Two operations of a priority queue:

- Insert an element in the set

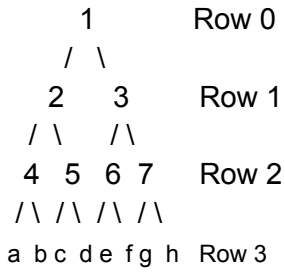
- Find smallest element in the set

	Insert	Find
Array	O(1)	O(n)
Sorted array	O(n)	O(1)
Linked List	O(1)	O(n)
Sorted Linked List	O(n)	O(1)

Binary Tree

- **Full Binary Tree:** A binary tree of depth k (total rows of tree) which has exactly (2^k) - 1 nodes. (Every node has two children, except the last row)

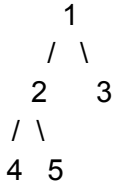
  Example:



depth = total rows, start at 0
depth: 3
Nodes: (2^4) - 1 = 15 nodes

- **Complete Binary Tree:** A binary tree with n nodes and of depth k is complete if its nodes correspond to the nodes with are numbered 1 to n in the full binary tree of depth k. (Every level is completely filled (except possibly the last level))

  Example:

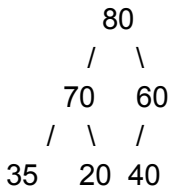


- **Traversing Complete Binary Tree:**
  - This tree is stored as an array
  - Parent (i) if (i != 1) parent is at floor(i/2)
    if (i == 1) i is the root, therefore parent is the root
  - Lchild (i) if (2i <= total nodes) left child is at 2i
    else: no left child exists

- Rchild (i) if ((2i + 1) <= n) right child is at (2i + 1)
  else: no right child exists

- **Maximum Binary Heap:** 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. (The node values go up as the tree grows from top to bottom)

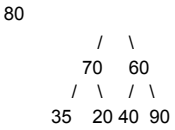
Example: Find Max: **O(1)**



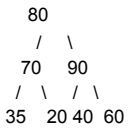
Example: Insert into Max Binary Heap: **O(lg n)**

Insert 90 into the tree:

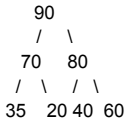
1. Insert it at the very bottom



2. Compare value of 90 to its parent and if 90 is greater than the parent, swap them

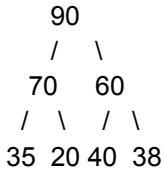


3. If 90 is not at the root then repeat step 2. Compare 90 with its parent and if 90 is greater then swap the two.

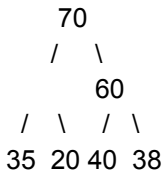


4. Check if 90 is at the root, if it is, then compare it to both of the children to make sure it is greater than them both. If it is not then more swapping is needed.

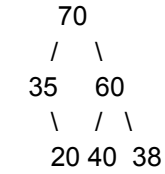
Example: Extract Max from Max Binary Heap **O(lg n)**



1. Remove head of the binary tree, in this case remove 90
2. Compare the value of children, whichever one is greater move it to the top

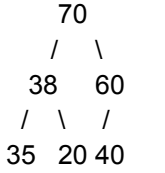


3. Now move to the children of the one you just used, compare the children and move up the child that is greater.



4. Now the tree is incomplete because the first child of last row is missing and the rest are there.. It is still complete if the last children of the row are missing.

5. Need to make it complete. To do so, compare the last value of the tree: 38 to the value that is missing a child: 35. Because 38 is greater than 35 move 38 in place of 35 and make 35 it's child



6. Now it is complete binary tree

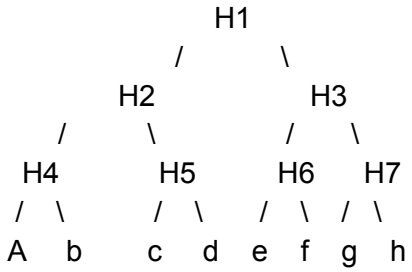
Example: Delete from Max Binary Heap **O(lg n)**

Example: Find and decrease a specific node value from Max Binary Heap **O(lg n)**

- **Construction of the Binary Heap:**
  O(n lg n) - O(n) - inserting into array, O(lg n) - sorting the array

  - If there are a total of n nodes for a tree, how many nodes are going to be at the bottom (leafs)? max/ceil of (n/2)
  - How many nodes from a level above from the leafs?
    ½ * the child nodes
    ½ * max/ceil of (n/2)
    = max/ceil of (n/4)

  - Total nodes in a row from bottom to top: n/2
    n/4
    n/8
    n/16
    etc.
  - How many swaps are needed to see that two children share a parent (making a heap of size 3: parent and two children)?
    - Go from bottom to top on the tree
    - Count total number of heaps: Total number of parents that have 2 children (not 1), a complete parent set is considered a heap
      Following example contains a heap of size 7:



- How many swaps are needed to make sure that these are all heaps of size 7?

  The leaf level does not need any swaps.
  So we start a level up, a level up has 4 heaps (H4, H5, H6, H7) - need 1 swap

  A level up needs 2 swaps at most

  A level up needs 3 swaps at most

  Max swaps: add swaps of each level

  H1 - (total nodes) * 3 swaps => (n/16) * 3
  H2, H3 - (total nodes) * 2 swaps => (n/8) * 2
  H4, H5, H6, H7 - (total nodes) * 1 swap => (n/4) * 1

  Theoretically: Calculating swaps from bottom to top:
  n/4 * 1
  n/8 * 2
  n/16 * 3
  …
  1 * lg n
  Sum: T = (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + …
  Assume T/2 = (n/8 * 1) + (n/16 * 2) + (n/32 * 3) + …
  Subtract T-T/2 = n/4 + n/8 + n/16 + …
  T/2 = n/2

  T = n

- How long will merging two binary heaps of size n will take?
  O(n) - using Linear time construction

- Challenge:
  - We have an unsorted array of length n
  - Print/Pop top k values in the array where (k < n) - so basically print/pop n-1 values
  - Cannot use pointers
  - The algorithm should run in **O(n lg k)** time

  Solution:
  - Create a min heap of size k (using first k values of array n)

- **O(k)** -- construction of the heap is linear (ex. O(n) for inserting into heap)
- Pick a value from array n after the first k values (because those are used for min heap)
- Compare the selected value of n to the top value of k (min heap)
- If the value of n > k, pop k.

  Basically, you do an insert which takes O(k)
  And you extract values after comparison which takes O(lg n)
  These operations together take log, and you do these operations n-k times

  Overall: O(k) + O(n-k)lg k = O(n lg k)

Binomial Tree B(k)

- B(k) is an ordered tree defined recursively
- B(k) consists of 2 binomial trees B(k-1) that are linked together such that the root of one is the leftmost child of the root of the other.
- B(o) consists of one node

Example:

- .
- .
-

- .
- .
- .
- .
- .
- .
- .
- .

-
-