CS 180: Lecture 1
Thursday, Week 0
9/28/17

Course Introduction

When we create algorithms, we want to use a **Universal Method of Computing**. It needs to be applicable across all OS/Systems an able to be used in the distant future.

When we design and analyze algorithms, there some things we need to consider

- How well the algorithm performs (resources)
- We want to try to find a Lower Bound Algorithm
- Analyze the time complexity of our algorithm

Random vs. Arbitrary

Many people think of random and arbitrary as the same things, but in terms of designing our algorithms, they are quite different

Random - based on finite/set probability function

- ex: Normal Distribution/ Binomial Distribution
- use this predetermined probability function to decide what we want
- Disadvantage: Can be very time/resource consuming to go through this random process

Arbitrary - determined by something categorical, or just chosen on a whim

- ex: can be chosen by height, weight, alphabeitcal, or just chosen on a whim
- Advantage: Arbitrary choice makes proofs simpler

As we can see, having to rely on random choice probably will not be best for our algorithms, so we want to construct algorithms with **arbitrary chioce** when applicable.

Egg Problem

Our goal is that we want to find the highest floor of an n story building such that when when you drop an egg from that floor, the egg won't break. However, our constraints on the problem are that we only have two eggs to test with, and we only are allowed 10 drops. Given our constraints, what is the maximum amount of floors that we can test on to guarantee we find a solution?

Break this problem down.

- What if the egg breaks on the first drop?

    - Then we have nine more tries with our last egg to find the floor.
    - Implies we should start at floor 10.
- What if the first egg doesn't break but the second egg breaks
    - Then we have eight more tries with our last egg
    - We know that floors 1-10 don't need to be tested because the first egg survived
    - Therefore, we need to test floors 11-18 with our 8 tries, so we should move up 9 floors to floor 19.

Following this pattern, we see that the amount of floors we can test on is

$$10 + 9 + \ldots + 1 = \sum_{k=1}^{10} k = \boxed{55}$$

CS 180: Lecture 2
Tuesday, Week 1
10/3/17

Problem Set 1: Due Next Thursday, 10/12/17

Serial Computation

When we talk about a **serial model of computation**, we are saying that one action is executed at a time in the computer. There are several components of a computer to consider when using this model.

ALU (Arithmetic Logic Unit) - performs basic arithmetic operations

RAM (Random Access Memory) - for our purposes, we have infinite RAM to solve problems

I/O (Input/Output) - devices to relay information to the computer/for the computer to relay information back to the user.

This model is very basic/very universal - applies to most/all systems, which is what we desire when creating algorithms.

Example Problem: Adding Numers in Series

Lets consider a possible algorithm for adding numbers in a series, i.e.

$$x_1 + x_2 + \ldots + x_n = S$$

How would we write this down according to our serial method of computation?

- Read x_1, put it in ALU

- Read x_2, add it to x_1
- $\ldots$
- Read x_n add it to x_1 + $\ldots$ + x_{n-1} stored in ALU
- Output Answer

Analysis of the algorithm

- Total Operations: 1 + 2(n-1)
  - x_1 iteration doesn't have an add operation
- $2n$ for input, and 1 for output
- Total: $2n + 1 + 2n - 2 + 1 = \boxed{4n}$

Matching Problem

Consider a group of Men and women that have interest in each other.

Defining the problem

- We have n men and n women (same of each)
- Each man and women has an order of preference for matching with a partner
- If we have two people that aren't matched that prefer each other over their partners, then we refer to this as an UNSTABLE matching
  - i.e. m matched with w and m' matched with w' but m and w' prefer each other.
- If we do not have this solution anywhere, the solution is STABLE

Find an algorithm that guarantees a stable solution every time.

Algorithm

While ∃ an unmatched man...

- Pick a man that hasn't been matched yet (arbitrary), called m
- Take the highest woman left in his priority list, called w and attempt to temporarily match them
  - If the woman is unmatched, she automatically accepts
  - If the woman is matched with another man m', compare in her priority list
    - If m is higher than m' on her list, she leaves m' who is now unmatched again, and matches with m.
    - Else, m' is higher than m on her list, she keeps m' and m remains unmatched
  - In either case, remove w from the priority list of m, since she has been considered either way

Note: Write solutions in bullet form for clarity

Analysis

Does this solution leave everybody matched at the end?

- Because each man will have every woman in his list, each woman will be matched with somebody eventually, and since ever woman is matched, then every man will be matched also

Now we need to prove the stability of our solution. Let's assume the unstable case occured, and prove by contradiction that it isn't possible.

- Unstable case: m matched with w and m' matched with w' but m and w' prefer each other.
- Case 1: m proposed to w' before
    - If they were matched previously, then w' will never match with m' in our algorithm since he is lower on her priority list than m
    - If they were not matched previously, then her match, called m'', must be higher than m on her priority list, and thus m'' is also higher than m' on her priority list.
- Case 2: m did not propose to w' before
    - This case is not possible. The unstable case has m matched with w, who is lower on his priority list than w', but according to our algorithm, that means he must have proposed to w' at some point before considering w.

Therefore, we have shown our solution is valid and always produces a stable matching.

CS 180: Lecture 3
Thursday, Week 1
10/5/17

Task Problem

Assume you are given a set of daily tasks. Each task is represented by an interval on the time axis (ex: 10-12, 9am - 4pm, etc...). Given the entire set of tasks, find a subset of non-overlapping tasks.

Simple Solution - just pick one task arbitrarily

- When given a question, read it thoroughly to see exactly what it asks
- Don't overcomplicate it and try to solve something it isn't asking
- Give the simplest possible answer

Clarification: We want the solution that maximizes the number of tasks in this subset.

Algorithm 1: Pick Smallest First

A possible algorithm to find the solution is given as follows

- Pick the smallest interval, and add it to the timeline
- Eliminate all other tasks that overlap it

- Repeat until all tasks are eliminated

Does the algorithm solve our problem? NO. Proof by counter-example:

|————————————||————————————————————|

|————————|

- According to our algorithm, we would take the smallest one (the one on the bottom) and eliminate all overlaps (ones on top)
- However, we see that we only have one task, where we could possibly have two, so this algorithm is bad.

Algorithm 2: Pick First Event

- Pick the first event point and add it to the timeline
- Eliminate all overlaps
- Repeat

Does this algorithm solve our problem? NO. Proof by counter-example:

|————————————————————————————————————|

|———||————————||————————||———————||————————————|

- According to our algorithm, we take the first task (top one) and eliminate the overlaps (whole bottom row)
- It is clear that the bottom row is the far more optimal solution

Algorithm 3: Pick the Event that Ends First

- Pick task that ends first
- Eliminate overlaps
- Repeat

Does this algorithm solve the problem? YES. Let's prove it by contradiction

- Assume our solution isn't optimal. Then ∃ an optimal solution that is different than ours.
- Assume the first i intervals, $0 < i < k$, are the same
- Compare the next interval, the $i + 1$ interval.
- Because of the construction of our algorithm, the $i + 1$ interval of our solution will end before the $i + 1$ interval of the other solution. Replace the $i + 1$ interval in this proposed solution with the one from our solution.
- Now the first $i + 1$ intervals between our solution and proposed optimal solution match
- Continue this process through the k_{th} interval
- Therefore, proposed optimal solution is now the same as our solution, and we have proved that our algorithm produces an optimal solution by contradiction.

CS 180: Discussion 1
Friday, Week 1
10/6/17

Problem 1: Combine Sorted Arrays

We are given two separate sorted arrays of numbers. Combine them into one sorted array

- Compare the first two numbers in each array
- Take the smaller one, put into first element of sorted array and advance index of array it came from
- Repeat.

Time complexity: If array1 has size n, and array2 has size m, then this is $O(m+n)$

Problem 2: Up-Down Array Sorting

Given a random array, ex: 4 1 2 6 5 3. Return an array sorted such that the relationship between each value is up, down, up, down.... ex: 1 3 2 5 4 6. ($\mathbf{1} \rightarrow \mathbf{3} =$ Up, $\mathbf{3} \rightarrow \mathbf{2} =$ down, etc...)

Naieve approach

- Sort the array
  - $O(nlog(n))$
- Take the first value, put in new array
- Take the last value, put it in next element of new array
- Repeat until finished.

Better approach

- Compare the first two values
  - If the relation is correct, ignore and move to next pair
  - If the relation is not correct, switch the two values
- Repeat until done.

This method has $O(n)$ time which is better than $O(nlog(n))$.

Problem 3: Proper Parenthesis

We are given a string of parenthesis. How do we tall if it is a valid string (i.e. all openings are closed and all openings and closings are proper)

Examples

- "(() (()))" is valid
- ") () () (())" is invalidL closing parenthesis before open parenthesis
- "( ))" is bad - extra closing parenthesis
- "( ))" is bad - extra closing parenthesis

My Solution

```
bool solve(string parenthesis_string)
{
  n = 0;
  counter = 0;
  while (n < parenthesis_string.length())
  {
    //Check the individual character and change the counter
      if (parenthesis_string[n] == "(")
      {
        counter++;
      }
      else if (parenthesis_string[n] == ")")
      {
          counter --;
      }
      else
      {
          cout << "Error: Invalid string given.";
          return false;
      }

    //Check counter. If it's negative at any point, we have too many closing parenthesis
      if (counter < 0)
      {
          cout << "Error: Extra closing parenthesis or closing parenthesis before open parenthesis."
          return false;
      }

     //Increment n
      n++
  }

  //End of loop over string, check the counter
  if (counter == 0)
  {
      cout << "Valid parenthesis string given";
      return true;
```

```
38    }
39
40    else if (counter > 0)
41    {
42        cout << "Error: Too many open parenthesis"
43        return false;
44    }
45
46    else
47    {
48        cout << "Error: Too many closing parenthesis";
49        return false;
50    }
51  }
```

- Keep a counter for the parenthesis string
    - Increment the counter if we have an opening parenthesis
    - Decrement the counter if we have a closing parenthesis
- If the counter is ever negative, then we have a closing parenthesis before an opening parenthesis , so this is not valid.
- If we get to the end and the counter is not 0, that means something hasn't been closed or there's an extra closing paren
- If the counter is 0, then we return true.

Another Solution: The Stack!

- If open parenthesis, push onto the stack
- If closed parenthesis, pop from the stack
    - If we can't pop the stack, too many closed paren, return false
- At the end, check stack
    - If stack is empty, the parenthesis string is valid, return true
    - If the stack is nonempty, some open parenthesis didn't get closed, return false.

Dont forget about using Stacks/Queues in solutions to problems.

Problem 4: Greatest Numer after Each Element

We have the following problem:

- Given an array of numbers, we want to find the greatest number after each element
    - Put -1 if such a number does not exist.
- Example array and solution below: 1 2 1 5 20 10 15

  20 20 20 20 -1 15 -1

Think about this problem and come back next week with a solution.

CS 180: Lecture 4
Tuesday, Week 2
10/10/17

Complexity

- When we consider time complexity, we want to assume n to be very large.
- If we compare algorithms with $O(n), O(2n), O(\frac{n}{2})$, we say they run about the same speed
    - Same with something like $O(2n + 500 + log(n))$, etc...
- We only really care about the **highest order of complexity**

Order: a function $T(n) = O(f(n))$ if ∃ constants n_0, c such that $\forall\ n \geq n_0$, we have $T(n) \leq cf(n)$

Can think of order as an **upper bound**.

Examples

- $n^2 = O(18n^2 + 10) \implies$ True
- $18n^2 + 10 = O(n^2) \implies$ True
- $n^2 = O(4n^2 + log(n) + \frac{n^3}{10}) \implies$ True
- $n^3 = O(n^2) \implies$ False

Omega Notation: a funtion $T(n) = \Omega(f(n))$ if ∃ constants n_0, c such that $\forall n \geq n_0$, we have $T(n) \geq cf(n)$

Can think of Omega notation as a **lower bound**

What is the **difference in application** between Order and Omega notation

- Order primarily used to describe **algorithms**
- Omega primarily used to describe **problems**

If a problem/algorithm is $O(n_1)$ and $\Omega(n_2)$ such that n_1 = n_2, then we say the problem is $\Theta(n)$

Graphs/Graph Algorithms

Graphs and their Properties

Graph - a graph is defined by a set of verticies (nodes) and edges (links). we write this formally by saying $G = (V, E)$ with, for example, $V = \{a, b, c, d\}$, $E = \{(b, c), (a, b), (a, c)(a, d)\}$

Graphs can have direction.

- A **directed** graph travels in a certain direction along its edges
- An **undirected** graph has no directions between its verticies

Graphs can have weighted or unweighted edges

- A **weighted** graph means that edges have different weights, one edge has a larger value than another
- An **unweighted** means that all edges mean the same thing

Graphs can be connected/disconnected

- If we can start at one vertex and travel along edges to touch all other vertex, the graph is **connected**
- If, when we start at one vertex, we cannot touch every other vertex by traversing edges, the graph is **disconnected**
    - Make sure to consider wether or not the graph is directed in the context of the problem

Eulerian Graphs

Problem: We want to see wether or not we can draw a given shape without picking up our pencil.

Rules:

- We must go over every edge and touch every vertex
- We cannot go over an edge twice

We are able to do this if and only if $\exists$ 2 or less vertexes with an odd number of edges attatched to it. Graphs that meet this quality are called **Eulerian graphs**.

Graph Traversal

There are different methods used to traverse graphs to search for something.

Breadth First Search - search within neighborhood of starting point

- Starting from a vertex, v_0
- Examine all of the vertecies connected to v_0 first.
- If we don't find what we need, then consider each of v_0's neighbors, lets call each one v_n
- For each neighbor v_n look at their immediate neighbors that haven't been checked
- etc...

Properties of BFS

- We can make a BFS tree starting with our initial point at the top, and traveling down one level that represents each of the neighbors we searched at that iteration
- Levels and Distance from Starting Point: a vertex is on level i of the BFS tree, then the shortest path between our original vertex v and this new point is i.

Depth First Search - search completely down a path from a starting point

- Start from vertex v_0
- Pick a vertex connected to it, lets say v_1, and test that
- Make v_1 our new temporary vertex
- If v_1 has any unexplored neighbors, lets say v_2
    - Test that vertex, make v_2 new temporary vertex
- else v_1 has all neighbors explored
    - backtrack to the edge that we were at before v_1, in this case v_0, and test it again.

General Graph Properties

Assuming our graph is connected, and undirected, we have

- $|V| = n$
- $n - 1 \leq |E| \leq \frac{n(n-1)}{2}$
    - There cannot exist two edges between vertices in an undirected graph.
    - Therefore we have $(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$

CS 180: Lecture 5
Thursday, Week 2
10/12/17

Note about Exams:

- Disguising algorithms for new problems
- Try problems on the FB page, see if you can get them

Graphs

Notation: $G = (V, E)$

- A graph with V verticies and E edges.

We have also learned two graph serach methods: BFS and DFS. Both create a tree-like structure. We can use this fact when considering the following problem.

Finding a Shortest Simple Path

Given a graph G (non-weighted), we want to find the shortest path such that no vertex is revisted. It follows from our definition of BFS that we get the following therom.

Therom: Shortest Simple Path

The length of the shortest path from a starting point, s, and an ending point, x, is i, where i is the level of x in the BFS tree.

Proof

Two counter-cases to consider

1. $SP(s, x) > i$ (shotest path between s and x is greater than i)
    - If x is in i, then it was first visited from $i - 1$
    - If x is in $i - 1$, it was first visited by $i - 2$
    - ...
    - If x is in 1, it was first visited by 0
    - Therefore, the distance between s and x is i -> Contradiction
2. $SP(s, x) < i$
    - level # of 1^{st} node from s must be 1, next must be two or less, subsequent node j must be at level j or less.
    - Lable of node is number on path from $s \rightarrow x$
    - Inductive: node with label j is at level j or less
    - x is at level i or less, which is a contradiction $\implies$ x is at level i, and distance i from s

Representing Graphs as Data Structures

Matrix

Given a Graph $G(V, E)$ with n verticies and e edges

- represented by an n x n matrix, with each row and column representing a vertex
- entry at (a, b) is 1 if $\exists$ an edge between verticies a and b and 0 if not
    - If graph is non-directed, matrix is symmetric
- $\implies$ n^2 total entries
- Good for dense graphs with many edges

Linked List

Given a Graph $G(V, E)$ with n verticies and e edges

- Represent graph as an array of linked lists
- Each node points to a linked list of the edges it is directed to
- Add reverse edge for undirected graphs
    - Every edge will appear twice

BFS Search

- Queue search
    - "First in first out"

DFS Search

- Stack search
  - "Last in First out"

CS 180: Discussion 2
Friday, Week 2
10/13/17

Graphs

Storage of Graphs

- Matrix
  - Query : $O(1)$
  - Add Value: $O(n)$
  - Add edge $O(1)$
- Linked List
  - Query: $O(n)$
  - Add value: $O(1)$
  - Add edge: $O(1)$

Searching Graphs

- **DFS** - depth first search
  - Search as far down levels as you can
  - When you can't go down further, backtrack to the last one and check any other paths.
  - If out of paths for that node, backtrack again until we find one an unchecked path
  - DFS == **STACK**
- **BFS** - breadth first search
  - Search in neighborhood of the original point
  - When all points nearby are searched, move to one of the neighbors and check all of his neighbors
  - Works level by level.
  - BFS == **QUEUE**

Implementing Each Searching Algorighm

DFS

```
def DFS(root)
    check root
    if root.child == None:
        return
    else for child in root.children:
        if child has already been checked
            do nothing
        else
            return DFS(child)
```

- How can we know if we have a cycle or not?
    - Implement this using a stack
    - If the number you have is already in the stack, we have a cycle

```
def DFSStack(root)
    check root
    if root.children== None:
        stack.pop(root)
        if stack.isempty
            return
    else:
        for child in root.children:
            check child
            add child to stack
            return DFSStack(child)
```

BFS:

Which is better to use when?

- If we want a search graph, we should use BFS.
    - Can search all nearby nodes, and most often graphs wont be too deep
    - For DFS, we may get stuck going down a really long path and doing a lot of unnecessary searches.

Trees

Different kinds of Trees

- Trees
    - Acyclic
- Binary Trees

    - Each parent node has at most 2 children
- Binary Search Trees
    - Looking at a node, everything will a value smaller than that node will be down the left path, and everything with a larger value than that node will be down the right path.

Tree Traversal

- In order
    - Go down the left edge and call In Order on that
    - Print node we are on
    - Go down the right edge and call In Order on that.
- Pre-order
    - Print the node we are on
    - Go down the left edge and call In Order on that
    - Go down the right edge and call In Order on that.
- Post-order
    - Go down the left edge and call In Order on that
    - Go down the right edge and call In Order on that.
    - Print the node we are on

Each of these traversals is $\mathbf{O(n)}$

Binary Search Trees

Let us consider the different operations that we can do

Insertion:

We cannot assume all BSTs are balanced, so this has $O(log(n))$ on average, but $O(n)$ worst case

```
def insert(num, root):
    if num > root.data:
        if root.right == NULL:
            root.right = new node(num)
            return
        else
            return insert(num,root.right)
    else
        if root.left == NULL:
            root.left = new node(num)
            return
        else
            return insert(num,root.left)
```

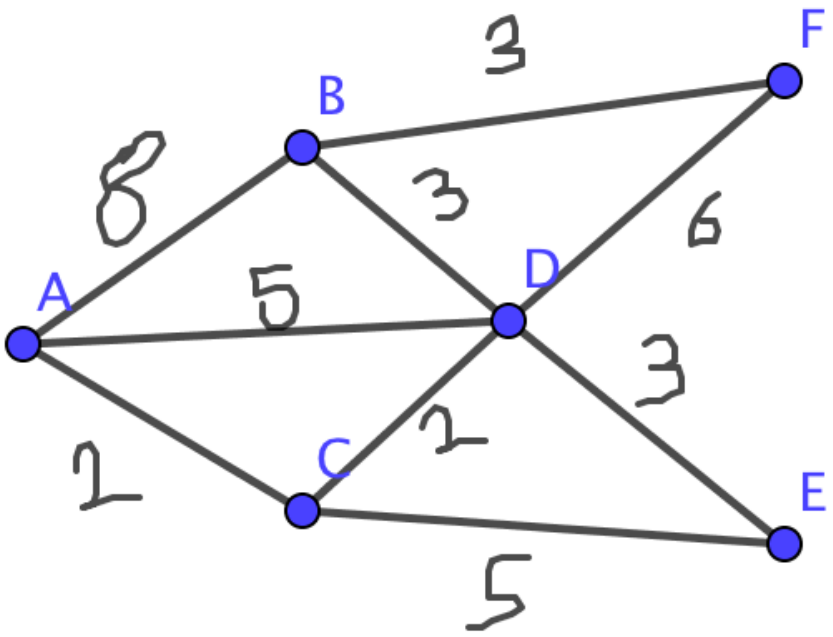
Shortest Path Problems

The problems we deal with are single source shortest path problem

- Find the distance from one point to every other point.

If graph is unweighted, then we can just use BFS, but most problems will be concerned with graphs with weighted edges. The famous algorithm that defines how to solve this problem is called Dijkstra's Algorithm

Let us consider the following graph



We want to find the shortest path from A to F. Here's how we do this

```
#DIKJSTRA'S ALGORITHM (for this problem)

* give each node a value. our starting node A = 0, every other node is infinity
* update each neighbor node of A based on size of edge
    - C = 2, D = 5, B = 8
* choose the smallest node to consider next (C), and mark it as checked
* update all neighbors of C with a new value V = value(C) + size(edge)
    - If a value is already there, keep the smallest value
    - B = 8, D = 4, E = 7
* choose D to consider since it is smallest
* update again, this time from D = 4
    - E = 7, F = 10, B = 7
* now consider E, but it has no neighbors to consider, so move on to B
* update again from B = 7
    - F = 10 (doesn't change)
* Since F is the smallest value left, our algorithm is done

CONCLDE: Shortest path to F is 8 units. It is from A -> C -> D -> F


```

In general we can generalize Dikjstra's Algorithm like so

```
#DIKJSTRA'S ALGORITH FOR SHORTEST PATH

1) Give starting vertex a value of 0, all other vertexes a value of infinity
2) Set initial node as current, all other nodes as unvisited
3) Consider each neighbor of initial node and calculate the tentative distance. The tentative distance is denoted as value(current_vertex) + value(edge_to_new_vertex). If the new vertex already has a value, keep the smaller of the two
4) Once we are done, mark the current node as visited.
5) Consider stopping conditions
   a) If unvisited node with smallest tentative distance is desired node, we are
   finished
   b) If smallest unvisited node has tentative distance infinity, then no connection
   between initial node and desired node, return failure
6) Otherwise, set the node with smallest tentative distance as current and return to step 3.

```

CS 180: Lecture 6
Tuesday, Week 3
10/17/17

Bredth First Search

Time Complexity

Not all graphs are connected, and wether or not a graph is connected can change the time complexity of BFS. Assume $G(e, n)$

- If the graph is connected, then we can just say that the search is $O(e + n)$
- If the graph is disconnected however, then we must visit different components as well, so the complexity becomes $O(e + n)$

Directed Graphs

The problem is essentially the same for directed graphs, but the most important thing that changes is the definition of **reachability**.

Problems with Directed Graphs

Topological Sort

If we have a directed graph that doesn't have a cycle, we call that a **Directed Acyclic Graph (DAG)**. If we are looking to find a cycle in a directed graph, we can do this using a DFS algorithm.

Given these graphs, the problem of trying to sort these graphs is called **Topological Sorting**. TS has the unique propery that it does not necesarrily have a unique solution , there can be more than one solution ot the problem. We want to find an algorithm for this, but first we need to define a couple of things

in-degrees - the number of directed edges coming in to that vertex, $deg^{-}(v)$

out-degrees - the number of directed edges leaving a vertex. Denoted $deg^{+}(v)$

source - a vertex v such that $deg^{-}(v) = 0$

The algorithm, known as **Kahn's Algorithm** is as follows

```
1) Choose a source
2) Output the source
3) Update in-degrees of all of the vertecies connected to the source
   a) If any of these become a source, add them to the source list
4) Move back to step 1 and perform this recursively.
```

Now we will investigate the time complexity of this algorithm. Evaluate the inner loop of the algorithm

- Initial the degrees of each vertex = $O(e)$
- Find a source such that the in-degrees of the vertex is 0 $O(n)$
- Updating the in-degrees of each vertex = $O(n)$
    - Maximum, we need to check $n-1$ verticies

The inner loop runs at $O(n)$, and we need to run this loop for n points at most, so we get a final complexity of $\boxed{O(n^2)}$.

However we can do a different analysis of the algorithm

- Initialize the degrees of each vertex = $O(e)$
- Find a source such that in-degrees of vertix is 0 = $O(n)$
- There are e edges, and we process each edge once, so updating in-degrees = $O(e)$

Therefore, we have gotten rid of the loop and we have a runtime of $\boxed{O(e+n)}$

How did analyzing the algorithm in two different ways give us two different runtimes

- One method analyzed the algorithm as a loop and considered the number of times the loop could be run
- The other method analyzed entire algorithm in full

(For further resources on Kahn's Algorithm: http://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/)

Two-Color Graphs

Given a graph G, how can we make it so that no two adjacent vertecies have the same color?

- Easy solution: Give every vertex a different color.

The more interesting problem arises when we attempt to limit on the number of colors. To help understand this problem, we define a different term that will lead us to the solution.

Bipartite Graph - a bipartite graph is a graph such that the verticies can be separated into two disjoint sets such that no two verticies within the same graphs are adjacent

From this we can get the following conclusions about 2-colorability of a grpah

- A graph is 2-colorable if and only if it is bipartite
- If a graph contains any cycles with odd length, it is not 2-colorable

To figure out wether a graph is bipartite/2-colorable, we can use a BFS algorithm.

Creative Problem

Given

- G is a directed, acyclic graph
- K is the maximal number of edges in any path

Goal: Partition V into $K+1$ groups such that there are no edges between verticies in a group

How can we approach this problem? Look at givens for hints.

Solve for Thursday lecture.

CS 180: Lecture 7
Thursday, Week 3
10/19/17

Graphs: Continued

Connectivity

All of the vertexes that can be reached in a grpah (nondirected) is called a **connected component**. Graphs can have several connected components, but the definition is more murky for directed graphs. We will define a strongly connected component to clear up our ambiguity for directed graphs.

strongly connected component - two points a and b are strongly connected if ∃ a path from a to b as well as a path from b to a

- The set of strongly connected components is a disjoint set
- Any strongly connected component must be a cycle

To find the strongly connected components, run BFS twice. Once on normal graph G, once on G^r (reverse graph).

TL;DR: Components not very well defined for directed graphs. Strongly Connected Components very well defined.

Paradigms: Greedy Paradigm

Take a problem. Look at one or two items in the problem and make a decision very quickly without really haven't seen the entire problem. Once make decision you stick to it.

Ex: Bunch of numbers to sort. First we see number 5. Guess it's the second smallest one in the set.

Ex: Famous problem. We only considered two people, and made a decision right away. That is an example of a greedy decision.

Proving decision is optimal is a challenging task.

Problem: Scheduling Processorsr

We have the scheduling problem we have done in the past, but instead we have k processors that we can schedule these tasks on, so instead of throwing an overlapping task out, we schedule it on a different processor.

To solve this put the next ending task on the lowest available processor. Doing this actually minimizes the amount of processors we need to use.

Prove yourself (proof in beginning of chapter 4 to check).

Shortest Path: Weighted Graphs

Given a connected, weighted graph, and a starting vertext s, find the shortest path from s to some point c.

- Only non-negative edges
- Non-directed graph
- Shortest path in terms of weights not length (number of edges)

Solution

Let's consider a greedy solution. We want to find the shortest path and stick to it.

- Choose the neighbors of s. Say that the shortest path between s and a point d is the weight of the edge between them, and don't change mind.

However, we can see that there can be a shorter path via two differenet edges with much smaller weight, so our greedy hypothesis was not good. Therefore, we can only make a greedy decision on the neighbor of s with the minimum weight edge.

- Choose the neighbor of s with minimum weight, lets say d. Say that the shortest path between s and d is the weight of the edge between them, and don't change mind.

Now we have the first step. We need to figure out the next step that works in the same manor.

- Consider all neighbors of verticies that we have finalized, and pick the one that has the shortest path to s

Why does this work?

- If there was another path that was shorter, then we would have to go through one of the other neighbors plus another nonnegative edge, and since our inductive greedy step considers the shortest path each time, then the other path must be longer.

This algorithm is known as **Dikjstra's Shortest Path Algorithm**

Important Note: Analysis

When analyzing algorithms, we need to do **worst case analysis.** We don't care how it can run on the best possible case, and for now we don't really care about average case, but we care about the longest possible time our algorithm can take.

CS 180: Discussion 3
Friday, Week 3
10/20/17

Topological Sort

We can only apply topological sort on a **Directed Acyclic Graph**

Algorithm for Topological Sort

```
1) find all sources (in-degree = 0), and add them to a queue
2) for all elements in queue
    a) print that element
    b) reduce in-degree of each child by 1 (remove that edge)
    c) if we reduce it to 0, add it to the queue
    d) if we have no points left in queue and there are still points in
graph, there was        a cycle, return FAIL
3) return the printed list of nodes.
```

Algorithm for Topological Sort using DFS

```
1) For each initial source
    a) Do DFS from source
        b) when done processing a node, add to a stack
```

Problem 1: Maximum Path Sum

We are given a binary tree, where each node has a certain value assigned to it. Find the path in the tree such that thte sum of values along this path is maximal. A path is from root to leaf

Algorithm: Recursive

- Return the max(left path, right path) + root.

Problem 2: BST

In a binary search tree, doing in-orer search gives us a list of roted number. The definition of a successor is as follows

- Next greatest node
- Next node given by in-order search

Given a pointer to a certain node, find its successor.

Note: The rules of this tree allow us to travel up and down

```
struct tree
{
  int value;
  int* leftChild;
  int* rightChild;
  int* parent;
};
```

Simple solution

- Do in-order search on the whole tree
- Find our given node, and return the next element in the list.

Complexity: $O(n)$

Better Solution

- If our node has a right child
  - Go down to the right, then as far left as you can, and return that value.
- If our node has no right child
  - Go back up until the current node we looking at on is a left child
  - Go up one more and return that node.

Complexity : $O(log(n))$ for a balanced tree, worst case $O(n)$ for unbalanced tree

Problem 3: Building Unique BST

We are given the in-order and pre-order printing results of a Binary Search Tree. Given both of these, construct a UNIQUE BST that satisfies both of these lists.

Solution

- The first node in the pre-order list must be the root, so place it.
- For each element in the pre-order list
  - find it in the in-order list
  - Take left-and right of it
  - Run recursively on left and right

Problem 4: Next Greater Element

Given an array, find the next element that is greater than our current element.

Simple Solution

- For each node, traverse the rest of the array and find the next largest node

Complexity: $O(n^2)$

Better Solution: Start from the Right, Use a Stack

- Push last value on the stack
- For each remaining value in the array
    - Compare with top of stack
        - If smaller than that value, put that value in new array
        - If bigger than that value, pop value from stack and compare with next one until we can do it
            - If stack is empty, put -1 for that element.
        - Push new value on stack

CS 180: Lecture 8
Tuesday, Week 4
10/24/17

MIDTERM NEXT TUESDAY:

- Chapter 1 - Chapter 4 (not all of 4)
- THIS LECTURE is cutoff for the midterm
- Start studying now.
- Review session Thursday

Dijkstra's Shortest Path: Continued

Time Complexity

- Find the minimum in a list of the uninvestigated path lengths
    - Inner Loop: $O(n)$
- Update the weights of all verticies connected to the one we are.
    - Inner Loop: $O(n)$
- How many times do we do these steps? Have to check all points accept first.
    - Outer loop: $O(n)$

Dikjstra's Time Complexity: $O(n^2)$

Is this optimal? Let's consider a new algorithm.

MISSED AN ALGORITHM THAT RAN $O(en^2)$

To try to obtain minimum runtime, consider implementing this algorithm with a heap

heap: a balanced binary tree such that for any subtree, the minimum of that subtree is the root.

For each of the e edges

- Get the minimum value
  - $O(1)$, minimum will always be the root
- Heapify the tree
  - Compare the original two children, move the smaller one up
  - Move down that node's child list and repeat recursively
  - $O(log(n))$

Therefore, this algorithm runtime is $O(elog(n))$, but is this the best?

- Worst case: $e = n^2$, so we have that worst case, this runs in $O(n^2log(n))$
- Dense graphs:
  - Use first algorithm
- Sparse graphs:
  - Use heap algorithm

Spanning Trees

spanning tree (ST): a graph with the following properties

- it is a tree
- it has the minimum number of edges ($n - 1$ assuming graph has n verticies)
- it spans/touches every vertex

How do we find a spanning tree?

- Trivial: Just run DFS

To make this more interesting, consider a different kind of spanning tree

minimum spanning tree (MST): a spanning tree such that it has minimum weight

We want to come up with an algorithm that determines the minimum spanning tree of a graph. To do this, we will state a very useful therom that we will use

Minimum Spanning Tree Therom

Take a graph, and split it into two partitions that has the following properties

- The two partitions are disjoint
- Each of the two partitions is non-empty
- All points in the graph are in either of two partitions

Consider all edges between partition 1 and partition 2, and denote the minimum weighted edge e_{min}.

Then ∃ a Minimum Spanning Tree that includes e_{min}.

Before we prove this therom, let's consider a couple algorithms that apply it

Prim's MST Algorithm

```
• Given a graph G, n verticies, e edges. Assume MST exists
• Create two partitions
    - L: contains an arbitary vertex
    - R: contains all other verticies
• While there are still verticies in R
    - Find the minimum edge between L and R, denoted as e[min]
    - Move e[min] into list for MST
    - Take vertex connected to e in partition R, and move it to partition L
```

Similarly to Dijkstra's, we get complexity $O(n^2)$ or $O(elog(n))$ depending on the implementation

Kruskal's MST Algorithm

```
• Given a graph G, n verticies, e edges, Assume MST exists
• Sort all the edges by weight in non-decreasing order
• While there are less than (n-1) edges in the MST
    - Consider the minimum edge that has not been considered
    - If including this edge in the MST creates a cycle
        * Discard and move on
    - Else including this edge in MST doesn't create cycle
        * Add to MST and move to next edge
```

CS 180: Lecture 9
Thursday, Week 4
10/26/17

Spanning Trees cont...

Last time we stated the MST Therom and used it a couple of times, but we have yet to formally prove it. Now we will do that

Minimum Spanning Tree Therom

Take a graph, and split it into two partitions that has the following properties

- The two partitions are disjoint
- Each of the two partitions is non-empty
- All points in the graph are in either of two partitions

Consider all edges between partition 1 and partition 2, and denote the minimum weighted edge e_{min}.

Then $\exists$ a Minimum Spanning Tree that includes e_{min}.

Proof

Assume by contradiction that we have a Minimum Spanning Tree that does not include e_{min}. We call this tree T'. Split the graph into two partitions such that e_{min} is one of the edges in the gap. Since this isn't the optimal tree, there must be another edge that goes between these two partitions, e_x. Consider the graph created by $T' \cup e_{min}$. Then we are guaranteed to have a cycle, because we now have n edges with n verticies, and two of the edges in the introduced cycle must be e_{min} and e_x for some edge in between them. If we replace e_x in our tree with e_{min} and consider $T' \cup e_{min} - e_x$, then we will have a tree that is more optimal than our first tree, so we have a contradiction.

CUTOFF FOR MIDTERM 1

Disjoint Set (Union-Find) Structure

We know how to do Kruskal's algorithm, but representing it as a data structure is not trivial. How can we best represent the graph to simplify Kruskal's algorithm the most? Consider the following problem.

We are given a bunch of numbers/nodes. We are allowed two operations in this problem

- Union
  - Take a vertex, and take another vertex, and find their union
- Find
  - Take two verticies, and see if they belong to the same group.

A problem of this kind is called a **Union-Find Problem**

How do we structure this?

1. Series of linked lists representing a union
   - Find: $O(n)$

    - Union: $O(1)$
2. All Elements in Union Point to Same Point
    - Find: $O(1)$
    - Union: $O(n)$
3. **Balanced Binary Search Trees**
    - Find: $O(log(n))$ (complexity of search in binary tree)
    - Union: $O(1)$ (just point the top of the smaller tree to the larger tree)

We can clearly see that number 3 is the best implementation of this type of structure. Now how does this apply to Kruskal's algorithm so we can simplify it?

- To add the smallest edge in our tree
    - Create a union between the two verticies
- What if we try to add an edge that will create a cycle
    - Use a find operation to see wether or not the two edges are in the same group
    - If they are, do not add to union, it would create a cycle.

Clustering

We are given a graph G. Our goal is to create k clusters such that

- The distance between clusters is large
- The distance between verticies within the same cluster is small.

To do this problem we want to group things together that have short edges, similar to what we do in Kruskal's algorithm, so for this algorithm we might want to take a similar approach that we took then.

What we can do is perform Kruskal's until we have the k clusters required, then stop. This is actually an optimal solution. Because of the way we choose edges in Kruskal's, if we had another group of clusters that we claim is more optimal, there would be a clustering we have in Kruskal's that would be split into two, but because Kruskal chooses shortest edges first, then the distance between the two clusters in the proposed optimal solution is shorter, so the solution is less optimal, and we have a contradiction.

CS 180: Lecture 10
Tuesday, Week 5
10/31/17

MIDTERM EXAM NO LECTURE

CS 180: Lecture 11
Thursday, Week 5
11/2/17

Divide and Conquer

The basic premise of this is that we will take a problem, split it into two smaller problems, and solve those problems recursively. This is very common and applys to many different problems, especially sorting.

Sorting: Merge Sort

We are given an array we want to sort

11 9 3 8 1 4 12 2

Divide in conquer breaks down the list into different sublists, and then solve the problem for smaller

11 9 3 8 1 4 12 2

11 9 3 8 1 4 12 2

11 9 3 8 1 4 12 2

So now each sublist is sorted since it only has 1 element, so now we will go back up the groups, each time merging the sorted subgroups into a group.

9 11 3 8 1 4 2 12

Merging

Now when we need to merge two lists, we do the following method

- Given: two sorted arrays, A and B, Output: A sorted array C that merges A and B
- Have two pointers, one pointing to A[0], one pointing to B[0]
- While both pointers are not at the end
    - Compare the two values at those two pointers
    - Take the smaller one, add it to C, and increment that pointer

NOTE: BOTH LISTS NEED TO BE SORTED TO APPLY MERGE

Now lets apply this to finish our problem

3 8 9 11 1 2 4 12

1 2 3 4 8 9 11 12

Time Complexity of Merging

Method 1:

- For each value in A, we will consider a value in B at most n times (assuming size of B is n and size of A is m).
- Therefore, this algorithm is $O(mn)$ ~ $O(n^2)$
- Can do better. Instead of charging time complexity to value in A or B, consider a value in the output
    - *How long does it take to get a value k in the output?*

Method 2:

- For each value k in the output, we will do $m + n$ comparisons max
- Therefore, merging is $O(m + n)$

Make sure analysis is the best it can be before redoing an algorithm

Time Complexity of MergeSort

Breaking down the steps we have the following facts

$$T(n) = 2T(\frac{n}{2}) + cn$$

$$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{cn}{2} \implies T(n) = 2^2T(\frac{n}{2^2}) + \frac{cn}{2}] + cn$$

$$\vdots$$

$$T(n) = 2^iT(\frac{n}{2^i}) + icn$$

We know eventually that this will become constant time, because we will break down the equation into lists with 1 element, so eventually we will have $T(1)$, therefore

$$\frac{n}{2^i} = 1 \implies 2^i = n \implies \boxed{i = log(n)}$$

Therefore, our formula becomes the following

$$T(n) = 2^{log n}T(1) + log(n)cn = cnlog(n) \implies \boxed{O(nlog(n))}$$

Graph Crossing Problem (pg. 224/225)

We are given a graph and we want to output the pairs of lines that cross

- For each line, check to see if it crosses each other line
- $O(n^2)$

We probably can't do better than $O(n^2)$ for this. What if we changed the problem though. Now, we only want to output *the number* of crossings. How can we use Divide and Conquer in order to solve this?

We basically will run mergesort, but each time we will count the number of inversions in each half. During the merge element, we do the following step

- Maintain pointers to the beginning of each list A and B
- Which pointer points to smaller element
    - If pointer in B, add it to list, and add number of elements left in A to be considered to SUM.
    - If pointer in A, add it to list.

Binary Search

Use divide and conquer to calculate the rutime of Binary search

$$T(n) = T(\frac{n}{2}) + c$$
$$T(\frac{n}{2}) = T(\frac{n}{4}) + c \implies T(n) = T(\frac{n}{4}) + 2c$$
$$\vdots$$
$$T(n) = T(\frac{n}{2^i}) + ic$$

Like before, we have the following

$$\frac{n}{2^i} = 1 \implies 2^i = n \implies \boxed{i = log(n)}$$

Therefore, as expected, binary search runs in $\boxed{O(log(n))}$ time.

Closest Pair of Points (pg. 226)

We will go over again Tuesday, read book for review/preview.

CS 180: Lecture 12
Tuesday, Week 6
11/7/17

Absent

CS 180: Lecture 13
Thursday, Week 6
11/9/17

Dynamic Programming

Introduction

For divide and conquer, we split the problem into two (usually) disjoint subsets to solve the problem. We will now introduce a new method known as **dynamic programming**. This method has a couple of major key differences from divide and conquer

- We split the problem into multiple subsets
- The subsets are not necessarily disjoint
- For each subproblem, we get an optimal solution, then combine these to get a solution to the problem as a whole.
    - Continue to do this recursively until each subproblem is small enough

Divide and conquer can be difficult to understand, but once you understand it it becomes a very powerful tool to use.

Weighted Interval Scheduling (pg. 252)

Same interval scheduling problem as before. We are given a bunch of intervals, however this time each interval has a certain weight that corresponds to it. Our goal is to find the schedule such that we maximize the total weight of the schedule and that none of the intervals are overlapping.

We start with a couple of assumptions

- Up to a certain coordinate x_i, we must know the optimal solution from the beginning to x_i
- For any $x_j < x_i$, we must know the optimal solution from the beginning to x_j

If there is an interval I_j such that it overlaps and goes past x_j then we compute two assumptions

- $I_j \notin S$ (solution)
    - $S_j = S_{j-1}$
- $I_j \in S$
    - $S_k = S_{l_j} + w_j$
    - S_{l_j} refers to the solution at x_i, the first ex coordinate before the beginning of j

We store all of the tentative solutions that we build up, and once we get to the end, we backtrack to find the optimal solution.

Knapsack Problem (pg. 267)

We are given a knapsack with size S. We have many items of different values, and sizes: $I_k = (s_k, v_k)$. There is an unlimited amount of each of the given items. Our goal is to place items in the knapsack such that we have the maximum size.

What parameters do we have that we can use dynamic programming on

- Number of items
- Size of knapsack

We have a table below that shows the Size of the knapsack against which items we are considering. We want to use dynamic programming to fill out each cell.

	S = 1	S = 2	S = 3	S = 4	S = 5
Item 1					
Items 1, 2					
Items 1, 2, 3					

In our algorithm, we will fill

- For item j, for knapsack size i
  - Item $j \in S$
    - Denote v_x as the value at item j, index $i - S_j$
    - $S_{i,j} = v_x + v_j$
  - Item $j \notin S$
    - $S_{i,j} = S_{i,j-1}$
  - Take $S_{i,j}$ as the maximum of these two values.
- Eventually, we will have the value for all items and knapsack size S
- The comparisons in each value of the array take constant time, so for this algorithm we have $O(nC)$
  - This is not polynomial because we have two different variables that the order depends on, but we could have, for example, $C = 2^n$
  - We call functions of this type **pseudo-poynomial**.

CS 180: Lecture 15 Thursday, Week 7 11/16/17

Dynamic Programming cont...

Segmented Least Squares: Multi-way choices (pg. 261)

Let's say we have a set of points in 2-D, and we want to approximate the points with a straight line. The straight line will attempt to minimize some error function E. In some cases, we possibly cannot do a good linear estimation with just one line, so in this case we are allowed to do multiple line segments.

- Consider if we chose n line segments, we would have zero error, but problem would be complex
- If we chose 1 line segment, we would have large error, even though the problem is simple

What is the right approach. Let us denote a cost function that we look to minimize:

$$C = \lambda l + S_1$$

Where l denotes the number of lines (multiplied by a constant λ), and S_1 is the error function. Let's use dynamic programming to solve

- Assume points are sorted by increasing x
- Assume that we know the optimal solution for the first i points
- When considering the optimal solution for the first $i+1$ points, we have two choices
  - We do not add a new line for the $i+1$ point. Just extend the line that goes through point i
  - We are going to add a new line in this instance
    - $\forall k < i+1$
      - add a new line segment from k to $i+1$ in the optimal solution up to k points
    - compute the error of all of these lines, and take the line that has the smallest error
    - $O(n^2)$ because we have n error calculations to do when finding a line/error and n lines at most
  - Calculate the cost function in each instance, and take the smaller of the two as the optimal solution up to $i+1$ points.

Algorithm runs in $O(n^3)$ time. Computing each optimal line addition is $O(n^2)$ time, and we do this n times for a total of $O(n^3)$.

Subsequence Problem

We are given a sequence of unique integers. We define the following

- A subsequence is increasing if for all i, j in the subsequence such that $i < j$, $S_i < S_j$
- A subsequence is contiguous if all elements in the subsequence are adjacent and in the same order as in the regular sequence

Find the largest increasing subsequence (not necesarrily contiguous)

Assume that for each of the first i elements of the sequence, we know the optimal solution. We now want to find the optimal solution for the first $i+1$ points. We have two cases to consider

- x_{i+1} is not in the solutions
    - the optimal solution up to x_i is the same as the optimal solution up to x_{i+1}
- x_{i+1} may be in the solution
    - for each of the the optimal soutions at x_k in $x_1, \ldots, x_i$
        - if $x_{i+1} > x_k$
            - add 1 to the solution corresponding to x_k
    - denote the optimal solution in this case as the

CS 180: Lecture 17
Thursday, Week 9
11/30/17

Network Flow

The runtime of Ford-Fulkerson is $O(e|f|)$

- There are at MOST $2e$ edges in the residual graph
- $|f|$ corresponds to the total amount of flow in the graph

This is psuedo-polynomial as of now, if edges have a very large capacity this could take a long time

Reminder: a **bipartite graph** is a graph where the nodes can be split between two disjoint groups, A and B such that every edge travels from $a_i \in A$ to $b_i \in B$

Matching Problem

Given a bipartite graph, find the maximum number of matches. This is a difficult problem, but when we consider this as a bipartite graph the problem is a little easier. We can model the bipartite graph as a network flow problem in orer to solve this

- Given two bipartite groups, A and B
- Direct all edges from A to B
- Create a source, s, and create an edge (s, a_i) for each $a_i \in A$
- Create a sink, t, and create an edge (b_i, t) for each $b_i \in B$
- Every edge has capacity 1
- Find max-flow using Ford-Fulkerson, max-flow corresponds to maximum matches

You can see how network flow transforms the problem, into one that is not only solvable, but also efficient. Since all of the capacities are 1, this problem will run at $O(n^2)$

Many of the problems relating to network flow are about **transforming the problem** to a network flow problem, solving that, and showing that the solution to the network flow problem is the same as the solution to the original problem.

Cell-Phone Problem

Suppose we have a collection of cell phone towers. At a given time, we have a collection of different cell phones that want to talk to each other. Each of the base stations have a certain capacity. Assume that the capacity of every base station is C. A phone can be connected to the base station if it is a distance of R or less. Find the maximum number of phones that can be connected to the base station system.

Again we utilize network flow to solve this. The algorithm is as follows

- Split graph into two bipartite sets
  - C for the set of cell phones
  - B for the set of base stations
- Direct all edges in C to their respective destinations in B
- Create a source, s, and direct source to each cell phone
- Create a sink, t, and direct each base station to it.
- Update capacities of all edges
  - Edges from s to a cell-phone have a capacity 1
  - If a cell-phone is less than distance R from a towers, ad a directed edge of capacity 1 between them
  - Edges from a base station to t have capacity of C
- Run Fork-Fulkerson, and the max flow is the answer

CS 180: Discussion 9
Friday, Week 9
12/1/17

Problem 1: Longest Consecutive Subsequence

Ex: Given a sequence: {1,9,3,10,4,20,2}. Longest consecutive subsequence is: 1,3,4,2 (order does not matter). Find an algorithm that finds the longest consecutive subsequence

A natural upper-bound to this problem is $O(nlog(n))$, because if we sort the sequence, then the solution to the problem is fairly trivial. This gives us the intuition that we want to solve this problem in $O(n)$ time.

We obtain a linear time solution by doing the following

- Put all numbers in a hash map
- For each number n in a sequence
    - If the number $n-1$ is also in the sequence, ignore this case
    - Else, this begins some consecutive subsequence, keep looking for greater values and note how long subsequence is

General strategies to take away from this problem for interviews

- In the beginning, try to come up with a trivial upper bound solution to get an idea of what your solution should be
- Try to make code that is simple

Problem 2: Sorted Rotated Array

We are given a sorted array A that is rotated to the right by k places. Find an efficient algorithm to see how many places the array was rotated/find the minimum

Upper bound solution: $O(n)$

- Traverse the array and fnd when the number decreases

This suggests that we want to find the solution in $O(log(n))$ time, which would suggest that we probably want to use binary search to some extent. The intuition for performing binary search on this problem is as follows.

- While we haven't found the number
    - Call our current search partition of the array A_i, center point a_c, beginning a_1, end a_k
    - If $a_c < a_1$, define new A_i on left half, redefine center,beginning, and end points
    - Else if $a_c > a_k$, define new A_i on right half, redifine center/beginning/end points
    - Else, array is already in sorted order, no rotation happened.

Problem 3a: All Permutations

Given a set of numbers, generate all permutations of these numbers.

The solution to this algorithm is known as **Heap's Algorithm for Generating Permutations**. The basics of this algorithm are as follows

- Start by building all permutations for the list of size 1
- Keep adding elements one by one, adding them in all possible slots for each of the permutations

Runtime: $O(n! * n)$

Problem 3b: Finding next sorted permutation

Given a permutation p, find what would be the next permutation in sorted order

- Find the longest increasing sequence from the right of the permutation
- Change the number directly to the left of this sequence with the first number in the sequence that is greater
- Reverse the sequence

Ex:

$$3, 2, 4, 1 \rightarrow 3, 4, 2, 1 \rightarrow 3, 4, 1, 2$$

Ex:

$$2, 5, 4, 6, 3, 1 \rightarrow 2, 5, 6, 4, 3, 1 \rightarrow 2, 5, 6, 1, 3, 4$$

Problem 4: Merge k sorted lists

We have N sorted lists of size $k_1, k_2, \ldots, k_n$. Merge these sorted lists.

Trivial algorithm:

- Have pointers in each list that track where we are
- Compare all pointers until we find the minimum
- Advance that pointer then continue until we are done

Time complexity of trivial algorithm is $O(N * \sum k_i)$. We can see that the bottleneck of this operation is the N comparisons that we do when finding the minimum. Luckily, we have a great data structure we can use that easilly finds the minimum: a **heap**

The following algorithm uses a heap in order to make this problem easier

- Have pointers in each list that tracks where we are
- Add all of these pointers to a heap
- While we are not done
    - Take the minimum from the top
    - Advance that pointer, and add the new pointer/value to the heap
    - Re-heapify

Now, this brings the time complexity of the problem to $O(log(N) * \sum k_i)$ because of the following

- Finding the minimum in the heap is constant time (it's always at the top)
- Re-Heapifying is $O(log(N))$