CS 161: Design and Analysis of Algorithms

Samuel Wong
Department of Statistics
Stanford University

Abstract

The course starts with the application and the proof of the Law of Large Numbers and the Central Limit Theorem. Monte Carlo methods are then introduced as well as convergence and measure-theoretic probability. Statistical models, hypothesis testing, and parameter estimation both in the parametric and non-parametric setting are discussed. Both frequentist and Bayesian methodologies are covered. Lastly, the course moves to discussions about Markov chains/processes. The settings with discrete and continuous time and space are examined with focus on first transition analysis and stochastic control.

Contents

1 General Ideas/Concepts **3**
1.1 Time Complexity . . . 3
1.2 Proof by Induction . . . 3
1.3 Recurrence Relations . . . 3
1.3.1 Master Theorem . . . 3
1.3.2 Substitution Method . . . 4
1.4 Randomized Algorithms . . . 4

2 Multiplication **5**
2.1 Naive Method . . . 5
2.2 Divide and Conquer: Method 1 . . . 5
2.3 Divide and Conquer: Method 2 (Karatsuba) . . . 5

3 Sorting **7**
3.1 InsertionSort . . . 7
3.2 MergeSort . . . 7
3.3 QuickSort . . . 7
3.4 Comparison Based Sorting . . . 8
3.5 CountingSort . . . 9
3.6 RadixSort . . . 9

4 Select **11**
4.1 Naive Method . . . 11
4.2 Linear Selection . . . 11
4.2.1 Selecting the Pivot . . . 11
4.2.2 Partitioning around this Pivot . . . 11
4.2.3 Recurse . . . 12

5 Binary Search Trees **13**
5.1 Red-Black Trees . . . 13
5.2 2-3-4 Trees . . . 13

6 Hash Tables **14**
6.1 Hash Functions . . . 14

7 Graphs/BFS/DFS **15**
7.1 Breadth First Search . . . 15
7.2 Depth First Search . . . 15
7.3 Strongly Connected Components . . . 15
7.4 Dijkstra's Algorithm . . . 16
7.5 Bellman-Ford . . . 16
7.6 Floyd-Warshall . . . 17

8 Dynamic Programming **19**
8.1 Longest Common Subsequence (LCS) . . . 19
8.2 Unbounded Knapsack . . . 20
8.3 0/1 Knapsack . . . 21

9 Greedy Algorithms **22**
9.1 Activity Selection . . . 22
9.2 Scheduling . . . 22

10 Minimum Spanning Trees **24**
10.1 Prim's Algorithm . . . 24
10.2 Kruskal's Algorithm . . . 25

11 Min Cuts and Max Flows **26**
11.1 Karger's Algorithm . . . 26
11.2 s-t Min Cuts and Max Flows . . . 26

1 General Ideas/Concepts

1.1 Time Complexity

There are a few different ways to analyze the runtime of an algorithm

- Worst Case Analysis (mainly focused on this since it tells us how the algorithm will perform on ANY input)
- Best Case Analysis
- Average Case Analysis (used for randomized algorithms)

Let $T(n)$ be the runtime of our algorithm. We say that $T(n) = O(f(n))$ iff there exists positive constants c and n_0 such that for all $n \geq n_0$, that $T(n) \leq cf(n)$. For example, if our algorithm has $T(n) = 3n^2 + 5n$, we say that this algorithm is $O(n^2)$, since we can find $c = 4$ and $n_0 = 5$, where this is true. When $n \geq 5$, $4n^2$ is always an upper bound to $3n^2 + 5n$. This is what we call Big-O (upper bound).

We say that $T(n) = \Omega(f(n))$ iff there exists positive constants c and n_0 such that for all $n \geq n_0$, that $T(n) \geq cf(n)$. This is what we call Big-Ω (lower bound).

We say that $T(n) = \theta(f(n))$ iff $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. In other words, there exists positive constants c_1, c_2, n_0 such that for all $n \geq n_0$, that $c_1 f(n) \leq T(n) \leq c_2 f(n)$. This is what we call Big-θ notation (tight bound).

1.2 Proof by Induction

There are 4 ingredients to Proof by Induction:

1. Inductive Hypothesis (IH): This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. i). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole range of values for i.
2. Base Case: First establish that the inductive hypothesis holds for some base case value(s) of i.
3. Inductive Step (Weak): Next, assume that the inductive hypothesis holds when i takes on some value k. Now prove that the IH holds as well when i takes on the value $k + 1$. Inductive Step (Strong): Next, assume that the IH holds when i takes on any value between [base case value(s)] and some number k. Now prove that the IH holds as well when i takes on the value k.
4. Conclusion: By induction, conclude that the IH holds across the range of i you're dealing with.

As a rule of thumb, we can use weak induction for iterative cases and strong induction for recursive cases. The difference is that for strong induction, we assume that the IH holds for all values up to the current one, which we need to hold when we are trying to prove something that depends on more than just the previous iteration.

1.3 Recurrence Relations

Recurrence Relations give us a recursive way to express runtimes for recursive algorithms! For a recurrence relation, we add the work it takes for each of the sub-trees + the amount of work done at the current level. For example, in MergeSort, we split the problem recursively into the left subtree and the right subtree and at each level we do $O(n)$ work to combine the sorted sub-lists from the children. Therefore, we would have $T(n) = 2T(\frac{n}{2}) + O(n)$.

We also need a base case, so for MergeSort, we would have $T(1) = O(1)$.

1.3.1 Master Theorem

Sometimes, our recurrence relation looks nice and is of the form $T(n) = a * T(n/b) + O(n^d)$. The Master Theorem states that:

$$T(n) = \begin{cases} O(n^d log n); \ a = b^d \\ O(n^d); \ a < b^d \\ O(n^{log_b a}); \ a > b^d \end{cases}$$

where a is the number of subproblems, b is the factor by which the input size shrinks, and d is the amount of work in the current problem.

The intuition behind the master theorem is that in the case where $a = b^d$, we are shrinking the problem and growing the number of subproblems at the same rate, so the same amount of work is done at every level, so the total amount of work is the number of levels times the amount of work per level. When $a < b^d$, the problem is shrinking faster than the amount of work done per level, so the highest level dominates the total amount of work. When $a < b^d$, the converse happens, so the leaves dominate (there are $log_b n$ leaves and $a^{log_b n} = n^{log_b a}$). Each leaf has work $O(1)$.

1.3.2 Substitution Method

When we don't have something in the nice form where we can apply the Master Theorem, we can fall back on the substitution method. Using the substitution method, we make an educated guess (sometimes by unraveling the recursion a few steps) and then prove our educated guess.

An example of this is if we have the recurrence relation of $T(n) = T(\frac{n}{5}) + T(\frac{7}{10}) + n$, where $T(n) = 1$, when $1 \leq n \leq 10$. Let's guess $O(n)$. In this case we cannot guess by unraveling so we may run a Python script to find a guess. Using induction, we have an inductive hypothesis that $T(n) \leq C * n$, though we still need to find the C. In our inductive step we let $k > 10$ and we assume that the base case holds for all $1 < n < k$. Then we have

$T(n) = k + T(\frac{k}{5}) + T(\frac{7}{10})$
$\leq k + C * (\frac{k}{5}) + C * (\frac{7}{10})$
$= k * (1 + \frac{C}{5} + \frac{C*7}{10})$
$\overset{?}{\leq} C * k$

To solve $k * (1 + \frac{C}{5} + \frac{C*7}{10}) \leq C * k$, we obtain that $C \geq 10$. Therefore, we have found a constant where this works and therefore it is $O(n)$. If the guess is too low, we won't be able to find a constant C that satisfies the equation. If the guess is too high, the equation will be obvious.

1.4 Randomized Algorithms

A randomized algorithm is an algorithm that incorporates randomness as part of its operation. Basically, we'll make random choices during the algorithm. There are two categories of randomized algorithms:

- Las Vegas Algorithms: This type of algorithm guarantees correctness but the runtime is a random variable
- Monte Carlo Algorithms: This type of algorithm guarantees runtime but does not always guarantee correctness

For randomized algorithms, we want to know both the expected runtime as well as the worst-case runtime. For the expected runtime, we select the worst input, and we reason about the expected runtime from the algorithm. For the worst-case runtime, we select the worst input, as well the the worst possible algorithm choice in the "random" step, and reason about the runtime. Note that the worst-case runtime is not random. It is fixed as the worse possible runtime for the algorithm. Also, note that in both expected and worse-case, we use the worst possible input.

Let's take an example where we want to try to sort a list. We take as input a list, randomly rearrange the elements, and hope that it is sorted. In this scenario, note that the worst possible input does not matter because we randomly rearrange the elements anyways. Therefore, the expected runtime is $O(n^2)$. The worst-case runtime is $O(\infty)$, since there is a possibility that the randomness never comes up with the elements all being sorted.

2 Multiplication

Given 2 non-negative numbers x, y with n digits each, we want to output the product $x * y$.

2.1 Naive Method

Similar to what was learned in grade school, we take the first number and multiply it by each digit of the second number, and then sum them all up. For example, if we did $45 * 63$, we would perform $45 * 3 = 135$ and $45 * 6 = 270$, and then add up $135 + 2700 = 2835$ (the 270 is shifted by 10 since it is in the tens digits place).

Assume both x, y have n digits. To do each partial multiplication, we need about $2n^2$ operations. This is because for each digit of the first number, we need to multiply it by the digit of the second number. For each of these, there is a possible carry (number bigger than 10 which would result in addition of 1 for the next multiplication). Therefore, we have $2n$ operations, and we need to repeat it n times (one for each digit of the second number), so $2n^2$ operations.

Adding these partial products together is also $2n^2$ operations. There are n of these partial products, with approximately $2n$ digits, so $2n^2$.

Therefore the total runtime complexity is $4n^2$, which is $O(n^2)$.

2.2 Divide and Conquer: Method 1

Divide and conquer is an algorithm design paradigm where we

1. Break up the problem into smaller subproblems
2. Solve those subproblems recursively
3. Combine the results of those subproblems to get the overall answer

In this case, let's say we want to multiply 2 different 4 digit numbers, which are 1234 and 5678. We can rewrite 1234 * 5678 as (12 * 100 + 34) * (56 * 100 + 78). Expanding out, this equals (12 * 56) * 100^2 + (12 * 78 + 34 * 56) * 100 + (34 * 78). What we have done is split one problem of multiplying 4 digit numbers into 4 problems of multiplying 2 digit numbers (plus some bit-shifting).

More generally, when we have $[x_1x_2...x_{n-1}x_n] * [y_1y_2...y_{n-1}y_n]$, we can rewrite this as $(a * 10^{\frac{n}{2}} + b) * (c * 10^{\frac{n}{2}} + d)$, which then equals $(a * c) * 10^n + (a * d + b * c) * 10^{\frac{n}{2}} + (b * d)$. A recursive pseudocode would look something like:

```
MULTIPLY(x, y):
    if (n = 1):
        return x * y

    write x as a * $10^{\frac{n}{2}} + b$
    write y as c * $10^{\frac{n}{2}} + d$

    ac = MULTIPLY(a, c)
    ad = MULTIPLY(a, d)
    bc = MULTIPLY(b, c)
    bd = MULTIPLY(b, d)

    return ac * $10^n$ + (ad + bc) * $10^{\frac{n}{2}}$ + bd
```

Let us look at the efficiency. In level 0, we have 1 problem of size n. In level 1, we have 4 problems of size $\frac{n}{2}$. In level t, we have 4^t problems of size $\frac{n}{2^t}$. In order to get down to the last level, where each problem is of size 1, it takes $log_2 n$ levels. Therefore, in the last level, we will have $4^{log_2 n}$ problems of size 1. We can calculate that $4^{log_2 n} = n^{log_2 4} = n^2$. Therefore, the runtime complexity is $O(n^2)$, the same as the naive algorithm.

2.3 Divide and Conquer: Method 2 (Karatsuba)

Recall that with the divide and conquer strategy that when we have $[x_1x_2...x_{n-1}x_n] * [y_1y_2...y_{n-1}y_n]$, we can rewrite this as $(a * 10^{\frac{n}{2}} + b) * (c * 10^{\frac{n}{2}} + d)$, which then equals $(a * c) * 10^n + (a * d + b * c) * 10^{\frac{n}{2}} + (b * d)$. The key difference in this Karatsuba method is the divide the problem into 3 subproblems, not 4.

The trick is to notice that the middle term, $(a * d + b * c)$, equals $(a + b)(c + d) - a * c - b * d$. We already compute $a * c$ and $b * d$ for the other two terms, so we can reuse those values. We look at the $(a + b)(c + d)$ term. Each of these a, b, c, d are $\frac{n}{2}$ long, and thus $a + b$ takes $\frac{n}{2}$ operations as does $c + d$. Therefore, we can see that the $(a + b)(c + d)$ term is of the same order as the $a * c$ or $b * d$ term. Thus, we can solve this in 3 subproblems rather than 4!

Thus, the runtime of this algorithm is $3^{log_2 n} = n^{log_2 3} \approx n^{1.6}$. Therefore, the runtime complexity is $O(n^{1.6})$, which is better than the naive algorithm.

3 Sorting

Given an unsorted list(or array) of n elements, output a sorted list.

3.1 InsertionSort

Intuition: We start from the first element and iterate through the list in order. If the element we are iterating through is larger than the previous one, then nothing is done. If the element is smaller, then it is swapped with the one previous. Then it is compared again to the one previous and if it is smaller, then it is swapped again. This repeats until the element is larger than the previous one, and then we resume traversing the list from where we left off.

```
InsertionSort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = cur_value
    return A
```

The outer for loop runs n times, and the inner while loop runs at most n times, and therefore the overall run time for this algorithm is $O(n^2)$.

3.2 MergeSort

Intuition: We divide the list into halves and recursively keep dividing into halves until each sub-list is only one element. Then we re-join the sub-lists, starting by comparing the 1 by 1 elements. Now we have sorted 2 element sub-lists and combine them in order. Now we have 4 element sub-list etc. until we return back to the original size.

```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = A[0:n/2]
    R = A[n/2:n]
    return MERGE(L,R)

#helper function MERGE
MERGE(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

We can see that the MERGE helper function is $O(n)$ since the for loop runs n times and each time there is $O(1)$ work. MERGESORT divides the problem in half each time, so therefore at each layer we double the number of problems, and each MERGE then does half the work from the previous layer since n has decreased by a factor of 2. Therefore, we have $O(n)$ work at each layer, and we have $log_2 n$ layers of recursion, so the overall run time for this algorithm is $O(nlogn)$.

3.3 QuickSort

QuickSort is a randomized algorithm for sorting. We will see that the expected running time is $O(nlogn)$ and the worst-case running time is $O(n^2)$. The intuition behind QuickSort is that we select a pivot at random, partition L and R around that pivot, and then recurse. This is a divide-and-conquer strategy. The pseudocode is below:

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Therefore the recurrence relation looks like $T(n) = T(|L|) + T(|R|) + O(n)$, where the base case is $T(0) = T(1) = O(1)$. If the randomness were to select the worst pivot each time (the min or max), then we have $T(n) = T(0) + T(n-1) + O(n)$. Therefore, the worst-case running time is $O(n^2)$. Another way to think of it is that in the worst case, we would have to run n recursions with each doing n amount of work for the partitioning, so $O(n^2)$.

To find the expected runtime, we make two observations. The first is that the comparisons between elements are made with the pivot and not each other. The second is that once an element is partitioned to L or R, it will never again be compared with any element from the other group (nor the pivot). To calculate the expected runtime, we first define a Bernoulli random variable as $X_{a,b} = 1$ if elements at indeex a, b are compared and 0 otherwise. Then our total number of comparisons is $E[\sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} X_{a,b}] = \sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} E[X_{a,b}]$

So what is $E[X_{a,b}]$? We know that it is $P[X_{a,b} = 1]$ since it is a Bernoulli random variable. We now use our observations. If a pivot that is not at index a or b is selected that takes the value between the values of the ones at a and b, then we know that a and b will never be compared because one will go to L and one will go to R. Therefore, we deduce that the $P[X_{a,b} = 1]$ is the probability of selecting a or b divided by the probability of selecting a value between the ones at a and b (inclusive). This equals $\frac{2}{b-a+1}$. Therefore, we have

$\sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} E[X_{a,b}] = \sum_{a=0}^{n-2} \sum_{b=a+1}^{n-1} \frac{2}{b-a+1}$
$= \sum_{a=0}^{n-2} \sum_{c=1}^{n-a-1} \frac{2}{c+1}$
$\leq \sum_{a=0}^{n-1} \sum_{c=1}^{n-1} \frac{2}{c+1}$
$= 2n \sum_{c=1}^{n-1} \frac{1}{c+1}$
$\leq 2n \sum_{c=1}^{n-1} \frac{1}{c}$
$= O(nlogn)$

Therefore our expected runtime is $O(nlogn)$.

One of the reasons why QuickSort is great is that it can do the sorting in place, so it is space efficient. So in the partition step, in practice, we use a different method than creating an L and an R. In practice we would:

- Take the pivot and swap it with the last element.
- Start two counters from index 0.
- Traverse the list moving one counter up by 1 each time
- When traversing, if the element is greater than the pivot, do not increment the second counter. If the element is less than the pivot, swap it with the element at this second counter and increment the second counter by 1.
- When we reach the end of the list, swap the pivot with the element at the second counter.

Note that in theory, rather than selecting our pivot randomly, we could use median of medians (discussed in a later section) to select an approximate median. The selection process is $O(n)$ and then this algorithm would not be random anymore, and therefore always be $O(nlogn)$. However, in practice, the $O(n)$ selection process has a high constant factor, so it is fairly expensive, so we pick the pivot randomly, and most likely we will not fall into the worst case runtime, but rather expected runtime of $O(nlogn)$.

3.4 Comparison Based Sorting

All the methods above, InsertionSort, MergeSort, and Quicksort are all comparison based sorting algorithms. Using this framework, we want to sort an array of items, but we cannot access the item's values directly, but instead can only compare two items to find out which one is bigger or smaller. "Comparison-based sorting algorithms" are general-purpose. The algorithm

makes no assumption about the input elements other than that they belong to some totally ordered set.

For a comparison-based sorting algorithm, we cannot do better for runtime than $O(nlogn)$. We can prove this by the following. We can diagram out comparison-based sorting as a binary decision tree, which takes in some input and outputs some combination of those inputs in a sorted order. Therefore, given the input is of length n, there are $n!$ different combinations of outputs. Therefore, the shallowest tree with at least $n!$ leaves has a depth of $log(n!)$. Thus, in all binary trees with at least $n!$ leaves, the longest path from the root to the leaf has length at least $log(n!)$. Through math, we can show that $log(n!) = O(nlogn)$. Therefore, we cannot do better than $O(nlogn)$, which is the runtime of MergeSort and the expected runtime of QuickSort.

However, if we do not limit ourselves to comparison-based sorting, say if we are able to work with the values directly, or knew what they were beforehand, we can do better than $O(nlogn)$.

Why then do we bother with comparison-based sorting? It is because comparison-based sorting can handle handle arbitrary comparable elements, such as arbitrarily large values, irrational numbers, etc. In addition, some comparison-based sorting algorithms can allow us to save space by doing the algorithm in place, such as QuickSort.

3.5 CountingSort

CountingSort does not use the comparison-based sorting paradigm, but instead we assume that there are only k different possible values in the array (and we know these k values in advance). In CountingSort, we allocate space (1 bucket for each of the k values). The buckets are allocated the sorted order that we want (for example, a bucket for 10, a bucket for 20, ..., etc.). Each bucket is a queue, so first in, first out. Creating the space can be viewed as taking k time ($k < n$).

We traverse the input array, and for each value we see, we place it into the corresponding bucket for that value. This takes n time. We then traverse the buckets (which are in sorted order) and remove the values one at a time. This takes n time. We now have a sorted array in $O(n)$ time.

Again, the assumptions that allow us to use CountingSort are that we are able to know what bucket to put something in, we know what values might show up ahead of time, and there aren't too many such values.

If there are too many possible values that could show up, then we need a bucket per value, and this can easily amount to a lot of space. Therefore, we can improve on CountingSort by using RadixSort.

3.6 RadixSort

RadixSort builds off of CountingSort, but is a better algorithm because it can save space. It is used for sorting integers where the maximum value of any integer is M. (Note that this can be generalized to lexicographically sorting strings as well, we would just use 26 buckets instead of 10. Let us discuss the integer case.

The idea behind RadixSort is to perform CountingSort on the least-significant digit first, then perform CountingSort on the next least-significant, and so on. Therefore, instead of a bucket per possible value, we just need to maintain a bucket per possible value that a single digit (or character) can take on!

To outline the steps more concretely, assume we have an array of size n (n integers). We create 10 different buckets in order labeled 0 through 9. We then traverse the array and placing each integer into the corresponding bucket of its least significant digit. Using the same methodology as CountingSort, we then visit the buckets one by one and remove the elements in order (FIFO because each bucket is a queue). Note that this is a stable sort, since for two entries with the same value, they will come out the same order that they came in. We then repeat for the next least significant digit and we repeat this until we have reached the most significant digit for the largest number in the input array. Hence, we repeat this d times, if the largest number has d digits.

Let's look at the runtime. We make d iterations and each iteration is $O(n)$, so the total runtime is $O(nd)$. Assume our largest integer is M. In the integer sorting case, we are using 10 buckets, so we will make $d = \lfloor log_{10}M \rfloor + 1$ iterations. Therefore, our runtime is $O(nlogM)$, so if M is about the size of n, we do not have improvement from MergeSort.

We solve this by increasing the number of buckets, call the number r. So more generically, the number of iterations is $d = \lfloor log_r M \rfloor + 1$. To be more specific, each iteration we said takes $O(n)$ before, but we were actually saying it took $O(n + 10)$, because we need to add in initializing the 10 buckets. Now with r buckets, each iteration takes $O(n + r)$. Therefore, our total runtime is $O((\lfloor log_r M \rfloor + 1)(n + r))$. Therefore, we have a tradeoff. More buckets means less iterations, but each iteration has

more runtime because there are more buckets to initialize.

A good sweet spot to use is to let $r = n$. Then we have $d = \lfloor log_n M \rfloor + 1$ iterations. Each iteration takes $O(n + n) = O(n)$ time. Therefore, the total runtime is $O((\lfloor log_n M \rfloor + 1)(n))$, which is $O(n)$. Therefore, more generally, if $M \leq n^c$ for some constant c, then RadixSort is $O(n)$.

4 Select

Given an unsorted array A of n elements, we want to output the kth smallest element of A.

We can easily come up with an $O(nlogn)$ algorithm for this. We can just run MergeSort and once the array is sorted, return the appropriate index. But can we do better?

4.1 Naive Method

If we were asking for 1st smallest element, we could simply store the first element, then traverse the list while comparing each element to the stored element. If it is smaller, then replace the stored element with this new, smaller element. This is one traversal through so it is $O(n)$. However, what if we want to find the $\frac{n}{2}$th smallest element? Then we would have to save the $\frac{n}{2}$ smallest elements we have seen so far and then every time a small element comes along, we would have to replace all these values with the correct one, therefore it would be $O(n^2)$.

4.2 Linear Selection

We claim that linear selection can do this in $O(n)$ time. The intuition behind linear selection is as follows:

1. Select a pivot (an element in the array)
2. Partition around this pivot (move all the elements smaller than the pivot into the left array and all the elements larger than the pivot into the right array)
3. Recurse (If there are exactly k elements in the left array and we want to return the kth smallest element then return it, otherwise repeat step 1 only for the subarray that we know if where the kth element must lie

4.2.1 Selecting the Pivot

Intuitively, we can see that the best pivot to select would be one that is the median or close to the median. That way when we partition, we'll have half the elements in L and half in R, and then we can discard the half we don't need. If we select a pivot close to the min or max, we won't get much gain from this splitting (for example if we have 1 element on one side and n - 1 elements on the other). Note that to find the exact median, we are essentially solving the SELECT problem for the $\frac{n}{2}$th smallest element, which we cannot do because that is exactly what we are trying to solve in the first place. So instead of trying to do that directly, we will instead do this using the median of medians method.

The goal of median of medians is to divide the list into smaller lists. Then we take medians from each of those smaller lists, and then take the median of those medians. This will be our proxy for the true median.
With the median of medians approach, we divide the list into 5 subgroups. We recursively keep dividing these subgroups by 5 until we reach a subgroup small enough that we can compute the submedian of that group. We then go up one layer and find the median of all these submedians and repeat until we're back at the top layer. We have now found an approximate median for our entire list.

How close are we to the true median? In the worst case scenario, we are in the 30th or 70th percentile (instead of 50th). Let's check for the lower percentile and the upper will hold by symmetry. We see that if we have 5 groups, our median of medians is the submedian that is larger than 2 submedians and smaller than 2 submedians by definition. We also know half of the group that contaited this median of medians is smaller than it. Therefore, in the worst case scenario, half of each of the two smaller submedian groups and half the elements in this median group are smaller. Thus we have at most $3 * \frac{1}{2}\frac{n}{5}$, which equals $\frac{3n}{10}$ less than the median. We can do the same for the upper bound which would be $\frac{7n}{10}$.

4.2.2 Partitioning around this Pivot

We take all the elements that are smaller than the pivot and put them into L, and then take all the elements that are larger than the pivot and put them into R.

```
PARTITION(A, pivot):
    L, R = [], []
    for i in [1,...,len(A)]:
        if (A[i] == pivot):
            continue
        elif (A[i] < pivot):
            add A[i] to L
```

```
        else:
            add A[i] to R
```

This partitioning is $O(n)$.

4.2.3 Recurse

We recursively make calls until we find the element, or until there is only 1 element left and we return that one. The SELECT code is below:

```
SELECT(A, k):
    if (len(A) == 1):
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A, p)
    if (len(L) == k - 1):
        return p
    elif (len(L) > k - 1):
        return SELECT(L, k)
    elif (len(L) < k - 1):
        return SELECT(R, k - len(L) - 1)
```

The median of medians is $O(n)$. The partition is $O(n)$. From our analysis before, we know that in the worst case scenario, we partition at the $\frac{3n}{10}$ or $\frac{7n}{10}$, so then we need to recurse on $\frac{7n}{10}$ elements. Therefore, our recurrence relation is $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$. We can prove that this recurrence relation is $O(n)$, therefore SELECT is $O(n)$!

We can prove $O(n)$ using the substitution method and picking a constant $C = 10$.

$T(k) = T(\frac{k}{5}) + T(\frac{7k}{10}) + k$
$\leq 10 * \frac{k}{5} + 10 * \frac{7k}{10} + k$
$= k * (1 + 2 + 7)$
$= 10 * k$

5 Binary Search Trees

A Binary Search Tree (BST) is a data structure that we use to store data. Compared to other data structures such as linked lists and sorted arrays, here are some common runtimes.

Operation	Sorted Array	Unsorted Linked List	BST (Worst Case)	BST (Balanced)
Search	$O(log(n))$	$O(n)$	$O(n)$	$O(log(n))$
Delete	$O(n)$	$O(n)$	$O(n)$	$O(log(n))$
Insert	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$

We can see that Binary Search Trees, when balanced, give us the best of both worlds. In a BST, the root node is at the "top" of the tree. Each parent can have up to two children. What make a BST special from any binary tree is that every LEFT descendant of a node has key less than that node, and every RIGHT descendant of a node has key larger than that node.

The importance of a balanced tree can be see in a runtime. It's possible for an unbalanced tree to basically be an increasing sorted array (or decreasing), in which case to do operations on it takes $O(n)$. However, if we make sure that our tree is balanced, the time it takes to get from the root node to any leaves is $O(log(n))$, so that is also the runtime to perform operations.

Thus, we have Self-Balancing Binary Trees. Some examples of this are Red-black trees, AVL trees, B-trees, and 2-3-4 trees. In order to self balance, they use the property called "rotations." There are left rotations and right rotations. For an example of left rotation, imagine a node A that has some subtree in the left child and node B as the right child. When we perform the left rotation, B moves to where A was, and A becomes B's left child. The whole left subtree that B previously had becomes the right subtree for A. In right rotation, imagine that A's left child is B and has some subtree as a right child. In the right rotation, B moves to where A was and A becomes B's right child. The subtree that was the former right subtree of B then becomes A's left subtree.

5.1 Red-Black Trees

A Red-Black Tree is a type of BST with the following properties:

- Every node is either red or black
- The root is a black node
- No red node has a red child
- Every root-NIL path has the same number of black nodes on that path

The point of these rules is to maintain balance. Since there needs to be the same number of black nodes for every root-NIL path, we are guaranteed that the longest path is not too much longer than the shortest path from root to leaf. Red nodes can extend a path, but only by so much since red nodes cannot have red children, so red nodes must be followed by black nodes if at all. More precisely, one path can be at most 2 times another path, since we can only pad the path with red nodes every other generation.

We have the following theorem: Any Red-Black Tree with n nodes has height $O(logn)$. Therefore, searching has the runtime $O(logn)$. As it turns out, inserting and deleting in Red-Black trees is also $O(logn)$. This can be seen since there is an isometry with 2-3-4 Trees. To see this, we will need to understand 2-3-4 Trees.

5.2 2-3-4 Trees

6 Hash Tables

Hash Tables are a data structure that helps to store data in a quickly accessing way using a hash function. We can compare the runtime below to sorted arrays and unsorted linked lists.

Operation	Sorted Array	Unsorted Linked List	Hash Table (Worst Case)	Hash Table (Expected)
Search	$O(log(n))$	$O(n)$	$O(n)$	$O(1)$
Delete	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Insert	$O(n)$	$O(1)$	$O(n)$	$O(1)$

In order to accomplish the $O(1)$ runtime for search, delete, and insert, a naive way would be to use direct addressing (create one bucket for each entry). For example, if we were storing numbers from 1 to 1000, we could create 1000 buckets and then store those numbers. The problem here is the space requirements. If we were storing numbers that could be anywhere between 1 and 1 billion, then we would need 1 billion buckets, possibly to store just a few numbers if we only have a few entries.

We then can think to iterate on this idea by creating 10 buckets (and then store each entry we see into the bucket with its least significant digit). However, now the problem is that we may have collisions (more than one element put into a bucket). In fact, if an adversary wanted to make things really bad, they could only provide inputs with the same least significant digit and then all the entries would be stored in one bucket, hence the runtime would then be $O(n)$, not $O(1)$. Although collisions are inevitable (and when we have them we can use chaining or linear probing), we want to find a way to make them very infrequent. Therefore we need the idea of hash functions.

6.1 Hash Functions

Let us assume we have a universe U of possible keys (possible entries we may see, such as all natural numbers, all reals, numbers from 1 to 100 etc.), and the size of the set U is M. Our job is to store n keys. We assume that M is much larger than n, or else the problem would be trivial. What a hash function does is it maps elements of U to $\{1, 2, ..., n\}$.

For the rest of the explanation, we assume that the number of keys we need to store n, is also equal to the number of buckets. This doesn't have to be the case, but in practice we usually aim for the number of buckets to be O(# elements to store), otherwise we are using "too much" space.

What we do is we have a hash family (a set of hash functions). At the time of storing entries, we then select one hash function at random and then hash the entries into their proper buckets. Since we have chosen the hash function at random, an adversary cannot prepare beforehand how to foil our plans and make all the inputs go to the same bucket.

How do we define a good hash family? We define a good hash family by one where if we pick a hash function h at random, that for any item, if we hash it to a bucket with h, that the expected number of other items (after we hash all the items) in that bucket (collisions) is $O(1)$. Mathematically, this is equivalent to wanting that for two different items u_i and u_j, the $P(h(u_i) = h(u_j)) \leq \frac{1}{n}$. If a hash family satisfies this requirement, it is called a Universal Hash Family.

Although we can prove that the exhaustive set of all hash functions can fulfill this property, it is computationally intensive and impossible to store all different combinations of hashes for the whole exhaustive set of functions. We want to look for an easier way.

In practice, the Universal Hash Family that is commonly used is the following. Pick a prime number $p \geq M$.

- Define $h_{a,b}(x) = ((ax + b) mod p) mod n$
- This Universal Hash Family is $H = \{h_{a,b} : a \in \{1, ..., p-1\}, b \in \{0, ..., p-1\}\}$
- To pick a random hash function, simply uniformly select a value for a and one for b

Therefore, we are able to create a Universal Hash Family and use it by only storing two values, a and b.

7 Graphs/BFS/DFS

Graphs are a way that we can represent many types of problems that we face. Graphs consists of nodes (vertices) and edges. The graph can be undirected or directed. The edges can also assume different weights.

A graph can be represented as an adjacency matrix or an adjacency list. As an adjacency matrix, if we have n nodes, we have an $n \times n$ matrix, and where we have an edge from i to j, we have an entry there in the matrix, otherwise 0. As an adjacency list, we can have a list of length n (each entry representing one node), and each entry of that list is a linked list with all the nodes that are connected to that node.

7.1 Breadth First Search

Breadth First Search (BFS) is a way to explore a graph. In BFS, we have a starting node, then we visit all of its neighbors. We then visit all the neighbors of those neighbors and proceed like that. Therefore BFS can be thought of as layers. The starting node is layer 0. Layer 1 is all the nodes that are neighbors with Layer 0. Layer 2 is all the nodes (that have not yet been visited) that are neighbors of Layer 1, etc. BFS will eventually find all the nodes in $O(n + m)$ time. This is because we traverse every node n once, as well as every edge m once.

BFS can be used to find the shortest path (from a starting node to a node of interest). We perform BFS from the starting node, then to the next layer, then the next etc., and once we first hit the node of interest, that layer number is the shortest path.

BFS can also be used to test bipartiteness. A graph is bipartite if there exists a 2-coloring of the graph such that there are no edges between two of the same colored vertices. We can check this by starting from a node and coloring it one color. Then we do BFS, coloring the next layer the second color, then the next layer the initial color etc. If we end up in a situation where we try to color an already colored node a different color, then the graph is not bipartite. If we successfully color the graph, then it is bipartite.

7.2 Depth First Search

In Depth First Search (DFS), we go asfar down a path as we can before we comes back to explore other options. We also can visit the entired graph using DFS in $O(n + m)$. We can use DFS to print out the nodes in a BST in order.

One useful task for DFS is Topological Sorting. In this scenario, we have a graph that is a DAG (Directed Acyclic Graph). This means we have no directed cycles and the graph is directed. In Topological Sorting, we have a DAG and we want to find an ordering of vertices so that all of the dependency requirements are met. Essentially, we want to produce an ordering such that: for every edge (v, w) in E, v must appear before w in the ordering. The result is a "priority" ordering of the nodes.

How we can use DFS is that we can start at any node, and then run DFS. We keep track of all the starting and finishing times of the nodes of when we visit them (finishing means that we will not visit that node again). If we sort the nodes in descending order by finishing time (largest finishing time first), we will get the correct topological sorting order.

Note that for DFS (and also BFS), if we can't reach every node from one run, we then do another DFS starting from a node that has not yet visited until we visit every node in the graph. This is called a DFS forest.

7.3 Strongly Connected Components

We say that a graph $G = (V, E)$ is strongly connected if there is a directed path from any vertex in V to any other vertex in V. A strongly connect component (SCC) of a directed graph is MAXIMAL subset of vertices $S \subseteq V$ such that there is a directed path from any vertex in S to any other vertex in S. SCCs provide information about communities in a graph.

How do we find the SCCs in a directed graph? We first notice that we can break down any directed graph into SCCs and that those SCCs form a DAG. We also notice that for those SCCs, we have source SCCs (upstream) and sink SCCs (downstream). We can see that if we are able to identity the sink SCCs first, we can remove them from the graph and then we can find the next sink SCC etc. and continue.

We notice that if we run DFS on the graph, the node with the largest finish time will be a node in the source SCC. However, we wanted to start with the sink SCC. Therefore, we just reverse the edge directions! We then run DFS again starting with the node with the largest finish time and now everything that is traversed is in one SCC. We then repeat with the node with the next largest finish time that was not in the SCC just discovered. This is called Kosaraju's algorithm and specifically the steps are:

- Run DFS to explore the entire graph and get finish times for all the nodes
- Reverse all the edges in the graph
- Run DFS again, starting with vertices with the largest finish time

Note that this algorithm's total runtime is $O(n+m)$ since it is essentially just two DFS, which are each $O(n+m)$ and also bookkeeping the finish times which is $O(n)$.

7.4 Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm that gives us the single-source shortest paths for weights graphs with non-negative edges. For example, we want to figure out the shortest distance between node 1 and node 2, but each edge is a different length. How this algorithm works is that we first label each edge with an edge weight (we can think of this as a cost for going from v to w).

We start by giving each node a value of ∞. Assume we want to figure out the cost of going from node 1 to any node in the graph. We change the starting node weight to 0 (distance between that node and itself is 0). Out of all the nodes, we check for the one with the lowest weight (in this case it will be the starting node with weight 0), call it N_1. We then traverse all of its neighbors, updating their values to the minimum of their current weight and N_1 + the edge weight connecting N_1 to that node. The first time a node is reached, it will clearly be updated since the starting value was ∞, however in later iterations it may be less clear and we may or may not update it depending on which value is lower. After we have traversed all of N_1's neighbors, we remove N_1 from the graph (mark it as sure). We then repeat the process by selecting our new N_1 to be the node now in the graph with the lowest value. We proceed until there are no more nodes in the graph (unsure nodes), and then we have the minimum distance between node 1 to any node in the graph. The pseudocode is below:

```
DIJKSTRA(G, s):
    Set all vertices to not-sure
    d[v] = infinity for all v in V
    d[s] = 0
    while there are not-sure nodes:
        pick the not-sure node with the smallest estimate d[u]
        for v in u.neighbors:
            d[v] = min(d[v], d[u] + w(u,v))
        mark u as sure

    now all d[v] indeed equals the true distance d(s,v)
```

The total runtime of Dijkstra's algorithm is O(n * (T(findMin) + T(removeMin)) + m * T(updateKey)). Therefore if we used an array, we would have $O(n*(n+n))+O(m*(1)) = O(n^2)+O(m)$. If we instead used a Red-Black Tree, we would have $O(n*(logn+logn))+O(m*logn) = O((n+m)logn)$. Therefore, we can see that the tree is better than the array when the data is sparse, (m is small), however when the data is dense, then the array is better.

In practice, Dijkstra's algorithm is implemented using a Fibonacci heap. The runtime is $O(n*(logn+logn))+O(m*1) = O(nlogn+m)$.

Some weaknesses of Dijkstra is that it doesn't work with graphs with negative edge weights. Also, if the weights change, then we need to re-run the whole thing.

7.5 Bellman-Ford

While Dijkstra works to find the shortest path from a single source in $O(nlogn+m)$ time, but only for non-negative path lengths, there is an algorithm named Bellman-Ford that finds the shortest path from a single source for any type of path lengths (positive or negative), but at the cost of a higher run time.

The Bellman-Ford will return the shortest path if one exists, or no-solution if one doesn't exist. A case where there is no solution is where there exists a negative cycle in a graph: a cyclical path where the sum of the edge weights in that cycle is a negative number. There is no solution in this case, because in theory one could continuously loop through this cycle, reducing the total path weight until $-\infty$, which would be the lowest path length.

In Dijkstra, a node's estimate was potentially updated whenever it had an incoming neighbor u that was the node with the smallest distance estimate at the time. This is because u is that node's "best chance" for now at getting close to s (the source

node).

But in Bellman-Ford, now that we have to handle negative edges, so instead of only considering the u with the smallest $d[u]$ as the node's "best chance", we're going to consider any incoming neighbor as a possible lead! This basically has to do with the fact that with a negative edge has the potential to drop the cost of a path (even a path that's looking quite expensive so far) by a lot, so we never really know where the shortest paths will come from, so we need to try everything!

Let's see how we run Bellman-Ford. Let's assume we have a directed graph with n nodes and m edges and the source node is s. We create n different arrays each of length n, and denote them $d^{(k)}$, where $0 \leq k \leq n$. Each index of these arrays of length n represents a node. Each $d^{(k)}$ represents the shortest distance between a node and the source s, using at most k edges. Thus, the first array, $d^{(0)}$, has 0 for the index with the source node and ∞ for all other entries (for a length of at most 0 edges, the distance from s to $s = 0$, and we cannot get to any other node so all their distances are ∞).

Then for $d^{(1)}$, we look at all the nodes b again, and when we update, we realize there are two cases:

- Case 1: the shortest path from s to b with at most k edges could be one with at most $k\breve{}1$ edges! In other words, allowing k edges is not going to change anything.
- Case 2: the shortest path from s to b with at most k edges could be one with exactly k edges! I.e. this length-k shortest path is [length $k\breve{}1$ shortest path to some incoming neighbor a] + $w(a, b)$. Which of b's incoming neighbors will offer this shortest path?

Thus to account for these cases, for each node b in $d^{(1)}$, we update it as $d^{(1)} = min\{d^{(0)}[b], min_a\{d^{(0)}[a] + w(a, b)\}\}$. We then repeat for $d^{(2)}, d^{(3)}, ..., d^{(n)}$. The generic update step is $d^{(k)} = min\{d^{(k-1)}[b], min_a\{d^{(k-1)}[a] + w(a, b)\}\}$.

Note that since we must fill in each of the arrays of length n, essentially we need to traverse all edges m times where we look at the distances for n nodes, the total runtime of Bellman-Ford is $O(mn)$, which is larger than Dijkstra's runtime of $O(nlogn + m)$. In terms of storage, instead of keeping around all the arrays from $d^{(0)}$ to $d^{(n)}$, for each update step, we only require the information from the one prior, so we can actually only keep space for 2 of these arrays at a time if we want to save space (the current one and the previous one).

One thing about Bellman-Ford is that we can detect a negative cycle. How we do this is that we run Bellman-Ford as before, thus ending with the last array of $d^{(n)}$, which is our optimal solution. If we run one more iteration, $d^{(n+1)}$, and the values end up decreasing more than $d^{(n)}$, we know we have a negative cycle. Pseudocode for Bellman-Ford is below

```
BELLMAN_FORD(G, s):
    d^{(k)} = 0, for k = 0, 1, ..., n - 1
    d^{(0)}[v] = infty for all v in V (except s)
    d^{(0)}[s] = 0

    for k = 1 to n:
        for b in V:
            d^{(k)}[b] = min{d^{(k-1}[b], min_a{d^{k-1}[a] + w(a,b)}}

    if d^{(n-1)} != d^{(n)}:
        return "NEGATIVE CYCLE!"

    return d^{(n-1)}
```

One really nice thing about Bellman-Ford is that the computation is more distributed. Each vertex just needs to know the previous distance estimates of itself and its immediate neighbors to perform an update. This makes it very practical to implement for networks like the Internet! Instead of each node having to keep state of the entire network, it just needs this local information to perform Bellman-Ford or any updates. (e.g. Routing Information Protocol, or RIP, uses it!)

7.6 Floyd-Warshall

Floyd-Warshall is all algorithm that finds the shortest paths from v to w for ALL pairs v, w of vertices in the graph, (not just shortest paths from a special single source s). Therefore, if we want to find all the shortest paths in the whole graph, we can run Bellman-Ford with each node as the source node, but that runtime would be $O(mn) * O(n) = O(n^2 * n) * O(n) = O(n^4)$! Bellman-Ford can do this in a faster runtime that $O(n^4)$. Note that similarly to Bellman-Ford, Floyd-Warshall can handle negative edge weights.

In Floyd-Warshall, our smaller subproblem that we are trying to solve is what is the shortest path from v to w such that all the internal vertices on the path are in the set of vertices $\{1, ..., k\}$. We start by choosing one node, call it the "middle", and calculating all the shortest paths from different nodes v and w using that one. We then increase the "middle" to that node plus one more node. Then we recalculate all the shortest paths from different nodes v and w using that "middle." We repeat until our "middle" is comprised of all the nodes.

In the recalculation step, we consider two cases. Let $D^{(k)}[v, w]$ denote when the "middle" is comprised of k nodes, and we want to find the shortest distance between nodes v and w that goes through the "middle."

- Case 1: We don't need this new vertex k, so $D^{(k)}[v, w] = D^{(k-1)}[v, w]$
- Case 2: We need vertex k! So, $D^{(k)}[v, w] = D^{(k-1)[v,k]+D^{(k-1)}[k,w]}$

The pseudocode for Floyd-Warshall is below:

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays D^{(k)} for k = 0,...,n
    D^{(k)}[v, v] = 0 for all v, for all k
    D^{(k)}[v, w] = infty for all v not equal to w, for all k

    for k = 1,...,n:
        for pairs v, w in V^2:
            D^{(k)}[v,w] = min{D^{(k - 1)}[v, w], D^{(k - 1)[v, k] + D^{(k - 1)}[k, w]}}

    for v in V:
        if D^{(n)}[v, v] < 0:
            return "NEGATIVE CYCLE!"

    return D^{(n)}
```

The runtime for the Floyd-Warshall algorithm is $O(n^3)$!

BFS	DFS	Dijkstra	Bellman-Ford	Floyd-Warshall
$O(m+n)$	$O(m+n)$	$O(m+nlogn)$	$O(mn)$	$O(n^3)$
Unweighted	Unweighted	Weighted (weights must be non-negative)	Weighted (can handle negative weights)	Weighted (can handle negative weights)
Single source shortest path; Test bipartiteness; Find connected components	Path finding (s,t); Toposort (DAG!!); Find SCC's; Find connected components	Single source shortest paths: Compute shortest path from a source s to all other nodes	Single source shortest paths: Compute shortest path from source s to all other nodes; Detect negative cycles	All pairs shortest paths: Compute shortest path between every pair of nodes (v,w)

8 Dynamic Programming

Dynamic programming (DP) is another algorithm design paradigm (an example of a different paradigm is Divide and Conquer). It's often used to solve optimization problems (e.g. shortest path). Bellman-Ford and Floyd-Warshall are examples of DP.

The elements of Dynamic Programming are:

- Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.
- Overlapping sub-problems: the subproblems overlap a lot! This means we can save time by solving a sub-problem once and cache the answer. (this is sometimes called "memoization")

The key difference between dynamic programming and divide and conquer is that the subproblems in DP overlap! In Divide-and-Conquer, this is not the case. There are two approaches for how to write a DP algorithm:

- Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first and building up). This method will have an iterative form.
- Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort). This method will have a recursive form.

The recipe for Dynamic Programming is as follows:

1. Identify optimal substructure: What are your overlapping subproblems?
2. Define a recursive formulation: Recursively define your optimal solution in terms of sub-solutions. Always write down this formulation.
3. Use dynamic programming: Turn the recursive formulation into a DP algorithm.
4. If needed, track additional information: You may need to solve a related problem, e.g. step 3 finds you an optimal value/cost, but you need to recover the actual optimal solution/path/subset/substring/etc. Go back and modify your algorithm in step 3 to make this happen.
5. (Optional) Can we do better: Any wasted space? Other things to optimize?

8.1 Longest Common Subsequence (LCS)

We are given two sequence X and Y, and we want to find their LCS. A sequence Z is a subsequence of X if Z can be obtained from X by deleting symbols. The LCS is defined as the longest subsequence of both X and Y. Some applications of LCS include: bioinformatics (detecting similarities in DNA strands, computational linguistics (similarites in word forms), and the diff Unix commmand.

Following the recipe for DP, we first identify the subproblem here. The subproblem here is to find the length of LCS's of the prefixes to X and Y (the first few elements of X and Y). Let $C[i, j] = \text{length\_of\_LCS}(X_i, Y_j)$. If we consider the ends of our prefixes, we have two cases:

- Case 1: $X[i] = Y[j]$. Then $C[i, j] = 1 + C[i - 1, j - 1]$
- Case 2: $X[i] \neq Y[j]$. Then $C[i, j] = max\{C[i - 1, j], C[i, j - 1]\}$.

Therefore, our DP algorithm is as follows:

```
LCS(X, Y):
    Initialize an (m + 1) x (n + 1) array #0-indexed array C
    C[i, 0] = C[0, j] = 0 for all i = 0,...,m and j = 0,...,n

    for i = 1,...,m:
        for j = 1,...,n:
            if X[i] = Y[j]:
                C[i, j] = C[i - 1, j - 1] + 1
            else:
                C[i, j] = max{C[i, j - 1], C[i - 1, j]}
    return C[m, n]
```

The algorithm above tells us the maximum length of the LCS. Suppose we want to find the string of the LCS. Then after running this algorithm, we can do a backwards pass starting at $C[m, n]$. How we do this is we check to see if $X[i] = Y[j]$. If so, then we append that to the LCS string and decrease i and j by 1. If they do not match, then we either decrease i by 1 or j by 1 and check again. Pseudocode is below:

```
RECOVER_LCS(X, Y, C):
    //C is already filled out
    L = []
    i = m
    j = n
    while i > 0 and j > 0:
        if X[i] == Y[j]:
            append X[i] to the beginning of L
            i = i - 1
            j = j - 1
        else if C[i, j] == C[i, j - 1]:
            j = j - 1
        else:
            i = i - 1
    return L
```

The runtime of creating C is $O(mn)$, since we have to traverse every element of X and Y to look at the longest subsequence. The runtime of recovering once we have C is the $O(max(m, n))$. Therefore, the total runtime is $O(mn)$. Can we do better? It is unclear since this problem is NP-hard.

8.2 Unbounded Knapsack

In the unbounded knapsack problem, we want to find the most valuable way to fill a knapsack with items. It is unbounded since there is an unlimited amount of each item that we can put into the knapsack as long as we don't exceed the capacity of the knapsack. Each item has a weight and a value. The sum of the items in the knapsack cannot exceed its weight, and the goal is to maximize the value.

The optimal subproblem in this case is to solve the problem for smaller knapsacks and use the answers to those smaller knapsacks for the larger knapsacks. The high level intuition is that for each item i that we can fit in our knapsack, we calculate the value of adding that item + the best value with capacity $x - w_i$. If we denote the optimal value we can fill a knapsack of capacity x as $K[x]$, then the recursive relation is that $K[x] = max_i\{K[x - w_i] + v_i\}$. The pseudocode is below:

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize a size W + 1 array, K
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i less than or equal to x:
                K[x] = max{K[x], K[x - w_i] + v_i}

    return K[W]
```

The runtime of this algorithm is $O(nW)$ since for each weight from 1 to W we try out all the n elements to get the optimal knapsack.
In actuality, we'd like our runtime to scale "nicely" with our input size, which is dependent on the capacity of our knapsack, W. Our input size is actually $O(nlogW)$, because it takes $logW$ bits to write down W, and it takes $nlogW$ bits to write down all n weights (we assume the values of each item are not the dominating factor here).

Thus, $O(nW)$ is not actually polynomial in the input size. We call these algorithms "pseudo-polynomial".
What we have found so far is the optimal value for the knapsack. Suppose we want to find the items in that knapsack. All we need to do it modify our algorithm from before to store the itemset.

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize size W + 1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = {}
```

```
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = {}
        for i = 1,...,n:
            if w_i less than or equal to x:
                K[x] = max{K[x], K[x - w_i] + v_i}

                if K[x] was updated:
                    ITEMS[x] = ITEMS[x - w_i] union {item i}
    return ITEMS[W]
```

This would also be $O(nW)$.

8.3 0/1 Knapsack

The 0/1 Knapsack problem has the same setup as the Unbounded Knapsack problem, except the difference is that we only have 1 copy of each item. We need to change our optimal subproblem from the Unbounded Knapsack to solving a smaller subproblem with a smaller knapsack AND fewer items. Now we index our knapsack by the items $K[x, j]$, where x is the capacity of the knapsack and j is the set of the items used.

Now, our recursive relation is that $K[x, j] = max\{K[x, j-1], K[x - w_j, j-1] + v_j\}$. Thus rather than just a list, we create a matrix that is $j \times x$. The pseudocode for this algorithm is

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
    Initialize a (n + 1) x (W + 1) table, K
    K[x, 0] = 0 for all x = 0,...,W
    K[0, W] = 0 for all j = 0,...,n

    for x = 1,...,W:
        for j = 1,...,n:
            K[x, j] = K[x, j - 1]
            if w_j less than or equal to x:
                K[x, j] = max{K[x, j], K[x - w_j, j - 1] + v_j}

    return K[W, n]
```

The runtime is $O(nW)$. This is because we have to examine knapsacks with weights from 1 to W, and for each we need to look through n different ways of ordering the items (without replacement).

9 Greedy Algorithms

The greedy paradigm takes the approach to commit to choices one-at-a-time, never look back, and hope for the best. The greedy solution does not always give us the optimal solution, but in some cases it does work!

9.1 Activity Selection

Activity selection is a problem in which a greedy algorithm works. In this problem, we have n activities with their own start times and finish times. The goal is to maximize the number of activities you can do. However, we can only do one activity at a time. Therefore we cannot do two activities with overlapping times.

The greedy algorithm that works is this: pick an available activity with the smallest finish time and repeat. Therefore, what we will do is sort all the activities by finish time, and then select the one with the smallest finish time that is possible to do. Pseudocode below:

```
ACTIVITY_SELECTION(activities A with start and finish times):
    A = MERGESORT_BY_FINISHTIMES(A)
    result = {}
    busy_until = 0
    for a in A:
        if a.start >= busy_until:
            result.add(a)
            busy_until = a.finish
    return result
```

The total runtime for this is $O(nlogn)$ since the mergesort takes $O(nlogn)$ and the traversal is $O(n)$. This is a greedy algorithm because at each step in the algorithm, we make a choice (pick the available activity with the smallest finish time) and never look back.

The proof of correctness is as follows:

- Inductive Hypothesis: After adding the tth activity, there is an optimal solution that extends the current set of activities.
- Base Case: After adding 0 activities, there is an optimal solution extending that (any solution extends 0 activities).
- Inductive Step (Weak): Suppose we've already chosen $(k-1)$ activities, and there's still an optimal solution that extends these choices. After adding the kth activity, we'll show that there is still an optimal solution that extends these k activities.
- Conclusion: After adding the last activity, there is an optimal solution that extends the current solution. The current solution is the only solution that extends the current set of activities (there is no remaining activity that we could still fit in). So, the current solution is optimal.

Let's expand on the Inductive Step. We assume that we have already chosen $(k-1)$ activities and that the optimal solution extends these choices. Let T^* be the optimal solution. Let's say our algorithm chooses the next activity, call it a_k. If a_k is in T^*, then we're good! Let's suppose a_k is not in T^*, and that actually a_j is in T^* instead. Our algorithm chooses the activity with the earliest finish time so the finish time of a_k is less than that of a_j. Therefore a_k doesn't conflict with anything in T^* after a_j, thus preserving the optimality of T^*. Thus, swapping a_j with a_k preserves optimality and we're still good!

9.2 Scheduling

Another problem that has a greedy solution is the scheduling problem. In this problem, we have a set of n jobs. Job i takes t_i hours. For every hour until job i is done, we pay the cost c_i. We want to find the order in which to complete the jobs to minimize the total cost. We can only perform one job at a time.

To solve this problem, we need to think about two variables, time and cost. It is clear that we want to perform the higher-cost jobs first. (We can see this by setting all the jobs to take the same amount of time and then looking at an example). It is also clear that we want to perform the shorter jobs first. (We can see this by setting all the jobs to be the same cost and then looking at an example). So which jobs do we do first? The higher cost ones or the shorter time ones?

For two jobs A and B, we can calculate the cost of doing A first then B. The cost is $t_A c_A + (t_A + t_B)c_B$. The cost of doing B first before A is $t_B c_B + (t_A + t_B)c_A$. Using these two equations, we find that we prefer to do A before B when $\frac{c_B}{t_B} \leq \frac{c_A}{t_A}$.

Therefore our greedy algorithm is as follows. Compute the cost to time ratio for all jobs. Sort jobs in descending order of cost/time ratios. Return the sorted jobs! The total runtime for this algorithm is $O(nlogn)$, due to the sorting.

10 Minimum Spanning Trees

Let's first define what a Minimum Spanning Tree is. A tree is an undirected, acyclic, connected graph. A spanning tree is a tree that connects all of the vertices. Let us assume we have an undirected, weighted graph. Thus, a Minimum Spanning Tree (MST) is a tree that spans all the vertices of this graph with the lowest total summed weight across its edges.

Another important definition is a "cut." A cut is a partition of the vertices into two non-empty parts. We say a cut respects a set of edges S if no edges in S cross the cut. We say an edge is "light" if that edge has the smallest weight of any edge that crosses the cut.

An important lemma that will be used for proof of correctness is the following: Consider a cut that respects a set of edges S. Suppose there exists an MST T^* containing S. Let (u, v) be a light edge crossing this cut. Then there exists an MST containing $S \cup (u, v)$.

Why is this lemma true? Well let's suppose that (u, v) is not in T^*. We know T^* is an MST so there must be some edge of T^* that crosses the cut since it connects all the vertices. Call this edge (x, y). Replace this edge (x, y) with (u, v), and we must get another spanning tree T, since this (u, v) edge crosses the cut so it will connect both sides. We know that (u, v) is a light edge, so it has the lowest weight. Therefore, T is optimal and an MST.

10.1 Prim's Algorithm

The idea of Prim's Algorithm is that we can create an MST by greedily adding the shortest edge. The algorithm goes like this. We start with a random vertex and then greedily add the edge to the vertex with the lowest edge weight. We then repeat until all the vertices are reached. The naive pseudocode is below

```
NAIVE_PRIM(G = (V,E), s):
    MST = {}
    visited = {s}
    while len(visited) < n:
        find the lightest edge (x,v) in E such that x in visited and v is not in visited

        MST.add((x,v))
        visited.add(v)

    return MST
```

The total runtime for naive Prim is $O(nm)$ since for each node n we visit all m edges each time to find the one with the lowest weight where one node has already been visited and the other not.

We can implement this with a faster runtime by keep track of the following in nodes that have not yet been reached by the growing tree:

- the distance from itself to the growing spanning tree using only one edge
- how to get to there (the closest neighbor that's reached by the tree already)

Therefore, each node starts with a value of ∞ which gets sequentially updated. Note that this is a lot like Dijkstra's! Pseudocode below:

```
PRIM(G = (V,E), s):
    MST = {}
    visited = {s}
    for all v besides s: d[v] = infty and k[v] = NULL
    for each neighbor v of s: d[v] = w(s,v) and k[v] = s

    while len(visited) < n:
        x = vertex with smallest d[v] value
        MST.add((K[x], x))
        for each unreached neighbor v of x:
            d[v] = min(w(x,v), d[v])
            if d[v] was updated:
                k[v] = x
```

```
        visited.add(x)

    return MST
```

Since this is very similar to Dijkstra, it actually have the same runtime. If implemented using a RB tree, the runtime is $O(mlogn)$. If implemented using a Fibonacci heap, the runtime is $O(m + nlogn)$.

Why does Prim's algorithm work? Well assume we have visited S nodes and S is in the MST T^*. Then we denote our cut that separates out the nodes which we have already visited vs. the nodes which we haven't visited yet. The next edge we add is the light edge on this cut. By our lemma, the MST contains S union the light edge.

10.2 Kruskal's Algorithm

In Kruskal's Algorithm, we start by denoting each vertex its own tree. We then add the edge with the lowest weight that combines two trees. We repeat until all the vertices are connected and we have our MST. The pseudocode is below:

```
KRUSKAL_NOT_VERY_DETAILED(G = (V,E)):
    E_SORTED = E sorted by weight in non-decreasing order
    MST = {}
    for v in V:
        put v in its own tree

    for (u,v) in E_SORTED:
        if u's tree and v's tree are not the same:
            MST.add((u,v))
            merge u's tree with v's tree
    return MST
```

To implement this above algorithm, we can use a data structure named Union-Find that supports 3 operations: MAKE SET, FIND, and UNION all in $O(1)$ time. With this Union-Find data structure, we can run Kruskal's Algorithm in $O(mlogn)$. If we look beyond comparison sorting and are allowed to use RadixSort, then we can run Kruskal's Algorithm in $O(m)$ time.

Why does Kruskal's Algorithm work? Well assume we have visited S nodes and S is in the MST T^*. The next edge we would add is the lowest weight edge that would connect two trees T_1 and T_2. Then we denote our cut as T_1 vs. all vertices including T_2 (we can also denote our cut as T_2 vs. all vertices including T_1 instead). Thus, the edge we choose is a light edge on this cut and by our lemma, the MST contains S union this light edge.

11 Min Cuts and Max Flows

Recall that a cut in a graph is defined as a partition of the vertices in a graph into two empty parts. A global minimum cut is a cut in this graph that has the fewest edges possible crossing it. Finding the global minimum cut in a graph has practical applications such as clustering and image segmentation. To find the global minimum cut, we can use Karger's Algorithm.

11.1 Karger's Algorithm

Karger's Algorithm is a randomized algorithm to find global minimum cuts in undirected graphs. It is a Monte Carlo randomized algorithm, which means that the runtime is guaranteed, but not necessarily the correctness. With Karger's Algorithm, the procedure is as follows:

1. Pick a random edge in the graph at uniform and contract it
2. To perform the edge contraction, we make those two nodes into a super-node while edges now become super-edges
3. Repeat until you only have 2 vertices left

Looking at this algorithm, we see that we obtain the correct global minimum cut if our random choice of edges does not select an edge that is part of the global minimum cut. What is the probability of this?
What we are trying to find is that after selecting $n-2$ edges (since we repeat until 2 nodes are left), that we do not select an edge that is part of the global minimum cut. The probability of success can be proved to be at least $\frac{1}{\binom{n}{2}}$. This seems low. However, what we can do it repeatedly run Karger's algorithm and pick the smallest cut in order to increase our success.

To succeed with probability $1-\delta$, it can be shown that we need to run Karger's algorithm $\binom{n}{2} ln(\frac{1}{\delta})$ times.

Each run of Karger's has a runtime of $O(n^2)$. To run it many times to obtain a higher probability of success, the total runtime is $O(n^4)$.

11.2 s-t Min Cuts and Max Flows

Now, assume we have a source node s and a sink node t. A minimum s–t cut is a cut which separates s from t with minimum cost. Note that this is different that Karger's because we have defined a particular sink and source node. Also, now, we are talking about directed and weighted graphs. Therefore, how we measure the cost of the cut is we sum up the weights for all the edges that cross the cut (that go from the s side to the t side). How do we find the minimum s-t cut? We need to first talk about Flows.

The value of a flow is the amount of stuff coming out of s (aka the amount of stuff flowing into t, due to flow conservation!) The flow needs to follow the following rules:

- Every edge has a flow (can be 0)
- Capacity constraint (the flow on every edge must be $\leq$ its capacity (the capacity is the weight for that edge)
- Flow Conservation Constraint (at each vertex, the incoming flows must equal the outgoing flows)

Therefore, the maximum flow can be found either by summing up all the flows coming out of s, or by summing up all the flows going into t.

There is a theorem called the Min-Cut/Max-Flow theorem that states that the value of the max-flow from s to t equals the value of the min-cut s-t cut.

Therefore to find the min-cut, we can just find the max-flow. To find the max-flow, we use an algorithm called the Ford-Fulkerson. The Ford-Fulkerson pseudocode is

```
FORD_FULKERSON(G, s, t):
    1. Start with arbitrary flow f (let's say flow of 0)
    2. Construct residual graph G_f
    3. Check if there's a path in G_f from s to t
        - if there is a path, update the flow f, and go back to step 2
        - if there isn't a path, then f is the max flow
```

So what is a residual graph? A residual graph is created when we are provided an opportunity graph (any graph of the flow) with certain flows. We take those flows and subtract them from the capacity of the original graph, and also add reversed edges of the capacity of the original graph. This is called the residual graph.

If we find a path from s to t in this residual graph, then we can add or "augment" this to the opportunity graph. If there is no way to "augment" this opportunity graph, then we have found the max-flow graph.

Note that not all augmenting path-finding procedures are created equal. It is possible to keep augmenting the opportunity graph is efficient ways, or slow and incremental ways. No matter which procedure we use, the algorithm will ultimately be correct, but the way in which we discover augmenting paths determines how efficient our algorithm is! Also, if the capacities of the input graph are all integers, then the value of any max flow is also an integer!

If we use BFS as our path-finding procedure, it can be shown that Ford-Fulkerson can be run in $O(nm^2)$ runtime.