1 Introduction

1.1 What is an algorithm?

Formally, an algorithm is just a procedure to turn one form of input into another one.

1.2 The champion problem

Given a set of n elements, how do we select the minimum or the maximum ones? The *naive* solution involves performing a linear scan on all the items in the array. However, for each item in the array, we would need to perform some amount of computation. Thus, the time it takes to solve the problem would scale as the size of the input scales up.

1.3 Selection Sort

Given a set of n elements, output the entire sequence in non-decreasing order. Using the champion problem, we see that that we can *reduce* the sorting problem to the champion problem. We pull the min number from the unsorted section of the array, and insert it into the sorted prefix array. Since we pull the minimum number from n, then $n-1$ elements, then we will pull the elements in a non-decreasing order.

During the first call to champion(i, n), we get the smallest value and swap it with the first item of a sorted array. Repeating this process eventually gives us the correct answer.

1.4 Insertion Sort

This is similar to the way we sort a deck of cards. For card i, we move from position i, until we find the previous item that is less than the number.

1.5 Asymptotic notation

When we talk about how many calculations some operation takes, we mean to say *how the number of calculations* scale with the size of the input. We can assign a formula for the amount of calculations, then we take the leading exponent, and call it the complexity. We classify it into several portions:

$$f(n) \leq C \times g(n) \forall n \geq n_0$$

Thus, if a function $f(n)$ is $O(n)$, then $f(n)$ varies from $g(n)$ by no less than a constant factor. This makes it easier to discuss the *time-complexity*.

1.6 Merge Sort

The main idea is that we have two different arrays, and we want to just compare item j and item i for every one in the two arrays. Using this way, we then recursively sort the two halves of each subarray until we hget all the base cases. This then gives us an algo with $O(n \log(n))$.

1.7 Recurrence Relations

We can describe the way how the instances realte to previoius instances of the problem. There are several methods: The substitution method, where we unroll the loop; the Recursion-tree method, where we guess an draw the recursion tree method; the final one is the Master Theorem.

2 Lower Bounds

2.1 Membership Query Problem

Basically, we want to know whether for a given array x with n elements, there exists an x and i such that $A[i] = x$. The naive solution involves a *linear scan*. If the array is sorted, then we can sort the array in $O\log(n)$ with *Binary Search*. Within the comparison-based model, the ordering between the elements can only be determined with comparisons.

2.2 More Asymptotic Notation

After we learned $O(n)$, we define $\Omega(n)$ and $\Theta(n)$, which act as a lower and upper bound, respectively.
After that, we introduce $o(n)$ and $\omega(n)$, which are defined as:

$$\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0$$

and

$$\lim_{x\to\infty} \frac{f(n)}{g(n)} = \infty$$

You want to remember: **if** $f(n) = o(g(n)),$ **then** $f(n) = O(g(n)),$ **but not-vice-versa**. Use L'Hopital's rule to show this is sometimes easier to do.
asymptotically optimal: When an algorithm runs in $\Omega(f(n))$ and $O(f(n))$, we know that the algorithm can't be improved by a *superconstant* factor.

2.3 Sorting Lower Bound

So far, we have a limit of $\Omega(n \log(n))$ for the comparison-based model, becuase both its upper bound and lower bound estimates are the same. If we can choose one of the $n!$ permutations, we shold then ask the evil adversary to remove a certain number of the possible permutations based on the ordering of the elements. We can show that $n \log(n)$ is asymptotically optimal.

3 Divide and Conquer

3.1 What exactly is D&C?

D&C involves recursion, or solving the same problem for a smaller instance of the problem. The idea is that to solve a problem of size n, we need the solutions to smaller instances of the problem and then somehow merge them together (i.e. the conquer step). In the mergesort this would be merging the two arrays after they have been sorted for lower instances of the problem.

3.2 Quick Sort

This algo chooses a *pivot* from the array (different ways of choosing it), then creates two arrays , S and L. In S we place all the elements less than k(our pivot), and in L the elements larger. We then recursively call the function on these two arrays.
What is the running time? It depends on the relative sizes of S and L. If either is always empty, then we will just be recursing on $(n-1)$ every time, which will lead to $O(n^2)$. If $|S| - |L| < 1$, then our recursion is *balanced* and thus reaches the desired $O(n \log(n))$ time. However, for a arbitrary ratio $|S| : |L|$, the worst-case is $(n, n-1, ..., 1)$, so $O(n^2)$

3.3 Randomized Quick Sort

Random selections of pivots usually result in balanced trees. Why? By the *Chernov Inquality*, we know that a bounded portion of the values will be a larger distance away from the mean.

3.4 Selection

AKA the $k-th$ order statistics, we seek to find the index of the k-th smallest number. We can split the arrays into S and L again, and if

4 Linear-Time Algorithms

4.1 Seelction

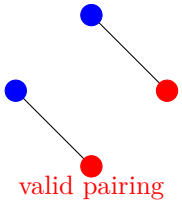
The naive way to find the min element is to do a linear scan. How to find the

order statistic in linear time? It depends on the selection of a good pivot.

5 Geometric Problems

5.1 Bichromatic Planar Matching

Given a set of n points on a plane, we want to find a pairing of points such that we can draw lines to all of them without them crossing.



Notice how neither line crosses when connecting to the other points. Generalizing to n points, no lines should intersect when trying to find one such pairing. A possible solution: assign an arbitrary arrangement and individually fix it.

5.2 Line Segment Intersection

To decide whether two line segments intersect, we need to check for the direction between two different line segments, whether we need a *clockwise rotation* or *counter-clockwise rotation*.

$$\begin{pmatrix} x(q)-x(a) & y(q)-y(a) \\ x(b)-x(a) & y(b)-y(a) \end{pmatrix} \quad (1)$$

Here, q is the endpoint of the vector we're measuring Θ from, a is the point where the two vectors intersect, and b is the other vector. If the result of the calculation is >0, it is in the right halfplane, <0 is in the left halfplane, and 0 means colinear.

5.3 Ham-Sandwich Cut

Given a set of blue points and red points on a plane, draw a straight line so that we can split the region into two containing equal amounts of red and blue points. Takes $O(n+m)$ time. With BMP, we can solve the HSC by joining together two points on opposite sides of the splitting line, or to each other if there is just one point on the area. This way we can get a pairing og red and blue points whose lines do not intersect in $O(n\log(n))$.

5.4 closest pair of points

Given S points, find the shortest distance among any of the two points. We have three scenarios: the shortest distance might lie on the left or right section, otherwise one of the endpoints might be across the side from the other. We then take the min of the closest point from the two sides, and to check for the center one, we calculate the distance δ from the dividing line. The closest pair can be at most 2δ away if it is to be the closest point.

5.5 Convex Hulls

The convex hull of a ste of points is the smallest polygon which can contains the given input set. We can know that the points with the largest and smallest x coordinates will be on the convex hull. Once we find those, we find the longest point L whose perpendicular distance to PQ is the longest. We do this for the line segment PQ and QP. We iterate until no new points can be found. This algorithm will run in $O(nh)$.

5.6 Farthest Pair

Given a set of points on the plane, output the longest distance between any two points. The two points must be on opposite ends of the convex hull. If they were both inside, then we could draw them out a bit farther and still be within the covnex hull. Therefore, the longest distance has to be held by two points who lie on the convex hull. This has a worst-case running time of $O(n^2)$, when all the points lie on the convex hull. How do we fix it?

5.7 antipodal pair

An *anitpodal pair* are two points such that two parallel lines between them can encompass all the points (basically two points on opposite ends of the graph). Thus they have to be on the convex hull and can be enumerated in $O(n)$..

6 Dynammic Programming

6.1 What is it?

For DP, the main idea is to compute the problems only once, and store the solutions. This way we can bring a problem from naive $O(2^n)$ down to polynomial time. *memoization* is just storing the solutions, so that when we *recurse*, we don't do the same problems over and over again.

6.2 Fibonacci Numbers

For fibs, the main recurrence is $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$. In the recursion tree, for each instance of the problem, we recurse on $n-1$ and $n-2$ until the base case, so for n elements, we recurse 2^n times. This problem becomes linear-time after a simple memoization.

6.3 Maximum Subarray

Given an n-length array, find the lower and upper bounds of the contiguous subarray with greatest sum. By naively trying all possible combinations until we find the best one, with a simple memoization we store the sum of

$$\sum_{i=1}^{j} A_i$$

Then, we can just compare the differences of A_i and $A_j, (i \leq j)$. With guessing the y, we can bring it down to $O(n)$, otherwise $O(n^2)$.

6.4 Rod-Cutting

Here, we're given an array of the values per inch that we can cut the rod, and our goal is to maximize the possible value. The main recurrence relation is for every i in $(0, ..., n-1)$, we take the max(currentMax, p[i]+cut(i-1)). After we calculate this for i, we store it in an array and check if we've calculated the subproblem every following time we recurse.

6.5 Longest Common Subsequence

Given two strings, we have to find the not-necessarily contiguous longest common subsequence of characters. We need to keep track of the current letter we're checking from the first string and second string. There are 3 possibilities:

- The longest common subsequence occurs in $(i-1, j)$ characters.
- " $(i, j-1)$
- If $s1[i] == s2[j]$, then we recurse on $(i-1, j-1)$, since i and j will both be part of the LCS.

To memoize, we can just keep a 2d array for the index i and j. Or, we can use a *bottom-up* approach and pre-emptively fill out an array. The time complexity is $O(nm)$, or the product of the lengths of the two strings. For a problem such as finding palindromic subsequences, we can reduce to LCS.

6.6 Longest Increasing Subsequence

Similar to the previous ones, here we find the longest subsequence of strictly increasing numbers in an array. A clever solution is to sort the original array S and find the LCS of S and the sorted version. This works because the numbers in the LCS won't change places in the array.However, this takes $O(n^2)$. A better solution is to mix binary search and DP. For every member of the LIS, we perform BS to find the smallest element that can fit in the array.

6.7 Integer Multiplication

To make the multiplication algorithm more efficient, we first attempt to recursively split the bit sequence into secitons, and multiply recursively those sections together. The karatsuba algorithm takes $O(n^{\log_2 3})$.

6.8 Matrix Multiplication

Again, Strassen's algorithms uses 7 recursive splits to solve the problem in $O(n^{\log_2 7})$

6.9 Matrix-Chain Multiplication

Given a sequence of n matrices and their descriptions, we need to find the minimum cost of multiplying them all together. Essentially, find the most efficient parenthesiztion. At every step, we need to check whether the current call to McMul()ether the current call to McMul(a,k)+McMul(k+1,b). Again, we can memoize this problem by storing the result of multiplying $A_i...A_j$.

7 Dynamic Sets

7.1 Binary Search Trees

BSTs all have the property that all the elements in the left subtree are all less than the current node x, and the elements in the right subtree are all greater than our current value.
When we traverse the tree, we can do the *pre-order*, *in-order*, and *post-order*, depending on the place where we print the root node.
The difference between the BSTs and the heaps depends on the cost of building and searching in them.

8 Greedy algorithms

When we talk about the greedy algorithms, we always choose the most convenient step at each iteration. However, this doesn't always lead to the optimal solution, like in DP.

8.1 Knapsack Problems

Given a set of weights and values for a set of n objects, we want the maximum sum of the elements. Again, we want to maximize the cost of including vs. not including the item in the ideal combination. so max(values[i] + knap(i-1, weight-weights[i]), knap(i, weight)).

9 Matroids

9.1 what?

A matroid is a structure that contains a finite colleciton of subsets S. A non-empty colleciton of subsets F is *hereditary*, that is , if $A \subseteq B$ and $B \subset F$, then $A \subset F$.
It also posseses the *exchange property*.

9.2 Weighted Matroid Problem

Given a matroid and a weight function, we want a set R in F whose weight is maximum among all the sets in F. (Many problems can be reduced to a weighted matroid problem).