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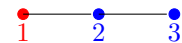
During the first call to champion(i, n), we get the smallest value and swap it with the first item of a sorted array. Repeating this process eventually gives us the correct answer.

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.

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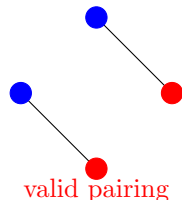
Notice how neither line crosses when connecting to the other points. Generalizing to n points, no lines should intersect when trying to find one such pairing. A possible solution: assign an arbitrary arrangement and individually fix it.

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 $O(n \log(n))$.