1 Introduction

1.1 What is an algorithm?

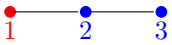
Formally, an algorithm is just a procedure to turn one form of input into another one.

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**.