1 Linear Time Selection (with Median as an Oracle)

We start with one more example of a divide-and-conquer algorithm. Consider the problem of finding the kth element in an array. That is, given an array $A[1...n]$, if we let $B = \text{SORT}(A)$ (in increasing order), we would like to find $B[k]$. This can be done by sorting the array, which takes $O(n \log n)$ time. We will see that we can do faster, via a rather intricate routine whose detail we will not cover.

**Lemma 1.1.** Given any list of $n$ numbers, we can find the median (defined as the $(n/2)^{th}$ element) in $O(n)$ time.

(the alternative, as in Section 2.4. of the textbook, is to use a randomized algorithm, but that’s also beyond the scope of what we’re covering).

Even though we did not go into the details of how such a median finder works, we can still invoke it as part of a modified binary search.

```plaintext
Select(A[1...n], k) {
    if(l > r) return NOT FOUND;
    m = MEDIAN(A[1...n]);
    let the number of times that m appears be m_freq
    Create A1 from all entries in A that are less than m.
    Create A2 from all entries in A that are greater than m.
    if(k <= A1.size()) {
        return Select(A1, k);
    } else if(k <= A1.size() + m_freq) {
```
return m (and corresponding index) as answer;
} else { // know k > A1.size() + m_freq, so we’re selecting from A2
    return Select(A2, k - A1.size() - m_freq);
}
}

Here the invocation of MEDIAN takes $O(n)$ time. Once we have $m$, we can also construct $A1$ and $A2$ in $O(n)$ time by sequentially walking through every element in $A$. Furthermore, we’re making one recursive call to a problem whose size is at most half as big. So the runtime recurrence becomes:

$$T(n) = 1 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil \right) + O(n),$$

which fits into Master theorem with $a = 1$, $b = 2$, and $d = 1$. This time though we have:

$$\log_b a = \log_2 1 = 0 < 1,$$

so we’re in the $d > \log_b a$ case of Master theorem, which means the total cost is $O(n^d) = O(n)$. Note that this is indeed faster than sorting the list.

## 2 Recursive Inversion Counting

Recall the inversion counting problem. Given array $A[1 \ldots n]$ of distinct numbers, finds the number of pairs $i < j$ such that $A[i] > Arr[j]$. We start with the observation the number of inversions that involves $j$ in some range $[L, R]$ and some $i < L$ is the number of elements in $A[L \ldots R]$ that are less than it.

This means if by sorting $A[L, R]$, and binary searching on it, we can compute the number of inversions involving any $i < L$ in $O(\log n)$ time. In other words, given some value $mid$, we can count the number of inversions with

$$i \leq m < j$$

in $O(n \log n)$ time. The gains here are from the sorting of $Arr[m + 1 \ldots n]$ being used on each query for $i \leq m$. In contrast, the naive algorithm of using a linear sweep to find the rank of $A[i]$ in $A[i + 1 \ldots n]$ does linear work for each of the $n$ elements.

Going back to the example from before, we split the array into

$$A[1 \ldots 4] = 8, 7, 12, 3; \quad A[5 \ldots 8] = 5, 11, 2, 9,$$


Then we can recurse in the usual way onto $A[l, m]$ and $A[m+1, r]$. This leads to the runtime recurrence of:

$$T(n) = 2T(n/2) + O(n \log n).$$

2
This doesn’t totally fit the form of master theorem: the non-recursive overhead has an extra factor of \(\log n\). However, we note that \(\log n\) is largest on the top level of the recursion, so we can treat it as an extra factor to be multiplied in at the end. This means we can first solve the recurrence:

\[
\hat{T}(n) = 2\hat{T}(n/2) + O(n),
\]

and then infer \(T(n) \leq \hat{T}(n)\).

For \(\hat{T}\), we have \(a = b = 2\), and \(d = 1\). This is the case with \(d = \log_b a\), so we get the result is \(\hat{T}(n) \leq O(n^d \log n) \leq O(n \log n)\). Putting the extra factor of \(\log n\) back then gives a total of \(T(n) \leq O(n \log^2 n)\).