Chapter 3

Master Theorem and Linear Time Sorts.

3.1 Master Theorem, one Theorem to bound them all.

The master theorem is the result of the recursive expansion. It handles recursive functions at the form of $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, for positive function $f : \mathbb{N} \to \mathbb{R}^+$.

Master Theorem, simple version.

First, Consider the case that $f = n^c$. Let $a \geq 1, b > 1$ and $c \geq 0$. then:

1. if $\frac{a}{b^c} < 1$ then $T(n) = \Theta(n^c)$ **(f wins)**.
2. if $\frac{a}{b^c} = 1$ then $T(n) = \Theta(n^c \log_b(n))$.
3. if $\frac{a}{b^c} > 1$ then $T(n) = \Theta\left(n^{\log_b(a)}\right)$ **(f loose)**.

Example 3.1.1.

1. $T(n) = 4T(\frac{n}{2}) + d \cdot n \Rightarrow T(n) = \Theta(n^2)$ according to case (3).
2. $T(n) = 3T(\frac{n}{2}) + d \cdot n \Rightarrow T(n) = \Theta(n^{\log_2(3)})$ also due to case (3).

Master Theorem, strong version.

Now, let's generalize the simple version for arbitrary positive f and let $a \geq 1$, $b > 1$.

1. if $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$ for some $\varepsilon > 0$ then $T(n) = \theta\left(n^{\log_b(a)}\right)$ **(f loose)**.
2. if $f(n) = \Theta\left(n^{\log_b(a)}\right)$ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$
3. if there exist $\varepsilon > 0, c < 1$ and $n_0 \in \mathbb{N}$ such that $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ and for every $n > n_0$ $a \cdot f\left(\frac{n}{b}\right) \leq cf(n)$ then $T(n) = \theta(f(n))$ **(f wins)**.

Example 3.1.2.

1. $T(n) = T(\frac{2n}{3}) + 1 arrow f(n) = 1 = \Theta(n^{\log_{\frac{3}{2}}(1)})$ matches the second case. i.e $T(n) = \Theta(n^{\log_{\frac{3}{2}}(1)} \log n)$.

2. $T(n) = 3T\left(\frac{n}{4}\right) + n \log narrow f(n) = \Omega(n^{\log_4(3)+\varepsilon})$ and notice that $af(\frac{n}{b}) = \frac{3n}{4}\log(\frac{n}{4})$. Thus, it's matching the third case. $\Rightarrow T(n) = \Theta(n \log n)$.

3. $T(n) = 3T(n^{\frac{1}{3}}) + \log\log n$. Let $m = \log n \Rightarrow T(n) = T(2^m) = 3T(2^{\frac{m}{3}}) + \log m$. Denote by $S = S(m) = T(2^m) arrow S(m) = 3T(2^{\frac{m}{3}}) + \log m = 3S(\frac{m}{3}) + \log m$. And by the fact that $\log m = O(m^{\log_3(3)-\varepsilon}) arrow T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log(n))$.

3.2 Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer with a size at most k. It runs in $\Theta(n + k)$ time, so when $k = O(n)$, counting sort runs in $\Theta(n)$ time. It first determines, for each input element x, the number of elements less than or equal to x. Then, it uses this information to place element x directly into its position in the output array. For example, if there are 17 elements less than or equal to x, then x will be placed in position 17. However, we need to make some adjustments to this method to handle cases where multiple elements have the **same value**. We do not want all of them to end up in the same position.

```
1  let B and C be new arrays at size n and k
2  for i ∈ [0, k] do
3  |  C_i ← 0
4  end
5  for j ← [1, n] do
6  |  C_{A_j} ← C_{A_j} + 1
7  end
8  for i ∈ [1, k] do
9  |  C_i ← C_i + C_{i−1}
10 end
11 for j ∈ [n, 1] do
12 |  B_{C_{A_j}} ← A_j
13 |  C_{A_j} ← C_{Aj}−1   // to handle duplicate values
14 end
15 return B
```

Algorithm 1: Counting-sort(A, n, k)

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

Note 1: That introduction was copied word by word from a web source. Do not use for commercial purposes.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
1 for i ∈ [1, d] do
2 |   use a stable sort to sort array A on digit i
3 end
```

Algorithm 2: radix-sort(A, n, d)

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.
- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.
- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:
  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n + k)\right)$.

Note 2: We will return to analyze the expected (average) running time after the lecture on probability.

Bucket sort. Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

```
1  let B[0 : n − 1] be a new array
2  for i ← [0, n–1] do
3  |  make B_i an empty list
4  end
5  for i ← [1, n] do
6  |  insert A_i into list B_⌊nA_i⌋]
7  end
8  for i ← [0, n–1] do
9  |  sort list B_i
10 end
11 concatenate the lists B_0, B_1, .., B_{n–1} together and
12 return the concatenated lists
```

Algorithm 3: bucket-sort(A, n)

3.3 Sorting in Comparison Model.

We have learned in the lecture that the runs of any sorting algorithm, which does not assume anything about the input's structure except the ability to compare elements in pairs, can be modeled as a binary tree. At each node, the algorithm compares two elements and, based on the result, moves to either the left or right child. Eventually, we reach the leaf of the tree and output the sorted elements. Notice that the nodes only exist in our imagination; the algorithm is not aware of their existence. We will name that tree the comparison tree.

The height of the comparison tree is (at least) the running time of the algorithm. We are going to demonstrate how to use these ideas to lower-bound the running time of the find-peak problem in the comparison model.

Example 3.3.1. Prove that no algorithm can find a peak in less than $\Theta\left(\log n\right)$ time.

Proof. Consider the following inputs for the problem, A^j will be a tringle that get's is point at position j. Namely:

$$A_i^j = \begin{cases} i & i < j \\ j + j - i & \text{else} \end{cases}$$

Now, let's assume, in contradiction, that there exists an algorithm $\mathcal{A}$ in the comparison model that runs in less than $\log n$ time. This implies that the comparison tree representation of its running has a height less than $\log n$ and the number of its leaves, each associated with a possible output, is less than n. Remember that we have n inputs of the form A^j.

By applying the pigeonhole principle, we can conclude that there are distinct j and j' such that $\mathcal{A}$ reaches the same leaf when given A^j and $A^{j'}$ as inputs. Since their peaks are set at different positions, it follows that the algorithm will output a wrong answer for at least one of them. □

Example 3.3.2. Prove that no algorithm, in the comparison model, can merge two sorted arrays in time $\Theta(n^{1-\varepsilon})$ for some $\varepsilon > 0$.

Proof. Assuming, for contradiction, that there exists an algorithm $\mathcal{A}$ in the comparison model that merges two given sorted arrays in time $\Theta\left(n^{1-\varepsilon}\right)$. Now, consider the sorting algorithm obtained by replacing the merge subroutine in merge-sort with $\mathcal{A}$. On one hand, our new algorithm is a sorting algorithm in the comparison model (otherwise it would contradict the correctness of $\mathcal{A}$), while on the other hand, its running time is equal to:

$$T(n) = n^{1-\varepsilon} + 2T(n/2) \Rightarrow T(n) = \Theta(n)$$

This contradicts our $\Omega(n \log n)$ lower bound for sorting in the comparison model. □