CS 3510 — Algorithms, Fall 2022

# Lecture 4: Master Theorem

January 20, 2022

*Lecturer: Frederic Faulkner* *Scribe: Aditya Diwakar*

Same announcement as previous lecture regarding homework. Homework 1 due Saturday. Homework 2 released Saturday. Also, join Piazza.

## 1 Master Theorem

Most of us can agree that expanding a recurrence relationship is painful, and very easy to make a mistake while simplifying. The Master Theorem is a powerful theorem that lets us find the running time of any recurrence relation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

In fact, based on $a, b, d$, we can find $T(n)$ as:

$$T(n) = \begin{cases} O\left(n^d\right) & a < b^d \\ O\left(n^d \log n\right) & a = b^d \\ O\left(n^{\log_b a}\right) & a > b^d \end{cases}$$

The proof of this theorem is not reproduced here, but can be easily found online. Essentially, the proof follows by manually expanding the generic recurrence from above. The geometric sum rules from the previous lecture helps here.

### 1.1 Example

Our naive slow multiplication method can be solved using this theorem:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n) \implies a = 4, b = 2, d = 1$$

and hence (by the Master Theorem), $4 > 2^1$ so $T(n) = O(n^{\log_2 4}) = O(n^2)$.

Hypothetically, let's say instead the recurrence was:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n^2) \implies a = 4, b = 2, d = 2$$

and hence (by the Master Theorem), $4 = 2^2$ so $T(n) = O(n^d \log n) = O(n^2 \log n)$

# 2 Divide and Conquer

We've briefly mentioned on what divide and conquer is, but let's talk about it some more. Divide and Conquer is a general algorithm strategy where if we can break a question into smaller pieces, and combine the solutions to those smaller problems.

## 2.1 MergeSort

An alteration of this sort of this problem is something where we are performing some action on a list, such as what is done by MergeSort.

1. Split list recursively
2. When list contains $\leq 2$ elements, a base case is reached and sort this
3. Reconstruct by performing a `merge` operation on the sub-solutions

How does the `merge` operation work?

1. Create a pointer on the first element of each list
2. Pick the smaller of the two pointed elements, move the relevant pointer
3. Continue this process until no more elements in one list
4. Add the rest of the elements from the other list

The recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ where $a = 2, b = 2, d = 1$ implying that $T(n) = O(n \log n)$.

```
1  Function MergeSort(L):
       /* split L into two halves                                   */
2      RetL ← [] L_a ← L[: |L|/2]
3      L_b ← L[|L|/2 :]
       /* MergeSort two halves                                      */
4      ML_a ← MergeSort(L_a)
5      ML_b ← MergeSort(L_b)
       /* Merge sorted halves                                       */
6      Merge(L_a, L_a, RetL)
7      return RetL
8  Function Merge(L_1, L_2, L):
       /* append the rest of other list if one is empty             */
9      if |L_1| = 0 ∨ |L_2| = 0 then
10         return L + L_1 + L_2
       /* pick smaller element; recursively move pointer            */
11     if L_1[1] ≤ L_2[1] then
12         L.append(L_1[1])
13         Merge(L_1[1 :], L_2, L)
14     else
15         L.append(L_2[1])
16         Merge(L_1, L_2[1 :], L)
```

## 2.2 Median

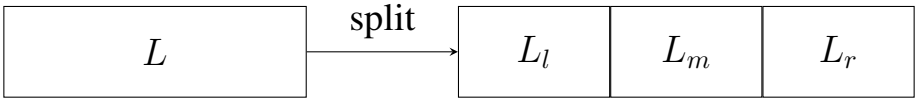
Another form of this question is `Median`. This type of algorithm attempts to find the median (middle) element of an $n$ list. A naive way is to sort the list and return the $n/2$ element. Since sorting has a runtime of $O(n \log n)$, then this method also has that same runtime.

However, instead, we choose to generalize. Instead of solving for only the median, we can make a new algorithm: `QuickSelect` which gives the $k^{th}$ smallest element from a list $L$. This will work for the median as we can set $k = |L|/2$.

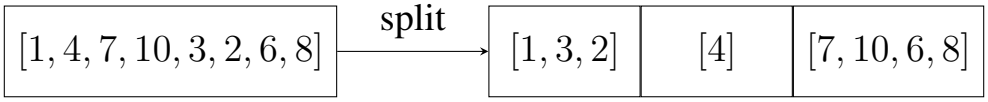
## 2.3 Quick Select

The objective of `QuickSelect` is to pick the $kth$ smallest element from a list $L$. Hence, it is parameterized as: `QuickSelect(L, k)`.

The first step of QuickSelect is to pick a random pivot $p$ from the elements of $L$. This takes $O(1)$. Then, we perform a linear search through $L$ partitioning $L$ into:




where $L_l$ are all elements in $L$ less than $p$, $L_r$ are all elements in $L$ greater than $P$, and $L_m$ are all the elements equal to $p$. For example, $L = [1, 4, 7, 10, 3, 2, 6, 8]$ and $k = 5$ and randomly let $p = 4$, then we split as so:




Since $k = 5$, then we know that it can't be in $L_l$ or $L_m$ because $|L_l| + |L_m| = 4$. Also, because of our pivot, we know that both $L_l$ and $L_m$ are $\leq 4$. Hence, we know our answer lives in $L_r$. When recursing into $L_r$, we can't use $k = 5$ (obviously: $|L_r| < 5$), but instead we choose $k - |L_l| - |L_m|$. In this example, that yields $1$ meaning we look for the smallest element in $L_r$.

Another rule: if the smallest element lies in $L_m$, then we don't need to recurse as $L_m = \{x \mid x \in L \wedge x = p\}$ so all elements in $L_m$ are equal to the same element and we can simply return the pivot.

```
1  Function QuickSelect(L, k):
       /* get random pivot                                          */
2      p ← random pivot ∈ L
3      L_l ← L[0 : |L|/3]
4      L_m ← L[|L|/3 : 2|L|/3]
5      L_r ← L[2|L|/3 : |L|]
       /* three cases of where kth smallest lives                   */
6      if k ≤ |L_1| then
7          return QuickSelect(L_l, k)
8      else if |L_l| < k ≤ |L_l| + |L_m| then
           /* simply return pivot as L_m only contains p            */
9          return p
10     else if |L_l| + |L_m| < k then
11         return QuickSelect(L_r, k − |L_l| − |L_m|)
```

Clearly, each *level* of this function call takes $O(n)$ time as the partitioning requires a linear scan to compare based on the pivot (a total of $n-1$ comparisions). However, what isn't clear is the number of recursions and the size of that recursion.

Since the pivot is picked randomly, it complicates the runtime calculation. However, we know if our pivot is between the $25^{th}$ and $75^{th}$ percentile, then we can reduce the size of the list to $3/4n$.

Further, $P\,(\text{picking good pivot}) = P\,(\text{picking pivot in middle } 50\%) = \frac{1}{2}$ meaning we expect to take 2 iterations before getting a good pivot.

Hence, the runtime can be computed as:

$$T(n) = T\left(\frac{3n}{4}\right) + \underbrace{O(n) + O(n)}_{\text{expected 2 tries to get good pivot}}$$

However, $2O(n) = O(n)$ giving us a final runtime of:

$$T(n) = T\left(\frac{3n}{4}\right) + O(n)$$

which is equal to $O(n)$ (by Master Theorem: $a = 1, b = 4/3, d = 1$).