Recursive Analysis - Recitation 3

Master theorem and recursive trees.

November 5, 2022

Abstract

One of the standard methods to analyze the running time of algorithms is to express recursively the number of operations that are made. In the following recitation, we will review the techniques to handle such formulation (solve or bound).

1 Bounding recursive functions by hands.

Our primary tool to handle recursive relation is the Master Theorem, which was proved in the lecture. As we would like to have a more solid grasp, let's return on the calculation in the proof over a specific case. Assume that your algorithm analysis has brought the following recursive relation:

1. **Example.** $T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + c \cdot n & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$. Thus, the running time is given by

$$T(n) = 4T\left(\frac{n}{2}\right) + c \cdot n = 4 \cdot 4T\left(\frac{n}{4}\right) + 4c \cdot \frac{n}{2} + c \cdot n = ... =$$
$$\overbrace{4^h T(1)}^{\text{critical}} + c \cdot n \left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 ... + \left(\frac{4}{2}\right)^{h-1}\right) = 4^h + c \cdot n \cdot \frac{2^h - 1}{2 - 1}$$

   We will call the number of iteration till the stopping condition the recursion height, and we will denote it by h . What should be the recursion height? $2^h = n \Rightarrow h = \log(n)$. So in total we get that the algorithm running time equals $\Theta\left(n^2\right)$.

   Question, Why is the term $4^h T(1)$ so critical? Consider the case $T(n) = 4T\left(\frac{n}{2}\right) + c$ .One popular mistake is to forget the final term, which yields a linear solution $\Theta(n)$ (instead of quadric $\Theta(n^2)$).

2. **Example.** $T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + c \cdot n & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$, and then the expanding yields:

$$T(n) = 3T\left(\frac{n}{2}\right) + c \cdot n = 3^2 T\left(\frac{n}{2^2}\right) + \frac{3}{2}cn + c \cdot n = \overbrace{3^h T(1)}^{\text{critical}} + cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + ... + \left(\frac{3}{2}\right)^{h-1}\right)$$
$$h = \log_2(n) \Rightarrow T(n) = 3^h T(1) + c \cdot n \cdot \left(\left(\frac{3}{2}\right)^{\log_2 n}\right) / \left(\frac{3}{2} - 1\right) = \theta\left(3^{\log_2(n)}\right) = \theta\left(n^{\log 3}\right)$$

   where $n^{\log 3} \sim n^{1.58} < n^2$.

2 Master Theorem, one Theorem to bound them all.

As you might already notice, the same pattern has been used to bound both algorithms. The master theorem is the result of the recursive expansion. it classifies 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. $T(n) = 4T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^2\right)$ according to case (3). And $T(n) = 3T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^{\log_2(3)}\right)$ 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)**.

Examples

1. $T(n) = T\left(\frac{2n}{3}\right)+1 \to f(n) = 1 = \Theta\left(n^{\log_{\frac{3}{2}}(1)}\right)$ matches the second case. i.e $T(n) = \Theta\left(n^{\log_{\frac{3}{2}}(1)} \log n\right)$.
2. $T(n) = 3T\left(\frac{n}{4}\right) + n \log n \to f(n) = \Omega\left(n^{\log_4(3)+\varepsilon}\right)$ and notice that $f\left(a\frac{n}{b}\right) = \frac{3n}{4} \log\left(\frac{3n}{4}\right)$. Thus, it's matching to the third case. $\Rightarrow T(n) = \Theta(n \log n)$.
3. $T(n) = 3T\left(n^{\frac{1}{3}}\right) + \log\log n$. let $m = \log n \Rightarrow T(n) = T(2^m) = 3T\left(2^{\frac{m}{3}}\right) + \log m$. denote by $S = S(m) = T(2^m) \to S(m) = 3T\left(2^{\frac{m}{3}}\right) + \log m = 3S\left(\frac{m}{3}\right) + \log m$. And by the fact that $\log m = O\left(m^{\log_3(3)-\varepsilon}\right) \to T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log(n))$.

3 Recursive trees.

There are still cases which aren't treated by the *Master Theorem*. For example consider the function $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$. Note, that $f = \Omega\left(n^{\log_b(a)}\right) = \Omega(n)$. Yet for every $\varepsilon > 0 \Rightarrow f = n \log n = O\left(n^{1+\varepsilon}\right)$ therefore the third case doesn't hold. How can such cases still be analyzed?

Recursive trees Recipe

1. draw the computation tree, and calculate it's height. in our case, $h = \log n$.
2. calculate the work which done over node at the k-th level, and the number of nodes in each level. in our case, there are 2^k nodes and over each we perform $f(n) = \frac{n}{2^k} \log\left(\frac{n}{2^k}\right)$ operations.
3. sum up the work of the k-th level.
4. finally, the total time is the summation over all the $k \in [h]$ levels.

applying the above, yields

$$T(n) = \sum_{k=1}^{\log n} n \cdot \log\left(\frac{n}{2^k}\right) = n \sum_{k=1}^{\log n} \left(\log n - \log 2^k\right) = n \sum_{k=1}^{\log n} (\log n - k) =$$
$$= \Theta\left(n \log^2(n)\right)$$

za Example. Consider merge sort variation such that instead of splitting the array into two equals parts it's split them into different size arrays. The first one contains $\frac{n}{10}$ elements while second contains the others $\frac{9n}{10}$ elements.

non-equal-merge alg.

```
Result: returns the sorted permutation of x_1...x_n ∈ R^n
1
2  if n ≤ 10 then
3  |  return bubble-sort (x_1...x_n)
4  end
5
6  else
7  |  define S_l ← x_1...x_{n/10 - 2}, x_{n/10 - 1}
8  |  define S_r ← x_{n/10}, x_{n/10 + 1}..., x_n
9  |
10 |  R_l ← non-equal-merge(S_l)
11 |  R_r ← non-equal-merge(S_r)
12 |
13 |  return Merge(R_l, R_r)
14 end
```

Note, that the master theorem achieves an upper bound,

$$T(n) = n + T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) \leq n + 2T\left(\frac{9n}{10}\right) \Rightarrow T(n) = O\left(n^{\log_{\frac{10}{9}}(2)}\right) \sim O\left(n^6\right)$$

Yet, that bound is far from been tight. Let's try to count the operations for each node. Let's try another direction.

Claim. Let n_i be the size of the subset which is processed at the i-th node. Then for every k:

$$\sum_{i \in \text{k level}} n_i \leq n$$

Proof. Assuming otherwise implies that there exist index j such that x_j appear in at least two different nodes in the same level, denote them by u, v. As they both are in the same level, non of them can be ancestor of the other. denote by $m \in \mathbb{N}$ the input size of the sub array which is processed by the the lowest common ancestor of u and v, and by $j' \in [m]$ the position of x_j in that sub array. By the definition of the algorithm it steams that $j' < \frac{m}{10}$ and $j' \geq \frac{m}{10}$. contradiction.

The height of the tree is bounded by $\log_{\frac{9}{10}}(n)$. Therefore the total work equals $\Theta(n \log n)$.

Thus, the total running time equals to:

$$T(n) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} f(n_i) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} n_i \leq n \log n$$

4 Appendix. Recursive Functions In Computer Science. (Optional)

[**COMMENT**] The current section repeats over part of the content above as it was designed to be self-contained. Also, notice that this part is considered as optional material and you are not required to remember the following algorithms for the final exam. Its primary goal is to expose you to "strange" running times.

Leading Example. numbers multiplication Let x, y be an n'th digits numbers over $\mathbb{F}_2^n$. It's known that summing such a pair requires a linear number of operations. Write an algorithm that calculates the multiplication $x \cdot y$.

Example. Long multiplication. To understand the real power of the dividing and conquer method, let's first examine the known solution from elementary school. In that technics, we calculate the power order and the value of the digit separately and sum up the results at the end. Formally: $x \leftarrow \sum_{i=0}^{n} x_i 2^i$ Thus,

$$x \cdot y = \left(\sum_{i=0}^{n} x_i 2^i\right)\left(\sum_{i=0}^{n} y_i 2^i\right) = \sum_{i,j \in [n]\times[n]} x_i y_j 2^{i+j}$$

the above is a sum up over n^2 numbers, each at length n and therefore the total running time of the algorithm is bounded by $\theta(n^3)$. [**COMMENT**] notice that adding 1 to 1111111111...1 requires $O(n)$.

Example. Recursive Approach. We could split x into the pair x_l, x_r such that $x = x_l + 2^{\frac{n}{2}} x_r$. Then the multiplication of two n-long numbers will be reduced to sum up over multiplication of a quartet. Each at length $\frac{n}{2}$. Thus, the running time is given by

$$\begin{aligned} x \cdot y &= \left(x_l + 2^{\frac{n}{2}} x_r\right)\left(y_l + 2^{\frac{n}{2}} y_r\right) = x_l y_l + 2^{\frac{n}{2}}\left(x_l y_r + x_r y_l\right) + 2^n x_r y_r \\ &\Rightarrow T(n) = 4T\left(\frac{n}{2}\right) + c \cdot n = 4 \cdot 4T\left(\frac{n}{4}\right) + 4c \cdot \frac{n}{2} + c \cdot n = ... = \\ &c \cdot n\left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 ... + \left(\frac{4}{2}\right)^{h-1}\right) + 4^h T(1) = n^2 + c \cdot n \cdot \frac{2^h - 1}{2 - 1} \end{aligned}$$

We will call the number of iteration till the stopping condition the recursion height, and we will denote it by h . What should be the recursion height? $2^h = n \Rightarrow h = \log(n)$. So in total we get that multiplication could be achieved by performs $\Theta\left(n^2\right)$ operations.

Example. Karatsuba algorithm. Many years it's was believed that multiplication can't done by less then $\Omega\left(n^2\right)$ time; until Karatsuba found the following algorithm. denote by

$$z_0, z_1, z_2 \leftarrow x_l y_r, x_l y_r + x_r y_l, x_r y_r$$

Notice that $z_1 = (x_l + x_r)(y_l + y_r) - z_0 - z_1$. summarize the above yields the following pseudo code.

Karatsuba alg.

Result: returns the multiplication $x \cdot y$ where $x, y \in \mathbb{F}_2^n$

if $x, y \in \mathbb{F}_2$ **then**
  return $x \cdot y$
end

else
  define $x_l, x_r \leftarrow x$ and $y_l, y_r \leftarrow x$ // $O(n)$.

  calculate $z_0 \leftarrow \text{Karatsuba}(x_l, y_l)$
    $z_2 \leftarrow \text{Karatsuba}(x_r, y_r)$
    $z_1 \leftarrow \text{Karatsuba}(x_r + x_l, y_l + y_r) - z_0 - z_2$

  return $z_0 + 2^{\frac{n}{2}} z_1 + 2^n z_2$ // $O(n)$.
end

Let's analyze the running time of the algorithm above, assume that $n = 2^m$ and then the recursive relation is

$$T(n) = 3T\left(\frac{n}{2}\right) + c \cdot n = 3^2 T\left(\frac{n}{2^2}\right) + \frac{3}{2}cn + c \cdot n = cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + ... + \left(\frac{3}{2}\right)^{h-1}\right) +) + 3^h T(1)$$

$$h = \log_2(n) \Rightarrow T(n) = n^{\log_2 3} + c \cdot n \cdot \left(\left(\frac{3}{2}\right)^{\log_2 n}\right) / \left(\frac{3}{2} - 1\right) = \theta\left(3^{\log_2(n)}\right) = \theta\left(n^{\log 3}\right)$$

where $n^{\log 3} \sim n^{1.58} < n^2$.

Heaps - Recitation 4 Correctness, Loop Invariant And Heaps.

Apart from quantifying the resource requirement of our algorithms, we are also interested in proving that they indeed work. In this Recitation, we will demonstrate how to prove correctness via the notation of loop invariant. In addition, we will present the first (non-trivial) data structure in the course and prove that it allows us to compute the maximum efficiently.

Correctness And Loop Invariant.

In this course, any algorithm is defined relative to a task/problem/function, And it will be said correctly if, for any input, it computes desirable output. For example, suppose that our task is to extract the maximum element from a given array. So the input space is all the arrays of numbers, and proving that a given algorithm is correct requires proving that the algorithm's output is the maximal number for an arbitrary array. Formally:

Correctness.

We will say that an algorithm $\mathcal{A}$ (an ordered set of operations) computes $f : D_1 \to D_2$ if for every $x \in D_1 \Rightarrow f(x) = \mathcal{A}(x)$. Sometimes when it's obvious what is the goal function f, we will abbreviate and say that $\mathcal{A}$ is correct.

Other functions f might be including any computation task: file saving, summing numbers, posting a message in the forum, etc. Let's dive back into the maximum extraction problem and see how correctness should be proven in practice.

Task: Maximum Finding. *Given $n \in \mathbb{N}$ numbers $a_1, a_2, \cdots a_n \in \mathbb{R}$ write an Algorithm that returns their maximum.*

Consider the following suggestion. How would you prove it correct?

Maximum finding.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 let b ← a_1
2 for i ∈ [2, n] do
3     b ← max(b, a_i)
4 return b
```

Usually, it will be convenient to divide the algorithms into subsections and then characterize and prove their correctness separately. One primary technique is using the notation of Loop Invariant. Loop Invariant is a property that is characteristic of a loop segment code and satisfies the following conditions:

Loop Invariant.

1. Initialization. The property holds (even) before the first iteration of the loop.
2. Conservation. As long as one performs the loop iterations, the property still holds.
3. (optional) Termination. Exiting from the loop carrying information.

What is the Loop Invariant here? *"at the i-th iteration,* $b = \max\{a_1 ... a_{i-1}\}$*"*. The proof is almost identical to the naive case.

Claim. Consider the while loop. The property: *"for every* $j' < j \leq n+1 \Rightarrow a_{j'} \leq a_i$*"* is a loop invariant that is associated with it.

Proof: first, the initialization condition holds, as the at the first iteration $j = 1$ and therefore the property is trivial. Assume by induction, that for every $m < j$ the property is correct, and consider the j-th iteration. If back again to line (5), then it means that $(j-1) < n$ and $a_{j-1} \leq a_i$. Combining the above with the induction assumption yields that $a_i \geq a_{j-1}, a_{j-2}, ... a_1$.

Correctness Proof. Split into cases, First if the algorithm return result at line (9), then due to the loop invariant, combining the fact that $j = n+1$, it holds that for every $j' \leq n < j \Rightarrow a_i \geq a_{j'}$ i.e a_i is the maximum of $a_1, ... a_n$. The second case, in which the algorithm returns Δ at line number (10) contradicts the fact that n is finite, and left as an exercise. the running time is $O(n^2)$ and the space consumption is $O(n)$.

Task: Element finding. *Given $n \in \mathbb{N}$ numbers $a_1, a_2, \cdots a_n \in \mathbb{R}$ and additional number $x \in \mathbb{R}$ write an Algorithm that returns i s.t $a_i = x$ if there exists such i and* False *otherwise.*

Element finding.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 for i ∈ [n] do
2     if a_i = x then
3         return i, a_i
4 return Δ
```

Task: The Superpharm Problem. *You are requested to maintain a pharmacy line. In each turn, you get one of the following queries, either a new customer enters the shop, or the pharmacist requests the next person to stand in front. In addition, different customers have different priorities, So you are asked to guarantee that in each turn, the person with the height priority will be at the front.*

Before we consider a sophisticated solution, What is the running time for the naive solution?(maintaining the line as a linear array) ($\sim O\left(n^2\right)$).

Heaps.

Heaps are structures that enable computing the maximum efficiency in addition to supporting adding and removing elements.

We have seen in the Lecture that no Algorithm can compute the max function with less than $n-1$ comparisons. So our solution above is indeed the best we could expect for. The same is true for the search problem. Yet, we saw that if we are interested in storing the numbers, then, by keeping them according to sorted order, we could compute each query in logarithmic time via binary search. That raises the question, is it possible to have a similar result regarding the max problem?

Heap

Let $n \in \mathbb{N}$ and consider the sequence $H = H_1, H_2 \cdots H_n \in \mathbb{R}(*)$. we will say that H is a Heap if for every $i \in [n]$ we have that: $H_i \leq H_{2i}, H_{2i+1}$ when we think of the value at indices greater than n as $H_{i>n} = -\infty$.

Checking vital signs. Are the following sequences are heaps?

1. 1,2,3,4,5,6,7,8,9,10 (Y)
2. 1,1,1,1,1,1,1,1,1,1 (Y)
3. 1,4,3,2,7,8,9,10 (N)
4. 1,4,2,5,6,3 (Y)

How mach is cost (running time) to compute the the min of H? (without change the heap). ($O(1)$). Assume that option 4 is our Superpharm Line, let's try to imagine how should we maintain the line. After serving the customer at top, what can be said on $\{H_2, H_3\}$? or $\{H_{i>3}\}$? (the second heighset value is in $\{H_2, H_3\}$.)

Subtask: Extracting Heap's Minimum. *Let H be an Heap at size n, Write algorithm which return H_1, erase it and returns H', an Heap which contain all the remain elements.* **Solution:**

Heappop.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 ret ← H_1
2 H_1 ← ∞
3 Heapify-down(1)
4 return ret
```

Heapify-down.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 next ← i
2 if 2i < n and H_next ≤ H_2i then
3     next ← 2i
4 if 2i + 1 < n and H_next ≤ H_2i+1 then
5     next ← 2i + 1
6 if i ≠ next then
7     H_i ↔ H_next
8     Heapify-down(next)
```

Claim. Assume that H satisfies the Heap inequality for all the elements except H_j. Namely for any $i \neq j$ we have that $H_i \leq H_{2i}, H_{2i+1}$. Then applying Heapify-down on H at index j returns an heap.

Proof. Induction on $n - j$. Base case, $n - j < n/2 \Rightarrow j > n/2$, Hence $2j, 2j + 1 \notin [n]$ and we have that the Heap property holds for any i.

Assume the correctness of the claim for any j' satisfy $j' > j$ and consider j. Denote by H' the state of the heap after the swapping at line number 7 and let $i \in [n]$. First Notice that if $j \neq 2i, 2i+1$ and $i \neq 2j, 2j+1$ then $H'_i = H_i \leq H_{2i} = H'_{2i}$ (and in similar to $2i + 1$). So it left to consider the case where $i = \lfloor j/2 \rfloor$ and $i = j$.

Insertion. blabla

Heapify-up.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 parent ← ⌊i/2⌋
2 if parent > 0 and H_perent ≤ H_i then
3     H_i ↔ H_perent
4     Heapify-up(perent)
```

Heappush.

Result: returns the maximum of $a_1 ... a_n \in \mathbb{R}^n$

```
1 H_n ← v
2 Heapify-up(n)
3 n ← n + 1
```