16. RECURRENCE RELATIONS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

December 2024

It is now time to zoom-in on the *tractable problems* and start classifying and comparing them using more precise complexity characteristics.

Up until now, we were content to split the world of problems into "polynomial" and "non-polynomial"; but it is time to move on and check which problems can be solved in $O(n), O(n \log(n))$, $O(n^2)$ etc. time.

This means that we will have to refer to some particular model of computation (remember that, for example, a 2-tape TM can solve `PALINDROME` in time $\Theta(|w|)$, but a single-tape TM can only do it in $\Theta(|w|^2)$).

Remember the problem `PALINDROME` and its single-tape and two-tape solutions. We discussed that a single-tape TM can solve it in $\Omega(|w|^2)$ time, but a two-tape TM can do it in $\Omega(|w|)$ time. So the precise choice of model matters if we intend to make a classification more fine-grained than "polynomial" vs. "non-polynomial".

Going forward, our model of computation will not be the Turing Machine, but a generic (and vaguely defined) "RAM machine". Ideally, we would like here to make a gentle introduction to this concept, dedicating a whole lecture to the intuitive functioning, formal definition, computability and complexity perspective of the RAM machine. However, due to time constraints, we will skip this presentation; instead we'll just present these relevant features:

- A RAM machine has integer registers and an arbitrarily large "memory": an array of locations that can each hold an integer; accessing any value in this array can be done in constant time.
- A RAM machine cannot compute more functions that a TM; every RAM machine can be simulated by a TM.
- Every RAM machine can be simulated by a TM with only a polynomial slowdown.

Familiarity with the x86 architecture and assembly language should help with understanding of the RAM machine model. However, do note that x86 registers and memory locations are of fixed sizes and cannot store arbitrarily large integers. The "instruction set" of the RAM machine is a complicated matter: most of the time, you can treat as "a single-step action" (so in constant time $\Theta(1)$) any operation for which there exists an x86 instruction. One immediate objection to this is that (for the RAM machine to be theoretically useful), multiplication cannot be a primitive, constant-time operation. To circumvent in-depth discussions about why this is so and what exactly are the operations possible in constant time, we will usually specify some specific operation(s) that we're counting as "basic operations". For example, in the context of "comparison-based sorting algorithms", we will "count" the number of comparisons performed by the algorithms. When we say: *"insertion sort has a complexity of* $\Theta(n^2)$*"*, we mean that the number of comparisons performed by insertion sort grows asymptotically in a quadratic manner with the number of elements in the input array.

1 Merge sort

Our first case study is the algorithm known as "merge sort". Note that sorting is not a decision problem, we have to compute a function: given a list of integers, what is the sorted form of the list?

The algorithm works recursively. The base case is a list consisting of a single element, or the empty list: these are already sorted.

Any bigger list is split in two, then merge sort is applied twice, recursively, on each half-list. The sorted lists are then *merged* together. *Merging* is a procedure that takes two sorted lists and produces from them a third one in sorted order; because the input lists are already sorted, this is quite easy to do.

Algorithm 16.1 Merge Sort pseudocode

```
1: function MERGE_SORT(A, length)
2:     if (length = 0) ∨ (length = 1) then
3:         return A
4:     middle ← ⌊length/2⌋
5:     A_L ← MERGE_SORT(A, middle)
6:     A_R ← MERGE_SORT(A + middle, length − middle)
7:     return merge(A_L, middle, A_R, length − middle)

8: function MERGE(A, length_A, B, length_B)
9:     length_R ← length_A + length_B
10:    R ← NEW_ARRAY(length_R)
11:    i ← 0
12:    j ← 0
13:    k ← 0
14:    while i < m ∧ j < n do
15:        if A[i] ≤ B[j] then
16:            R[k] ← A[i]
17:            k ← k + 1
18:            i ← i + 1
19:        else
20:            R[k] ← B[j]
21:            k ← k + 1
22:            j ← j + 1
23:    while i < length_A do
24:        R[k] ← A[i]
25:        k ← k + 1
26:        i ← i + 1
27:    while j < length_B do
28:        R[k] ← B[j]
29:        k ← k + 1
30:        j ← j + 1
31:    return R
```

Theorem 16.1. `MERGE_SORT` has a running time of $\Theta(n \log(n))$

Where n is the array length (called `length` for clarity in the pseudocode).

You are probably familiar with this result, but how can we prove it?

The first thing that the merge procedure does is to allocate a new array R; the precise details of this process are unimportant (in practice, we would have a `malloc` that we can treat as a black box); the only essential element is that we can think of it as happening in *linear time*, or faster.

It is easy to see that the rest of the procedure does $\Theta(\texttt{length}_A + \texttt{length}_B)$ operations. But the merged arrays are halves of the original, so we can say that *merge* runs in **linear time** $\Theta(n)$. The difficulty with merge sort itself comes from the recursive calls.

Let $T(n)$ be the time complexity of merge sort on an array of n elements. Then the recursive calls lead us to this recursive relation:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

When given an array of n elements, merge sort calls itself twice with arrays half as long, then does some extra work: it merges them in linear time[†].

For now, let us make the simplifying assumption that we are only dealing with arrays whose length is a power of 2: at each step, the array is split exactly in half, two parts of equal length. This gives us the relation:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$$

In order for this relation to make sense, we need some *base case*. For an array with no elements, or one elements, the function just does some checks in constant time:

$$T(0) = \Theta(1) \tag{1}$$
$$T(1) = \Theta(1) \tag{2}$$

In this lecture we will learn how to solve these types of recurrences, a technique that allows us to determine the time complexity of recursive algorithms.

2 The recursion-tree method

Our first approach is very natural and general. We will construct a *recursion-tree* of the formula. The structure of the tree is as follows:

- We start from the root; we label this with the time complexity required to split/combine solutions and anything else. We shall think of this as "the amount of work" (in basic operations[‡]) that need to be performed in a single recursive call.

  In our case, this "amount of work" is $\Theta(n)$

- Each node has a number of children equal to the number of recursive calls made.

  In our case, each node has two children, except for the leaves of the tree, which have no children. In general, level i of the tree has 2^i nodes (the root is on level 0).

- As mentioned before, each node gets labeled with its "local amount of work".

  In our case, for the children of the root this is $\Theta(\frac{n}{2})$. In general, a node on level i has a label of $\Theta(\frac{n}{2^i})$.

- The leaves are those nodes that correspond to the base cases.

  In our case, the leaves are those nodes which have to sort arrays with one item. They are all on the same level, let's call it k. Because we get to these leaves by halving n a total number of k times, we have that $k = \log_2(n)$.

In the recursion tree of merge sort, on each level i, we have 2^i nodes, each doing $\frac{n}{2^i}$ operations; a total number of n operations per each level.

Starting with the root node on level 0, we have $\log(n) + 1$ levels.

Thus the total "amount of work" of the tree is $n \cdot \log(n) + 1$; this is $\Theta(n \log(n))$.

In general, we need to describe the tree structure, including the "amount of work" performed by each node; after doing so, we just have to sum up the work of all the nodes.

For merge sort, each level performed the same amount of work, which made it easy – we only had to multiply two terms. For other relations, this might not be the case.

In the next lecture we will learn about the *master method*, with which we can directly obtain an answer for all recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$.

[†]It also does the extra work of checking the array length for the base case, but this is some extra constant that gets hidden by $\Theta(n)$.
[‡]Here, by "basic operations" we mean: comparing two values, incrementing a value, setting the value of a variable.

3 Dealing with numbers that are not powers of 2

Let us return to our assumption that n is a power of 2. If it isn't, how does that change our result?

In the tree described above, it will not be true that each child of a node has the same amount of work to do; but the amount of work per level will be the same.

The tree also becomes unbalanced and some paths from root to leaf will be longer than others. But this difference can be at most 1.

As a consequence, the asymptotic growth we found for powers of 2, also holds for other numbers.

Note that this is only true for "natural" recurrences, such as the one described for merge sort, where one of the children gets the floor of $\frac{n}{2}$ and the other gets the ceil of $\frac{n}{2}$. We can imagine an artificial recurrence that checks whether n is a power of 2 and changes its behavior based on this answer, but we shall ignore these cases, as they don't really appear in practice.

4 The substitution method

A more general method of determining the complexity of a recurrence relation is using the *substitution method*.

This involves making a guess of the complexity, then proving it rigorously, using induction. The recursion-tree method is a good way of getting such a guess: we can apply it sloppily, get an idea of the true answer, then prove it using the substitution method. Generally, this should involve less work than writing a precise, rigorous proof using the recursion-tree method alone.

Let us return to merge sort and its recurrence relation and show how the substitution method works to prove that $T(n) \in O(n \log(n))$. We only prove an upper-bound here; this allows us to easily give up our assumption that n is a power of 2; we shall actually consider a modified version of the recurrence $T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$. Note that this is "bigger" than the actual recurrence of merge sort.

The term $\Theta(n)$ seems difficult to work with; we can choose instead some particular function from $\Theta(n)$, say n, and rewrite $T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$.

Remember the formal definition of O! We need to show that there exists two positive constants c and n_0, such that for any $n \geq n_0$, $0 \leq T(n) \leq cn \log(n)$.

Before we begin our proof, there is an important upper bound we need:

Theorem 16.2.

$$\log\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq \log\left(\frac{n}{2}\right) + \frac{2}{n}$$

Proof.

$$\log\left(\left\lceil \frac{n}{2} \right\rceil\right) = \log\left(\frac{n}{2}\right) + \log\left(\frac{\left\lceil \frac{n}{2} \right\rceil}{\frac{n}{2}}\right) \tag{3}$$

$$< \log\left(\frac{n}{2}\right) + \log\left(\frac{\frac{n}{2}+1}{\frac{n}{2}}\right) \tag{4}$$

$$= \log\left(\frac{n}{2}\right) + \log\left(1 + \frac{2}{n}\right) \tag{5}$$

$$\leq \log\left(\frac{n}{2}\right) + \frac{2}{n} \tag{6}$$

□

This proof hinges on the fact that $\log(1+x) \leq x$ for $x \geq 0$.

Now we can get back to our inductive proof that $T(n) \in O(n \log(n))$.

Base case: $T(1)$ equals some constant $k > 0$. This is problematic, because we obtain $0 \leq k \leq 0$ which is false.

But remember that the asymptotic notations only tell us something about the growth of a function, *from some point n_0 onwards*. So we can choose a different base case for our inductive hypothesis, $n = 2$; we obtain: $0 \leq 2k+2 \leq 2c$.

This holds for any $c \geq k + 1$.

For reasons that we will clarify later, we must also choose $T(3)$ as a base case:

$$T(3) = 2T(2) + 3 = 4k + 7$$

$0 \leq 4k + 7 \leq c3\log(3)$ holds for all $c \geq \dfrac{4k+7}{3\log 3}$.

We have two constraints that c should be bigger than some value, so we must choose the maximum between the two. Later, we will get another constraint.

Induction step: Assume our relation holds for any number lower than n; this includes $\lceil \frac{n}{2} \rceil$: $0 \leq T\left(\lceil \frac{n}{2} \rceil\right) \leq c\lceil \frac{n}{2} \rceil \log\left(\lceil \frac{n}{2} \rceil\right)$.

Multiplying this relation by 2 and adding n we get:

$$0 \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \leq 2c\left\lceil \frac{n}{2} \right\rceil \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + n$$

But the middle part is precisely $T(n)$, so we have:

$$\begin{aligned}
0 \leq T(n) &\leq 2c\left\lceil \frac{n}{2} \right\rceil \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + n && (7)\\
&= 2\left\lceil \frac{n}{2} \right\rceil c\left(\log\left(\frac{n}{2}\right) + \frac{2}{n}\right) + n && (8)\\
&\leq 2\left\lceil \frac{n}{2} \right\rceil c\left(\log(n) - 1 + \frac{2}{n}\right) + n && (9)\\
&\leq cn\left(\log(n) - 1 + \frac{2}{n}\right) + c\left(\log(n) - 1 + \frac{2}{n}\right) + n && (10)\\
&= cn\log(n) - cn + 2c + c\log(n) - c + \frac{2c}{n} + n && (11)\\
&= cn\log(n) + (1-c)n + c\log(n) + c + \frac{2c}{n} && (12)
\end{aligned}$$

$c\log(n) + c + \frac{2c}{n}$ grows slower than n; so if n's coefficient is negative, everything after $cn\log(n)$ is less than zero for sufficiently large n.

We can set c to whatever value we need, but we need this to work starting from $n = 4$ (the first non-base case).

In other words, we want the **monotonously decreasing function**: $f(n) = (1-c)n + c\log(n) + c + \frac{2c}{n}$ to be already **smaller than 0** at $f(4)$:

$$\begin{aligned}
f(4) &= (1-c)4 + c\log(4) + c + \frac{2c}{4} && (13)\\
&= 4 - 4c + 2c + c + \frac{c}{2} && (14)\\
&= 4 - \frac{c}{2} && (15)
\end{aligned}$$

So $f(4) \leq 0 \Leftrightarrow c \geq 8$; so we can just choose c to be $\max\left(k+1, \dfrac{4k+7}{3\log(3)}\right) + 8$.

For $n = 3$, there is no positive value of c for which $f(3) < 0$; that is why we had to treat 3 separately, as a base case.

4.1 Simplifying

From now on, we will generally not bother with specifics, such as using floors and ceils. We will think of recurrences as of the form $T(n) = 2T(\frac{n}{2}) + n$.

This will keep proofs much simpler:

$$\begin{aligned}
&T(n) \in O(n\log(n)) \Leftrightarrow && (16)\\
&\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N},\ \text{s.t. } \forall n \geq n_0, 0 \leq T(n) \leq cn\log(n) && (17)
\end{aligned}$$

Proof. **Base case**: $n = 2:\quad 0 \leq 2k + 2 \leq 2c$, which holds for $c \geq k + 1$.

Induction step: Assume that $0 \leq T(\frac{n}{2}) \leq c\frac{n}{2}\log(\frac{n}{2})$.

Multiply by 2 and add n to get:

$$T(n) \leq 2c\frac{n}{2}\log(\frac{n}{2}) + n \tag{18}$$
$$= cn\log(\frac{n}{2}) + n \tag{19}$$
$$= cn(\log(n) - \log(2)) + n \tag{20}$$
$$= cn\log(n) - cn + n \tag{21}$$
$$\leq cn\log(n) \tag{22}$$

The final relation holds for all $c \geq 1$, so $c = \max(1, k + 2)$.

□

4.2 Complications

Sometimes our proofs may fail in a specific way. Consider the recurrence $T(n) = 2T(\frac{n}{2}) + \log(n)$. Sketch the recursion tree to get the guess that $T(n) \in O(n)$.

We then proceed with the substitution method:

Proof. **Base case**: $n = 1:\quad 0 \leq k \leq c$, which holds for $c \geq k$.

Induction step: Assume that $0 \leq T(\frac{n}{2}) \leq c\frac{n}{2}$.

Multiply by 2 and add 1 to get:

$$T(n) \leq 2c\frac{n}{2} + 1 \tag{23}$$
$$= cn + 1 \tag{24}$$

... and we're stuck...

□

We showed that $T(n) \leq cn + 1$, but we are supposed to show that $T(n) \leq cn$, which doesn't follow from what we have. What bothers us is that $+1$ that we can't get rid of.

To address this, we will change the form of what we are trying to prove, by showing instead that $0 \leq T(n) \leq c(n - r)$, for some *fixed* constant r. Proving this amounts to showing that $T(n) \in O(n - r)$; but this is the exact same set as $O(n)$!

Proof. **Base case**: $n = 1:\quad 0 \leq k \leq c(1 - r)$.

r is a constant that we can choose for our convenience, before starting the whole proof[§], so we can avoid the problematic values that yield something that isn't strictly positive (so $r < 1$).

Then the relation holds for: $c \geq \dfrac{k}{1 - r}$.

Induction step: Assume that $0 \leq T(\frac{n}{2}) \leq c(\frac{n}{2} - r)$.

Multiply by 2 and add 1 to get:

[§]We gave it a symbolic name so that we can start the proof and see what problems appear along the way.

$$T(n) \leq 2c(\frac{n}{2} - r) + 1 \tag{25}$$
$$= 2c\frac{n}{2} - 2cr + 1 \tag{26}$$
$$= cn - cr + (1 - cr) \tag{27}$$
$$= c(n - r) + (1 - cr) \tag{28}$$
$$\leq c(n - r) \tag{29}$$

Now we have a way out! The final relation holds if $1 - cr \leq 0$, i.e. if $c \geq \frac{1}{r}$.

□

5 References and further reading

Much of the treatment of recurrences in these notes is inspired by and adapted from section 4 "Divide-and-Conquer" of the first chapter of the legendary textbook "Introduction to Algorithms" [1] by Cormen, Leiserson, Rivest and Stein.

Bibliography

[1] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 1990.