Algoritmer og Datastrukturer

zts900, chs380

18-02-2025

---

1 Merge sort

How many calls for size 21?:

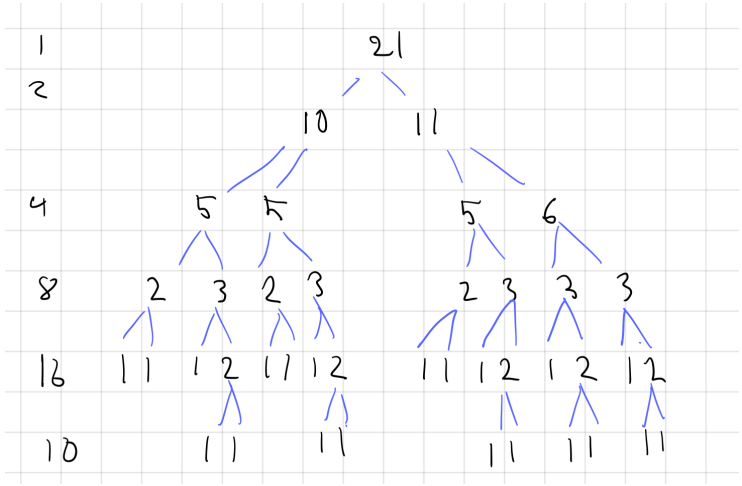

Figure 1: A sample caption describing the picture.

We have one call when we first call teh fucntion then each time we split the list

- recursive depth: 1 - 1 main call
- recursive depth: 2 - main call
- 4
- 8
- 16

- 10 - we end up with uneven lists that results in not the full nr of recursive calls for this level as those lists were 1 longer.

Total 41 calls

How many calls in general?

In our recursive tree we know that we will end up with n leaf nodes since every list will be split down to lenght 1. We also know that the number of internal nodes will be 1 less than our leaf nodes in a full binary tree. We have L nr leafs, I internal, T nr nodes in tree:

$$T = I + L$$

$$I = L - 1$$

$$T = (L - 1) + L = 2L - 1$$

Each node in the tree T represents a recursive call, this includes the leaf nodes since we need a call each to trigger the if statement that will return them, we have the same nr leafnodes as items in our list L=n we derive:

$$T = 2n - 1$$

---

2 Recurrences

In the first 4 recurrences we want to apply the master theorem to solve them, because this theorem applies to recurrences of the form $T(n) = aT(n/b) + f(n)$ where **a** is the number of subproblems, **b** is the factor by which **n** is reduced and $f(n)$ is the non recursive cost.
In the last 2 recurrences we will use recursion-tree method for solving recurrences.

1. $T(n) = 4T(\lfloor n/2 \rfloor) + n\ log\ n$

a=4 , b=2, $f(n) = n\ log\ n$
We analyze the critical exponent $\log_b a$ which tells us how the recursive calls contribute to the overall complexity.
$\log_2 4 = 2$, therefore the recurrence relations complexity is directly influenced by the term $n^{\log_b a} = n^{\log_2 4} = n^2$
The exponent helps us determine wheather the function $f(n)$ (the extra work outside the recursion) is smaller or equal to or larger than n^2, which allows us to apply the master theorem correctly.

We check if $f(n)$ is asymptocally smaller than n^2 and observe that $f(n) = n\ log\ n < n^2$ because n^2 grows faster that log.
That's why we are dealing with Case 1. Since $f(n) = n\ log\ n = O(n^{2-\epsilon})$ for any constant $\epsilon \leq 1$, we conclude that the solution is $T(n) = \theta(n^2)$.
Therefore the first reccurence has the solution we are looking for.

2. $T(n) = 4T(\lfloor n/2 \rfloor) + n^2$

a=4 b=2, which mean that the watershed function is $n^{log_b a} = n^{log_2 4} = n^2$
Case 2 applies when $f(n) = \theta(n^{log_b a}\ log^k n)$ which means that $f(n)$ has the exact same polynomial growth as $n^{log_b a}$ but might have an additional log factor (like $n^2 \log n$).
In this case there is no additional log factor, **k=0**, so $f(n) = n^2 = \theta(n^2 log^0 n)$ and $T(n) = \theta(n^2\ lg\ n)$ which is not equal to the solution that we are looking for.

3. $T(n) = 2T(\lfloor n/4 \rfloor) + n^2$

a=2 ; b=4 which means that the watershead function is $n^{log_b a} = n^{log_4 2} = n^{1/2}$
Since n^2 grows faster than $n^{1/2}$ we have $f(n) = \Omega(n^{log_b a+\epsilon}) = \Omega(n^{1/2+\epsilon})$ which satisfies the first condition in Case 3.
The regularity condition states than $af(n/b) \leq cf(n)$ for some $c < 1$, meaning that the extra work should not explode too quickly as n gets smaller:
$af(n/b) \leftrightarrow 2f(n/4) \leftrightarrow 2f(n/4)^2 \leftrightarrow 2n^2/16 \leftrightarrow n^2/8$
Since $n^2/8 \leq cn^2$ for c=1/8 which is less than 1, the regularity condition hold and we have the solution $T(n) = \theta(f(n) = \theta(n^2)$

4. $T(n) = 9T(\lfloor n/3 \rfloor) + n^3$

a=9 , b=3 which means that the watershed function is $n^{log_b a} = n^{log_3 9} = n^2$
Since n^3 grows faster than n^2: $f(n) = \Omega(n^{log_b a+\epsilon}) = \Omega(n^{2+\epsilon})$ which satisfies the first condition of case 3.
The regularity condition states than $af(n/b) \leq cf(n)$ for some $c < 1$, meaning that the extra work should not explode too quickly as n gets smaller:
$9f(n/3) \leftrightarrow 9(n/3)^3 \leftrightarrow 9(n^3/27) \leftrightarrow n^3/3$
Since $n^3 \leq cn^2$ for **c=1/3** which is equal 1, the regularity condition holds and we have the solution $T(n) = \theta f(n) = \theta(n^3) \neq \theta(n^2)$

5. $T(n) = T(n-1) + n^2$

We can see how the formula changes recursively by substituting $T(n-1)$ into the equation:

$$T(n-1) = T(n-1) - 1 + (n-1)^2 \qquad = T(n-2) + (n-1)^2$$

Substituting into $T(n)$:

$$T(n) = (T(n-2) + (n-1)^2) + n^2$$

Repeating for $T(n-2)$:

$$T(n-2) = T(n-2) - 1 + (n-2)^2 \qquad = T(n-3) + (n-2)^2$$

Substitutiing again into $T(n)$:

$$T(n) = (T(n-3) + (n-2)^2) + (n-1)^2 + n^2$$

Repeating the process for $T(n-3)$:

$$T(n-3) = T(n-3) - 1 + (n-3)^2 \qquad = T(n-4) + (n-3)^2$$

So we get:

$$T(n) = T(n-4) + (n-3)^2 + (n-2)^2 + (n-1)^2 + n^2$$

We can build a recursion tree:

Recursion Tree

Level 0: n^2
Level 1: $(n-1)^2, (n-1)^2$
Level 2: $(n-2)^2, (n-2)^2, (n-2)^2, (n-2)^2$
We stop when $(n-k) = 1 \Rightarrow k = n-1$, meaning there are $n-1$ levels.
The total cost accumulates as a sum of all the cost in the different levels which means that it accumulates as $T(n) = T(1) + \sum_{i=0}^{n-1}(n-i)^2$
If we rewrite by setting $j = n-1-i$:

$$\sum_{i=0}^{n-1}(n-i)^2 = \sum_{j=1}^{n} j^2$$

We use the sum formula:

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$$

And approximating we get:

$$\frac{n(n+1)(2n+1)}{6} \approx \frac{2n^3}{6} = \frac{n^3}{3}$$

Therefore, the recurrence has the solution:

$$T(n) = \theta(n^3)$$

which is not equal to the solution we are looking for.

6. $T(n) = T(n-1) + n$

We expand recursively:

$$T(n-1) = T(n-2) + (n-1) \Rightarrow T(n) = (T(n-2) + (n-1)) + n$$

Again:

$$T(n-2) = T(n-3) + (n-2) \Rightarrow T(n) = (T(n-3) + (n-2)) + (n-1) + n$$

Again:

$$T(n-3) = T(n-4) + (n-3) \Rightarrow T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n$$

We construct the recursion tree:

Recursion Tree

Level 0:n
Level 1: $(n-1), (n-1)$
Level 2: $(n-2), (n-2), (n-2), (n-2)$
Level i: $(n-i)$
We stop when $(n-k) = 1 \Rightarrow k = n-1$, meaning there are $n-1$ levels. We therefore solve:

$$\sum_{i=0}^{n-1} (n-i)$$

Using the sum formula for the first n natural numbers:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We approximate and get:

$$\frac{n(n+1)}{2} = n^2 + n + \frac{1}{2} = n^2$$

We therefore have that the solution for the recurrence is :

$$T(n) = \theta(n^2)$$

which is equal to the solution we are looking for

---

3 Divide & Conquer

We know that we have a recursive algorithm that divides the input **X** in 4 subproblems X_1, X_2, X_3, X_4, each with size $\lfloor n/2 \rfloor$.
It calls the algortihm recursively in the 4 subproblems. It takes **O (n log n)** time to calculate the 4 subproblems and **O(n^2)** to combine the results. Therefore we can write the following reccurence:

$$T(n) = 4T(\frac{n}{2}) + O(n\ log\ n) + O(n^2)$$

But because **O(n^2)** grows faster than **O (n log n)**, we can ignore **O (n log n)** and write the reccurence as:

$$T(n) = 4T(\frac{n}{2}) + O(n^2)$$

We can now solve this using the master theorem:
We have **a = 4** and **b = 2**, which means that the waltershed function is $n^{\log_b a} = n^{\log_2 4} = n^2$.
Here we have that Case 2 applies when $f(n) = \theta(n^{\log_b a} \log_n^k)$ which means that $f(n)$ has the exact same polynomial growth as $n^{\log_b a}$, but it might have an additional log factor.
In this case there is no additional log factor and its solution is

$$T(n) = \theta(n^2 \log n)$$

---

4 Runtime

We can try to find the runtime of the pseudocode : We observe that the outer-loop runs from **i=1 to i= n/2** which means that it runs exactly **n/2** times.

The inner loop runs depending on the value of **i**. It runs from **j = n/2-i to j=n**. Therefore the iterations for a given **i** is:

$$n - (n/2 - i) = n/2 + i$$

We can now calculate the total iterations of the code, which is the times the inner loop runs for each **i**:

$$\sum_{i=1}^{n/2}(n/2 + i)$$

$$\sum_{i=1}^{n/2}(n/2) + \sum_{i=1}^{n/2} i$$

The first sum is so $(n/2)(n/2) = n^2/4$ And the second sum is the sum of the first k natural numbers:

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

Where $k = n/2$: $\frac{(n/2)(n/2\ +1)}{2} = \frac{n^2/4+n/2}{2} = \frac{n^2}{8} + \frac{n}{4}$
Due to ignoring constants and focusing on the highest term which is n^2, we conclude that the overall running time is $T(n) = O(n^2)$
It is now easy to identify the correct statements: The first one $T(n) = O(n)$ is not correct because the running time is higher than linear; The second one $T(n) = O(n\ log\ n)$ is not correct because the running time time higher than $O(n\ log\ n)$; The third one $T(n) = O(n^2)$ is correct because upper bound surely is $O(n^2)$; The fourth one $T(n) = \Omega(n)$ is correct because $O(n^2)$ implies $\Omega(n)$; The fifth one $T(n) = \Omega(n\ log\ n)$ is correct because $O(n^2)$ implies $\Omega(n\ log\ n)$; The sixth one $T(n) = \Omega(n^2)$ is correct because $T(n) = O(n^2)$ also applies as $\Omega(n^2)$

---

5 Invariants

What are vaild loop invariants at the start of each itteration of the for loop. Choose one or several correct answers.

The code finds the second smallest nr

We loop through a list of numbers looking at element A[i] each loop. For each itteration the smallest observed element is stored in m, while the next smallest element is stroed in r. The update of the elements is done thorugh two conditionals checking if A[i]<m and then updating A[i] and r accordingly, and one checking m<A[i]<r, meaning that we found a new second largest element and update r accordingly. With this in mind we look at the statements:

1. TRUE: $m = min(A[1], .., A[i-1])$ meaning m is the smallest number in the list range index 1 to i-1, the loop goes to n meaning at the end n-1. This is true since m is the smallest of the observed elements and alle elements A[1] to A[i-1] are obaserved at step i, and m is initialised to A[1].

2. FALSE: $m = min(A[1], .., A[n])$ at the start of each loop we have yet to look at A[i], which we do in the itteration, here max A[n], meaning at step i=n we have only observed elements up to n-1. Hence the statement is false.

3. TRUE: $r \geq m$ meaning r is larger or equal to m. This is true, r is the second largest obseved element, if we have two elements of the same size r=m otherwise r<m.

4. FALSE: $m \geq 0$ this is false we have no element limitation to possitive numbers.

5. TRUE: At least 1 element in $A[1], .., A[i-1]$ is smaller or equal to r. This is true, in the case where i = 2, $r = \infty$, makeing any element less or equal. When r is initialised, even if the list is of lenght 1 or only have one element size r is equal otherwise r is less. This is true.

6. FALSE At most 2 elements in $A[1], ..., A[i-1]$ are les tahn or equal to r. If we have multiple of the same elements there we can have f.ex 5 smallest elements m would be the first observed of these r would be the second but we would have 3 left that are equal.

---

6 Induction

Prove that: the number of diagonals d_n in a n-conrner shape is equal to $\frac{n(n-3)}{2}$ for $3 \leq n$. We know that: a diagroanl splits a ncorner into a n1-shape and n2shape with a common border where $n_1 + n_2 = n + 2$.

We want to prove:

$$d_n = \frac{n(n-3)}{2}$$

B.s

A triangle with n = 3 corners has d3 = 0 diagonals

$$d_3 = \frac{3(3-3)}{2} = 0$$

holds.

I.h

how does the nr diagonals change going from n to n+1?

- $n = 3 \rightarrow d_n = 0$
- $n = 4 \rightarrow d_n = 2 : \ d_n - d_{n-1} = 2$
- $n = 5 \rightarrow d_n = 5 : \ d_n - d_{n-1} = 3$
- $n = 6 \rightarrow d_n = 9 : \ d_n - d_{n-1} = 4$
- $n = 7 \rightarrow d_n = 14 : \ d_n - d_{n-1} = 5$

The difference between step n and n+1 grows as is: 2, 3, 4, 5... We observe that it grows with n-2. Hence we conclude:

$$d_n - d_{n-1} = n - 2$$

$$d_n = n - 2 + d_{n-1}$$

$$d_{n+1} = (n+1) - 2 + d_n$$

I.s

We must prove that for some $3 \leq k \mid k \in Z$:

$$d_{k+1} = \frac{(k+1)((k+1)-3)}{2} = \frac{(k+1)(k-2)}{2} = k^2 - k - 2$$

We do the following steps:

$$\begin{aligned} d_{k+1} &= d_k + (k+1) - 2 \\ &= \frac{k(k-3)}{2} + k - 1 \\ &= \frac{k(k-3) + 2k - 2}{2} \\ &= \frac{k^2 - k - 2}{2} \\ &= \frac{(k+1)((k+1)-3)}{2} \end{aligned}$$

Hence it is proven that our formula applies for all n.

7 Correctness and runtime

a)

From the given informaotion we want to argue for

$$Ax = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

We have:

$$Ax = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix}$$

If follow from mx multiplication that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_{n-2} + F_{n-1} \end{pmatrix}$$

We know from the fibonacci sequence that $F_n = F_{n-1} + F_{n-2}$ for $n > 1$ hence

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

$$Ax = (F_{n-1}, F_n)^T$$

Our identity holds

b)

We want to prove: $A^{n-1}x = (F_{n-1}, F_n)^T$ for $n \geq 1$.

B.s

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

for $n = 2$ we have

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

Holds

I.h

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

I.s

We need to prove:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

We porceed as follows:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} =$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} =$$

Using our I.h

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} =$$

We know from a) that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

c)

With multiplication addition and subtraction beeing executed in constant time we would in a simple implementation to derive A^{n-1} have some constant time for conducting a 2x2 matrix multiplication c, times the number of matricies we combine, here n-1, hence we would complete the calculation in $(n-1)c$ time which is bounded from above by $O(n)$.

To be more efficinet we can attempt to "jump" as fast as possible to the desired $n-1$ exponent by using squares. F.ex instead of for $n-1=5$ calculate $AAAAA = A^5$ we can use the already derived step $AA = A^2$ and then calculate $A^2A^2A = A^5$, requiering ony 3 matrix multiplications instead of 4 as in the first case. Lets derive and algorithm to formaise this method and then argue for its time complexity.

To argue for the time complexity i derive the following algorithm, where i recurively divide up the multiplication operation into its sub operations by halving or near halving the exponent each time, calling the function with m/2 when even and with m-1 when uneven and adding a *A.

recurrance

We have constant time O(1) for multiplication of a 2x2 matrix, in the even case we have the recursive case T(m/2) and one multiplication of the result O(1), in the odd case we have the recursive case T(m-1) and one matrix multiplication O(1):

Algorithm 1 Exponentiate a Matrix using Exponentiation by Squaring

```
1: function EXP_MATRIX(A, m)
2:     if m = 1 then
3:         return A
4:     end if
5:     if m is even then
6:         Half ← EXP_MATRIX(A, m/2)
7:         return Half × Half
8:     else
9:         return EXP_MATRIX(A, m − 1) ×A
10:    end if
11: end function
```

$$T(m) = \begin{cases} O(1) & m = 1 \\ T(m/2) + O(1) & m - even \\ T(m-1) + O(1) & m - odd \end{cases}$$

Lets first think abou it in the case when m is a power of 2 $m = 2^k$we get the recurrance: $T(m) = T(m/2) + O(1)$, with the mater theorem we have:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$$T(m) = T(m/2) + O(1)$$

Hence $a = 1$, $b = 2$ and $f(n) = O(1)$ implying $f(n) = O(n^{log_2 1})$, $log_2 1 = 0$ implying n^0. Since any ϵ added to the exponenet woudl make our funciton non constant we can rule out case 1 and 3. However case 2 is still viable for which we have to fulfill that there exsists $K \geq 0$ s.t $f(n) = \Theta(n^{log_2 1}(log_2 n)^k)$, we see that for our case we can set $k = 0$ giving $f(n) = \Theta(n^0(log_2 n)^0) = \Theta(1)$, which fits.

For case 2 the master theorem states that $T(n) = \Theta(n^{log_b a}(log_2 n)^{k+1})$ giving us time complexity:

$$T(n) = \Theta(n^{log_2 1}(log_2 n)^1) = \Theta(log_2 n)$$

d)

1. For c) showed that we can derive the matrix A^{n-1}in $O(lgn)$ time.
2. For b) we proved: $A^{n-1}(0,1)^T = (F_{n-1}, F_n)^T$ for $n \geq 1$.
3. We are assuming artimic operations are conducted in constant time.

We want to argue that F_n can be calculated in $O(lgn)$ time. The argument goes as follows.

We can derive A^{n-1}in $O(lgn)$ time (1), which means that given constant time for artimetric operations (3), we can derive $A^{n-1}(0,1)^T = (F_{n-1}, F_n)^T$, in $O(lgn) + O(1)$ time, which is $O(lgn)$ time. The resulting matrix contains F_n. Hence we have derived F_n in $O(lgn)$ time.