

## Assignment 1 — AD

Schmidt, Victor Alexander, `rqc908`

Ibsen, Helga Rykov, `mcv462`

Larsen, Christian Vadstrup, `1jq919`

Monday 08:00, February 20th

## Task 1: Master Theorem

1.

we have the following pricing scheme  $p(n) = 8p(n/2) + n^2$

we set:  $a = 8$   $b = 2$   $f(n) = n^2$

Since  $n^{\log_b a} = n^3$  we are in the first case.

Solution:  $p(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$

2.

we have the following pricing scheme  $p(n) = 8p(n/4) + n^3$

we set:  $a = 8$   $b = 4$   $f(n) = n^3$

since  $n^{\log_b a} = n^{\frac{3}{2}}$  we are in the third case.

we have  $af(\frac{n}{b}) = 8(\frac{n}{4})(\frac{n^3}{4}) < \frac{1}{2}n^3 = \frac{1}{2}f(n)$

pick  $c = \frac{1}{2}$

Solution:  $p(n) = \Theta(f(n)) = \Theta(n^3)$


3.

we have the following pricing scheme  $p(n) = 10p(n/9) + n \log_2 n$

we set:  $a = 10$   $b = 9$   $f(n) = n \log_2 n$  since  $n^{\log_b a} = n^{1.05}$  we are in the first case.

which is because we know that positive polynomials are bigger than polylogarithms, which mean that we can rearrange our statement as  $O(n \cdot n^{1.05}) = n \log_2 n$

which can be rewritten as  $O(n^{1.05}) = \log_2 n$  which gives us a

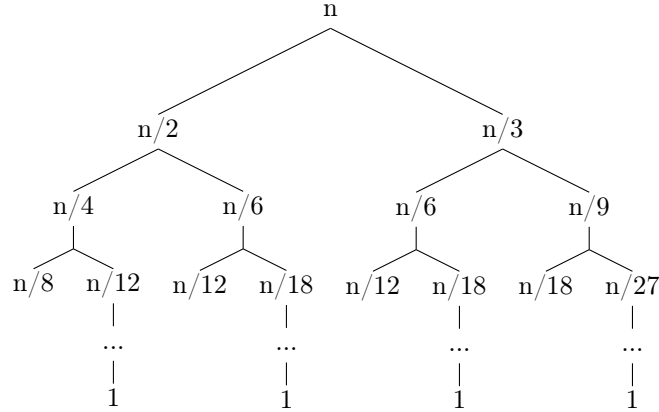
Solution:  $p(n) = \Theta(f(n)) = \Theta(n^{1.05})$  

## Task 2: Recursion tree and the substitution method

1.

A recursion tree for

$$p(n) = p(n/2) + p(n/3) + n$$



The cost of each level is the sum of the costs of all the leaves at that level:

Level 0:  $n = n$

Level 1:  $(n/2 + n/3) = 5/6 \cdot n$

Level 2:  $(n/4 + 2n/6 + n/9) = 5^2/6^2 \cdot n$

Level 3:  $(n/8 + 3n/12 + 3n/18 + n/27) = 5^3/6^3 \cdot n$

Level k:  $(5/6)^k \cdot n$

The total cost of the function is approximately the sum of the costs at each level, which is:

$$T(n) = n + (5/6)n + (5/6)^2n + (5/6)^3n + (5/6)^4n$$

This is a converging geometric series, which has a finite sum. Therefore, the overall cost of the function is dominated by the term  $n$ . So, our guess of the upper bound on the recurrence is:

$$T(n) = \mathcal{O}(n)$$

Now, we make use of the **substitution method** by substituting into the original recurrence:

$$\begin{aligned} T(n) &\leq cn \\ T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + n \\ &\leq \frac{1}{2}cn + \frac{1}{3}cn + n \\ &= \frac{5}{6}cn + n \end{aligned}$$

Base cases, assuming  $n/2$  and  $n/3$  have to be integers, have to include  $T(1) = 1, T(2) = 2, T(3) = 3, T(4) = 4, T(5) = 5$ , which is the same as saying that  $n \geq 6$ . The previous equation will hold, when:

$$\begin{aligned}
 T(n) &\leq cn \\
 \frac{5}{6}cn + n &\leq cn \\
 \frac{5}{6}c + 1 &\leq c \\
 1 &\leq \frac{1}{6}c \\
 \Rightarrow c &\geq 6
 \end{aligned}$$

... We have now shown that  $T(n) = O(n)$  holds, when  $c \geq 6$ . If we set  $c = 6$  and  $n_0 = 6$  we get that  $T(n) \leq 6n$   
 $\square$

## 2.

A recursion tree for

$$p(n) = \sqrt{n} \cdot p(\sqrt{n}) + \sqrt{n}$$

$$\begin{array}{c}
 n \quad [\text{cost} : n] \\
 | \\
 \sqrt{n} \quad [\text{cost} : n^{1/2}] \\
 | \\
 \sqrt[4]{n} \quad [\text{cost} : n^{1/4}] \\
 | \\
 \sqrt[8]{n} \quad [\text{cost} : n^{1/8}] \\
 | \\
 \dots \quad [\text{cost} : n^{(1/2)^k}]
 \end{array}$$

At each level of the recursion tree contributes a term of the form  $n^{(1/2)^k}$  to the total cost, where  $k$  is the level number. As the input value gets smaller with each level, the total number of levels in the recursion tree is proportional to the iterated logarithm of  $n$ ,  $\log^* n$ . Therefore, the total cost of the function is:

$$\begin{aligned}
T(n) &= \sum_{k=0}^{\log^* n} n^{(\frac{1}{2})^k} \\
&\leq \sum_{k=0}^{\infty} n^{(\frac{1}{2})^k} \\
&= n + n^{1/2} + n^{1/4} + \dots \\
&= n \left( 1 + \frac{1}{\sqrt{n}} + \frac{1}{n^{1/4}} + \dots \right) \\
&\leq n \sum_{k=0}^{\infty} \left( \frac{1}{2} \right)^k
\end{aligned}$$

The sum is a geometric series with a common ratio of  $1/2$ , which means that:

$$\sum_{k=0}^{\infty} \left( \frac{1}{2} \right)^k = \frac{1}{1 - 1/2} = 2$$

Therefore, we have:

$$n \sum_{k=0}^{\infty} \left( \frac{1}{2} \right)^k = n \cdot 2 = 2n$$

Our guess of the upper bound on the recurrence is therefore  $T(n) \leq 2n = \mathcal{O}(n)$

... Now we make use of the **substitution method** by substituting into the original recurrence:

$$\begin{aligned}
T(n) &\leq cn \\
T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + \sqrt{n} \\
&\leq \sqrt{n} \cdot c\sqrt{n} + \sqrt{n} \\
&= cn + \sqrt{n}
\end{aligned}$$

As we are at an impasse, because  $cn + \sqrt{n} \not\leq cn$  (if  $n > 1$ ), we start over where  $T(n) \leq cn - d$ :

$$\begin{aligned}
T(n) &\leq cn - d \\
T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + \sqrt{n} \\
&\leq \sqrt{n} \cdot (c\sqrt{n} - d) + \sqrt{n} \\
&= cn - d\sqrt{n} + \sqrt{n}
\end{aligned}$$

The previous equation will hold, when:

$$\begin{aligned}
T(n) &\leq cn - d \\
cn - d\sqrt{n} + \sqrt{n} &\leq cn - d \\
\sqrt{n} &\leq d\sqrt{n} - d \\
1 &\leq d \left( 1 - \frac{1}{\sqrt{n}} \right) \\
\Rightarrow d &\geq \frac{1}{1 - \frac{1}{\sqrt{n}}}
\end{aligned}$$

... We assume that  $\sqrt{n}$  has to be an integer and  $n \neq 1$ , as this would break  $d$ . Base case  $T(1) = 1$ , and for this to be true,  $T(1) = 1 \leq c \cdot 1 - d$  where  $c \geq d + 1$ . We can then insert the smallest value into  $n$  which is  $n = n_0 = 2$ . Now we then get that the maximum  $d = 2$ , which means that if  $d$  satisfies  $d \geq 2$ , then inequality always holds, no matter the value of  $n$ . We now choose some value  $c = 3$  and we get that:

$$T(n) \leq 3n - \sqrt{n}, \forall n \geq 2$$

□

### Task 3: Pseudo-code for IntroSort

Comments: the variable `counter` counts levels of recursion

---

**Algorithm 1** INTROSORT( $A$ ,  $low$ ,  $high$ ,  $counter$ )

---

```
1: if ( $low < high$ ) then
2:   case 1:
3:   if ( $counter > 2\log(n)$ ) then
4:     HEAPSORT( $A$ ,  $low$ ,  $high$ )
5:   case 2:
6:   if ( $high - low \leq 16$ ) then
7:     INSERTIONSORT( $A$ ,  $low$ ,  $high$ )
8:   case 3:
9:    $q = \text{RANDOMIZED-PARTITION}(A, low, high)$ 
10:  INTROSORT ( $A$ ,  $low$ ,  $q - 1$ ,  $counter+1$ )
11:  INTROSORT ( $A$ ,  $q + 1$ ,  $high$ ,  $counter+1$ )
```

---

## Task 4: Show that the running time of introsort is worst-case $\mathcal{O}(n \log n)$

The worst-case running time of INTROSORT is calculated on the basis of the worst-case running times of HEAPSORT, INSERTIONSORT and RANDOMIZED-PARTITION. In accord with the results given in the course book, we have the following:

$$HEAPSORT : \mathcal{O}(n \lg n)$$

$$INSERTIONSORT : \mathcal{O}(n^2)$$

$$RANDOMIZED - PARTITION : \mathcal{O}(n \lg n)$$

INSERTIONSORT in the context of INTROSORT is called on a small array of max 16 elements. That makes its worst-case running time as follows:

$$INSERTIONSORT : \mathcal{O}(n^2) = \mathcal{O}(16^2) = \mathcal{O}(256)$$

Since INTROSORT consists of the three if-statements, we cannot sum up the running times of the three sorting algorithms. Instead, we choose the if-branch with the worst running time to make our estimation of the upper bound of INTROSORT.

In big  $\mathcal{O}$  notation, we drop constant factors, so  $\mathcal{O}(256)$  of INSERTIONSORT is equivalent to  $\mathcal{O}(1)$ , which is a constant-time algorithm.

The term  $\mathcal{O}(n \cdot \log(n))$  grows much faster than a constant-time algorithm as  $n$  increases and dominates the constant low-order term.

Now, both HEAPSORT and RANDOMIZED-PARTITION have  $\mathcal{O}(n \cdot \log(n))$  as their worst-case and expected running time, respectively.

We can therefore choose either of them and estimate the worst-case running time of INTROSORT as  $\mathcal{O}(n \cdot \log(n))$ .



**5. Discuss why we use heap sort rather than another  $O(n \log n)$  sorting algorithm such as merge sort.**

Both MERGESORT and HEAPSORT sort the input in  $\mathcal{O}(n \cdot \log(n))$ . But in contrast to MERGESORT, HEAPSORT is an in-place algorithm that does not need an extra space and produces an output in the same memory by swapping the elements merely by changing the pointer(s). The same goes for QUICKSORT.

Therefore, HEAPSORT is more preferable to other sorting algorithms that require extra cache for creating temporary subsidiary lists for copying the elements.

**6. Why is it a good idea to run insertion sort on the nearly sorted data, when we know from CLRS that its worst-case running time is  $(n^2)$ ?**

It is correct that the worst-case running time of INSERTIONSORT is  $\mathcal{O}(n^2)$ . But that holds for the sorted (or nearly) sorted data in decreasing order.

However, if the data has already been (almost) sorted in increasing order, then it would max take  $\mathcal{O}(n)$  for INSERTIONSORT to sort the data.

If we take the example of the use of INSERTIONSORT in INTROSORT, then we can see that the number of unsorted elements is so small, that it almost takes constant time for INSERTIONSORT to sort the elements.