Exercise Sheet 3
COMS10007 Algorithms 2018/2019

26.03.2019

Reminder: $\log n$ denotes the binary logarithm, i.e., $\log n = \log_2 n$.

1 Countingsort and Radixsort

1. Illustrate how Countingsort sorts the following array:

4	2	2	0	1	4	2

See slides of lecture 12.

2. Illustrate how Radixsort sorts the following binary numbers:

100110 101010 001010 010111 100000 000101

100110		100110		10000**0**		100**0**00		10**0**000		1**0**0000		**0**00101
101010		101010		00010**1**		101**0**10		00**0**101		0**0**0101		**0**01010
001010		001010		10011**0**		001**0**10		10**0**110		1**0**0110		**0**10111
010111	$\rightarrow$	100000	$\rightarrow$	10101**0**	$\rightarrow$	000**1**01	$\rightarrow$	01**0**111	$\rightarrow$	1**0**1010	$\rightarrow$	**1**00000
100000		010111		00101**0**		100**1**10		10**1**010		0**0**1010		**1**00110
000101		000101		01011**1**		010**1**11		00**1**010		0**1**0111		**1**01010

3. Radixsort sorts an array A of length n consisting of d-digit numbers where each digit is from the set $\{0, 1, \ldots, b\}$ in time $O(d(n + b))$.

   We are given an array A of n integers where each integer is *polynomially bounded*, i.e., each integer is from the range $\{0, 1, \ldots, n^c\}$, for some constant c. Argue that Radixsort can be used to sort A in time $O(n)$.

   Hint: Find a suitable representation of the numbers in $\{0, 1, \ldots, n^c\}$ as d-digit numbers where each digit comes from a set $\{0, 1, \ldots, b\}$ so that Radixsort runs in time $O(n)$. How do you chose d and b?

We encode the numbers in A using digits from the set $\{0, 1, \ldots, n-1\}$, i.e., we set $b = n - 1$. To be able to encode all numbers in the range $\{0, 1, \ldots, n^c\}$ it is required that $(b+1)^d \geq n^c+1$ (we can encode $(b+1)^d$ different numbers using d digits where each digit comes from a set of cardinality $b+1$, and the cardinality of the set $\{0, 1, \ldots, n^c\}$ is $n^c + 1$). Since $(b+1)^d = n^d$, we can set $d = c + 1$, since

$$n^{c+1} \geq n^c + 1$$

holds for every $n \geq 2$ (assuming that $c \geq 1$). The runtime then is

$$O(d(n+b)) = O((c+1)(n+(n-1))) = O((c+1)2n) = O(n) \ ,$$

since 2 and $c+1$ are both constants.

2 Loop Invariant for Radixsort

Radixsort is defined as follows:

Require: Array A of length n consisting of d-digit numbers where each digit is taken from the set $\{0, 1, \ldots, b\}$

1: **for** $i = 1, \ldots, d$ **do**
2: Use a stable sort algorithm to sort array A on digit i
3: **end for**

(least significant digit is digit 1)

In this exercise we prove correctness of Radixsort via the following loop invariant:

At the beginning of iteration i of the for-loop, i.e., after i has been updated in Line 1 but Line 2 has not yet been executed, the following holds:

The integers in A are sorted with respect to their last $i-1$ digits.

1. *Initialization:* Argue that the loop-invariant holds for $i = 1$.

   In the beginning of the iteration with $i = 1$ the loop-invariant states that the integers in A are sorted with respect to their last $i - 1 = 0$ digits. This is trivially true.

2. *Maintenance:* Suppose that the loop-invariant is true for some i. Show that it then also holds for $i + 1$.

Suppose that the integers in A are sorted with respect to their last $i-1$ digits at the beginning of iteration i. We will show that at the beginning of iteration $i+1$ the intergers are sorted with respect to their last i digits.
Let A_{i+1} be the state of A in the beginning of iteration $i+1$. For an integer x, let $x^{(i)}$ be the integer obtained by removing all but the last i digits from x. Suppose for the sake of a contradiction that there are indices j, k with $j < k$ such that $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$. If such integers exist then the loop invariant would not hold. We will show that assuming that these integers exist leads to a contradiction.
First, suppose that digit i of $(A_{i+1}[j])^{(i)}$ and digit i of $(A_{i+1}[k])^{(i)}$ are identical. Note that this implies $(A_{i+1}[j])^{(i-1)} > (A_{i+1}[k])^{(i-1)}$. Observe that in iteration i, the digits are sorted with respect to digit i. Since the subroutine employed in Radixsort is a stable sort algorithm, the relative order of the two numbers has not changed since their ith digits are identical. This implies that the relative order of the two numbers was the same at the beginning of iteration i. This is a contradiction, since the loop invariant at the beginning of iteration i states that the digits are sorted with respect to their $i-1$ last digits, however, $(A_{i+1}[j])^{(i-1)} > (A_{i+1}[k])^{(i-1)}$ holds.
Next, suppose that digit i of $(A_{i+1}[j])^{(i)}$ and digit i of $(A_{i+1}[k])^{(i)}$ are different. Then, since $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$ we have that digit i of $(A_{i+1}[j])^{(i)}$ is necessarily larger than digit i of $(A_{i+1}[k])^{(i)}$. This however is a contradiction to the fact that the numbers were sorted with respect to their ith digit in iteration i.
Hence, the assumption that there are indices j, k such that $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$ is wrong. If no such indices exist then the integers in A are sorted with respect to their last i digits at the beginning of iteration $i+1$.

Hint: You need to use the fact that the employed sorting algorithm as a subroutine is stable.

3. *Termination:* Use the loop-invariant to conclude that A is sorted after the execution of the algorithm.

After iteration d (or before iteration $d+1$, which is never executed), the invariant states that the numbers in A are sorted with respect to their last d digits, which simply means that all numbers are now sorted with regards to all their digits.

3 Recurrences: Substitution Method

1. Consider the following recurrence:

$$T(1) = 1 \text{ and } T(n) = T(n-1) + n$$

Show that $T(n) \in O(n^2)$ using the substitution method.

Proof. We will show that $T(n) \leq c \cdot n^2$, for some integer c whose value we'll determine later.
We first substitute our guess into the recurrence and obtain:

$$T(n) = T(n-1) + n \leq c \cdot (n-1)^2 + n = cn^2 - 2cn + c^2 + n \ .$$

It is required that $-2cn + c^2 + n \leq 0$ for our guess to hold. This is equivalent to $n(2c-1) \geq c^2$. We select $c = 1$ and obtain $n \geq 1$, which always holds.
Next, we need to show that the choice $c = 1$ works as well for the base case. We have $T(1) = 1$ and $c1^2 = 1 \cdot 1^2 = 1$, and the base case $n = 1$ holds too.
We have thus proved that $T(n) \leq n^2$ for every $n \geq 1$, which implies $T(n) \in O(n^2)$. □

2. Consider the following recurrence:

$$T(1) = 1 \text{ and } T(n) = T(\lceil n/2 \rceil) + 1$$

Show that $T(n) \in O(\log n)$ using the substitution method.

Hint: Use the inequality $\lceil n/2 \rceil \leq \frac{n}{\sqrt{2}} = \frac{n}{2^{\frac{1}{2}}}$, which holds for all $n \geq 2$. Use $n = 2$ as your base case.

Proof. We will prove that $T(n) \leq c \cdot \log n$, for some constant c and $n \geq 2$.
We first substitute our guess into the recurrence:

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + 1 \leq c \cdot \log(\lceil n/2 \rceil) + 1 \\ &\leq c \cdot \log\left(\frac{n}{\sqrt{2}}\right) + 1 = c\log(n) - c\log(\sqrt{2}) + 1 = c\log(n) - \frac{1}{2}c + 1 \ . \end{aligned}$$

Observe that $-\frac{1}{2}c + 1 \leq 0$ for $c \geq 2$. Chosing such a c, we obtain $T(n) \leq c \log n$ as required.
Last, we verify the base case $n = 2$. We have $T(2) = T(1) + 1 = 2$ and $c\log(2) = c$. We can hence chose $c = 2$ and both the base case and the induction step hold. Hence, we have proved $T(n) \leq 2 \log n$ for every $n \geq 2$. This implies $T(n) \in O(\log n)$. □

4 Recurrences: Recursion Tree Method

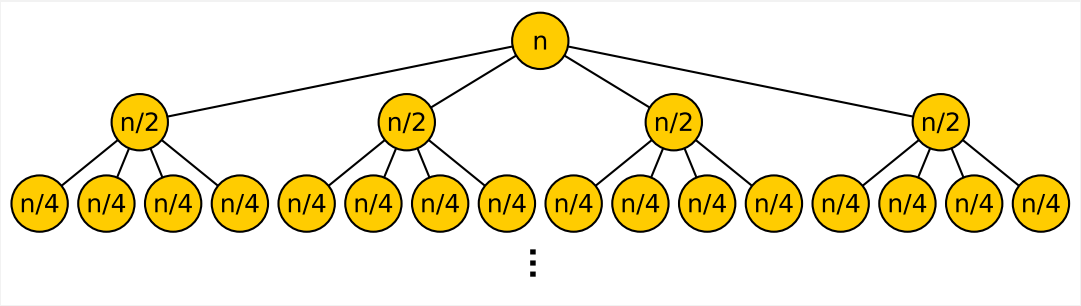
1. Use a recursion tree to determine a good asymptotic upper bound on the recurrence

$$T(n) = 1 \text{ for every } n \leq 10 \text{ and } T(n) = 4T(n/2 + 2) + n \text{ for every } n > 10 \ .$$

Use the substitution method to verify your answer. (this is a difficult question!)

Hint: Ignore the additive 2 for a rough analysis using the recursion tree. For the substitution method, use at least one lower order term.

Recursion Tree: We ignore the additive 2 and consider the recursion tree of the recurrence $T(n) = 4T(n/2) + n$:



We can see that the tree has less than $\log n$ levels, since the parameter n is halved from one level to the next, and we stop as soon as we have values of $n \leq 10$. We also see that the total work in layer i is $n2^{i-1}$. Our guess is thus:

$$\sum_{i=1}^{\log n} n2^{i-1} \quad = \quad n \sum_{i=1}^{\log n} 2^{i-1} = n \sum_{i=0}^{\log(n)-1} 2^i = n\left(2^{\log n} - 1\right) \leq n2^{\log n} = n^2 \ ,$$

where we used the equality

$$\sum_{i=0}^{j} 2^i = 2^{j+1} - 1 \ .$$

We will hence prove $T(n) \in O(n^2)$ in the following using the substitution method. (continued on next page...)

First Attempt: $T(n) \leq c \cdot n^2$
Plugging this guess into the recurrence gives:

$$T(n) = 4T(n/2+2)+n \leq 4c(n/2+2)^2+n = 4c(n^2/4+2n+4)+n = cn^2+8cn+16c+n\ .$$

Observe that the summand cn^2 is exactly what we need, however, since $8cn+16c+n$ is never ≤ 0 (we can only choose a positive c, since otherwise cn^2 would also be negative), our guess did not work out. We need to consider a lower order term.

Second Attempt: $T(n) \leq c \cdot n^2 + d \cdot n$ (d could as well be negative here). We obtain:

$$\begin{aligned} T(n) &= 4T(n/2+2)+n \leq 4c(n/2+2)^2 + 4d(n/2+2) + n \\ &= 4c(n^2/4+2n+4) + 2dn + 8d + n = cn^2 + \underbrace{8cn + 8c + 2dn + 8d + n}_{I}\ . \end{aligned}$$

We require that part I is at most 0. Hence:

$$\begin{aligned} 8cn + 8c + 2dn + 8d + n &\leq 0 \\ n\underbrace{(8c + 2d + 1)}_{A} + \underbrace{8c + 8d}_{B} &\leq 0\ . \end{aligned}$$

The rest of the proof is rather technical and complicated and may require a bit of work to verify the details:
For part B to be bounded by at most 0 we need to select $d \leq -c$. For part A to be bounded by at most 0 we obtain:

$$\begin{aligned} 8c + 2d + 1 &\leq 0 \\ d &\leq \frac{-1-8c}{2} = -4c - \frac{1}{2}\ . \end{aligned}$$

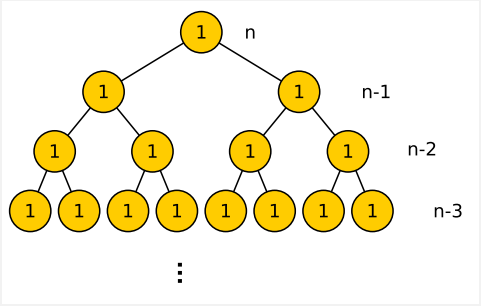
The condition $d \leq -4c - \frac{1}{2}$ is stronger than $d \leq -c$. For convenience, we will chose the even stronger choice $d = -4c - 4 = -4(c+1)$.
It remains to verify the base case and select a value for c on the way.
We have $T(n) = 1$ for every $1 \leq n \leq 10$. It is enough to prove that our guess is an upper bound on $T(n)$ for every $7 \leq n \leq 10$, since the smallest value on which we invoke the recurrence is larger than 10, and the recursive call is on a parameter $\geq \frac{10}{2} + 2 = 7$. We will select c and d such that $cn^2 + dn = cn^2 - 4(c+1)n \geq 1$, for every $7 \leq n \leq 10$.
Observe that $-4(c+1)n \geq -4(c+1)10 = -40(c+1)$ for $7 \leq n \leq 10$. Furthermore, $cn^2 \geq 49c$, for every $7 \leq n \leq 10$. We thus need to select a c such that $49c-40(c+1) \geq 1$. This yields $9c \geq 41$ or $c \geq 41/9$. We can hence select for example $c = 5$.
Recall that $d = -4(c+1) = -24$. We have thus proved that $T(n) \leq 5n^2 - 24n$ for every $n \geq 7$. This implies $T(n) = O(n^2)$. □

2. Use a recursion tree to determine a good asymptotic upper bound on the recurrence

$$T(1) = 1 \text{ and } T(n) = 2T(n-1) + 1\ .$$

Use the substitution method to verify your answer.

Recursion Tree: The recursion tree looks as follows:



We can see that in level i, the total work is 2^{i-1}. Furthermore, the tree has n levels. Our guess is thus:

$$\sum_{i=1}^{n} 2^{i-1} = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \ .$$

This guess is in fact *exact*, i.e., we have already precisely determined the value of the recurrence, i.e., $T(n) = 2^n - 1$. Nevertheless, we will verify this next using the substitution method.

First Attempt: We first try the guess $T(n) \leq c \cdot 2^n$:

$$T(n) = 2T(n-1) + 1 = 2c \cdot 2^{n-1} + 1 = c2^n + 1 \ .$$

We can see that using this guess we obtain an additive 1 that should not be here. A guess that works is as follows (which is little surprising):

Second Attempt: $T(n) \leq c \cdot 2^n - 1$:

$$T(n) = 2T(n-1) + 1 = 2(c \cdot 2^{n-1} - 1) + 1 = c2^n - 1 \ .$$

Last, to verify the base case, observe that $T(1) = 1$ and $c2^1 - 1 = 2c - 1$. We can hence select any $c \geq 1$ so that $2c - 1 \geq 1$. We thus pick $c = 1$.

We have proved that $T(n) \geq 2^n - 1$ which implies $T(n) \in O(2^n)$. □

5 Fibonacci Numbers

Consider the algorithm ImprovedDynPrgFib(n) for computing the Fibonacci numbers as presented on slide 13 of Lecture 15. In this exercise, the goal is to prove that the algorithm indeed computes the nth Fibonacci number.

1. Give a suitable loop invariant (it should involve at least variables a and b).
2. Prove that the loop invariant is correct: It holds at the beginning of the algorithm, it is maintained throughout the algorithm, and we can conclude from the loop invariant that the algorithm indeed computes the nth Fibonacci number.

I won't provide a solution to this exercise. I am curious to see your solutions.