Date:

Design and Analysis of Algorithms - Problem Sheet 1

1.

a) If the problem has size n, then on computer A would take n^2 seconds, and on computer B it would take $\frac{10^n}{10^{12}}$ seconds. Now we want to find out when $\frac{10^n}{10^{12}} < n^2$.

$$\frac{10^n}{10^{12}} < n^2 \iff$$
$$10^n < 10^{12} \cdot n^2 \iff$$
$$\log 10^n < \log(10^{12} \cdot n^2) \iff$$
$$n < 12 + \log(n^2) \iff$$
$$n < 12 + 2 \log n$$

So $n \leq 14$ it would be better to use implementation B on B, and when $n > 14$, then it would be better to use A on A.

b) Using A on A it would take 30^2 seconds, and using B on B it would take $\frac{10^{30}}{10^{12}} = 10^{18}$ seconds.

2.

From the definition we know that there $\exists c$ and n_0, such that for all $n \geq n_0$ $f(n) \leq c g(n)$. So we can choose a to be c and b we can choose it to be $1 + \max\{f(n), \text{ for } n < n_0\}$.

3.

a) $f = O(g)$ for $c = 2$, $n \geq 300$
$f = \Omega(g)$ for $c = 1$, $n \geq 0$
$\Rightarrow f = \Theta(g)$ for $n \geq 300$

b) $f = O(g)$, for $c = 1$ and $n \geq 1$

f is not lower bounded by $g \Rightarrow f$ is not asymptotically tight bounded by g.

c) ~~$f = O(g)$ for $c = 100$ and $n \geq 1$.~~
~~f is not upper bounded by g~~

let's assume $f = O(g)$ we have to find c and n_0:

$f \leq cg$ for $n \geq n_0 \iff$

$100n + \log n \leq c(n + (\log n)^2) = cn + c\log^2 n \iff$

$(100 - c)n \leq \log n(c\log n - 1)$

For $c > 100$ the LHS is negative and for $n > 10$ the RHS is positive $\Rightarrow c = 101$, $n_0 = 10$

g is not a lower bound for $f \Rightarrow g$ is not an asymptotically tight bound for f.

d) $f = O(g)$ for $c = 1$, $n \geq 1$

Let's assume that g is a lower bound for $f \Rightarrow \exists c, n_0$:

$\forall n \geq n_0$: $f(n) \geq cg(n)$

$n\log n \geq c(10n\log 10n)$ $\quad / : n$

$\log n \geq 10c(\log 10 + \log n)$

$\log n(1 - 10c) \geq 10c$, which is true for $c = 0{,}05$ and $n = 100$

$\Rightarrow f = \Theta(g)$

e) $f = O(g)$ for $n \geq 1$ and $c = 1$

$f = \Omega(g)$ for $c = \frac{1}{2}$ and $n \geq 10$

$\Rightarrow f = \Theta(g)$

Date:

f) If $f = O(g)$, there is c and n_0 such that for all $n \geq n_0$ $f(n) \leq c\,g(n)$ is true. let's assume that such c and n_0 exist.

$$\sqrt[10]{n} \leq c(\log n)^{10} \iff \text{(both sides are positive)}$$

$$n \leq c^{10}(\log n)^{100} \iff$$

$$\log_{100} n \leq \log_{100}\left(c^{10}(\log n)^{100}\right) \iff$$

$$\tfrac{1}{2}\log n \leq \log_{100} c^{10} + \log n \iff$$

$$\tfrac{1}{2}\log n \leq \tfrac{1}{2}\cdot 10 \log c + \log n \iff$$

$$0 \leq 5\log c + \tfrac{1}{2}\log n$$, which is true for $c = 1$, $n_0 = 10$

let's assume that $f = \Omega(g) \Rightarrow$ there exist c and n_0 such that for every $n \geq n_0$ $f(n) \geq c\,g(n)$ is true.

$$\sqrt[10]{n} \geq c(\log n)^{10} \iff$$

$$-5\log c \geq \tfrac{1}{2}\log n$$, but $\log n$ can't be bounded

$\Rightarrow f$ is not assymptotically lower bounded by g

$\Rightarrow f \neq \Theta(g)$

g) let's assume $f = O(g)$. We want to find c and n_0:

$\forall n \geq n_0 \quad f(n) \leq c\,g(n)$

$$n^{\frac{1}{2}} \leq c(\log n)^3 \iff \text{(both sides are positive)}$$

$$n \leq c^2(\log n)^6 \iff$$

$$\log_6 n \leq \log_6\left(c^2(\log n)^6\right) \iff$$

$$\log_6 n \leq 2\log_6 c + \log n$$

For big enough n, $\log_6 n > \log n$ no matter how big c is

$\Rightarrow f$ is not $O(g)$.

let's assume $f = \Omega(g) \Rightarrow \forall n \geq n_0 : f(n) \geq c\,g(n)$

$$\sqrt{n} \geq (\log n)^3$$

$$n \geq (\log n)^6$$

$\log_6 n \geq \log n$, which

is true for all $n > 10$. $\Rightarrow f \neq \Theta(g)$

h) Let's assume $f = O(g)$ $\Rightarrow$ ~~for~~ if $c = 1$, then for every $n \geq n_0$: $n2^n \leq 3^n$ is true

$n \leq \left(\frac{3}{2}\right)^n$, which is true for $n > 1$.

$\Rightarrow f = O(g)$ for $c = 1$, $n_0 = 1$.

f is not $\Omega(g)$ because $\frac{n2^n}{3^n} \geq c$ would have to be true.

$n \cdot \left(\frac{2}{3}\right)^n \geq c$, but because $\left(\frac{2}{3}\right) < 1$, the LHS will converge to 0.

$\Rightarrow f$ is not assymptotically lower bounded by g

$\Rightarrow f$ is not $\Theta(g)$.

i) If $c = 1$, then 2^n is obviously smaller than 2^{n+1} for $n > 0$. $\Rightarrow f = O(g)$.

It is also obvious that $2^n \geq \frac{1}{4} \cdot 2^{n+1}$ for $n > 0$.

$\Rightarrow f = \Omega(g) \Rightarrow f = \Theta(g)$

j) If $c = 1$ $\Rightarrow$ we need to prove $(\log n)^{\log n} \leq \left(2^{\log n}\right)^{\log n}$ for big enough n $\Leftrightarrow$ because we can assume $\log n > 1$, then the inequality is equivalent to prove that $\frac{2^{\log n}}{\log n} > 1$.

$\Leftrightarrow \frac{\log n}{2^{\log n}} < 1$

$$\frac{d}{dn}\left(\frac{\log n}{2^{\log n}}\right) = \frac{-\ln(2) \cdot \log(n) + 1}{\ln(10) \cdot 2^{\log_{10}(n)}\, n}$$

But $-\ln(2) \cdot \log(x) + 1 < 0$ for big enough n and $\ln(10) \cdot 2^{\log n} \cdot n > 0$ $\Rightarrow \frac{\log n}{2^{\log n}}$ will converge to 0

$\Rightarrow \frac{\log n}{2^{\log n}} < 1$ for big enough n.

Date:

Let's assume $f = \Omega(g) \Rightarrow$
$\left(\frac{\log n}{2\log n}\right)^{\log n} \geq c$ for some $c > 0$ and $n \geq n_0$.

From the previous result we know $\frac{\log n}{2\log n}$ converges
$\Rightarrow$ there is no such c that the inequality is true
for all $n \geq n_0$. $\Rightarrow f$ is not $\Omega(g)$
$\Rightarrow f$ is not $\Theta(g)$

4.

$\log(n!) = \log(n.(n-1)(n-2)\dots 1) = \log n + \log(n-1) + \dots + \dots + \log 1 \leq n.\log n.1$

$\Rightarrow \log(n!) = O(n\log n)$ for $c=1$, $n \geq 10$

$\log(n!) = \sum_{i=1}^{n} \log i > \sum_{i=\frac{n}{2}}^{n} \log i \geq \frac{n}{2} \cdot \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2)$

$\Rightarrow \log(n!) = \Omega(n\log n) \Rightarrow \log(n!) = \Theta(n\log n)$

5.

a) I am going to prove that $f_k = O(n^k)$ by induction on k.
For the base we have $f_0 = O(n^0) = O(1)$, which is true.
For the induction hypothesis $f_{k-1} = O(n^{k-1})$. I am going to prove that $f_k = O(n^k)$.

From question 2 we have that if $f = O(n^k)$, then there are $a, b > 0$, such that $f(n) \leq an^k + b$ for all $n \geq 0$.
$\Rightarrow f_k(n) \leq f_k(n-1) + f_{k-1}(n) \underset{IH}{\leq} f_k(n-1) + \sum_{i=1}^{n} ai^{k-1} + b = O(n^k)$

b) I am going to prove it by induction on k. For the base case $g_0 = \Omega(n^0) = \Omega(1)$, which is true. My induction hypothesis

NOTEBOOK Date:

is that $g_{k-1} = \Omega(n^{k-1})$ => by definition for Big-Omega there is c such that $g_{k-1}(n) \geq cn^{k-1}$.

$$\Rightarrow g_k(n) \geq g_k(n-1) + g_{k-1}(n) \geq g_k(0) + \sum_{i=1}^{n} ci^{k-1} \geq$$

$$\geq c \sum_{i=\frac{n}{2}+1}^{n} i^{k-1} \geq c \cdot \frac{n}{2} \cdot \left(\frac{n}{2}\right)^{k-1} = \Omega(n^k).$$

4.

a) Let the numbers be a_1, a_2, a_3, a_4. We can compare a_1 and a_2, a_3 and a_4. Then we compare the two bigger integers to find the largest number and compare the two smaller ones to find the smallest number.

b) Let the numbers be $a_1, a_2, \dots, a_n$. I am going to prove that to find the largest number I need $2^k - 1$ comparisons (2^k is the number of numbers).

Base case: $k = 0$ -> true

For the inductive hypothesis: I need $2^k - 1$ comparisons to find the biggest number.

Let the numbers be 2^{k+1}. We need to do 2^k comparisons to find the larger from a_i and a_{i+1} (i - odd). From IH, another $2^k - 1$ to find the largest number.

=> in total $2^k - 1 + 2^k = 2^{k+1} - 1$ comparisons.

The same is true for finding the smallest number

=> to find the biggest and smallest number first we need to do $\frac{n}{2}$ comparisons to devide them into two groups - „bigger" and „smaller" and in each of

Date:

those groups do another $\frac{n}{2}-1$ comparisons to find respectively the biggest and smallest number => in total $3\cdot\frac{n}{2}-2$ comparisons.

6.

a) $T(n) \le 2T(n-1)+n$

$$T(n) \le 2T(n-1)+n \le 2(2T(n-2)+n-1)+n \le 2(2(2T(n-3)+n-2)+n-1)+n \le \dots$$

$$\le 2^k T(n-k) + \sum_{j=0}^{k-1} 2^j(n-j) \underset{k=n-1}{=}$$

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

~~$2^{n-1}T(1) = O(2^{n-1})$, because $T(1)=1$~~

$$\sum_{j=0}^{n-2} 2^j n - \sum_{j=0}^{n-2} 2^j j = n(2^{n-1}-1) - \frac{n\cdot 2^n - 3\cdot 2^n + 4}{2} =$$

$$= 3\cdot 2^{n-1} - n - 2 \Rightarrow$$

$$T(n) \le 2^{n-1} + 3\cdot 2^{n-1} - n - 2 = 2^{n+1} - n - 2 = O(2^{n+1})$$

b) $T(n) \le T(\frac{n}{2}) + n\log n$

$$T(n) \le T(\tfrac{n}{2}) + n\log n \le T(\tfrac{n}{4}) + \tfrac{n}{2}\log\tfrac{n}{2} + n\log n \le \dots \le$$

$$T(1) + \sum_{j=0}^{\log_2 n} \frac{n}{2^j}\log\frac{n}{2^j} =$$

$$= 1 + \sum_{j=0}^{\log_2 n} \frac{n}{2^j}\log n - \sum_{j=0}^{\log_2 n} \frac{n}{2^j}\cdot\log 2^j =$$

$$= 1 + n\log n \sum_{j=0}^{\log_2 n} \frac{1}{2^j} - n\log 2 \sum_{j=0}^{\log_2 n} \frac{j}{2^j} = O(n\log n)$$

NOTEBOOK

Date:

c) $T(n) \leq T(n-1) + 3n^2$

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

$\leq T(n-3) + 3(n-2)^2 + 3(n-1)^2 + 3n^2 \leq \dots \leq$

$\leq T(1) + 3\sum_{j=0}^{n-2} (n-j)^2 = \frac{\cancel{3}(n-2)(n-1)(2(n-2)+1)}{\cancel{6}\,2} =$

$= \frac{(n-2)(n-1)(2n-3)}{2} + 1 = O(n^3)$

d) $T(n) \leq 2T(\frac{n}{2}) + n^2$

From the master theorem $b = 2, a = 2, d = 2 \Rightarrow d > \log_b a$

$\Rightarrow T(n) = O(n^2)$

8.

At each iteration, the "ternary" search algorithm has to do 4 things: 1) test x for equality with the element at position $\frac{n}{3}$, 2) test for equality with element at position $\frac{2n}{3}$, 3) test if x is bigger than $arr(\frac{x}{3})$, 4) test if x is bigger than $arr(\frac{2x}{3})$.

$\Rightarrow T_1(n) = T_1(\frac{n}{3}) + 4$

Binary search has to 1) test x for equality with $arr(\frac{n}{2})$, 2) test if x is bigger than $arr(\frac{n}{2})$.

$\Rightarrow T_2(n) = T_2(\frac{n}{2}) + 2$

$T_1(1) = T_2(1) = 1 \Rightarrow T_1(n) = 4\log_3 n + 1$

$T_2(n) = 2\log_2 n + 1$

Date:

We have to compare $4\log_3 n$ and $2\log_2 n$

$$4\log_3 n \geq 2\log_2 n \iff$$
$$2\log_3 n \geq \log_2 n \iff$$
$$2\frac{1}{\log_n 3} \geq \frac{1}{\log_n 2} \iff$$
$$2 \geq \frac{\log_n 3}{\log_n 2} \iff$$

$$2 \geq \log_2 3$$, which is true

=> binary search is more efficient than „ternary" search.

9.

Let the first array be arr1, and the second one - arr2.
mid1 is going to be middle element of arr1, and mid2 - the middle element of arr2.

If mid1 + mid2 is less than n

1) arr1(mid1) > arr2(mid2), then we can throw out the first half of arr2. Reduce n by (mid2+1).

2) arr1(mid1) < arr2(mid2), we can throw away the first half of arr1. Reduce n by (mid1+1).

If mid1 + mid2 > n

1) arr1(mid1) > arr2(mid2), throw away second half of arr1.

2) arr1(mid1) < arr2(mid2), throw away second half of arr2.

Stop when one of the arrays is empty The answer is the n-th element of the other array.

10.

We can find the minimal element with binary search. Then check with the first elements of the two arrays (possibly only one array) in which array to search x.

Then do normal binary search in whichever array.

11.

We can first sort the array with merge sort. Then for every element search with binary search if there is an element that is equal to $x - A[i]$ ($A[I]$ is the element that is being searched.)

12.

To find the inversion, we will use algorithm which is similar to merge sort. We will divide the problem until we get to the individual elements of the array. When we start merging the subarrays, for example $a[0..j]$ and $b[0..j]$ (*), we do it like merge sort compare the heads of the two arrays. But when $b(i) < a(t)$, we add $j-t+1$ to the counter of inversions, because the indexes of the elements of a are smaller than the indexes ~~of b~~ from the original array of b => whenever an element from a is bigger than an element in b, it is an inversion.

(*) $a[0..j]$ and $b[0..j]$ are sorted.