DESIGN AND ANALYSIS OF ALGORITHMS

GABRIEL MOISE

HT 2018

PROBLEM SHEET 1

Big-O and asymptotic notations

Question 1

$a(n) = 10^6 n^2$

$b(n) = 10^n$

Computer A $\to 10^6$ operations / second

Computer B $\to 10^{12}$ operations / second

Algorithm α – problem $P \xrightarrow{\text{worst case}} a(n)$ on A

Algorithm β – problem $P \xrightarrow{\text{worst case}} b(n)$ on B

(a) Let $T(A,n)$ be the time needed for the algorithm used for P on computer A in the worst case scenario, depending on the size n of the problem and $T(B,n)$, similarly. Then:

$$T(A,n) = \frac{a(n)}{10^6} = \frac{10^6 n^2}{10^6} = n^2 \text{ seconds}$$

$$T(B,n) = \frac{b(n)}{10^{12}} = \frac{10^n}{10^{12}} = 10^{n-12} \text{ seconds}$$

Now, by comparing them, we observe that for $n \le 14$, we have $T(A,n) > T(B,n)$ and for $n > 14$, $T(A,n) < T(B,n)$. The point here is that $n^2 = O(n^2)$ and $10^{n-12} = \Omega(10^n)$, and since $n^2 = O(10^n)$, we get that $n^2 = O(10^{n-12})$. It can also be seen as $n^2 \le 1 \cdot 10^{n-12}$ from $n \geq 15$.

Therefore, for $n \le 14$ we choose the algorithm β and for $n > 14$, the algorithm α.

(b) $n = 30 \Rightarrow T(A,30) = 30^2 = 900$ seconds $= 15$ minutes

$T(B,30) = 10^{18}$ seconds $\approx 31,668,087,814$ years

* **Question 2**

$k \in \mathbb{N}_+$

$f = O(n^k) \Rightarrow (\exists)\, c \in \mathbb{R}\ (\exists)\, N \in \mathbb{N}$ such that $f(n) \le c \cdot n^k\ (\forall)\, n \geq N$.

Let $a \geq c$, $b = \left|\max\{f(n) \mid 0 \le n < N\}\right| \Rightarrow f(n) \le b \le a \cdot n^k + b\ (\forall)\, n \in \{0, 1, \dots, N-1\}$ and

and $f(n) \le c \cdot n^k \le a \cdot n^k \le a \cdot n^k + b\ (\forall)\, n \geq N \Rightarrow f(n) \le an^k + b\ (\forall)\, n \in \mathbb{N}$

For us to be sure that $a, b > 0$ we can choose $a = 1 + c > 0$

$b = 1 + \left|\max\{f(n) \mid 0 \le n < N\}\right| > 0$

Question 3

	$f(n)$	$g(n)$	$f=O(g)$	$f=\Omega(g)$	$f=\Theta(g)$
a.	$n-100$	$n-200$	YES	YES	YES
b.	$n^{1/2}$	$n^{2/3}$	YES	NO	NO
c.	$100n+\log n$	$n+(\log n)^2$	YES	YES	YES
d.	$n\log n$	$10n\log(10n)$	YES	YES	YES
e.	$\log(2n)$	$\log(3n)$	YES	YES	YES
f.	$n^{0.1}$	$(\log n)^{10}$	NO	YES	NO
g.	$\sqrt{n}$	$(\log n)^3$	No	YES	NO
h.	$n\cdot 2^n$	3^n	YES	NO	NO
i.	2^n	2^{n+1}	YES	YES	YES
j.	$(\log n)^{\log n}$	$2^{(\log n)^2}$	YES	NO	No

Question 4

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

To prove this, we'll use Stirling's approximation for $n!$:

$$n! = \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n\cdot\left(1+\Theta\left(\frac{1}{n}\right)\right),$$ which means that $(\exists)\, a,b>0$ and $n_0\in\mathbb{N}$ such that

$$\sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n\cdot\left(1+\frac{a}{n}\right)\le n! \le \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n\cdot\left(1+\frac{b}{n}\right)\quad (\forall)\, n\ge n_0 \quad \Big|\ \log(\,)$$

$$\underbrace{\underbrace{\log(\sqrt{2\pi})}_{=\Theta(1)}+\underbrace{\log(\sqrt{n})}_{=\frac{1}{2}\log n = \Theta(\log n)}+\underbrace{\log(n^n)}_{=\Theta(n\log n)}-\underbrace{\log(e^n)}_{=n=\Theta(n)}+\underbrace{\log\left(\frac{n+a}{n}\right)}_{\text{less than } n = \Theta(n)}}_{=\Theta(n\log n)}\le\log(n!)\le\underbrace{\underbrace{\log(\sqrt{2\pi})}_{\Theta(1)}+\underbrace{\log(\sqrt{n})}_{=\Theta(\log n)}+\underbrace{\log(n^n)}_{=\Theta(n\log n)}-\underbrace{\log(e^n)}_{=\Theta(n)}+\underbrace{\log\left(\frac{b+n}{n}\right)}_{=\Theta(n)}}_{\Theta(n\log n)}$$

Therefore $\log(n!) = \Theta(n\log n)$

Recurrences

* Question 5

(a) $f_0 = O(1)$, $k>0$, $n>0$

$$f_k(n)\le f_k(n-1)+f_{k-1}(n)$$

We will prove that $f_k = O(n^k)$ by induction on k:

Base case: $P(0)$: $f_0 = O(n^0) = O(1)$, which is true

inductive step: We suppose that $f_{k-1} = O(n^{k-1}) \overset{(Q2)}{\Rightarrow} \underbrace{f_{k-1}(n)\le a n^{k-1}+b,\ (\forall)\, n\ge 0}^{(\exists)\, a,b>0 \text{ such that}}$

$$f_k(n) \le f_k(n-1) + f_{k-1}(n) \le f_k(n-1) + an^{k-1} + b$$
$$f_k(n-1) \le f_k(n-2) + f_{k-1}(n-1) \le f_k(n-1) + an^{k-1} + b$$
$$\cdots\cdots\cdots\cdots$$
$$f_k(1) \le f_k(0) + f_{k-1}(1) \le f_k(0) + an^{k-1} + b$$
$$(+)$$
$$f_k(n) \le f_k(0) + an^k + bn \Rightarrow f_k = O(n^k)$$

(b) $g_0 = \Omega(1)$, $k>0, n>0$

$$g_k(n) \geq g_k(n-1) + g_{k-1}(n)$$

We will prove that $g_k = \Omega(n^k)$ by induction on k:

Base case: $P(0)$: $g_0 = \Omega(n^0) = \Omega(1)$, which is true

Inductive step: We suppose that $g_{k-1} = \Omega(n^{k-1}) \Rightarrow (\exists)\, c>0, n_0 \in \mathbb{N}$ such that $g_{k-1}(n) \geq c\cdot n^{k-1}$ $(\forall)\, n \geq n_0$. Then, for $n \geq n_0$ we have

$$g_k(n) \geq g_k(n-1) + g_{k-1}(n) \geq g_k(n-1) + c\cdot n^{k-1}$$
$$g_k(n-1) \geq g_k(n-2) + g_{k-1}(n-1) \geq g_k(n-2) + c\cdot n^{k-1}$$
$$\cdots\cdots\cdots\cdots$$
$$g_k(n_0+1) \geq g_k(n_0) + g_{k-1}(n_0+1) \geq g_k(n_0) + c\cdot n^{k-1}$$
$$(+)$$
$$g_k(n) \geq g_k(n_0) + (n-n_0)\cdot c\cdot n^{k-1} = \alpha + c\cdot n^k - c\cdot n_0 \cdot n^{k-1}, \text{ with } n_0 \le n \Rightarrow g_k = \Omega(n^k)$$

Question 6

$T(1) = 1$

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

We'll prove by induction on n that $T(n) \le \sum_{i=0}^{n-1} 2^i\cdot(n-i)$

Base case: $P(1)$: $T(1) \le \sum_{i=0}^{0} 2^i\cdot(1-i)$

$1 \le 2^0 \cdot 1$, true

Inductive step: We suppose that $P(n)$ is true and therefore $P(n+1)$ must also be true

$$T(n+1) = 2T(n)+n+1 \overset{P(n)}{\le} 2\cdot \sum_{i=0}^{n-1} 2^i\cdot(n-i) + n+1 = \sum_{i=0}^{n-1} 2^{i+1}\cdot(n-i) + (n+1) = \sum_{i=0}^{n} 2^i\cdot(n-i)$$

So, $T(n) \le \sum_{i=0}^{n-1} 2^i\cdot(n-i)$

As $2^i\cdot(n-i) = O(n\cdot 2^i)$, for $i \in \{0,1,\dots,n-1\}$ $\Big|\Rightarrow T(n) = O(n\cdot 2^{n-1})$

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

Let $k = \lfloor \log_2 n \rfloor \Rightarrow 2^k \le n < 2^{k+1} \xRightarrow{T \text{ is a non-decreasing function}} T(2^k) \le T(n) < T(2^{k+1})$

$$T(2^k) \le T(2^{k-1}) + 2^k \log 2^k = T(2^{k-1}) + k\cdot 2^k \log 2 \le T(2^{k-2}) + (k-1)\cdot 2^{k-1}\log 2 + k\cdot 2^k \log 2 \le \dots \le$$
$$T(1) + \left(\sum_{i=1}^{k} i\cdot 2^i\right)\log 2 = 1 + \log 2 \cdot \left(\sum_{i=1}^{k} i\cdot 2^i\right) \le 1 + \log 2 \cdot k\cdot 2^k = O(k\cdot 2^k)$$

Analogously, $T(2^{k+1}) = O((k+1)2^{k+1}) = O(2(k+1)2^k) = O(2k\cdot 2^k + 2^{k+1}) = O(k\cdot 2^k) \Rightarrow$

$\Rightarrow T(n) = O(k\cdot 2^k)$, where $k = \lfloor \log_2 n \rfloor \Rightarrow T(n) = O(n \log n)$

(c) $T(n) \leq T(n-1) + 3n^2 \leq T(n-2) + 3(n-1)^2 + 3n^2 \leq \ldots \leq T(1) + 3\sum_{i=2}^{n} i^2 = T(1) + 3\left(\frac{n(n+1)(2n+1)}{6} - 1\right) =$

$= 1 + 3\,\frac{n(n+1)(2n+1)}{6} - 3 = \frac{n(n+1)(2n+1)}{2} - 2 = n^3 + \frac{3}{2}n^2 + \frac{n}{2} - 2 = O(n^3) \Rightarrow T(n) = O(n^3)$

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

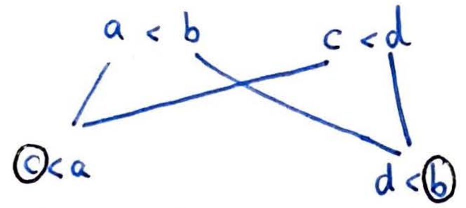
By using Master's Theorem, with $a=2, b=2, d=2$ (we have $\log_b a = \log_2 2 = 1 < 2 = d$), we get that $T(n) = O(n^2)$.

Comparison problems: Searching, sorting, selection

Question 7

(a) Let the 4 integers be a, b, c, d. We first compare them two by two: a with b and c with d. Then, to find the smallest element of the four, we compare the smallests elements from the two initial comparisons and we choose the smallest one here and this is the smallest of the group. We proceed in the same way for the biggest element with the comparison between the biggest elements of the two groups.

For example, if we had $c < a < d < b$, we'll have



(b) Let's call the procedure SMALLBIG. To get SMALLBIG(n), which returns the smallest and the biggest elements of a list of n elements, we apply SMALLBIG recursively for the two halves of the array and compare the results. Then, we get

SMALLBIG$(2^1) = 1$ ← base case (one comparison needed)

SMALLBIG$(2^k) = 2\cdot$ SMALLBIG$(2^{k-1}) + 2$

↑ the 2 halves ↑ 2 more comparisons needed

We'll prove by induction (on k) that SMALLBIG$(2^k) \leq \frac{3n}{2} - 2 = \frac{3\cdot 2^k}{2} - 2 = 3\cdot 2^{k-1} - 2$

Base case: P(1): SMALLBIG$(2^1) = 1 \leq 3\cdot 2^0 - 2 = 1$

Inductive step: We assume that P(k): SMALLBIG$(2^k) \leq 3\cdot 2^{k-1} - 2$ and we'll prove that SMALLBIG$(2^{k+1}) \leq 3\cdot 2^k - 2$

SMALLBIG$(2^{k+1}) = 2\cdot$ SMALLBIG$(2^k) + 2 \leq 2\cdot(3\cdot 2^{k-1} - 2) + 2 = 3\cdot 2^k - 4 + 2 = 3\cdot 2^k - 2$

Therefore, we proved that SMALLBIG$(n) \leq 3\frac{n}{2} - 2$, where $n = 2^k$, $k \geq 1$.

Question 8

The binary search algorithm does $B(n)$ steps to find if an element is in a sorted list or not, where n is the length of the list. For it, we have the recurrence:

$$B(1)=1$$
$$B(n)=1+B(n/2),\ (\forall)\ n>1$$

This algorithm does, in the worst-case scenario (when the element is not in the list, for example) $\lceil \log_2 n \rceil$ comparisons.

Now, we'll compare it to the worst-case scenario for "ternary" search. Let $T(n)$ the number of operations needed to solve a problem of size n with this algorithm. Then, if we want to find x, where x is bigger than all the elements from the list we are working with, then at each step we will do 2 comparisons and divide the problem by 3:

$$T(1)=1$$
$$T(2)=2$$
$$T(n)=2+T(n/3),\ (\forall)\ n>2$$

This algorithm does, in the worst-case scenario approximatively $2\cdot\lceil \log_3 n \rceil$. So, both algorithms are logarithmic. To compare them, we can say that $2\log_3 n = \log_{\frac{3}{2}} n$, which is bigger than $\log_2 n$, so the "ternary" search is quite slower than the "binary".

Question 9

We are given to sorted lists, A and B, each containing n elements and we want to find the n^{th} biggest element of the union of the list.

First of all, we notice that the n^{th} biggest element of the union of the two lists has the following property: it is bigger than k elements from list A (more precisely, the first k elements from A) and bigger than the first $(n-k-1)$ elements from list B, with k to be determined.

Why is that? The idea is that by concatenating the two lists with a merge function (the one from merge-sort) which keeps the elements in order, the n^{th} element of this list, the number we are interested about, is greater than exactly $(n-1)$ elements, k from A and $n-1-k$ from B, let's say.

The target here is to find k and then to compare $A[k+1]$ with $B[n-k]$ and the smaller one is the element we are looking for (because it's the next element in the union)

$A:\ A[1]\ A[2]\ A[3]\ \dots\ A[n-1]\ A[n]$

$B:\ B[1]\ B[2]\ B[3]\ \dots\ B[n-1]\ B[n]$

We will start by supposing that $k=n-1$ and looking "binarily" (same concept as in "binary search", which runs in $O(\log n)$ time) for the value of k for which $A[k]<B[n-k]$ and $A[k+1]>B[n-k-1]$ (if we get $A[0]$ or $B[0]$, we'll assume they are 0).

The interval on which we will search for k will always halve and we will choose the half of the interval depending on the comparisons we mentioned. Therefore, we will keep track of the left and right limits we have for k in the $\{0, 1, \dots, m-1\}$ range and uptade them accordingly. A program for the problem using this algorithm would look like:

```
INT search (INT l, INT r)
  { k = (l+r) / 2
    if ( A[k] > B[n-k]) {r=(l+r)/2; search(l,r)}
      else if (A[k+1] < B[n-k-1]) { l = (l+r)/2; search (l,r)}
           else if (A[k+1] < B[n-k]) return A[k+1]
                else return B[n-k] }

search(0,m)
```

Its complexity is $O(\log m)$ as at each step we reduce the interval we are searching for the result by half.

Question 10

$X = \langle x_0, x_1, \dots, x_{n-1} \rangle$ cyclically sorted

$(\exists)\ 0 \le j < n \quad (\forall)\ 0 \le i < n-1 \quad x_{(j+i) \bmod m} < x_{(j+i+1) \bmod m}$

First, we'll find the smallest element in X in $O(\log m)$ steps. We will do this with a binary search halving the interval (which initially is $[0, \dots, m-1]$) and continuing in the left half if $X[l] > X[half]$, or in the right one, otherwise. We know this does the right thing because we normally have only "<" between terms, except one ">" which is between $x[k-1]$ and $x[k]$, $x[k]$ being the element we are looking for.

After finding it, we proceed to search with "binary search" from that position (the left limit) to $k+n-1$ (the right limit), using the fact that $x[i] = X[i \bmod n]$ for all i. So, we basically need two "binary" searches, so the complexity of the algorithm is $O(\log m)$

Question 11

To find two elements from A, $A[i]$ and $A[j]$ that sum to z, we will try every element from A. sorted and search $z - A[i]$ with a "binary search".

The steps are:

1) Sort A - complexity $O(m \log m)$ with merge-sort
2) For $i=1$ to n - m times line 3) $\Rightarrow O(m \log m)$
3) Binary search $(z - A[i])$ in A $\rightarrow$ complexity $O(\log m)$

$\Rightarrow O(n \log m)$ complexity

Thus, we obtain, for some value $A[i]$, an $A[j] = z - A[i] \Rightarrow A[i] + A[j] = z$, as we wanted.
(or not, if it does not exist any pair $(A[i], A[j])$)

Question 12

The algorithm for counting the number of inversions of an array A, with n distinct numbers is simply the merge-sort algorithm after we add (in the function merge) that whenever an element from the left sub-array is greater that an element from the right sub-array, we add $(ln - pos + 1)$ to our variable that counts the number of inversions, where ln = length of the left sub-array, pos = the position in the left subarray where we are at the moment (with the checkings). This is because if an element from the left subarray, $L[pos]$ is greater than one from the right subarray, $R[pos']$ then all the elements $L[pos+1], L[pos+2], \ldots, L[ln]$ are also greater than $R[pos']$ and as they have smaller indexes than $R[pos']$ in the initial array A, they are all inversions.

This algorithm runs as fast as merge-sort, so it has $O(n \log n)$ steps, worst-case.

if A contains duplicates, the algorithm is the same, as the condition had a strict sign: $L(pos) > R(pos')$, so we count only the inversions.