Exercise sheet 1
Advanced Algorithms

Master Données et Systèmes Connectés

Master Machine Learning and Data Mining

Master Cyber-Physical Social Systems

Amaury Habrard

amaury.habrard@univ-st-etienne.fr

Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet Saint-Étienne
amaury.habrard@univ-st-etienne.fr

Semester 1

Exercise 1

Since f is $O(g)$, we have for some constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$:

$$f(n) \leq c \times g(n).$$

Dividing both sides by c, we get that $g(n) \geq \frac{1}{c} f(n)$ for all $n \geq n_0$.
So we have for all $n \geq n_0$ the existence of a constant $c' = 1/c$ such that

$$g(n) \geq c' \times f(n),$$

this implies that $g = \Omega(f)$.

Exercise 1 (ctd) - item 1

Since f is $O(g)$, there exist constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$: $f(n) \leq c \times g(n)$.

Since g is $O(h)$, there exist constants $c' > 0$ and $n'_0 \geq 0$ such that $\forall n \geq n'_0$: $g(n) \leq c' \times h(n)$.

Now when $n \geq \max(n_0, n'_0)$, we have:
$f(n) \leq c \times g(n) \leq c \times c' \times h(n)$

thus f is $O(h)$ using constants $n''_0 = \max(n_0, n'_0)$ and $c'' = c \times c'$.

The proofs for other claims are similar,

Exercise 1 (ctd) - item 2

Since f is $\Omega(g)$, there exist constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$: $f(n) \geq c \times g(n)$.

Since g is $\Omega(h)$, there exist constants $c' > 0$ and $n'_0 \geq 0$ such that $\forall n \geq n'_0$: $g(n) \geq c' \times h(n)$.

Now when $n \geq \max(n_0, n'_0)$, we have:
$f(n) \geq c \times g(n) \geq c \times c' \times h(n)$

thus f is $\Omega(h)$ using constants $n''_0 = \max(n_0, n'_0)$ and $c'' = c \times c'$.

The 3rd case with θ (item 3), is deduced by the two preceding cases

Exercise 1 (ctd) - item 4

Since f is $O(h)$, there exist constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$: $f(n) \leq c \times h(n)$.

Since g is $O(h)$, there exist constants $c' > 0$ and $n'_0 \geq 0$ such that $\forall n \geq n'_0$: $g(n) \leq c' \times h(n)$.

Now when $n \geq \max(n_0, n'_0)$, we have:
$f(n) + g(n) \leq c \times h(n) + c' \times h(n) = (c + c') \times h(n)$

thus $f + g$ is $O(h)$ using constants $n''_0 = \max(n_0, n'_0)$ and $c'' = c + c'$.

Exercise 1 (ctd) - item 5

Since g is $O(f)$, there exist constants $c > 0$ and $n_0 \geq 0$ that:
$g(n) \leq c \times f(n)$.

Then when $n \geq n_0$, we have:
$f(n) + g(n) \leq f(n) + c \times f(n) = (1 + c) \times f(n)$
thus $f + g$ is $O(f)$ using constants $n_0'' = n_0$ and $c'' = 1 + c$.

Also, for $n \geq n_0$, we have:
$f(n) + g(n) \geq f(n)$, since $g(n)$ is a positive function,
thus $f + g$ is $\Omega(f)$ using constants $n_0''' = n_0$ and $c''' = 1$.

$f + g$ is both $O(f)$ and $\Omega(f)$, thus $f + g$ is $\theta(f)$.

Exercise 2.1

This is false in general since it could be that $g(n) = 1$ for all n, $f(n) = 2$ for all n, and then $\log_2 g(n) = 0$, whence we cannot write $\log_2 f(n) \leq c \log_2 g(n)$.

On the other hand, if we simply require $g(n) \geq 2$ for all n beyond some n_1, then the statement holds. Since $f \in O(g)$ implies there exists[1] $c > 1, n_0 \geq 0$ s.t. $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, we have

$$\begin{aligned}
\log_2 f(n) &\leq \log_2(cg(n)) \\
&\leq \log_2(c) + \log_2(g(n)) \text{ by a property of } \log \\
&\leq \log_2(c) \times \log_2(g(n)) + \log_2 g(n) = (\log_2(c) + 1)(\log_2(g(n)))
\end{aligned}$$

once $n \geq max(n_0, n_1)$ since $\log_2 g(n) \geq 1$ beyond this point.

We have then found an index $n_0' = max(n_0, n_1)$ and a constant $c' = \log_2 c + 1$ allowing us to conclude that $\log_2(f) = O(\log_2(g))$.

[1] Note: we must also ensure $\log_2(c) \geq 0$ here, but we can assume $c > 1$ for $O(\cdot)$ relationships without loss of generality since even if a constant between 0 and 1 is considered at first, then a constant greater than one works as well.

Exercise 2.2

This is false: take $f(n) = 2n$ and $g(n) = n$.
Then $2^{f(n)} = 4^n$ while $2^{g(n)} = 2^n$.
If you assume that $2^f \in O(2^g)$, then there exists $c > 0$ and $n_0 \geq 0$ s.t. for every $n \geq n_0$: $4^n \leq c2^n$.
This implies that $2^n \leq c$ (or in other words $n \leq \log_2(c)$) which means that it cannot be true for every n greater than n_0, thus the claim is false.

Exercise 2.3

This is true. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $(f(n))^2 \leq c^2(g(n))^2$ for all $n \geq n_0$ (you just have to write the definition properly)

Exercise 3

First, f_1, f_2, f_4 are easy (they belong to classic functions: exponential, polynomial, logarithm): $f_4 \in O(f_2)$ and $f_2 \in O(f_1)$.

Now for f_3, it starts to be smaller than 10^n but once $n \geq 10$, then clearly $10^n \leq n^n$. This is exactly what we need for the definition of $O(\cdot)$ (take $c = 1$ and $n_0 = 10$), thus $f_1 \in O(f_3)$.

Now f_5 is a bit more complex. The solution here is to take logarithms to make things clearer. Here $\log_2(f_5(n)) = \sqrt{\log_2 n} = (\log_2 n)^{1/2}$.
For the other functions we have $\log_2(f_4(n)) = \log_2(\log_2(n))$ while $\log_2(f_2(n)) = \frac{1}{3}\log_2(n)$.
Let $z = \log_2 n$, we can see these as <u>functions of z</u>: $\log_2(f_2(n)) = \frac{1}{3}z$, $\log_2 f_4(n) = \log_2(z)$ and $\log_2(f_5(n)) = z^{1/2}$.

Exercise 3 (ctd)

Now it is easier to see what is going on. First, let's compare f_4 and f_5.

For $z \geq 16$, we have $\log_2(z) \leq \sqrt{z}$ (you can try to do a simple analysis of the function $f(z) = \ln(z)/\ln(2) - \sqrt{z}$, global maximum at $4/\ln(2)^2$)).

But the condition $z \geq 16$ is the same as $n \geq 2^{16} = 65,536$.
Thus once $n \geq 2^{16}$, we have $\log_2 f_4(n) \leq \log_2 f_5(n)$, and so $f_4(n) \leq f_5(n)$ since $\log_2$ is a monotonic increasing function.

Thus we can write $f_4(n) \in O(f_5(n))$.

Exercise 3 (ctd)

Similarly we have $z^{1/2} \leq \frac{1}{3}z$ once $z \geq 9$ (you can do a simple analysis of the function $f(z) = z^{1/2} - \frac{1}{3}z$, global maximum at $9/4$)
in other words, once $n \geq 2^9 = 512 = n_0$.

For n above this bound we have $\log_2 f_5(n) \leq log_2 f_2(n)$ and hence $f_5(n) \leq f_2(n)$, and so we can write $f_5(n) \in O(f_2(n))$.

Essentially, we have discovered that $2^{\sqrt{\log_2 n}}$ is a function whose growth rate lies somewhere between that of logarithms and polynomials.

Exercise 4.1

$$\begin{aligned} T(n) &= T(n-1)+n \\ &= T(n-2)+n-1+n \\ &\vdots \\ &= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \end{aligned}$$

$\Rightarrow O(n^2)$ (also $\theta(n^2)$)
Note: the master theorem cannot be applied here.

Exercise 4.2

$T(n) = T(n/2) + n,\ T(1) = 0,$
let $n = 2^k$

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 2^k \\ &= T(2^{k-2}) + 2^{k-1} + 2^k \\ &\vdots \\ &= T(1) + \sum_{i=1}^{k} 2^i = 2^{k+1} - 2 = 2n - 2 \end{aligned}$$

$\Rightarrow O(n)$ (also easy by induction)
By master theorem (1), we find easily the complexity:
$(a = 1, b = 2, d = 1)$

Exercise 4.3

$T(n) = 2T(n/2) + n^2$, let $n = 2^k$

$$\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + \left(2^k\right)^2 \\
&= 2^2 T(2^{k-2}) + 2\left(2^{k-1}\right)^2 + \left(2^k\right)^2 = 2^2 T(2^{k-2}) + 2^{2k-1} + 2^{2k} \\
&\vdots \\
&= 2^k T(1) + \sum_{i=0}^{k-1} 2^k 2^{k-i} \\
&\leq 0 + \sum_{i=0}^{k} 2^k 2^{k-i} \leq 2^k(2^{k+1} - 1) = 2n^2 - n
\end{aligned}$$

[For the last line (1st term), adding the term 2^k is required to use the geometric series property, if we want to have the exact value of the series we should substract the additional 2^k term which leads to $T(n) = 2n^2 - 2n$]

$\Rightarrow O(n^2)$

By master theorem 1, we get also the same complexity $(a = 2, b = 2, d = 2)$

Exercise 4.4

$T(n) = 2T(\sqrt{n}) + \log n$, let $n = 2^k$, then we have
$T(2^k) = 2T(2^{k/2}) + k$.

By fixing $F(k) = T(2^k)$, we have

$$F(k) = 2F(k/2) + k$$

which is a classic recurrence corresponding to complexity $O(k \log k)$,

thus $T(n) \in O(\log n \log \log n)$.

Exercise 4.5

$T(n) = T(\sqrt{n}) + \log\log n$, let $n = 2^{2^k}$, then
$T(2^{2^k}) = T(2^{2^{k-1}}) + k$. Let $F(k) = T(2^{2^k})$ then

$$\begin{aligned} F(k) &= F(k-1) + k \\ &= F(k-2) + k - 1 + k \\ &\vdots \\ &= F(0) + \sum_{i=2}^{k} i = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} \end{aligned}$$

Thus $F(k) \in O(k^2)$, thus $T(n) \in O((\log\log n)^2)$.

Exercise 4.5 with master theorem - formulation 2

$T(n) = T(\sqrt{n}) + \log\log n$, let $n = 2^k$, then
$T(2^{2^k}) = T(2^{k-1}) + \log k$. Let $F(k) = T(2^k)$ then

$$F(k) = F(k/2) + \log k$$

Now, we can interpret the last term of the series as
$f(k) = \log k = k^{\log_2 1} \log k$.
Thus $F(k) \in O(k^{\log_2 1} (\log k)^2)$ which is $O((\log k)^2)$, thus
$T(n) \in O((\log\log n)^2)$.

Bonus Recurrence

Bonus recurrence:

$T(n) = 3T(n/2) - 2T(n/4) + \log n.$

First, note that $T(1) = 3$, $T(2) = 3$, $T(4) = 5$, $T(8) = 12$, $T(16) = 30$, $T(32) = 71$, $\ldots$

Let $n = 2^k$, we have: $T(2^k) = 3T(2^{k-1}) - 2T(2^{k-2}) + k$.

Let $F(k) = T(2^k)$, we have $F(k) = 3F(k-1) - 2F(k-1) + k$ and $F(0) = 3$, $F(1) = 3$, $F(2) = 5$, $F(3) = 12$, $F(4) = 30$, $F(5) = 71$, $\ldots$

This a a non homogeneous linear relation, to solve it we need some theorems.

Theorem

Theorem 1: homogeneous equation

Consider the homogenous recurrence: $a_n = Aa_{n-1} + Ba_{n-2}$, A and B being real numbers.

The solution of this recurrence depends on the nature of the roots of the characteristic equation

$$s^2 - As - B = 0$$

We consider two cases:

1. if we have two distinct real roots s_1 and s_2 then $a_n = \alpha s_1^n + \beta s_2^n$.
2. if we have one real root s then $a_n = \alpha s^n + \beta n s^n$.

(if we have two complex conjugate roots, $a_n = r^n(\alpha cos(n\theta) + \beta sin(n\theta))$ where r and θ correspond to the polar form of the roots).

The result above is presented with an order 2, but can be generalized to any order k

Theorem 2

Theorem 2: non homogeneous equation

if $a_n^{(p)}$ is a particular solution of the non homogeneous recurrence equation

$$a_n = Aa_{n-1} + Ba_{n-2} + Q(n) \quad (1) \tag{1}$$

then every solution of (1) is of the form

$$a_n = a_n^{(h)} + a_n^{(p)}$$

where $a_n^{(h)}$ is a solution of the associated homogeneous recurrence relation $a_n = Aa_{n-1} + Ba_{n-2}$.

Finding a particular solution of (1) is difficult in general, however some methods exist for some simple forms of $Q(n)$.

Theorem 3

Theorem 3

Consider the relation (1) with $Q(n) = t^n \times$ (Polynom of degree N).
It t is not a root of $s^2 - As - B = 0$ then there is a particular solution of the form

$$a_n^{(p)} = t^n(p_0 + p_1 n + p_2 n^2 + \ldots + p_N n^N).$$

If t is a root of $s^2 - As - B = 0$ of multiplicity m (*ie* t is present m times as a root), then there is a particular solution of the form

$$a_n^{(p)} = t^n n^m(p_0 + p_1 n + p_2 n^2 + \ldots + p_N n^N).$$

Note that if $Q(n)$ is simply a polynomial like $Q(n) = 3n^2 - 3n + 1$ then $t = 1$ in the above theorem ($Q(n) = 1^n(3n^2 - 3n + 1)$).

Solution

Now, back to our case by considering $F(k) = 3F(k-1) - 2F(k-1) + k$, we have $Q(k) = k = 1^k k$, the associated homogeneous relation is $s^2 - 3s + 2 = 0$ which is equivalent to $(s-1)(s-2) = 0$, thus 1 and 2 are the roots.

Now $Q(k) = 1^k k$, thus by Theorem 3 $a_k^{(p)}$ is of the form $k(\gamma k + \delta)$. From Theorem 1, the solution of the associated homogeneous relation is of the form $\alpha 1^k + \beta 2^k$, thus from Theorem 2 the solution is of the form $F(k) = \alpha 1^k + \beta 2^k + \gamma k^2 + \delta k$. Now by using the values $F(0)$, $F(1)$, $F(2)$, $F(3)$ we get $\alpha = 0$, $\beta = 3$, $\gamma = -\frac{1}{2}$, $\delta = -\frac{5}{2}$, thus

$$F(k) = 3 * 2^k - \frac{k^2}{2} - \frac{5k}{2}.$$

Thus $T(n) = 3 * n - \frac{(\log n)^2}{2} - \frac{-5 \log n}{2} \Rightarrow \theta(n)$.

Exercise 5

Let $e_1, \ldots, e_n$ be the equivalence classes of the chips: chips i and j are equivalent if the machine says that $e_i = e_j$. We are looking for a value x so that strictly more than $n/2$ of the indices have $e_i = x$.

Divide the set of chips into two roughly equal piles: a set of $\lfloor n/2 \rfloor$ chips and a second for the remaining $\lceil n/2 \rceil$ chips. We will recursively run the algorithm on the two sides, and we will assume that if the algorithm finds an equivalence class containing more than half of the chips, then it returns a sample chip as a representer of the equivalence class.

Exercise 5 (ctd)

Note that if there are strictly more than $n/2$ chips that are equivalent in the whole set, say have equivalence class x, then at least one of the two sides will have more than half of the chips also equivalent to x. So at least one of the two recursive calls will return a chip that has equivalence class x.

The reverse of this statement is not true: there can be a majority of equivalent chips in one side without that equivalence class having more that $n/2$ chips overall (as it was only a majority on one side). So, if a majority chips is returned on either side we must test this chip against all the other chips.

Exercise 5 (algo for finding a majority equivalence class in a set of chips)

```
Input: S set of chips;
begin
    if |S| = 1 then Return one chip;
    if |S| = 2 then
        test if two chips are equivalent;
        Return either chip if they are equivalent;
    Let S1 be the set of the first ⌊n/2⌋ chips;
    Let S2 be the set of the remaining chips;
    Call the algorithm recursively for S1;
    if a chip is returned then
        test this chip against all other chips;
    if no chip with a majority equivalence has yet been found then
        Call the algorithm recursively for S2;
        if a chip is returned then
            test this chip against all other chips
    Return a chip from the majority equivalence class if one if found;
end
```

Exercise 5

The correctness of the algorithm follows from the observation above: that if there is a majority equivalence class, than this must be a majority equivalence class for at least one of the two sides.

To analyse the running time, let $T(n)$ denote the maximum number of tests the algorithm does for any sets of n chips. The algorithm has two recursive calls and does at most $2n$ tests outside of the recursive calls. So, we get the following recurrence (assuming n is divisible by 2):

$$T(n) \leq 2T(n/2) + 2n,$$

which leads to a complexity $T(n) \in O(n \log n)$.
(use master theorem or similar proof techniques as in exercise 5)

Exercise 5: in linear time

Pair up all chips and test all pairs of equivalence. If n was odd, one chip is unmatched. For each pair that is not equivalent, discard both chips. For pairs that are equivalent, keep one of the two. Keep also the unmatched chip, if n is odd. We call this subroutine ELIMINATE.

The observation that leads to the linear time is as follows. If there is an equivalence class with more than $n/2$ chips, then the same equivalence class must also have more than half of the chips after calling ELIMINATE.

This is true as when we discard both chips in a pair, then at most one of them can be from the majority equivalence class. One call to eliminate on a set of n chips takes $n/2$ tests and as a result we have only $\leq \lceil n/2 \rceil$ chips left. When we are down to a single chip, its equivalence is the only candidate for having a majority. We test this chip against all others to check if its equivalence has more than $n/2$ elements.

Exercise 5: in linear time

Little note: When you get an odd number of chips, it can happen that after 1st round you have 2 different chips remaining, then these 2 chips can be candidates for having a strict majority and you have to test the two.

This situation can occur for example when you have only 2 equivalence classes and an odd number of chips.
$\{2, 1, 2, 1, 2, 1, 2, 1, 1, 1, 2\}$

Exercise 5: in linear time

The complexity of the ELIMINATE procedures defines a series where the number of remaining chips is divided by 2 at each step. Let assume $n = 2^k$ for sake of simplicity (otherwise take 2^k as an upper bound for n and you can get the result):

$$\frac{n}{2}+\frac{n}{4}+\cdots+\frac{n}{2^k} = \frac{n}{2}(1+\frac{1}{2}+\ldots+\frac{1}{2^{k-1}}) = \frac{n}{2}(\sum_{i=0}^{k-1}\frac{1}{2^i}) = \frac{n}{2}(\frac{1-\frac{1}{2^k}}{1-\frac{1}{2}}) = n-1$$

This previous part bounds the successive calls of ELIMINATE, now we have an additional cost of $n-1$ for comparing the last chip with the others. Thus the global cost is $2n-2$ implying linear time $O(n)$. (It can be $3n-3$ if you need to compare the two final chips).