Solution TD 1 : Introduction to algorithms and data structures

Doriath Döhler Gabriel

October 4, 2020

1. $\mathcal{O}$, Ω and Θ notations
   Prove or disprove :

   (a) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$
   False : let $\forall n \in \mathbb{N}, f(n) = n$ and $g(n) = n^2$.
   $f(n) + g(n) \in \Theta(n^2) \neq \Theta(n) = \Theta(\min(f(n), g(n))) = n$
   $\forall n \in \mathbb{N}, f(n) = (-1)^n$ and $g(n) = (-1)^{n+1}$ also constitue a conterexample but negative complexity are rare...

   (b) $g(n) - h(n) = \Theta(f(n))$, where $g(n), f(n) \in \Theta(f(n))$
   False : let $\forall n \in \mathbb{N}, f(n) = g(n) = h(n)$
   We have $g(n), h(n) \in \Theta(f(n))$ but $g(n) - f(n) = 0 \notin \Theta(n) = \Theta(f(n))$

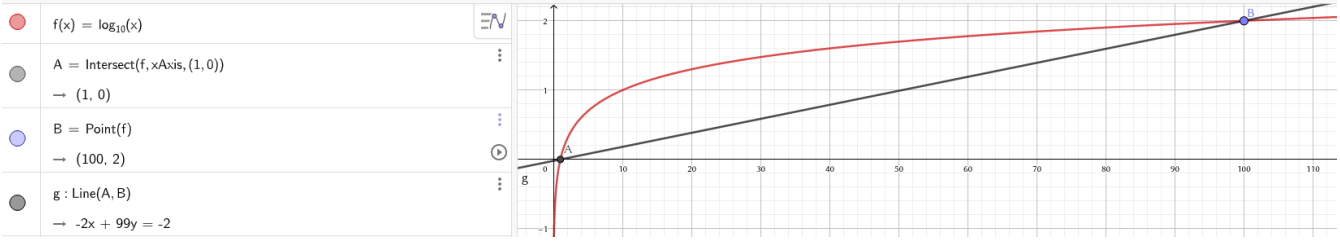
   (c) $\Theta(f(n)) + \Omega(f(n)) = \mathcal{O}(f(n))$
   False : let $\forall n \in \mathbb{N}, f(n) = n$.
   $n = \Theta(f(n))$ and $e^n = \Omega(f(n))$ but $n + e^n \notin \mathcal{O}(n)$

2. Classic assymptotic relations
   Show that $n! = o(n^n)$ and $\log(n!) = \Theta(n \log(n))$.

   ✘ Proof with Stirling.
   Reminder : $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$. $n! = o(n^n)$ follows. By composing with an equivalent we get : $\log(n!) = \Theta(n \log(n))$. Note that we can apply $\log$ on the Stirling formula because $n! \to \infty$.

   ✘ Elementary proof.
   $0 \leq \frac{n!}{n^n} = \frac{n \times (n-1) \cdots \times 1}{n \times \cdots \times n} = \prod_{k=1}^{n} \frac{k}{n} \leq \frac{1}{n} \to 0$ so $n! = o(n^n)$.
   $\log(n!) = \sum_{k=1}^{n} \log(k) \geq \sum_{n \geq k \geq \frac{n}{2}} \log(k) \in \Theta(n \log(n))$
   We conclude with : $\log(n!) = \sum_{k=1}^{n} \log(k) \leq \sum_{k=1}^{n} \log(n) = n \log(n)$

   ✘ Jérôme's elegant proof.
   By concavity of the logarithm we have : $\forall n \in \mathbb{N}^*, \forall 1 \leq k \leq n, \log(k) \geq \frac{\log(n)}{n-1}(k-1)$.
   For example with $n = 100$:



Hence : $\log(n!) = \sum_{k=2}^{n} \log(k) \geq \sum_{k=2}^{n} \frac{\log(n)}{n-1}(k-1) \geq \frac{\log(n)}{n} \sum_{k=2}^{n}(k-1) \in \Theta(n\log(n))$
We conclude with : $\log(n!) = \sum_{k=1}^{n} \log(k) \leq \sum_{k=1}^{n} \log(n) = n\log(n)$

3. Prove that : $\forall a \in \mathbb{R}, \forall b \in \mathbb{R}_+^*, (n+a)^b = \Theta(n^b)$
   If $a \geq 0$ then $\forall n \in \mathbb{N}^*, 0 \leq \frac{n^b}{(n+a)^b} \leq 1$ so $n^b = \mathcal{O}((n+a)^b)$. We also have $\forall n \in \mathbb{N}^*, \frac{(n+a)^b}{n^b} = (1+\frac{a}{n})^b \leq (1+a)^b$ so $(n+a)^b \in \mathcal{O}(n^b)$. Finally we have that $(n+a)^b \in \Theta(n^b)$.
   If $a < 0$ a similar method proves the result (left as an exercice to the reader).

4. Two stack in one array.

```
1  Function create(n):
2      a = Array.make(n, 0)
3      i = 0
4      j = n - 1
5      return a, i, j
6
7  Function pop1(a, i, j):
8      if i > 0 then
9          i = i - 1
10         return a[i]
11     else
12         raise Stack1Empty
13
14 Function pop2(a, i, j):
15     if j < n - 1 then
16         i = j + 1
17         return a[j]
18     else
19         raise Stack2Empty
20
21 Function push1(a, i, j, x):
22     if i = j + 1 then
23         raise Stack1Full
24     else
25         a[i] = x
26         i = i + 1
27
28 Function push2(a, i, j, x):
29     if i = j + 1 then
30         raise Stack2Full
31     else
32         a[j] = x
33         j = j - 1
34
```

$pop1$, $pop2$, $push1$ and $push2$ clearly take $\mathcal{O}(1)$ time. Here is an implementation in python :

```python
class Double_stack:
    def __init__(self, n):
        self.n = n
        self.a = [None] * n
        self.i = 0
        self.j = n - 1

    def __str__(self):
        return f'n : {self.n}, i : {self.i}, j : {self.j}, a : {self.a}'

    def __repr__(self):
        print(str(self))

    @property
    def is_not_full(self):
        return self.i != self.j + 1

    def pop1(self):
        assert self.i > 0, 'stack1empty'
        self.i -= 1
        return self.a[self.i]

    def pop2(self):
        assert self.j < self.n - 1, 'stack2empty'
        self.j += 1
        return self.a[self.j]

    def push1(self, x):
        assert self.is_not_full, 'stack1full'
        self.a[self.i] = x
        self.i += 1

    def push2(self, x):
        assert self.is_not_full, 'stack2full'
        self.a[self.j] = x
        self.j -= 1
```

Note that we could use a similar strategy to exercice 8 in order to deal with overflow.

5. Implementation of a stack with two queues.

(a) With two queues :

```
1  Function create():
2      Q1 = createEmptyQueue()
3      Q2 = createEmptyQueue()
4      sizeQ1 = 0
5      return Q1, Q2, sizeQ1
6
7  Function push(Q1, Q2, sizeQ1, x):
8      pushQueue(Q1, x)
9
10 Function pop(Q1, Q2, sizeQ1):
11     if sizeQ1 = 0 then
12         raise EmptyStack
13     initialSizeQ1 = sizeQ1
14     while sizeQ1 > 1 do
15         x = popQueue(Q1)
16         pushQueue(Q2, x)
17         sizeQ1 = sizeQ1 - 1
18
19     x = popQueue(Q1)
20     sizeQ1 = sizeQ1 - 1
21     Q1, Q2 = Q2, Q1
22     sizeQ1 = initialSizeQ1 - 1
23     return x
24
```

(b) With only one queue :

```
1  Function create():
2      Q = createEmptyQueue()
3      return Q
4
5  Function push(Q, x):
6      pushQueue(Q, x)
7
8  Function pop(Q):
9      if isEmptyQueue(Q) then
10         raise EmptyStack
11     push(Q, "end")
12     x = popQueue(Q)
13     last = Null
14     while x ≠ "end" do
15         if last ≠ Null then
16             push(Q, last)
17         last = x
18         x = popQueue(Q)
19
20     return last
21
```

Complexity : push and create are $\Theta(1)$ but pop is $\Theta(n)$. Maybe there is a better way to do it ?

6. Palindrome detection.

(a) Solution in Ocaml :

```
let rec rev ?(acc=[]) = function
    | []     -> acc
    | t :: q -> rev ~acc:(t :: acc) q

let rec equal l1 l2 = match l1, l2 with
    | [], []             -> true
    | [], _ | _, []      -> false
    | t1 :: q1, t2 :: q2 -> t1 = t2 && equal q1 q2

let isPalindrome l = egal l (rev l)
```

(b) Solution in Ocaml (we assume that the garbage collector does its jobs) :

```
let rec split ?(l1=[]) l k =
    if k = 0 then
        l1, l
    else (
        match l with
            | []     -> failwith "error ;)"
            | t :: q -> split ~l1:(t :: l1) q (k - 1)
    )

let isPalindrome l =
    let n = List.length l in
    n mod 2 = 0 && (
        let l1, l2 = split l (n / 2) in
        egal l1 (rev l2)
    )
```

Note that all recursive function here are tail recursive so no space for a stack is requiered.

7. Implementation of a queue using two stacks.

   - ✘ $create\_empty$: create two stacks S_1 and S_2.
   - ✘ $push\ x$: add x to S_1.
   - ✘ pop : if S_2 is not empty then pop from S_2 else transfert (one by one) all the elements of S_1 into S_2. If S_2 is still empty then the queue is empty else pop from S_2.

   Implementation is left as an exercice.
   Analysis : During n operations on the data structure, there are at most n elements going on the queue. Each element is at most added to S_1 then transfered from S_1 to S_2 and then popped from S_2. These three operations take constant time. As a consequence, doing n operations on the data struture takes $\Theta(n)$ time. Hence the $\Theta(1)$ amortized time complexity. In other words, if you need to tranfert the k elements of S_1 to S_2 (wich takes $\Theta(k)$ time), you have already done $\Theta(k)$ operations to fill S_1. Note that queues can also be implemented with dynamic circular arrays (tableau dynamique circulant en français) or doubly linked lists.

8. Dynamic arrays. (Similar to python's lists.)

   (a)
   - ✘ $create\_empty$: create an array a of length 2. Set $n \leftarrow 2$.
   - ✘ $access\ i$: if $i < n$ then return $a[i]$.
   - ✘ $set\ i\ x$: if $i < n$ then $a[i] \leftarrow x$.
   - ✘ $insert\ x$: create an array b of length $2n$, $b[0 : n] \leftarrow a$, $b[n] \leftarrow x$, $a \leftarrow b$ and $n \leftarrow 2n$. (The garbage collector will destroy b).

   Analysis : $access$, set and $create\_empty$ all take $\Theta(1)$ time. Space is always bounded by $3n + 3$. Insertion takes $\Theta(n)$ time but we only have to use it when a is full. And in order for a to be full we need to call set $\frac{n}{2}$ times wich takes $\Theta(n)$ time. All together, this gives an $\Theta(1)$ amortized runing time.

(b) Why would we want to reduce the size of a ? After all, if a is at some point of size n and later of size $k < n$ the space complexity will still be n. So why bother ? Lets describe an algorithm that uses n dynamic arrays :

```
1 create n dynamic arrays a_1, ..., a_n
2 for j ∈ [[1, n]] do
3 |   for k ∈ [[1, n]] do
4 |   |   store k in a_j
5 |   for k ∈ [[1, n]] do
6 |   |   remove k from a_j
7
```

Without reducing the size of the arrays when deleting elements the space complexity is $\Theta(n^2)$ whereas when reducing the size of the arrays, the space complexity is $\Theta(n)$. This is much better.

First strategy : keep in memory the number of elements stored in a (lets called that i). When $2i < n$ resize the array by cutting its size by 2.
Analysis : This sadly doesn't work. Indeed, if we have n elements stored in a of size $2n$, adding and then removing an element n times will take $\Theta(n^2)$ time for $\Theta(n)$ operations in total (including the insertion of the first n elements of a).

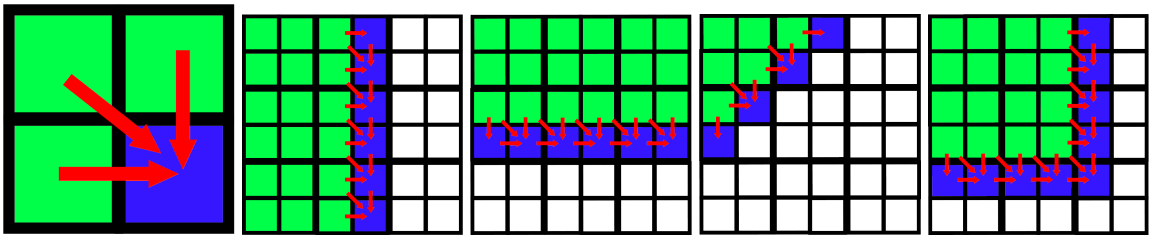
(c) Second strategy : quadruple the size of a when it is more than $\frac{3}{4}$ full and cut the array size in 4 when it is less than $\frac{1}{4}$ full. It can easily be proven (using the same techniques as before) that this strategy gives a $\Theta(1)$ amortized runing time for all operations.

9. Longest common subsequence.

(a) Clearly, a greedy algorithm doesn't work here so lets use dynamic programming. First of all lets define the subproblems :
$\forall i \in [\![0, m]\!], \forall j \in [\![0, n]\!], P_{x,y}[i, j] := |LCS((x_1, \dots, x_i), (y_1, \dots, y_j))|$.
For negative i or j, $P_{x,y}[i, j] := 0$
We then find a relation between subproblems :

- ✘ $P_{x,y}[0, \_] = P_{x,y}[\_, 0] = 0$
- ✘ $\forall i \in [\![0, m[\![, \forall j \in [\![0, n[\![, P_{x,y}[i+1, j+1] = \begin{cases} P_{x,y}[i, j] + 1 & \text{if } x_{i+1} = y_{j+1} \\ \max(P_{x,y}[i+1, j], P_{x,y}[i, j+1]) & \text{otherwise} \end{cases}$

Indeed : if $x_{i+1} = y_{j+1}$ we can just add x_{i+1} and y_{j+1} to $LCS((x_1, \dots, x_i), (y_1, \dots, y_j))$ to get a common subsequence of $(x_1, \dots, x_{i+1})$ and $(y_1, \dots, y_{j+1})$ of size $P_{x,y}[i, j] + 1$. This is optimal by optimality of $LCS((x_1, \dots, x_i), (y_1, \dots, y_j))$. If $x_{i+1} \neq y_{j+1}$ then either x_i or y_j is not in a longest common subsequence of $(x_1, \dots, x_i)$ and $(y_1, \dots, y_j)$. This proves $(*)$.
Finally we need to find an order to fill the table $P_{x,y}$. For example columns by columns or row by row or diagonal by diagonal. Here any one of these will work.



Implementation in Ocaml :

```
let lcs x y =
    let m = String.length x in
    let n = String.length y in
    let a = Array.make_matrix m n 0 in
    let p i j =
        if i >= 0 && j >= 0 then
            a.(i).(j)
        else
            0
    in

    for i = 0 to m - 1 do
        for j = 0 to n - 1 do
            a.(i).(j) <-
                if x.[i] = y.[j] then
                    p (i - 1) (j - 1) + 1
                else
                    max (p i (j - 1)) (p (i - 1) j)
        done;
    done;
    p (m - 1) (n - 1)
```

Analysis : filling a case of the table takes $\Theta(1)$ time so the complexity is $\Theta(nm)$ time and space.

(b) For each i, j we will store in $P'_{x,y}[i,j] \leftarrow \begin{cases} (i-1,j-1) & \text{if } x_i = y_j \\ (i,j-1) & \text{if } P_{x,y}[i,j-1] > P_{x,y}[i-1,j] \text{and } x_i \neq y_j \\ (i-1,j) & \text{otherwise} \end{cases}$
thus remembering the choices we make while constructing $P_{x,y}$.
From that we can recover the longest common subsequence. Note that the asymptotic time and space complexity doesn't change.

(c) Remark :
One row/column/diagonal only depends on the one/one/two previous row/column/diagonals. Therefore we only need to store the last row or the last column or the last two diagonals. This technique gives a $\Theta(\min(m,n))$ space complexity (not acounting for the space to store x and y. Sadly this only works if want the length of the LCS and not one actual LCS.

10. Interval graphs.

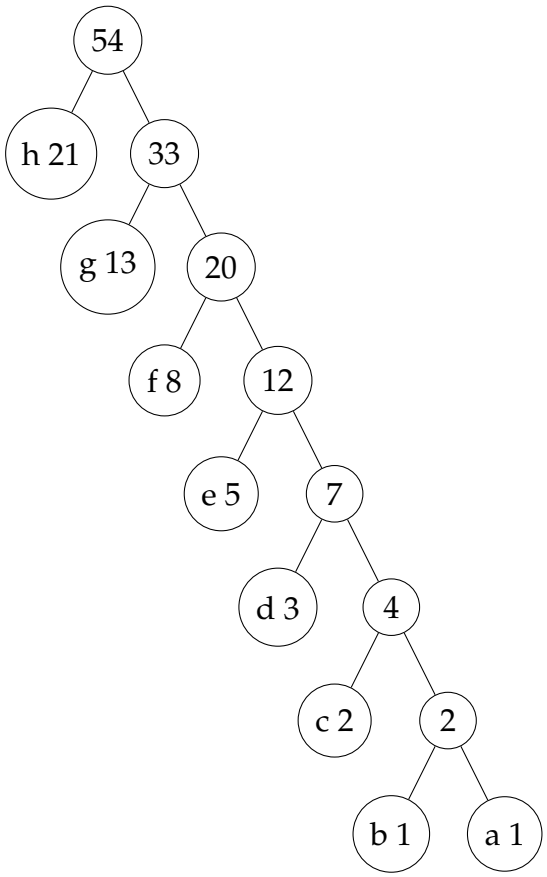
(a) Greedy strategy : choose the task that finish the soonest.
Analysis : sorting the tasks by finish time takes $\Theta(n\log(n))$ time then we make $\Theta(n)$ operations. Hence a runing time of $\Theta(n\log(n))$.
Correctness : let $x_1 < \cdots < x_p$ be the finishing times of the solution given by the greedy algorithm and let $y_1 < \cdots < y_q$ be the finishing times of the optimal solution having the largest prefix in common with $x_1, \ldots, x_p = x$. Let $k = \max\{i \in \mathbb{N} : \forall j \in]\!]0, i]\!], x_j = y_j\}$. Lets suppose $k \neq p$. By design of the greedy algorithm we have $x_{k+1} < y_{k+1}$. Thus $x_1, \ldots, x_k, x_{k+1}, y_{k+2}, \ldots, y_q$ is optimal and has a larger prefix in common with x than y with x. That contradicts the definition of $y_1, \ldots, y_q$. So we have $k = p$. That is : $x_1 = y_1$ and $\ldots$ and $x_p = y_p$. By optimality of

y we have $p \leq q$. $p < q$ means that they are other tasks $(y_{p+1}, \ldots, y_q)$ compatible with x wich is impossible by design of the greedy algorithm. So we have $p = q$ meaning that the greedy approch is optimal.

(b) We now add a priority ω_i to the task i. We want to maximaze : $\sum_{i \in I} \omega_i$ with the constraint : $\forall (i,j) \in I^2$ tasks i and j are compatible. Here, no efficient greedy strategy works. (An inefficient greedy strategy would be to compute all of the possibilities $\Omega(2^n)$ and then take the task with the lowest finishing time from the tasks in the optimal solution already computed by brute force.) So we will use dynamic programming. We will assume without loss of generality that $f_1 < \cdots < f_n$.
Subproblems : let $P[i]$ be the scheduling problem for the first i tasks.
We have : $(*) \begin{cases} P[0] = 0 \\ \forall i \in \mathbb{N}^*, P[i] = \max\{\omega_i + P[k(i)], P[i-1]\} \quad \text{where } k(i) = \max\{j \in \mathbb{N} : f_j \leq s_i\} \end{cases}$
Indeed : we either take the task i or not.
Analysis : sorting is done in $\mathcal{O}(n^2)$ and the actual dynamic programming takes $\mathcal{O}(n^2)$ so the runing time is in $\mathcal{O}(n^2)$. Correctness follows from the dynamic programming equation $(*)$.
Remark : We have just calculated the sum of the priorities of the tasks of the optimal solution but it is easy to also compute the optimal solution (see 9. (b)).

11. Huffman codes.

(a) Left corresponds to 0 and right to 1.



a	1111111
b	1111110
c	111110
d	11110
e	1110
f	110
g	10
h	0

(b) If x is a node with only one child y then x's code is a prefix of y's code. So the tree doesn't represent a prefix code.

(c)
(d)
(e)