COMP 251 Assignment 1

Jeremie Poisson - McGill ID 260627104

McGill School of Computer Science – October 1st, 2015

1 Exercise 1 : Resident Matching problem

Imagine a set of medical students $m \in M$ (set of all students) seeking for a position in hospitals $h \in H$ (set of all hospitals) who all have open positions for n med students, as initially described by the exercise 4.

- Initially all positions of hospitals $h \in H$ are available (i.e. unassigned);
- Initially every med student $m \in M$ is unassigned;

While (*there is an unassigned spot in an hospital* h)
- choose such hospital h
- let s denote the highest med student on h's preference list

if (s *is free* (i.e. has not been assigned))
- h assigns s to its available position

else (s *has already been assigned to an other hospital* h')
if (s *prefers* h *over* h')
- s releases its assigned spot in h'
- h assigns s to its available spot

- Return the set of matched students and hospitals

Question: *Show that there is always a stable matching.*

We first need to point out that this algorithm stays a typical stable matching problem. Therefore, most of the claims pointed out bellow are based off the typical G-S algorithm. These claims will be placed in context of the medical residence assignment problem and justified accordingly, if need be. Mainly, the main differences being :

(i) hospitals generally want more than one resident;
(ii) there is a surplus of medical students;

Claim 1: A student s is assigned from the point at which he/she receives its first assignment. Their assignment only gets better and better as he/she trades

up his/her assignment for an other hospital. This trade is solely based on s's preference list.

Proof. See the original G-S justification.

Claim 2: The sequence of students s assigned to an hospital h gets worse and worse throughout the execution of the algorithm.

Proof. See the original G-S justification.

Claim 3: The algorithm terminates after at most $|H| \cdot |S|$ iterations of the while loop.

Proof. Each time through the while iteration, an hospital h checks the availability of a student s. This is different from the original G-S algorithm as the number of spot available for an hospital h can be more than 1 (see modification (i)) (although we assume that there is always enough med students than available spots (see (ii)). Hence, in the worst case, an hospital h would need to check the availability of every student in $|S|$, making the total number of possible iterations $|H| \cdot |S|$.

Claim 4: If an hospital h has an open spot at some point in the algorithm, then there exists a student s to whom the availability has not been checked.

Proof. This one slightly differs from the original as there is not necessarily the same amount of positions and students (see (i)).

(by contradiction) assume that, at some point, such an hospital h (who have an open position) runs off of students to check the availability (none of $s \in S$ is still available). Therefore, by *claim 1*, it must follow that every students has been assigned to a position. In addition, it is clearly stated that there is more students than available positions (see (ii)). *Therefore, it is impossible for any hospital $h \in H$ run off of students.*

Claim 5: All students get matched to a position in an hospital (perfect matching).

Proof. See original G-S justification and modifications (i) and (ii).

Claim 6: An execution of this algorithm yields a set of pairs of assignments (position in hospital - med student). This set SM is a stable matching.

Proof. such a claim requires us to prove that the algorithm can't yield to a stable matching SM with an existing type of instability (described in the problem).

First type of instability : (contradiction) assuming there exists a pair (s, h) $\in SM$ where $s \in S$ and $h \in H$ and an unmatched student $s' \in S$ where h prefers s' over one of its assigned student s. In such execution of the G-S algorithm (where s' is preferred to s), h would first have checked the availability of s prior to s' (claim 2). Meaning that s ranks higher than s' (contradiction). Moreover,

if s' had been assigned to h at some point and then traded-up his assignment, s' would still be assigned to an hospital $h \in H$ (claim 1) (contradiction).

Second type of instability : (contradiction) assuming there exists two pairs $(s,\ h)$ and $(s',\ h') \in SM$ where $s,\ s' \in S$ and $h,\ h' \in H$. Such matching SM in which h prefers s' over s and s' prefers h to h'. In such execution, h clearly ended up checking the availability of both candidates. In which case, it would mean that h prefers s over s' (contradiction) because they have been assigned. In the case that s' was assigned to h but ended-up trading its assignation to h', it would mean that h' ranks higher than h (claim 1) (contradiction).

2 Exercise 2 : Chapter 2

Proof. Let f and g be two functions that :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

by the formal definition of the limit for a sequence (read function) that converges to a number L:

$$\frac{f(n)}{g(n)} \to L \quad as \quad n \to \infty$$

if and only if

$$\forall \epsilon > 0 \quad \in \ R$$

such that

$$|\frac{f(n)}{g(n)} - L| < \epsilon$$

beyond some n_0

In this case, $L = 0$, making the ratio $|\frac{f(n)}{g(n)}|$ bounded by any $\epsilon > 0$. Since we can assume that $f(n)$ and $g(n)$ are two functions that are only defined positively and that it is a ratio, *it is safe to drop the absolute values operators.* Hence,

$$\frac{f(n)}{g(n)} < \epsilon \Rightarrow f(n) < \epsilon \cdot g(n)$$

From an algorithm analysis viewpoint, it is equivalent to say that $f(n)$ is $O(g(n))$. However, as no lower bounds can be defined using the definition of the limit, it is impossible that $f(n)$ is $\Omega(g(n))$ when $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, as required by the definition of $\theta(\cdot)$.

3 Exercise 3.1

(A) 99n

Double the size.

$$= \frac{99(2n)}{99n} = 2$$

2 times slower.

Increase the size by 1.

$$= \frac{99(n+1)}{99n} = \frac{99n}{99n} + \frac{99}{99n} = 1 + \frac{1}{n}$$

$1 + \frac{1}{n}$ times slower.

(B) n^2

Double the size.

$$= \frac{(2n)^2}{n^2} = \frac{2^2 \cdot n^2}{n^2} = 4$$

4 times slower.

Increase the input size by 1.

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

$1 + \frac{2}{n} + \frac{1}{n^2}$ times slower.

(C) n^4

Double the size.

$$\frac{(2n)^4}{n^4} = \frac{2^4 \cdot n^4}{n^4} = 2^4 = 16$$

16 times slower.

Increase the input size by 1.

$$\frac{(n+1)^4}{n^4} = \frac{n^4+4n^3+6n^2+4n+1}{n^4} = 1+\frac{4n^3}{n^4}+\frac{6n^2}{n^4}+\frac{4n}{n^4}+\frac{1}{n^4} = 1+\frac{4}{n}+\frac{6}{n^2}+\frac{4}{n^3}+\frac{1}{n^4}$$

$1 + \frac{4}{n} + \frac{6}{n^2} + \frac{4}{n^3} + \frac{1}{n^4}$ times slower.

(D) $n2^n$

Double the size.

$$\frac{(2n) \cdot 2^{2n}}{n \cdot 2^n} = \frac{(2n) \cdot 2^n}{n} = 2 \cdot 2^n = 2^{n+1}$$

2^{n+1} times slower.

Increase the size by 1.

$$\frac{(n+1)2^{n+1}}{n2^n} = \frac{(n+1)\cdot 2}{n} = 2 + \frac{2}{n}$$

$2 + \frac{2}{n}$ times slower.

(E) 3^n

Double the size.

$$\frac{3^{2n}}{3^n} = 3^n$$

3^n times slower.

Increase the size by 1.

$$\frac{3^{n+1}}{3^n} = 3$$

3 times slower.

4 Exercise 3.2

To start, we will use the general fact that

exponential growth rates > polynomial growth rates > linear growth rates

to quickly organize those functions in groups of 2. We will then prove for each couple of functions that its child is, in fact, upper-bounded by its parent (the child f is O(its parent g)). Having a proper order, it is then correct to say that every couples keeps its order by the property of *transitivity* of the asymptotic growth rates. It is possible to compare every functions using the property seen in the exercise of chapter 2.

1. $f_5(n) = 3^n$
2. $f_4(n) = n \cdot 2^n$

   The above can be proven using the technique of *Exercise 1* to prove that $f_4(n)$ is O(3^n).

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{3^n}$$

taking the natural logarithm of the equation ...

$$y = \frac{n \cdot 2^n}{3^n} \Rightarrow \ln y = (\ln n + n \cdot \ln 2) - n \cdot \ln 3$$

and now taking the limit to infinity of $\ln y$ and applying l'Hospital rule we get:

$$\lim_{n\to\infty}(\ln n + n\cdot\ln 2) - n\cdot\ln 3 = \lim_{n\to\infty}(\frac{1}{n} + \ln 2) - \ln 3 = (0 + \ln 2) - \ln 3$$

considering that $e^{\ln y} = y$, we take the exponential value of the limit :

$$\lim_{n\to\infty}\frac{n\cdot 2^n}{3^n} = (e^0 + e^{\ln 2}) - e^{\ln 3} = (1+2) - 3 = 0$$

Hence, f_4 is $\mathrm{O}(f_5(n))$.

3. $f_3(n) = n^2\cdot\log\cdot\log n$
4. $f_1(n) = 99n^2$

   Let's prove that $f_1(n)$ is $\mathrm{O}(f_3(n))$.

$$\lim_{n\to\infty}\frac{99n^2}{n^2\cdot\log\cdot\log n} = \lim_{n\to\infty}\frac{99}{\log\cdot\log n}$$

   using l'Hospital rule, we get:

$$\lim_{n\to\infty}\frac{0}{(\log\cdot\log n)'} = 0$$

   Therefore, that $f_1(n)$ is $\mathrm{O}(f_3(n))$.

5. $f_2(n) = 2^{\log_2 n}$

   Finally, let's prove that $f_2(n)$ is $\mathrm{O}(f_1(n))$ ...

$$\lim_{n\to\infty}\frac{2^{\log_2 n}}{99n^2}$$

   We take the natural logarithm of the expression:

$$y = \frac{2^{\log_2 n}}{99n^2} \Rightarrow \ln y = (\log_2 n\cdot\ln 2) - (2\cdot\ln 99n)$$

   We take the limit of both part of the equation and applying l'Hospital rule :

$$\lim_{n\to\infty}\log_2 n\cdot\ln 2 - \lim_{n\to\infty}2\cdot\ln 99n = \lim_{n\to\infty}\frac{\ln 2}{n} - \lim_{n\to\infty}\frac{\ln 99\cdot 2}{n} = 0 - 0$$

   considering that $e^{\ln y} = y$, we take the exponential value of both limits :

$$\lim_{n\to\infty}\frac{2^{\log_2 n}}{99n^2} = e^0 - e^0 = 1 - 1 = 0$$

Therefore, it is safe to say that $f_2(n)$ is $O(f_1(n))$ ...

Again, by transitivity, it is possible to establish the following order of growth for all functions :

$$f_5 \leq f_4 \leq f_3 \leq f_1 \leq f_2$$

5 Exercise 3.3

(a) False. Counter-example.

By definition, we say that $f(n)$ is $O(g(n))$ if and only if :

$$0 \leq f(n) \leq c \cdot g(n)$$

where $c \in R_+$

now let's imagine that, in this context :

$$f(n) = 2 \quad g(n) = 1$$

Given these functions, it would still be the case that for $f(n)$ to be $O(g(n))$ (constant c). However, when we apply the $\log_2$ to both function, the equation above becomes :

$$\log_2 2 \leq c \cdot \log_2 1$$

$$1 \leq c \cdot 0$$

$$1 \leq 0$$

which does not make sense in this specific case. Therefore, $\log_2 f(n)$ is *not* $O(\log_2 g(n))$.

Aside. Important note: in various other cases, this would actually work; it would be possible to find a constant c that would make the whole expression true. However, given this case, it is impossible to generalize the statement.

(b) False. Counter-example.

Let's consider the following functions :

$$f(n) = 2n \quad g(n) = n$$

With these functions, it is true that $2n$ is $O(n)$. In order for $2^{f(n)}$ to be $O(2^{g(n)})$ the following property must always remain true:

$$2^{2n} \leq c \cdot 2^n$$

Let's show that that the equation above is not true $\forall\ c$.

$$= \frac{2^{2n}}{2^n} \leq c = 2^n \leq c$$

Now let's remove the n exponent on both sides :

$$= (2^n)^{\frac{1}{n}} \leq (c)^{\frac{1}{n}} = 2 \leq c^{\frac{1}{n}}$$

Let's find out what $c^{\frac{1}{n}}$ turns out to be when $n \to \infty$ by evaluating the limit.

$$\lim_{n\to\infty} c^{\frac{1}{n}} \Rightarrow\ y = c^{1/n} \Rightarrow \ln y = \frac{1}{n} \cdot c$$

$$\lim_{n\to\infty} \frac{1}{n} \cdot c = 0 \Rightarrow \lim_{n\to\infty} c^{\frac{1}{n}} = e^0 = 1$$

It follows that $c^{\frac{1}{n}}$ is 1 when $n \to \infty$.

$$2 \leq 1$$

Which is obviously not true. The following property, essential for this statement to be true, can't hold.

(c) True.

Once again, we prove that using the fact that for $f(n)^2$ to be $\mathrm{O}(g(n)^2)$, the following property must hold :

$$f(n) \leq c \cdot g(n)$$

We simply square out both sides of the equation :

$$f(n)^2 \leq (c \cdot g(n)) = f(n)^2 \leq c^2 \cdot f(n)^2$$

Since squaring is a monotonic function (that strictly increases), the property still holds even when squared. Hence, it is true that $f(n)^2$ is $\mathrm{O}(g(n)^2)$

6 Exercise 8

6.1 $k = 2$ jars

- let n be the total number of rungs in the ladder;
- let i ($i <= n$) be the highest rung position at which we can drop the jar, without breaking it;
- let k the number of jars available for us to break to determine i.

A useful strategy to find i would be to split up the latter into $\sqrt{n}$ parts. That way, we make sure that every section of the ladder has been split evenly; with the same number of rungs per section.

Given that we have $k = 2$, we first try dropping our 1st jar at the top-most rung of each sections until the first jar breaks. Positions are given as follows :

$$x_{rung} = a \cdot \sqrt{n} \quad \forall\ a \in \{1, 2, ..., \sqrt{n}\}$$

Aside. If the jar does not break after testing all of the top-most rungs of every sections, then $i = n$.

- let j denote the position of the rung at which the first jar breaks.

We now start dropping our second (and last) jar at each rungs of the section, starting from the top-most rung of the previous section $(j - \sqrt{n})$, until it breaks. The positions are given as follows :

$$x_{rung2} = (j - \sqrt{n}) + b \quad \forall\ b \in \{1, ..., (\sqrt{n} - 1)\}$$

- let m be the position at which the last jar breaks.

Then, by following the algorithm, $i = m - 1$, being the last rung that has been proven for not breaking the jar.

(End of the strategy)

Therefore, given that $k = 2$, the worst run of the algorithm would take, at most, $2 \cdot \sqrt{n}$ tries before determining i properly. That is, trying each sections and trying each rungs within the current section.

$$f_2(n) = \sqrt{n} + \sqrt{n} - 1 = 2 \cdot \sqrt{n} - 1$$

To prove that this algorithm is actually better than linear time, we need to make sure that it is O(n).

$$\lim_{n \to \infty} \frac{f_2(n)}{n} = \lim_{n \to \infty} \frac{2 \cdot \sqrt{n} - 1}{n} = \lim_{n \to \infty} \frac{2 \cdot 1}{2 \cdot \sqrt{n}} = 0$$

(using de l'Hospital rule)

6.2 $k > 2$ jars

Given that we have a number $k > 2$ jars, describe the best strategy to get the highest safe rung using at most k jars.

Given that we have more jar to spare, a trivial solution would be to split up the ladder as we previously did in 6.1 over and over again, each in equal sections. Once we get in either one of those situations :

- there is one jar left;
- the ladder (or its sections) are no more splittable

the only solution is either (a) "brute force" search the section (with the last jar) or (b) return the highest ladder rung.

- let n be the total number of rungs in the ladder;
- let i ($i <= n$) be the highest rung position at which we can drop the jar, without breaking it;
- let k the number of jars available for us to break to determine i.

We split up the ladder in $\sqrt{n}$ sections once again and test every top-most rung of all sections.

- let m denote the position of the rung at which the jar broke.

We have found the section containing the highest safe rung. The position of this section is $[m - \sqrt{n}, m]$.

There is now three possible scenarios at this point :

- there is one jar left

If there is only one jar left, we need to brute force search the highest safe rung of its section by starting from the bottom of the section and check out each rung moving upward.

let m be the position at which the jar breaks

then by definition, $i = m - 1$, being the last rung that has been proven for not breaking the jar (the last valid one).

- the ladder (or its section) are no more splittable

If the ladder (or sections) is no more splittable, then all of the rungs must have been tested. We simply need to return the last valid rung at this position.

let m be the position at which the last jar broke

$i = m - 1$

- else : (there is more than one jar left) and (the sub-section is still splittable)

We can split up this section of the ladder containing the highest safe rung as well. We do this trivially by considering this section as a new ladder recursively. Therefore, we restart the algorithm using this section as the "ladder" and $k-1$ jars.

(End of the strategy)

The function $f_k(n)$ being the maximum number of tries required to test the jar with k jar in a n rung ladder is the following :

$$f_k(n) = k(n-1)^{(\frac{1}{2^{k-1}})}$$

Finally, we also need to prove that typically, a every function $f_k(n)$ will be $O(f_{k-1}(n))$. That is, that every function involving a greater number of test jar grow slower than those with less "test jars".

$$\lim_{n\to\infty} \frac{f_k}{f_{k-1}} = \lim_{n\to\infty} \frac{(k)(n-1)^{\frac{1}{2^{k-1}}}}{(k-1)(n-1)^{\frac{1}{2^{k-2}}}} = \frac{(k)}{(k-1)} \cdot \lim_{n\to\infty} \frac{(n-1)^{\frac{1}{2^{k-1}}}}{(n-1)^{\frac{1}{2^{k-2}}}}$$

taking the natural logarithm of the expression :

$$y = \frac{(n-1)^{\frac{1}{2^{k-1}}}}{(n-1)^{\frac{1}{2^{k-2}}}} \Rightarrow \ln y = \frac{1}{2^{k-1}} \cdot \ln(n-1) - \frac{1}{2^{k-2}} \cdot \ln(n-1) \Rightarrow$$

$$\lim_{n\to\infty} \frac{1}{2^{k-1}} \cdot \ln(n-1) - \lim_{n\to\infty} \frac{1}{2^{k-2}} \cdot \ln(n-1) = \lim_{n\to\infty} \frac{1}{2^{k-1}} \cdot \frac{1}{n} - \lim_{n\to\infty} \frac{1}{2^{k-2}} \cdot \frac{1}{n}$$

(using the l'Hospital rule)

$$= \frac{1}{2^{k-1}} \cdot 0 - \frac{1}{2^{k-2}} \cdot 0 = 0 - 0$$

since we used the natural logarithm

$$y = e^{\ln y} \Rightarrow \frac{(k)}{(k-1)} \cdot (e^0 - e^0) = 0$$

It follows that every $f_k(n)$ is $O(f_{k-1}(n))$: it is true to say that every function $f_k(n)$ is somewhat faster than any $f_{k-1}(n)$ in terms of running times.