Homework Assignment #1

Due: Oct 10th, 2017

CMPUT 204
Department of Computing Science
University of Alberta

Note: All logarithms are in base 2 unless specified otherwise. We denote $\mathbb{R}^+ = \{x \in \mathbb{R} : \ x > 0\}$ and $\mathbb{R}_{\geq 1} = \{x \in \mathbb{R} : x \geq 1\}$.
This assignment is partitioned into Exercises and Problems. Exercises are optional and will not be graded. Problems are mandatory and will be graded. You are, however, strongly advised to work out a full solutions for both.
Your max-grade for this assignment is 110 (+2 pts bonus). However, you **must** submit answers to Problems 1 & 2.

Exercise I. In the binary-search problem we take as input a *sorted* array A with n elements, and a key k and we return the largest element in A that which isn't strictly greater than k. That is we return

$$\max\{x \in A : \ x \leq k\} \text{ , or } \perp \text{ if all } x \in A \text{ satisfy } x > k$$

Write a pseudo-code for binary search that (a) uses recursion and (b) runs in $O(\log(n))$ time.
Answer.

```
procedure BinarySearch($A, k, f, l$)
**Precondition: $A[f \ldots l]$ is an array of sorted elements, $1 \leq f \leq l$ and $f, l \in \mathbb{N}$.
if ($f = l$) then
    if ($A[f] \leq k$) then
        return $A[f]$
    else
        return $\perp$
end if
$m \leftarrow \lceil \frac{f+l}{2} \rceil$
if ($A[m] \leq k$) then
    return BinarySearch($A, k, m, l$)
else
    return BinarySearch($A, k, f, m-1$)
```

And of course we initially invoke `BinarySearch`$(A, k, 1, n)$.
The runtime of this algorithm is $O(\log(n))$ because every-time we recurse we cut down the size of the array in half, thus after $\log(n)$ recursive calls we are down to a single element.

Exercise II. Prove that for any $n \geq 1$ and $0 \leq k \leq n$ we have that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.
Use this fact to give a recursive algorithm that computes $\binom{n}{k}$. Prove the correctness of your algorithm.
Answer. There are multiple proofs.

- One can prove that $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ using combinatorial logic: how can enumerate all subsets of size k of a given set S of size n? On the one hand it is (by definition) $\binom{n}{k}$.
  On the other hand, fix some $a \in S$. Clearly, any subset of size k either contains a or not. If it doesn't contain n, it is a k-size subset out of the remaining $n-1$ elements, and we have $\binom{n-1}{k}$ of those; if it does contain a, the remaining $k-1$ elements must come from the raining $n-1$ elements in $S \setminus \{a\}$, and we have $\binom{n-1}{k-1}$ of those.
  Altogether: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. $\square$

- A purely arithmetic proof:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \frac{(n-1)!}{(k-1)! \cdot (n-k)!} + \frac{(n-1)!}{k! \cdot (n-k-1)!}$$
$$= \frac{n!}{k! \cdot (n-k)!}\left(\frac{k}{n} + \frac{n-k}{n}\right) = \frac{n!}{k! \cdot (n-k)!} \cdot 1 = \binom{n}{k} \quad \square$$

The recursive algorithm we propose uses this recursive formula, with the base case of $k = 0$ or $k = n$.

```
procedure Rec-Binom($n, k$)
if ($k = 0$ or $k = n$) then
   return 1
else
   return Rec-Binom($n-1, k-1$) + Rec-Binom($n-1, k$)
```

Correctness of this algorithm is proven via induction on n.

Claim: For any $n \geq 0$ we have that for any $0 \leq k \leq n$ it holds that `Rec-Binom(`n, k`)` correctly outputs $\binom{n}{k}$.

Proof: Prove by induction on n. Base case: For the case of $n = 0$ the claim is only meaningful for $k = 0$. In this case, `Binom-Rec(`$0, 0$`)` returns 1 by definition and $\binom{0}{0} = 1$.

Induction step: Fix any n. Assuming the claim holds for n, we show it holds for $n+1$. Given $n+1$ and any $0 \leq k \leq n+1$ we split into cases on k.
If $k = n+1$ then `Rec-Binom(`$n+1, n+1$`)` returns 1 and indeed $\binom{n+1}{n+1} = 1$.
If $k = 0$ then `Rec-Binom(`$n+1, 0$`)` returns 1 and indeed $\binom{n+1}{0} = 1$.
Otherwise $1 \leq k \leq n$. In which case, `Rec-Binom(`$n+1, k$`)` returns the summation of the output of `Rec-Binom(`n, k`)` and `Rec-Binom(`$n, k-1$`)`. By IH, `Rec-Binom(`n, k`)`$= \binom{n}{k}$ and `Rec-Binom(`$n, k-1$`)`$=$ $\binom{n}{k-1}$. Hence, the output of `Rec-Binom(`$n+1, k$`)` is $\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$ because of the above-proven claim.
In any case, `Rec-Binom(`$n+1, k$`)`$= \binom{n+1}{k}$ and so our claim holds for $n+1$ as well. $\square$

Exercise III. True or False? Prove.

1. $\sqrt{n} \in O(n)$.

   Answer. True. Set $c = 1$ and $n_0 = 1$. Indeed, $\sqrt{n} \leq n$ for any $n > 1$ as $n = \sqrt{n} \cdot \sqrt{n}$ hence $\sqrt{n} \leq n \Leftrightarrow 1 \leq \sqrt{n} \Leftrightarrow 1 \leq n$.

2. $\sqrt{2^n} \in O(2^{\sqrt{n}})$

   Answer. False. $\sqrt{2^n} = 2^{\frac{n}{2}}$ and for large enough ns we have that $n/2 > 2\sqrt{n}$ (for any $n \geq 4^2$ in fact). Thus, for any large enough ns we have

$$\frac{\sqrt{2^n}}{2^{\sqrt{n}}} = 2^{\frac{n}{2}-\sqrt{n}} \geq 2^{2\sqrt{n}-\sqrt{n}} = 2^{\sqrt{n}} \overset{n\to\infty}{\to} \infty$$

3. $n! \in O(2^{n\log(n)})$

   Answer. True. For any $n \geq 1$ we have that $n! = 1 \cdot 2 \cdot \ldots \cdot n \leq n \cdot n \cdot \ldots \cdot n = n^n = 2^{n\log(n)}$.

4. $n! \in \Theta(2^{n\log(n)})$

   Answer. False. Since $2^{n\log(n)} = n^n$, we simply look at the ratio

$$\frac{n!}{2^{n\log(n)}} = \frac{n!}{n^n} = \frac{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n}{n \cdot n \cdot n \cdot \ldots \cdot n} \leq \frac{1}{n} \cdot \frac{2 \cdot 3 \cdot \ldots \cdot n}{n \cdot n \cdot \ldots \cdot n} \leq \frac{1}{n} \overset{n\to\infty}{\to} 0$$

   implying $n! \in o(2^{n\log(n)})$ and hence $n! \notin \Theta(2^{n\log(n)})$.

5. For any function $f : \mathbb{N} \to \mathbb{R}^+$ it holds that $o(f) \cap \omega(f) = \emptyset$.

   Answer. True. Fix g to be any function in $o(f)$ arbitrarily. Then $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ and so, by the uniqueness of the limit, we have that $\lim_{n\to\infty} \frac{g(n)}{f(n)} \neq \infty$, and so it cannot hold that $g \in \omega(f)$.

6. For any function $f : \mathbb{N} \to \mathbb{R}^+$ it holds that $\Omega(f) = \omega(f) \cup \Theta(f)$.

   Answer. False. But this answer will be disclosed only when entire HW solution is released...

   Answer. False. Let $f(n) = n$. Let $g(n) = \begin{cases} n, & n \text{ is odd} \\ n^2, & n \text{ is even} \end{cases}$.

   Clearly, $g(n) \in \Omega(f(n))$: fix $c = 1$ and $n_0 = 1$ and indeed, for any $n > 1$ we have $g(n) \geq f(n)$.

   However, it is also clear that $g(n) \notin O(f(n))$ as for any $c > 0$ and any even n larger than c we have $g(n) = n^2 > cn$. As a result, $g(n) \notin \Theta(f(n))$.

   Moreover, $g(n) \notin \omega(f(n))$ as the limit of the series $\frac{g(n)}{f(n)}$ is not defined: $\frac{g(n)}{f(n)} = \begin{cases} 1, & n \text{ is odd} \\ n, & n \text{ is even} \end{cases}$ and in particular cannot be ∞ as on odd ns the series is the constant 1.

   Thus $g \in \Omega(f)$ yet $g \notin \omega(f) \cup \Theta(f)$.

Exercise IV. Pick your favorite programming language.
For each of the following values of n — from 1000 to $50,000$ in increments of 1000 — do the following:

1. Generate an array A large enough to hold n integers.
2. Start the clock.
3. Repeat n times:
   (a) Generate an arbitrary integer.
   (b) Insert the integer into A.
4. Stop the clock and compute the runtime.

Now draw the plot of the runtime as a function of n.

However, you should repeat this experiment twice: once where the insertion of Step 3(b) is done using the standard `Insert`, and once where the insertion is done using `Insert2` (see slides of Unit 01).

Draw both plots on the same figure. Explain the two plots.

Answer. We used Matlab for running the experiment. The plots are given in Figre 1. The runtime of implementing each insertion in constant time seems like a constant but when you zoom in – one can see it actually more linear; in contrast, the runtime where each insertion touches upon all existing instances (`Insert2`) looks like a parabola (again, one needs to accommodate for implementational issues especially when Matlab is concerned). Indeed, the theory assures us that when each insertion is done is time $O(1)$ then inserting n elements takes $O(n)$; and when each insertion takes $O(A.size)$ for an array with $A.size$ existing elements then the overall runtime of inserting n elements is about $O(1+2+3+...+n) = O(n^2)$.

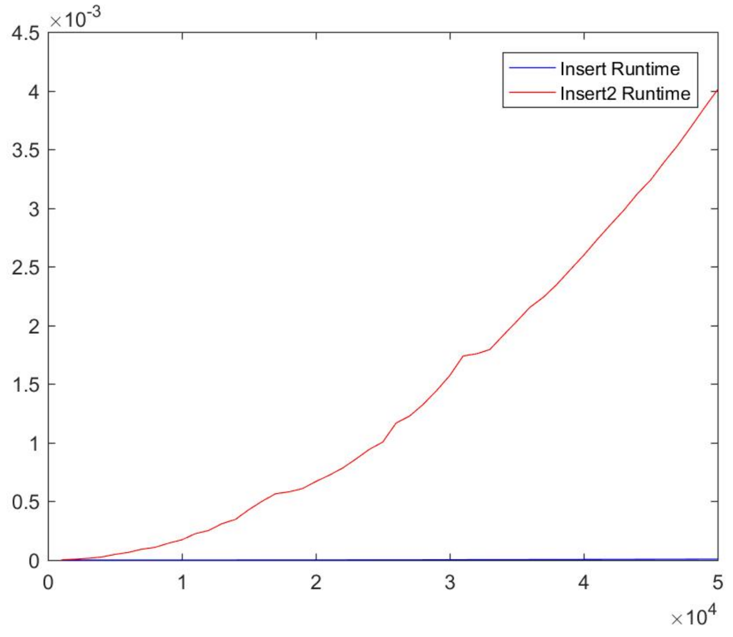

Figure 1: The runtime plots of inserting n elements into the array using `Insert()` (blue) or using `Insert2()` (red).

Here's the Matlab code of the experiment. We actually ran each experiment 10 times and averaged the runtime.

```
MyInsert.m:
----------
function [A,ASIZE] = MyInsert(A,ASIZE,ACAPACITY,x)
% Inserts x into an array A with ASIZE many elements already in A
% and a room for ACAPACITY elements overall
% returns the revised A as well as the new size of A
   if (ASIZE < ACAPACITY)
      A( (ASIZE+1) ) = x;
      ASIZE = ASIZE+1;
   end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
MyInsert2.m:
------------
function [A,ASIZE] = MyInsert2(A,ASIZE,ACAPACITY,x)
% Inserts x into an array A with ASIZE many elements already in A
% and a room for ACAPACITY elements overall
% returns the revised A as well as the new size of A
   if (ASIZE < ACAPACITY)
      for (i=(ASIZE:-1:1))
         A(i+1) = A(i);
      end
      A(1) = x;
      ASIZE = ASIZE+1;
   end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
InsertionExperiment.m:
------------------------
Ns = 1000*(1:50);

arbitraryNumber = 85;

y1 = [];
y2 = [];
trials = 10;
for (n=Ns)
        runtimes=[];
        for (t = 1:trials)
                A = zeros(n,1);
                Asize = 0;
                Acapacity = n;
                t1 = now;
                for (i=1:n)
                        x = arbitraryNumber;
                        [A,Asize] = MyInsert(A,Asize,Acapacity,x);
                end
                t2 = now;
                runtimes = [runtimes, t2-t1];
        end
        y1 = [y1,mean(runtimes)];


        runtimes=[];
        for (t = 1:trials)
                A = zeros(n,1);
                Asize = 0;
                Acapacity = n;
                t1 = now;
                for (i=1:n)
                        x = arbitraryNumber;
                        [A,Asize] = MyInsert2(A,Asize,Acapacity,x);
                end
                t2 = now;
                runtimes = [runtimes, t2-t1];
        end
        y2 = [y2,mean(runtimes)];
end
plot(Ns,y1,'b-', Ns,y2,'r-');
legend('Insert Runtime', 'Insert2 Runtime');
```

Problem 1. (10 pts) True or False? Provide a *short* explanation.

(i) $n^5(\log(n))^{29} + 52\sqrt{n^9} \in O(n^{\sqrt{529}})$ (Note: $\sqrt{529} = 23$)
Answer. True. For sufficiently large ns we have $\log^{29}(n) \leq n$, and so, for large enough ns we have $n^5 \log^{29} n + 52\sqrt{n^9} \leq n^6 + 52n^6 = 53n^6$. As $\lim_{n\to\infty} \frac{53n^6}{n^{23}} = \lim_{n\to\infty} \frac{53}{n^{19}} = 0$ the limit rule yields that $n^5 \log^{29} n + 52\sqrt{n^9} \in o(n^{\sqrt{529}})$, and in particular in $O(n^{\sqrt{529}})$.

(ii) $n^{52}(\log(n))^9 + 52\sqrt[9]{n} \in O(52^9 n^{\frac{52}{9}})$
Answer. False. $n^{52}\log^9 n + 52\sqrt[9]{n} \geq n^{52}$ for any natural n and $\lim_{n\to\infty} \frac{n^{52}}{52^9 n^{52/9}} = \frac{1}{52^9}\lim_{n\to\infty} n^{52(1-\frac{1}{9})} = \infty$. Thus $n^{52}\log^9 n + 52\sqrt[9]{n} \in \omega(52^9 n^{\frac{52}{9}})$ and cannot belong to $O(52^9 n^{\frac{52}{9}})$.

(iii) $52^{n^9} \in O(92^{n^5})$
Answer. False. $52^{n^9} = 2^{n^9\cdot\log(52)}$ and $92^{n^5} = 2^{n^5\cdot\log(92)}$. We compare the exponents: $n^9\log(52) > 2n^9$ for all naturals; whereas $n^5\log(92) < n^6 < n^9$ for all $n > \log(92)$. It follows that $n^9\log(52) - n^5\log(92) > n^9$ for large enough ns, hence $\lim_{n\to\infty} n^9\log(52) - n^5\log(92) = \infty$, thus $52^{n^9} \in \omega(92^{n^5})$.

(iv) $52^{9n} \in \Theta((52\cdot 9)^n)$.
Answer. False. $52^{9n} = 2^{n\cdot 9\cdot\log(52)}$ and $(52\cdot 9)^n = 2^{n\cdot(\log(52)+\log(9))}$. Comparing exponent: $9n\log(52) - n\cdot\log(52) - n\cdot\log(9) = 8n\log(52) - n\log(9) \geq 8n - n\log(9) = n(8 - \log(9)) > 4n$. Thus $\lim_{n\to\infty} 9n\log(52) - n\cdot\log(52) - n\cdot\log(9) = \infty$, and therefore $52^{9n} \in \omega((52\cdot 9)^n)$ implying $52^{9n} \notin \Theta((52\cdot 9)^n)$.

(v) $1.5^n + 1.29^n \in O(1.529^n)$.
Answer. True. For any n we have $1.5^n \leq 1.529^n$ and $1.29^n \leq 1.529^n$. Hence, $1.5^n + 1.29^n \leq 2\cdot 1.529^n$ for any n.

(vi) $\sqrt{1.5^n + 1.29^n} \in O(1.529^{\sqrt{n}})$.
Answer. False. For any n we have $\sqrt{1.5^n + 1.29^n} \geq 1.5^{n/2} = 2^{n\cdot\frac{\log(1.5)}{2}}$; whereas $1.529^{\sqrt{n}} = 2^{\sqrt{n}\log(1.529)}$. Comparing exponents, we look at the difference:

$$n\cdot\frac{\log(1.5)}{2} - \sqrt{n}\log(1.529) = \sqrt{n}\left(\sqrt{n}\frac{\log(1.5)}{2} - \log(1.529)\right)$$

Clearly, as $\frac{\log(1.5)}{2}$ is some positive constant and $\log(1.529)$ is some other positive constant, for all sufficiently large ns we have $\sqrt{n}\frac{\log(1.5)}{2} - \log(1.529) > 1$ implying that for all large enough ns we have $n\cdot\frac{\log(1.5)}{2} - \sqrt{n}\log(1.529) > \sqrt{n} \overset{n\to\infty}{\to} \infty$. We infer that $\sqrt{1.5^n + 1.29^n} \in \omega(1.529^{\sqrt{n}})$ and thus $\sqrt{1.5^n + 1.29^n} \notin O(1.529^{\sqrt{n}})$.

(vii) $n^{\frac{529\log\log n}{\log n}} \in O(n^{0.529})$.
Answer. True. Observe that: $n^{\frac{\log\log n}{\log n}} = 2^{\frac{\log(n)\cdot\log\log n}{\log n}} = 2^{\log\log(n)} = \log(n)$, thus $n^{\frac{529\log\log n}{\log n}} = \left(n^{\frac{\log\log n}{\log n}}\right)^{529} = (\log(n))^{529}$. As we know that $(\log(n))^k \in o(n^\epsilon)$ for any $\epsilon > 0$ and any k, we can plug in $k = 529$ and $\epsilon = 0.529$ and deduce that $\log(n)^{529} \in o(n^{0.529}) \subset O(n^{0.529})$.

(viii) There exists a constant $c > 1$ such that $n^c \in \Theta(n(\log(n))^{529})$.
Answer. False. ASOC that such a σ exists. Then we would have that there exists $C > 0$ and n_0 such that $n^c \leq Cn(\log(n))^{529} \implies C(\log(n))^{529} \geq n^{c-1} = n^{\frac{c-1}{2}}\cdot n^{\frac{c-1}{2}}$. Since for large enough n we have that $n^{\frac{c-1}{2}} \geq C$ then it holds that for large enough n we must have that $(\log(n))^{529} \geq n^{\frac{c-1}{2}}$. This contradicts the fact that for any $\epsilon > 0$ and any $k > 0$ we have $(\log(n))^k \in o(n^\epsilon)$ (in this case, for $k = 529$ and $\epsilon = \frac{c-1}{2}$).

(ix) $1^{529} + 2^{529} + \ldots + n^{529} \in \Theta(n^{530})$.
Answer. True. For every $i \leq n$ we have $i^{529} \leq n^{529}$. Thus, $1^{529} + 2^{529} + \ldots + n^{529} \leq n^{529} + n^{529} + \ldots + n^{529} = n\cdot n^{529} = n^{530}$. On the other hand, looking at only the largest $\frac{n}{2}$ summands, we have

$1^{529} + 2^{529} + \ldots + n^{529} \geq \left(\frac{n}{2}\right)^{529} + \left(\frac{n}{2}+1\right)^{529} + \left(\frac{n}{2}+2\right)^{529} + \ldots + n^{529} \geq \left(\frac{n}{2}\right)^{529} + ... + \left(\frac{n}{2}\right)^{529} = \frac{n}{2} \cdot \frac{n^{529}}{2^{529}} = \frac{1}{2^{530}} \cdot n^{530}$. As 2^{-530} is a positive constant (*ridiculously* small but still a constant), then $1^{529} + 2^{529} + \ldots + n^{529} \geq 2^{-530} n^{530}$.

(x) $\sum\limits_{i=1}^{n} \sum\limits_{j=i+1}^{n} (j-i)^{529} \in O(n^{531})$

Answer. True. For any given i between 1 and n, the sum $\sum\limits_{j=i+1}^{n} (j-i)^{529} = 1^{529} + 2^{529} + 3^{529} + ... + (n-i)^{529} \leq 1^{529} + 2^{529} + 3^{529} + ... + n^{529} \leq n^{530}$ as we have shown in the previous article. Hence, $\sum_{i=1}^{n} n^{530} \leq n \cdot n^{530} = n^{531}$, showing that $\sum\limits_{i=1}^{n} \sum\limits_{j=i+1}^{n} (j-i)^{529} \in O(n^{531})$.

Problem 2. (10 pts) Prove the following claims. Make sure you use statements such as "Let x be *any...*" or "We prove the statement by constructing the following example."

(i) There exists two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, such that $f \notin O(g)$ and $g \notin O(f)$.

Answer. We construct the two required functions as follows. Define $f(n) = \begin{cases} 2^n, & \text{if } n \text{ is even} \\ 1, & \text{if } n \text{ is odd} \end{cases}$ and $g(n) = \begin{cases} 1, & \text{if } n \text{ is even} \\ 2^n, & \text{if } n \text{ is odd} \end{cases}$. For any $c > 0$ we will always have that $f(n) > c \cdot g(n)$ for large enough *even* n and $g(n) > c \cdot f(n)$ for large enough *odd* n.

(ii) (Geometric series) Given a constant c, denote $f_c(n) = c^0 + c^1 + c^2 + ... + c^n$.

Then $f_c(n) \in \begin{cases} \Theta(c^n), & \text{if } c > 1 \\ \Theta(n), & \text{if } c = 1 \\ O(1), & \text{if } 0 < c < 1 \end{cases}$

Answer. The easiest case is when $c = 1$, as then $f_1(n) = 1 + 1 + 1 + ... + 1 = n + 1 \in \Theta(n)$. For any $c \neq 1$ we use the formula for geometric series to deduce that $f_c(n) = \frac{c^{n+1}-1}{c-1}$. Thus, for any $c > 1$ we have that $f_c(n) = \frac{c}{c-1} \cdot c^n - \frac{1}{c-1} \leq \frac{c}{c-1} \cdot c^n$, hence $f_c(n) \in O(c^n)$. Moreover, for large enough ns we have that $\frac{1}{c-1} < \frac{c}{2(c-1)} \cdot c^n$ (it is enough to have $c^n > \frac{2}{c}$, namely have $n > \log_c(2/c)$), thus $f_c(n) > \frac{c}{2(c-1)} \cdot c^n$. This proves that for any $c > 1$ we have $f_c(n) \in \Theta(c^n)$.

For any $0 < c < 1$, we can use the fact that $f_c(n) \leq \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$. As the latter is some constant, we have that $f_c(n) \in O(1)$ in this case.

(iii) For any two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, denoting $\max\{f, g\}(n) = \max\{f(n), g(n)\}$, then we have that $\max\{f, g\}(n) \in \Theta(f(n) + g(n))$.

Answer. Fix any f and g such that for all ns we have $f(n) \geq 0$ and $g(n) \geq 0$. Therefore, for any n, $\max\{f(n), g(n)\} \leq f(n) + g(n)$. Also, since for every n we have $2\max\{f(n), g(n)\} \geq f(n) + g(n)$ then for every n it holds that $\max\{f, g\}(n) \geq \frac{1}{2}(f(n) + g(n))$. So $\max\{f, g\}(n) \in \Theta((f + g)(n))$.

(iv) For any functions $f, g : \mathbb{N} \to \mathbb{R}^+$, $\sqrt{f(n)} + \sqrt{g(n)} \in \Theta(\sqrt{f(n) + g(n)})$.

Answer. Fix any f and g such that for all ns we have $f(n) \geq 0$ and $g(n) \geq 0$. Since $\sqrt{f(n)} \leq \sqrt{f(n) + g(n)}$ and $\sqrt{g(n)} \leq \sqrt{f(n) + g(n)}$ then we have that $\sqrt{f(n)} + \sqrt{g(n)} \leq 2\sqrt{f(n) + g(n)}$ for any n. Conversely, since $f(n), g(n) \geq 0$ then the fact that $(\sqrt{f(n)} + \sqrt{g(n)})^2 = f(n) + 2\sqrt{f(n) \cdot g(n)} + g(n) \geq f(n) + g(n)$ implies that $\sqrt{f(n)} + \sqrt{g(n)} \geq \sqrt{f(n) + g(n)}$.

(v) There exists two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ such that $f(n) \in \Theta(g(n))$ yet $2^{f(n)} \notin \Theta(2^{g(n)})$;
(2pt Bonus) and there exists two functions $f, g : \mathbb{N} \to \mathbb{R}^+$ such that $2^{f(n)} \in \Theta(2^{g(n)})$ yet $f(n) \notin \Theta(g(n))$.

Answer. We construct the required two functions as follows. Consider $f(n) = n$ and $g(n) = 2n$, where clearly $f(n) \in \Theta(g(n))$ yet it is also clear that $\frac{2^{f(n)}}{2^{g(n)}} = 2^{n-2n} = 2^{-n} \overset{n\to\infty}{\to} 0$ showing that $2^{f(n)} \in o(2^{g(n)})$.

As for the bonus part, we give the following two functions by considering $f(n) = \frac{1}{n}$ and $g(n) = \frac{1}{n^2}$. Clearly for $n \geq 2$ we have $g(n) \leq \frac{1}{2}f(n)$ thus the limit rule proves that

$$\frac{2^{1/n}}{2^{1/n^2}} \leq 2^{1/n} \overset{n\to\infty}{\to} 2^0 = 1$$
$$\text{and } \frac{2^{1/n}}{2^{1/n^2}} = 2^{\frac{1}{n} - \frac{1}{n^2}} \geq 2^{\frac{1}{n} - \frac{1}{2n}} = 2^{\frac{1}{2n}} \overset{n\to\infty}{\to} 2^0 = 1$$

Thus $2^{f(n)}/2^{g(n)} \overset{n\to\infty}{\to} 1$ so by the limit rule $2^{f(n)} \in \Theta(2^{g(n)})$. However, it is clear that $f(n) \notin \Theta(g(n))$ as $\frac{1/n}{1/n^2} = n \overset{n\to\infty}{\to} \infty$, hence $f(n) \in \omega(g(n))$.

Problem 3. (20 pts) Order the following list of functions by increasing big-O growth rate. Group together (for example, by underlining) those functions that are big-Θ of one another.

$529n$	5^{2^9}	$\log((\log n)^{529})$	$(\log(n))^{529}$	$2^{\log(n+529)}$
$\log(2^{2^{\cdot^{\cdot^{2}}}}\}^{n \text{ times}})$	$2^{\frac{\log(n)}{529}}$	$2^{(\log(n)+529)}$	$2^n + 2^{n-1} + 2^{n-2} + \ldots + 2^0$	4^n
$529n^{0.529} + 2^{\sqrt{\log(n)}}$	$2^{\log^*(n)}$	$n^2 \log_{529}(n) + 2^{529}n^2$	$n\log(n^{529})$	$n\log(529n)$
$529n^{0.529} + 925n^{9.25}$	$n^3 - n^{2.99}(\log(n))^{529}$	$\lfloor 529n(\log(n))^2 \rfloor$	$4^{\log n}$	$\log_{529}(n)$

Answer.

$$5^{2^9} \in \Theta(1) \quad (1)$$

$$2^{\log^*(n)} \in o\left(\underbrace{\log(\log(\log(\ldots\log}_{c}(n)\ldots))\right) \text{ for any fixed natural } c \quad (2)$$

$$\log((\log n)^{529}) \in \Theta(\log\log(n)) \quad (3)$$

$$\log_{529} n \in \Theta(\log(n)) \quad (4)$$

$$(\log(n))^{529} \in o(n^c) \text{ for any } c > 0 \quad (5)$$

$$2^{\frac{\log(n)}{529}} \in \Theta(n^{\frac{1}{1975}}) \quad (6)$$

$$529n^{0.529} + 2^{\sqrt{\log(n)}} \in \Theta(n^{0.529}) \quad (7)$$

$$529n\ ,\quad 2^{\log(n+529)}\ ,\quad 2^{(\log(n)+529)} \in \Theta(n) \quad (8)$$

$$n\log(529n)\ ,\quad n\log(n^{529}) \in \Theta(n\log(n)) \quad (9)$$

$$\lfloor 529n(\log(n))^2 \rfloor \in \Theta(n(\log(n))^2) \quad (10)$$

$$4^{\log n} \in \Theta(n^2) \quad (11)$$

$$n^2\log_{529} n + 2^{529}n^2 \in \Theta(n^2\log(n)) \quad (12)$$

$$n^3 - n^{2.99}(\log(n))^{529} \in \Theta(n^3) \quad (13)$$

$$529n^{0.529} + 925n^{9.25} \in \Theta(n^{9.25}) \quad (14)$$

$$2^n + 2^{n-1} + 2^{n-2} + \ldots + 2^0 \in \Theta(2^n) \quad (15)$$

$$4^n \in \Theta(2^{2n}) \quad (16)$$

$$\log(2^{2^{\cdot^{\cdot^{2}}}}\}^{n \text{ times}}) \in \Theta(2^{2^{\cdot^{\cdot^{2}}}}\}^{(n-1) \text{ times}}) \quad (17)$$

Problem 4. (15 pts) We define a series using the following rule: $a_0 = 0$, $a_1 = 1$ and $a_n = 2a_{n-1} + a_{n-2}$ for any $n \geq 2$.

(i) (3 pts) Denote $\phi = 1 + \sqrt{2}$. Let $f(x) = x^2 - 2x - 1$. Show that $f(\phi) = 0$, and that $f'(x) > 0$ for any $x > 1$. Infer that for any c, d such that $1 < c < \phi < d$ we have that $2c + 1 > c^2$ and $2d + 1 < d^2$.
Answer. It is merely a calculation to check that

$$f(\phi) = \phi^2 - 2\phi - 1 = (1+\sqrt{2})^2 - 2(1+\sqrt{2}) - 1 = 1 + 2\sqrt{2} + 2 - 2 - 2\sqrt{2} - 1 = 0$$

It is also simple to see that $f'(x) = 2x - 2$ and so for any $x > 1$ we have that $f'(x) = 2x - 2 > 2\cdot 1 - 2 = 0$. The two facts combined show that f is monotonically increasing on the interval $(1, \infty)$ and since $f(\phi) = 0$ it means that on $(1, \phi)$ the function f is negative, and on (ϕ, ∞) the function is positive. Namely, for any $c \in (1, \phi)$ we have $c^2 - 2c - 1 < 0 \Rightarrow c^2 < 2c + 1$, and for any $d > \phi$ we have $d^2 - 2d - 1 > 0 \Rightarrow d^2 > 2d + 1$.

(ii) (7 pts) Prove that for any two constants c, d such that $1 < c < \phi < d$ we have that $a_n \in O(d^n)$ and that $a_n \in \Omega(c^n)$.
Answer. Fix c, d as required. We prove that for any $n \geq 1$ we have $\frac{1}{3}c^n \leq a_n \leq d^n$ by induction on n.
(Recall: $a_0 = 0, a_1 = 1, a_2 = 2 \cdot 1 + 0 = 2$.)
For $n = 1$: $a_1 = 1$, $c^1 < \phi < 3$ and $d^1 > \phi > 1$ hence $\frac{1}{3}c^1 < 3/3 = 1 = a_1 < d^1$.
For $n = 2$: $a_2 = 1$, $c^2 < c + 1 < \phi + 1 < 4$ and $d^2 > d + 1 > 1 + 1 = 2$, hence $\frac{1}{3}c^2 < 4/3 < 2 = a_2 < d^2$.
The induction step: Fix any $n > 2$. Assuming that $\frac{1}{3}c^i \leq a_i \leq b^i$ for all $1 \leq i < n$ we show the required holds for n as well.
Indeed,

$$a_n = 2a_{n-1} + a_{n-2} \leq 2d^{n-1} + d^{n-2} = d^{n-2}(2d+1) \overset{\text{by (i)}}{<} d^{n-2} \cdot d^2 = d^n$$
$$a_n = 2a_{n-1} + a_{n-2} \geq \tfrac{1}{3}\left(2c^{n-1} + c^{n-2}\right) = \tfrac{c^{n-2}}{3}(2c+1) \overset{\text{by (i)}}{>} \tfrac{c^{n-2}}{3}c^2 = \tfrac{1}{3}c^n$$

This implies that for $c_1 = 1$ and $n_0 = 1$, we have that for all $n \geq n_0$, $a_n \leq c_1 d^n$; and for $c_2 = \frac{1}{3}$ and $m_0 = 1$ we have that for all $n \geq m_0$ it holds that $a_n \geq c_2 c^n$. Hence $a_n \in O(d^n)$ and $a_n \in \Omega(c^n)$. $\square$

(iii) (5 pts) Prove that for any c, d such that $1 < c < \phi < d$ we have that $a_n \in o(d^n)$ and that $a_n \in \omega(c^n)$. (Hint: use (ii))
Answer. Fix c, d such that $1 < c < \phi < d$. Let us now pick some $c' \in (c, \phi)$ and $d' \in (\phi, d)$. (For example, $c' = \frac{c+\phi}{2}, d' = \frac{d+\phi}{2}$.) As $c < c' < \phi$ and $\phi < d' < d$ it follows that $\frac{c'}{c} > 1 > \frac{d'}{d}$.

Due to (ii), we know that there exists $c_1, c_2 > 0$ and some n_0 such that for all $n \geq n_0$ it holds that $c_1(c')^n \leq a_n \leq c_2(d')^n$. Therefore,

$$\lim_n \frac{a_n}{c^n} > \lim_n \frac{c_1(c')^n}{c^n} = c_1 \cdot \lim_n \left(\frac{c'}{c}\right)^n = \infty$$
$$\lim_n \frac{a_n}{d^n} < \lim_n \frac{c_2(d')^n}{d^n} = c_2 \cdot \lim_n \left(\frac{d'}{d}\right)^n = c_2 \cdot 0 = 0$$

Proving that $a_n \in \omega(c^n)$ and $a_n \in o(d^n)$. $\square$

Problem 5. (15 pts)
(i) (7 pts) Give a *recursive* pseudo-code for an algorithm solving the following problem. Input: a linked-list L with integers; Output: (a pointer to) the sub-linked-list whose *head* holds the largest element in L or `nil` if L is empty.
Example: suppose L has 6 elements: $\boxed{1} \rightarrow \boxed{9} \rightarrow \boxed{6} \rightarrow \boxed{44} \rightarrow \boxed{3} \rightarrow \boxed{15}$, then the output should be the sub-list whose head is 44: $\boxed{44} \rightarrow \boxed{3} \rightarrow \boxed{15}$.
State a formal claim, and provide a formal proof, as to the correctness of your pseudo-code.
Answer.

```
procedure FindMax($L$)
** returns the sub-linked-list whose head holds the largest elements in $L$.
if ($L$ =nil) then
   return nil
$res$ ←FindMax($L.next$)
if ($res$ =nil   or   $L.head.data > res.head.data$) then
   return $L$
else
   return $res$
```

Claim: For any linked-list L of integers, `FindMax(`L`)` returns a list whose head holds the largest elements in L (or `nil` if L is empty).
Proof: We prove the required by induction on n, the number of elements in L.
Base case, $n = 0$: First, consider the special case of an empty list L =`nil`. Here `FindMax(`L`)` immediately returns `nil` as required.
Induction step: Fix any natural n. Assuming the claim holds for any linked-list of integers that holds n elements, we show it is also true for any linked-list holding $n + 1$ elements.
Since $n + 1 \geq 1$ then L isn't empty, namely, $L \neq$`nil`. The code therefore invokes `FindMax(`$L.next$`)` which is a linked-list with only n elements, so by IH res is now the sub-linked-list of $L.next$ whose *head* is the largest among all elements in $L.next$. Now, in the special case of a list L with a single element we have that $L.next$ =`nil` thus res =`nil`, and so we return L — which, for the case of list of length 1, is the only elements in L and thus the largest element in L. Otherwise, res is a linked-list such that $res.head$ is the largest among all elements in $L.next$. The code then compares $res.head$ with $L.head$: if $L.head$ is the largest, then $L.head$ is larger than the max-element in $L.next$ thus is the largest element in L altogether, and so we return L. Otherwise $L.head \leq res.head$ which means $res.head$ is the largest among all elements in L, and indeed the code returns res. □
(ii) (8 pts) Give a pseudo-code that executes `SelectionSort` on a list L of integers (repeatedly finds the largest element in L and puts it at the end of the unsorted elements). Claim (and prove) formally the correctness of your algorithm.
Here is an example of the code for `SelectionSort` on an *array*:

```
procedure SelectionSort($A, 1, n$)
for ($j$ from $n$ downto 1) do
    $ind$ ←FindMax($A, 1, j$)        ** finds the index of the largest element in $A[1, ...j]$
    exchange $A[ind] \leftrightarrow A[j]$
```

You should write a similar code for a *linked-list.*
Hint: It is probably easiest to create a new list $L2$, find and *remove* the largest element in L and insert this element into the right place in $L2$. You may refer to the code for `InsertHead()` and `DeleteHead` from the slides without re-writing them.

Answer.

```
procedure SelectionSort($L$)
$L2$ ←new empty list
while ($L$ ≠nil) do
   $res$ ←FindMax($L$)
   Insert($L2, res.head.data$)
   DeleteHead($res$)
return $L2$      $**$ or set $L \leftarrow L2$
```

Claim: For any linked-list L of integers, `SelectionSort(`L`)` returns a list that holds the same elements as in L only in ascending order.
Proof: We prove the claim using the following loop-invariant: at the beginning of each `while`-loop iteration, (i) all elements in L are $\leq$ than all elements in $L2$, (ii) $L2$ is sorted in ascending order, and (iii) all elements that originally appeared in L are either in L or in $L2$ but not in both.
Initialization: initially $L2$ is empty, so the invariant vacuously holds.
Maintenance: Assuming the claim holds in the beginning of some iteration, we show it also holds in the beginning of the next one.
Note how we iterate through the loop only when L isn't empty. Therefore, the correctness of `FindMax(`L`)` sets res to be the sub-linked-list of L whose *head* holds the largest element in L. Because of the loop-invariant, we also know that $res.head$ (which is an element of L) is $\leq$ than all elements in $L2$. By taking $res.head$ out of L and placing it in $L2$ we maintain invariant (iii) (since this is the only elements that moves between L and $L2$ in each iteration), and since $res.head$ is the largest among all elements in L then invariant (i) is also maintained: the elements remaining in L were $\leq$ than all elements originally in $L2$ and $\leq res.head$ which was moved to $L2$. Lastly, by inserting $res.head$ as the new *head* (the first element) of $L2$ we maintain the invariant (ii) that $L2$ is sorted: its first element is $\leq$ than all other elements, and all other elements were sorted already.
Termination 1: In each iteration we remove an element from L. Thus, after we remove all elements from L it becomes empty (set to `nil`). Hence the loop eventually concludes.
Termination 2: When the `while`-loop halts $L =$`nil`, namely, is empty. By invariant (iii) this means that $L2$ must hold all elements that were originally in L. By invariant (ii) we have that $L2$ is sorted in ascending order. So by returning $L2$ we return a linked-list that holds the same elements as in L only in a sorted order. □

Problem 6. (20 pts) Consider the following program:

```
procedure Foo($A, f, l$)
**Precondition: $A[f \dots l]$ is an array of integers, $f, l$ are two naturals $\geq 1$ with $f \leq l$.
if ($f = l$) then
    return $(A[f])^2$        ** i.e. $A[f] \times A[f]$
else
    $m \leftarrow \lfloor \frac{f+l}{2} \rfloor$
    return Foo($A, f, m$)+ Foo($A, m+1, l$)
end if
```

(i) (2 pts) In a sentence: what does Foo return? State a formal claim.
Answer. We claim that Foo returns the sum of the squares of all elements in the subarray $A[f...l]$.

(ii) (4 pts) Prove that indeed Foo returns what you claim it returns. (You may assume all elements in A are unique.)
Answer. We prove that Foo returns the sum of the squares of all elements in $A[f...l]$ by induction on $n = l - f + 1$.
Base case: $n = 1$ or $l - f = 0$. Here $f = l$ and Foo returns $(A[f])^2$ which is clearly the only element, and thus the sum of the squares of all the entries in $A[f...f]$.
Induction step: Fix some $n \geq 2$. Assuming the claim holds for any f, l with $l - f + 1 < n$ we show it also holds for f, l such that $f - l + 1 = n$.
Fix any l, f such that $f - l = n - 1$. We have that

$$m - f = \lfloor \tfrac{f+l}{2} \rfloor - f = \lfloor \tfrac{f+f+(l-f)}{2} \rfloor - f = \lfloor \left(f + \tfrac{l-f}{2}\right) \rfloor - f = f + \lfloor \tfrac{l-f}{2} \rfloor - f = \lfloor \tfrac{l-f}{2} \rfloor$$
$$l - (m+1) = l - 1 - \lfloor \tfrac{f+l}{2} \rfloor \geq l - 1 - \tfrac{f+l}{2} = \tfrac{2l-f-l}{2} - 1 = \tfrac{l-f}{2} - \tfrac{1}{2} - \tfrac{1}{2} \leq \lfloor \tfrac{l-f}{2} \rfloor - \tfrac{1}{2}$$

As $n \geq 2$ we have that $f - l \geq 1$ which means $\lfloor \frac{l-f}{2} \rfloor < l - f$. It follows that both recursion calls are done over subarrays where the difference between their resp. endpoints is at most $\lfloor \frac{l-f}{2} \rfloor < n - 1$. Therefore we can apply the IH on both recursive calls, thus Foo(A, f, m) returns the sum of the squares of all elements in $A[f...m]$ and Foo($A, m+1, l$) returns the sum of the squares of all elements in $A[(m+1)...l]$. Since Foo(A, f, l) returns the sum of the two, it follows that Foo(A, f, l) returns the sum of the squares of all elements in $A[f...l]$. ■

(iii) (4 pts) Using induction, argue that Foo invokes the ^{2}-operator at most n times.
Answer. Let $T(n)$ be the function defined as the number of times that Foo invokes a call to the ^{2}-operator on an (input) array of size n. We prove that $T(n) \leq n$ using induction on n.
Base case: For $n = 1$, the case where $f = l$, we have that $T(n) = 1$ and indeed $1 \leq 1$.
Induction step: Fix $n > 1$. Assuming $T(i) \leq i$ for any $1 \leq i < n$, we show that $T(n) \leq n$ as well.

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) \leq \lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil \overset{(*)}{=} n$$

where the equality marked in $(*)$ holds trivially if n is even, and if n is odd we have $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = \frac{n-1}{2} + \frac{n+1}{2} = \frac{2n}{2}$. □

(iv) (3 pts) Write a non-recursive version of Foo(A, f, l) (in pseudo-code).
Answer.

```
procedure Foo2($A, f, l$)
**Precondition: $A[f \dots l]$ is an array of elements, $1 \leq f \leq l$ and $f, l \in \mathbb{N}$.
**Postcondition: Returns the sum of all element of $A[f \dots l]$.
$sum \leftarrow 0$
for ($i$ from $f$ to $l$) do
    $sum \leftarrow sum + (A[i])^2$
end for
return $sum$
```

(v) (4 pts) Claim and prove formally the correctness of your algorithm.
Answer. We prove it using the following loop invariant: "At the start of each for-loop iteration, *sum* holds the sum of the squares of all elements in $A[f...(i-1)]$."

Initialization: initially $i = f$ and $sum = 0$, and clearly 0 is the sum of the squares of all elements in the empty subarray $A[f...f-1]$.
Maintenance: assuming the invariant holds at the beginning of the loop, then we now add $(A[i])^2$ to sum. Thus, after the loop executes *sum* holds the sum of the squares of all elements in $A[f,...,i]$. So at the beginning of the next loop (one i is incremented) it is the same as $A[f,...,(i^{\text{new}}-1)]$.
Termination #1: the for loop clearly terminates as we only increment i by 1 in each iteration.
Termination #2: at the last of for-loop iterations we have that $i = l+1$ which means that at the end of the loop it holds that *sum* is the sum of the squares of all element in $A[f...(l+1-1)] = A[f...l]$. ■

(vi) (3 pts) Find the worst case running time of this new program and express it in Θ notation.
Answer. The for-loop iterates for n times where $n = l - f + 1$ is the size of array A, and at each iteration it takes constant time. Therefore the running time is $\Theta(n)$.

Problem 7. (20 pts) Professor Harry Potter has devised an ingenious concoction: it looks like water, tastes like water and resembles water in any way, shape or form, and yet consuming even a single drop of this potion will cause the drinker's skin to turn bright blue for a day. However there's a delay in the effect of the potion — it takes about 45 minutes for the effect to kick in.

Alas, Prof. Potter's students, showing him the same respect undergraduates show to their professors (sigh), have scraped off the bottle's label and hid it among $n-1$ water bottles. So now Prof. Potter's cabinet holds n identically looking bottles, where one of them is the turn-skin-to-blue potion and the rest contain very ordinary water. To make matters worse, Prof. Potter has no more than 1 hour to find the potion before potion-class begins. Luckily, he can still call on a *few* of his trusted grad-students (and on you!) to assist him in finding the potion.

Help Prof. Potter! Design an algorithm that finds which one of n bottles contains the special potion in a single hour, using no more than $\lceil \log(n) \rceil$ tasters. Prove your claims.

Hint: Start by thinking recursively. The cases of $n = 1$ or $n = 2$ are easy. Assuming you have a "tasting scheme" for n bottles with k drinkers, devise a potion-finding tasting scheme for $2n$ bottles using $k+1$ tasters. Then "unravel" the recursion to see what bottles the j-th taster drinks from. It is OK if you think of n as a power of 2 for the purpose of finding the algorithm, but your algorithm should work for any n. (Also, it is probably a tad more convenient to number the bottles from 0 to $n-1$ rather than from 1 to n.)

Answer.

First, let's give a recursive algorithm. Clearly, if $n = 1$ then it is evident that this single bottle contains the potion, so no tasters needed. Assuming we have a scheme for n bottles, here's how we devise a scheme for $2n$ bottles using one more taster. We partition the $2n$ bottles into two equal sets with n bottles in each, A and B. Using the recursion, there's a tasting scheme for k tasters to determine which of the bottles in A is poisoned (if indeed the potion is in A), and the same for B. So let A_j and B_j denote the bottles which drinker j drinks from in each scheme — we now give drinker j a cup containing liquid from all the bottles in $A_j \cup B_j$. The $k+1$-th taster drinks a cup containing liquid from all the bottles in A and not from any bottle in B.

We argue that this is a valid scheme for finding the potion among the $2n$ bottles. If drinker $k+1$ turns blue then the potion is in A; otherwise, it is in B. Once we've established which of the two sets the potion is in, the colors of drinkers $1, 2, .., k$ tell us which bottle in A or B is the sought-after potion.

Let's unravel the recursion to see what happens. For each n we number the bottles from 0 to $n-1$.

- If $n = 1$, no tasters.
- If $n = 2$, then we have a single taster, drinking from bottle 0 but not from bottle 1.
- If $n = 4$, then we have two drinkers. The first taster drinks from the first bottle from the set $\{0, 1\}$ and from the first bottle in the set $\{2, 3\}$ – namely, the first taster drinks from bottles 0 & 2; the second drinks from bottles 0 & 1.
- If $n = 8$, then we have 3 tasters. The first taster drinks from the first and the third bottles in set $\{0, 1, 2, 3\}$ and from the first and third bottle from the set $\{4, 5, 6, 7\}$ – i.e., drinks from bottles $\{0, 2, 4, 6\}$; the second taster drinks from the first two bottles from the set $\{0, 1, 2, 3\}$ and from the first two bottles from the set $\{4, 5, 6, 7\}$ — i.e., drinks from bottles $\{0, 1, 4, 5\}$; the third taster drinks from bottles $\{0, 1, 2, 3\}$.
- If $n = 16$, then we have 4 tasters: first taster drinks from bottles $\{0, 2, 4, 6, 8, 10, 12, 14\}$; second taster drinks from $\{0, 1, 4, 5, 8, 9, 12, 13\}$; third taster drinks from $\{0, 1, 2, 3, 8, 9, 10, 11\}$ and the fourth drinks from $\{0, 1, 2, 3, 4, 5, 6, 7\}$.

The pattern now emerges: drinker j drinks from all the bottles that when written in binary, the j-th bit is 0. This is the algorithm we use:

```
procedure FindPotion($n$)
$J \leftarrow \lceil\log(n)\rceil$
for ($j$ from 1 to $J$) do
   for ($i$ from 0 to $n-1$) do
      if (($j$th bit in the binary representation of $i$) $=0$) then
         $j$ drinks from bottle $i$
WAIT 50 MINUTS
$B \leftarrow$ an array of size $J$
for ($j$ from 1 to $J$) do
   if (drinker $j$ is blue) then
      $B[j] \leftarrow 0$
   else
      $B[j] \leftarrow 1$
$Potionbottle \leftarrow$ the number whose binary representation is $B$
```

Clearly our algorithm uses $J = \lceil\log(n)\rceil$ tasters. As we traverse all bottles, it is clear that the jth taster drinks from all bottles with the jth bit $= 0$ in their binary representation. Therefore if the jth taster turns blue the potion has its jth bit set to 0, o/w the potion bottle jth bit is 1. Thus B is the binary representation of the bottle holding the potion.

Just so you learn something. Remember that in class we talked about how our abstract machine has this Random Access property, that allows it to reach each memory cell in $O(1)$? Well, in your less abstract computer, how does your CPU manages to get this random access capabilities?
As opposed to the abstract machine, the memory is in your computer pretty finite. Only $n = 2^{64}$ possible cells. And the CPU gets a 64-bit address and accesses the memory-cell whose address was given almost instantaneously — by doing something very similar to what Harry Potter did here (or you) in finding the right bottle. The memory access unit is basically one big AND gate, where the each bit in the 64-bit address controls which cells we select: will we do an AND of the $n/2$ cells whose jth bit is 1 or the $n/2$ cells whose bit is 0. Altogether, the specific 64-bits provided in the address determine what unique cell among the possible 2^{64} cells goes into the CPU's registries.