Homework Assignment #2

Due: Noon, Oct 23, 2017
Submit **printed** solutions via eClass
Submit **handwritten** solutions at dropbox on CSC level 1

CMPUT 204
Department of Computing Science
University of Alberta

Note: All log's are in base 2, unless specified otherwise.

You can use the fact $H(n) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$, and you might also require the fact that for the Fibonacci number of n, $F(n)$, we have $F(n) = \Theta(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

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.

Exercise I. Solve the following recurrence relations. Unless specified otherwise, you may assume $T(n) = O(1)$ for $n = 0, 1, 2$ or any other small constant.

1. $T(1) = 0$ and $T(n) = 1 + T(\log(n))$.

   Answer. Using iterative substitution,

$$\begin{aligned} T(n) &= 1 + T(\log(n)) \\ &= 1 + 1 + T(\log\log(n)) \\ &= 1 + 1 + 1 + T(\log\log\log(n)) \\ &= \underbrace{1 + 1 + \ldots + 1}_{k} + T(1) \\ &= k + 0 = k \end{aligned}$$

   for k such that $\underbrace{\log(\log(\ldots\log(n)))}_{k} = 1$. This is precisely $\log^*(n)$. Thus $T(n) = \log^*(n) + O(1) = \Theta(\log^*(n))$.

   We argue that for any $n = 2^{2^{\cdot^{\cdot^{2}}}}\}k \text{ times}$ we have $T(n) = k$. For $k = 0$ we have $n = 1$ and indeed $T(1) = 0$.

   Fix k, assuming the claim holds for $n = 2^{2^{\cdot^{\cdot^{2}}}}\}k \text{ times}$ we show it also holds for $n = 2^{2^{\cdot^{\cdot^{2}}}}\}k+1 \text{ times}$.
   Indeed $T(n) = T(2^{2^{\cdot^{\cdot^{2}}}}\}k+1 \text{ times}) = 1 + T(2^{2^{\cdot^{\cdot^{2}}}}\}k \text{ times}) \overset{\text{I.H}}{=} 1 + k$. ■

2. $T(n) = 3T(n/3) + \sqrt{n^3}$.

   Answer. Using Master Theorem, $a = 3$, $b = 3$, and $f(n) = n^{3/2}$ so $c = 3/2$. As $c > 1 = \log_3(3)$, and since $3\left(\frac{n}{3}\right)^{3/2} = \frac{3}{3^{3/2}}n^{3/2} < \frac{3}{5}n^{3/2}$ then case (3) holds and $T(n) = \Theta(n^{3/2})$.

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

   Answer. Assuming that n is an odd number, using iterated substitution:

$$\begin{aligned} T(n) &= T(n-3) + 3\log n \\ &= T(n-6) + 3\log(n-2) + 3\log n \\ \ldots \\ &= T(1) + 3\sum_{i=0}^{\lfloor \frac{n}{3} \rfloor} \log(n-3i) \end{aligned}$$

   To give a formal proof, it is easier now to first argue $T(n) = T(1) + 3\sum_{i=0}^{\lfloor \frac{n}{3} \rfloor} \log(n-3i)$ by induction, then give asymptotic bounds of the sum.

   Base case: For $n = 1$ the summation is solely over $i = 0$ so we have $T(1) + \log(1-0) = T(1) + 0 = T(1)$ as required.

   Induction step: Fix any $n \geq 1$. Assuming the claim holds for n we show it also holds for $n + 3$.

Indeed

$$\begin{aligned} T(n+3) = T(n) + (n+3)\log(n+3) &\overset{\text{IH}}{=} T(1) + 3\sum_{i=0}^{\lfloor \frac{n}{3} \rfloor} \log(n-3i) + 3\log(n+3) \\ &= T(1) + 3\sum_{i=1}^{\lfloor \frac{n}{3} \rfloor + 1} \log(n+3-3i) + \log(n+3-0) \\ &= T(1) + 3\sum_{i=0}^{\lfloor \frac{n+3}{3} \rfloor} \log(n+3-3i) \end{aligned}$$

as required.

We now get an asymptotic bound on $T(n) = T(1) + 3\sum_{i=0}^{\lfloor \frac{n}{3} \rfloor} \log(n-3i)$. Clearly, the above sum is upper bounded by $3 \cdot \frac{n}{3}\log(n) = n\log(n)$ so $T(n) \in O(n\log n)$. On the other hand, the first $\frac{n}{6}$ terms in the above sum are all at least $\log(n/2) = \log(n) - 1$. So the above sum is lower bounded by $3 \cdot \frac{n}{6}\log(\frac{n}{2}) \geq \frac{n}{2}\log(n) - \frac{n}{2}$, thus $T(n) \in \Omega(n\log n)$. Therefore, $T(n) \in \Theta(n\log n)$.

4. $T(n) = 4T(\frac{n}{7}) + n^{1.47474747...}$.

   Answer. Using Master Theorem: $a = 4$, $b = 7$, and $f(n) = n^{1.47474747...}$ with $c = 1.474747 > 1 > \log_7(4)$. Since $4 \times (\frac{n}{7})^{1.47474747...} = \frac{4}{7^{1.4747474...}} n^{1.47474747...} \leq \frac{4}{7} n^{1.474747...}$. Hence case (3) holds and Thus $T(n) \in \Theta(n^{1.47474747...})$.

5. Let $c > 0$ be some constant. $T(n) = T(n-1) + n^c$ with $T(0) = 0$.
   (Hint: You can use HW1.)

   Answer. Iterative substitution gives that

$$\begin{aligned} T(n) &= n^c + T(n-1) \\ &= n^c + (n-1)^c + T(n-2) \\ &= n^c + (n-1)^c + (n-2)^c + T(n-3) \\ &\vdots \\ &= n^c + (n-1)^c + ... + 2^c + 1^c \end{aligned}$$

   Based on HW1, we know this solves to $\Theta(n^{c+1})$. Indeed, let's prove by induction that $T(n) = \sum_{i=1}^{n} i^c$.

   Base case: $n = 0$, and indeed $T(0) = 0$.

   Induction step: Fix $n \geq 0$. Assuming $T(n) = \sum_{i=1}^{n} i^c$, we show the required also holds for $n+1$.
   $T(n+1) = (n+1)^c + T(n) = \sum_{i=1}^{n} i^c + (n+1)^c = \sum_{i=1}^{n+1} i^c$.

   Plugging in HW1's question, we have $T(n) = \sum_{i=1}^{n} i^c \in \Theta(n^{c+1})$.

Exercise II. Your boss has given you an assignment: to devise the fastest algorithm you can for solving a particular problem whose input is an array A with n elements. After mulling the problem over for a few days you come up with a few alternatives. Which alternative do you prefer and why? Explain.

- Algorithm 1:
  Iterate over each element in A and do at most $100n$ operations per element.
- Algorithm 2:
  For inputs of size $n > 1$:
  Repeat a hundred times the procedure of: (i) recursing over an input of size $n/2$ and (ii) make a single operation.

Answer. We prefer the first algorithm.

Algorithm 1 makes $O(n)$ iterations and in each iteration makes $O(n)$ steps. Hence its runtime is $O(n^2)$.

Algorithm 2's runtime can be expressed by the recurrence relation: $T(n) = 100(T(n/2) + 1) = 100T(n/2) + 100$.
Using Master Theorem, with $a = 100$, $b = 2$ and $f(n) = 100$, we can easily see that $100 \in O(1) \subset o(n) \subset o(n^{\log(100)})$, and so case 1 applies and $T(n)$ solves to $T(n) \in \Theta(n^{\log(100)})$.

As $100 > 4$ we have that $\log(100) > 2$ and so $n^2 \in o(n^{\log(100)})$. Thus we prefer the first algorithm.

Problem 1. (20 pts) Find asymptotic upper/lower bounds for $T(n)$. Show how you derived the solution and prove your bound.
Unless specified otherwise, assume that in each case $T(1) = 1$ (or any small constant).

(a) (4 pts) $T(n) = T(n-1) + \frac{1}{n}$ with $T(0) = 0$.
Answer. Using iterated substitution:

$$\begin{aligned}
T(n) &= \tfrac{1}{n} + T(n-1) \\
&= \tfrac{1}{n} + \tfrac{1}{n-1} + T(n-2) \\
&= \tfrac{1}{n} + \tfrac{1}{n-1} + \tfrac{1}{n-2} + T(n-3) \\
&= \tfrac{1}{n} + \tfrac{1}{n-1} + \tfrac{1}{n-2} + \tfrac{1}{n-3} + T(n-4) \\
&= ... = \tfrac{1}{n} + \tfrac{1}{n-1} + \tfrac{1}{n-2} + \tfrac{1}{n-3} + ... + \tfrac{1}{2} + \tfrac{1}{1} + 0 = \sum_{k=1}^{n} \tfrac{1}{k} = H(n)
\end{aligned}$$

We prove that for all n we have $T(n) = H(n)$ via induction.
$T(1) = 1 + T(0) = 1$ and indeed $H(1) = 1$.
Fix n. Assuming $T(n) = H(n)$ we show $T(n+1) = H(n+1)$. Indeed, $T(n+1) = \frac{1}{n+1} + T(n) \overset{\text{I.H}}{=} H(n) + \frac{1}{n+1} = \sum_{k=1}^{n+1} \frac{1}{k} = H(k+1)$. $\blacksquare$.
Thus $T(n) = H(n) = \ln(n) + O(1) \in \Theta(\log(n))$. **(b)** (4 pts) $T(n) = 2T(\frac{n}{2}) + \frac{n}{\lg n} + n$.

Answer. We use Master Theorem: $a = 2$, $b = 2$, and so $n^{\log_b(a)} = n^1 = n$, vs. $f(n) = n + \frac{n}{\log(n)}$. Clearly $n \leq f(n) \leq 2n$ so $f(n) \in \Theta(n) = \Theta(n^1(\log(n))^0)$ and so Master Theorem case 2 applies and we get $T(n) = \Theta(n\log(n))$.

(c) (4 pts) $T(n) = T(\lfloor\sqrt{n}\rfloor) + n$.
Answer. Using iterated substitution (we assume $n^{1/2^i}$ is well-defined for any i):

$$\begin{aligned}
T(n) &= T(\sqrt{n}) + n \\
&= T(n^{1/4}) + n + \sqrt{n} \\
&= T(n^{1/8}) + n + \sqrt{n} + n^{1/4} \\
&= T(n^{1/16}) + n + n^{1/2} + n^{1/4} + n^{1/8}) \\
&\cdots \\
&= T(n^{2^{-i}}) + \left(\sum_{j=0}^{i-1} n^{2^{-j}}\right) \\
&\cdots \\
{\scriptstyle \text{terminates at } k} \quad &= T(n^{2^{-k}}) + \left(\sum_{j=0}^{k-1} n^{2^{-j}}\right)
\end{aligned}$$

The iterations ends for k for which $n^{2^{-k}}$ is a small constant, i.e., when $2^{-k} = \log_n(2) = \frac{\log_2(2)}{\log_2(n)} = \frac{1}{\log(n)}$, i.e., for $k = \log\log(n)$. Alternatively, assuming that $n = 2^{2^j}$ for some j, then the iterations terminate at $k = j$.
(Note, $2^{-k} > 0$ for any k so $n^{2^{-k}} > 1$. That's why we terminate at $n = 2$ rather than the usual $n = 1$.)
Substituting $k = \log\log(n)$ we get that

$$T(n^{2^{-k}}) + \left(\sum_{j=0}^{k-1} n^{2^{-j}}\right) = T(n^{\frac{1}{\log(n)}}) + n + \left(\sum_{j=1}^{k-1} n^{2^{-j}}\right) = n + T(2) + \left(\sum_{j=1}^{k-1} n^{2^{-j}}\right)$$

so we get an immediate lower bound of $T(n) \geq n$. To get an upper bound, note that $T(2)$ is simply some constant, and that for any $j \geq 1$, each of the $k = \log\log(n)$ summands is upper bounded by $n^{1/2}$. Hence, $T(n) \leq n + T(2) + n^{1/2}\log\log(n) \leq 2n$ for large enough ns. We can infer that $T(n) = \Theta(n)$.

So we prove $n \leq T(n) \leq 2n$. Since $T(n)$ is non-negative, then for any n we have $T(n) = T(\sqrt{n}) + n \geq 0 + n = n$ so the lower bound is immediately proven (no need for induction). As for the upper bound: $T(1) = 1 \leq 2$, $T(2) = T(1) + 2 = 3 \leq 4$, and $T(3) = T(1) + 3 = 4 \leq 6$ so the induction holds for any base case. For the induction step: Fix $n \geq 4$. Assuming $T(i) \leq 2i$ for any $1 \leq i < n$ we show that $T(n) \leq 2n$. Indeed: $T(n) = T(\sqrt{n}) + n \leq 2\sqrt{n} + n \overset{n\geq 4}{\leq} \sqrt{n}\cdot\sqrt{n} + n = 2n$. ■

(d) (4 pts) $T(n) = 16T(\frac{n}{2}) + (n\log n)^4$.
Answer. Using Master Theorem: $a = 16$, $b = 2$, $\log_b a = 4$, and $(n\log n)^4 \in \Theta(n^4\log^4 n)$, by case (2) we have $T(n) \in \Theta(n^4 \log^5 n)$.

(e) (4 pts) $T(n) = T(n-1) + T(n-2) + 1$ with base case of $T(1) = 1$ and $T(2) = 2$.
Answer. We use iterated substitution to derive the solution, where $F(n)$ denotes the n-th Fibonacci number. $(F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8...)$

$$\begin{aligned}
T(n) &= 1 + T(n-1) + T(n-2)\\
&= 1 + (1 + T(n-2) + T(n-3) + T(n-2) = (1+1) + 2T(n-2) + T(n-3)\\
&= (1+1) + 2(1 + T(n-3) + T(n-4)) + T(n-3) = (1+1+2) + 3T(n-3) + 2T(n-4)\\
&= (1+1+2+3) + 5T(n-4) + 3T(n-5)\\
&= (1+1+2+3+5) + 8T(n-5) + 5T(n-6)\\
&= (1+1+2+3+5+8) + 13T(n-6) + 8T(n-7)\\
&\ \vdots\\
\text{at row } j &= \sum_{i=1}^{j} F(i) + F(j+1)T(n-j) + F(j)T(n-j-1)\\
&\ \vdots\\
\text{at row } n-2 &= \sum_{i=1}^{n-2} F(i) + F(n-1)T(2) + F(n-2)T(1) = \sum_{i=1}^{n-2} F(i) + 2F(n-1) + F(n-2)\\
&= \sum_{i=1}^{n-1} F(i) + F(n-1) + F(n-2) = \sum_{i=1}^{n} F(i)
\end{aligned}$$

We argue therefore that for every n it holds that $T(n) = \sum_{i=1}^{n} F(i)$, and prove it by induction.
Base case: For $n = 1$ we have $T(1) = 1 = F(1)$. For $n = 2$ we have $T(2) = 2 = 1 + 1 = F(1) + F(2)$.
Induction step: Fix some n. We assume the claim holds for all $i \leq n$ and prove it also holds for $n + 1$.

$$\begin{aligned}
T(n+1) &= T(n) + T(n-1) + 1\\
&= \sum_{i=1}^{n} F(i) + \sum_{i=1}^{n-1} F(i) + 1\\
&= \sum_{i=1}^{n} F(i) + \sum_{i=2}^{n} F(i-1) + 1\\
&= F(1) + \sum_{i=2}^{n} (F(i) + F(i-1)) + 1\\
&= F(1) + 1 + \sum_{i=2}^{n} F(i+1)\\
&= F(1) + F(2) + \sum_{i=3}^{n+1} F(i) = \sum_{i=1}^{n+1} F(i)
\end{aligned}$$

Using the fact that $F(n) = \Theta(\left(\frac{1+\sqrt{5}}{2}\right)^n)$ and the sum of geometric series $\sum_{i=1}^{n} \left(\frac{1+\sqrt{5}}{2}\right)^i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - 1}{\left(\frac{1+\sqrt{5}}{2}\right) - 1} = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, we deduce that $T(n) = \Theta(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

Problem 2. (10 pts) Consider the following recurrence for a function T that takes on nonnegative values and is defined on integers ≥ 1:

$$T(n) \leq \begin{cases} 10n, & \text{if } 1 \leq n \leq 9 \\ T(\frac{7}{10}n) + T(\frac{1}{5}n) + 3n, & \text{if } n > 9 \end{cases}$$

Find and prove asymptotic upper bound on $T(n)$.
Answer. We use iterated substitution to derive the solution.

$$\begin{aligned} T(n) &= 3n + T(0.7n) + T(0.2n) \\ &= (1 + 0.7 + 0.2) \cdot 3n + T(0.7^2 n) + 2T(0.7 \times 0.2n) + T(0.2^2 n) \\ &= (1 + 0.7 + 0.2 + 0.7^2 + 2 \times 0.7 \times 0.2 + 0.2^2) \cdot 3n \\ &\qquad + T(0.7^3 n) + 3T(0.7^2 \times 0.2n) + 3T(0.7 \times 0.2^2 n) + T(0.2^3 n) \\ &= \big(1 + (0.7 + 0.2) + (0.7 + 0.2)^2 + (0.7 + 0.2)^3\big) \\ &\qquad + T(0.7^4 n) + 4T(0.7^3 \times 0.2n) + 6T(0.7^2 \times 0.2^2 n) + 4T(0.7 \times 0.2^3 n) + T(0.2^4 n) \end{aligned}$$

and so — as long as n is large enough — it seems as though the i-th iteration would look like:

$$3n \sum_{j=0}^{i-1} (0.7 + 0.2)^j + \sum_{j=0}^{i} \binom{i}{j} T(0.7^{i-j} \times 0.2^j \times n)$$

The problem is that at some point we reach iterations is for which we recurse only on a subset of the $T(\cdot)$-terms. That is for such is where $0.2^i n < 10$ — and so we no longer recurse on $T(0.2^i n)$ because we reached the base case; whereas $0.7^i n > 10$ — so we will continue to recurse on the term $T(0.7^i n)$ (at least). This makes it tougher to analyze.

But let's do a thought experiment, let's assume we do continue recursing on all branches – indefinitely! In other words, suppose we extended T to be a function over the reals and we continue recursing forever. This may seem like an over-kill, but as it turns out — we actually get a linear upper-bound from it:

If we run indefinitely, then our overall runtime is

$$3n \cdot \sum_{j=0}^{\infty} (0.7 + 0.2)^j = 3n \cdot \sum_{j \geq 0} 0.9^j = 3n \tfrac{1}{1 - 0.9} = 30n$$

combine this we the fact that for every n we do at least $3n$ steps (as both $T(\frac{7}{10}n)$ and $T(\frac{1}{5}n)$ are non-negative), and we get that $3n \leq T(n) \leq 30n$.

This implies that we should have $T(n) \leq 30n$. We prove this using induction.
Base case: For any $n \leq 9$ we have $T(n) = 10n < 30n$.
Induction step: Fix $n \geq 10$. Assuming that $T(i) \leq 30i$ for any $i < n$ we show that $T(n) \leq 30n$. Indeed

$$\begin{aligned} T(n) &= 3n + T(0.7n) + T(0.2n) \\ &\leq 3n + 30 \times 0.7n + 30 \times 0.2n = (3 + 21 + 6)n = 30n \quad \blacksquare \end{aligned}$$

This implies that $T(n) \in O(n)$. On the flip side, it is obvious that $T(n) = 3n + T(\frac{7}{10}n) + T(\frac{1}{5}n) \geq 3n$ for any $n \geq 10$, thus $T(n) \in \Omega(n)$. Thus, $T(n) = \Theta(n)$.

Problem 3. (20 pts) Your boss has given you an assignment: to devise the fastest algorithm you can for solving a particular problem whose input is an array A with n elements. After mulling the problem over for a few days you come up with a few alternatives. Which alternative do you prefer and why? Explain.
(a) (8 pts)

- Algorithm 1:
  Iterate over each element in A and do at most 3 operations per element.

- Algorithm 2:
  For inputs of size $n > 1$:
  1. Recurse on the first half of the input
  2. Recurse on the latter half of the input
  3. Make a single arithmetic operation

Answer. Both algorithms are just as good for you, as both run in $\Theta(n)$.

The first algorithm takes $\Theta(n)$ time — there are n elements in the array and for each one with only do $O(1)$ work.

The second algorithm has a runtime $T(n)$ which is represented by the recurrence relation $T(n) = 2T(\frac{n}{2}) + 1$. Master Theorem applies here with $a = b = 2$ and $f(n) = 1$. As $1 \in O(n^1)$ the first case of Master Theorem applies and we get $T(n) \in \Theta(n)$.
(b) (12 pts)

- Algorithm 1:
  For inputs A of size $n > 1$:
  1. Recurse on an instance of size $n - 1$
  2. Make a single basic operation
  3. Recurse on a different instance of size $n - 1$

- Algorithm 2:
  For inputs A of size $n > 1$:
  1. Recurse on an instance of size $n - 1$
  2. Iterate over each element in A and do at most $O(2^{n^{0.99}})$ operations per each element.

- Algorithm 3:
  Iterate over all possible subsets of elements of A and do a single operation per subset.

Answer. We wish to use Algorithm 2, as it is the fastest.

The first algorithm's runtime is given by the recurrence relation $T(n) = 2T(n-1) + 1$. This is the Towers of Hanoi recurrence relation. Assuming $T(0) = 0$ this solves to $T(n) = 2^n - 1$. Indeed, by induction we can see that $T(0) = 2^0 - 1 = 0$; and for a given n where $T(n) = 2^n - 1$ then $T(n+1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$. Thus, the runtime of Algorithm 1 is in $\Theta(2^n)$.

The second algorithm is a recursive algorithm that in addition to the recursive call makes $O(n \cdot 2^{n^{0.99}})$ steps ($O(2^{n^{0.99}})$ per each of the n elements in A). Thus its runtime is given by the recurrence relation: $R(n) = R(n-1) + n \cdot 2^{n^{0.99}}$. Iterated substitution gives that

$$\begin{aligned}
R(n) &= R(n-1) + n \cdot 2^{n^{0.99}} \\
&= R(n-2) + n \cdot 2^{n^{0.99}} + (n-1) \cdot 2^{(n-1)^{0.99}} \\
&= R(n-3) + n \cdot 2^{n^{0.99}} + (n-1) \cdot 2^{(n-1)^{0.99}} + (n-2) \cdot 2^{(n-2)^{0.99}} \\
&\vdots \\
&= R(0) + \sum_{i=1}^{n} i \cdot 2^{i^{0.99}}
\end{aligned}$$

while finding an exact sum for this is complicated, it is fairly simple to see that each of the n summands in this sum is at most $n \cdot 2^{n^{0.99}}$. Hence, the overall runtime is at most $O(n^2 \cdot 2^{n^{0.99}})$. Seeing as $n^2 \cdot 2^{n^{0.99}} = 2^{n^{0.99}+2\log(n)} \leq 2^{n^{0.99} \cdot 2\log(n)}$, and since $\log(n) \in o(n^{0.009})$, then for large enough values of n we have $\log(n) \leq \frac{1}{2}n^{0.009}$, which means that for large enough values of n, $n^2 \cdot 2^{n^{0.99}} \leq 2^{n^{0.99} \cdot 2\log(n)} \leq 2^{n^{0.99} \cdot n^{0.009}} = 2^{n^{0.999}}$. We get that the runtime of Algorithm 2 is in $o(2^{n^{0.999}})$ which is asymptotically smaller than 2^n.

The third algorithm traverses all 2^n choices of subsets of elements from A and does $O(1)$ work per subset — so its overall runtime is $\Omega(2^n)$.

It is therefore easy to see that Algorithm 2's runtime is the better of all three.

Problem 4. (15 pts) Consider the following elegant(?) sorting algorithm:

```
procedure SomeSort($A, b, e$)        ** Sorts the subarray $A[b, .., e]$
if ($e = b + 1$) then
    if ($A[b] > A[e]$) then
        exchange $A[b] \leftrightarrow A[e]$
    end if
else if ($e > b + 1$) then
    $p \longleftarrow \lfloor \frac{e-b+1}{3} \rfloor$
    SomeSort($A, b, e - p$)
    SomeSort($A, b + p, e$)
    SomeSort($A, b, e - p$)
end if
```

(a) (2 pts) Illustrate the behavior of `SomeSort` on the input $A = [33, 22, 11]$ with $b = 1$ and $e = 3$.
Answer. The algorithm invokes three recursive calls:
First call for SomeSort$(A, 1, 2)$ which sorts $[33, 22]$ and sets $A = [22, 33, 11]$.
Second call for SomeSort$(A, 2, 3)$ which sorts $[33, 11]$ and sets $A = [22, 11, 33]$.
Third call for SomeSort$(A, 1, 2)$ which sorts $[22, 11]$ and sets $A = [11, 22, 33]$.

(b) (6 pts) Prove that `SomeSort` correctly sorts any input array A of size $n = e - b + 1$.
Answer. let $n = e - b + 1$ be the number of elements in the array A. We use an inductive argument to show that for any value of n this algorithm sorts array A. The claim is obvious for $n \leq 2$. So let's assume that `SomeSort` works for arrays of size smaller than $n > 2$ and assume that A is an array of size n. To make the arguments easier we assume that $n = 3p$, i.e. $e - b + 1 = 3p$. (To give a formal proof for $n = 3$ we basically give the same argument as the one below, only with $p = 1$.) By induction hypothesis and because $e - p - b + 1 < n$, the first recursive call to `SomeSort` sorts the first $2p$ elements.
The key point: After the first recursive call, we have that the all of the largest p elements of A now appear in positions $\{b + p, b + p + 1, .., e\}$. To see this, fix any element a_j which is among the largest p elements of A and consider the two possible cases:
(i) If initially a_j was among the elements in $\{b, b + 1, ..., e - p\}$ then after sorting them, it must be among the top p elements in this sub-array, which means it is located somewhere in $A[b + p, b + p + 1, ..., e - p]$.
(ii) Otherwise, the original position of a_j was at $A[e - p + 1, ..., e]$ and those have not changed. So in any case, a_j must be an element in $A[b + p, b + p + 1, ..., e]$.
Therefore, all the top p largest elements of A are now at $A[b + p, ..., e]$. This implies that the second recursive call to `SomeSort` puts these top p largest elements of A at their correct locations (at the end part of A) by the induction hypothesis. Now that the top p locations in A holds the correct largest p elements and in order, the last call to `SomeSort` puts the rest of the elements into their correct locations between b and $e - p$, by induction hypothesis. This results in a completely sorted array.

(c) (6 pts) Find a recurrence relation for the worst-case running time of `SomeSort`. Give a tight (i.e. Θ) asymptotic bound for the worse-case running time of `SomeSort`.
(Hint: for simplicity assume that $n = 3^k$ for some constant k.)
Answer. The running time function $T(n)$ is constant for small values of n, and for $n \geq 3$ we have that $T(n) = 3T(\lceil \frac{2n}{3} \rceil) + c$ for some constant c. The reason is that `SomeSort` makes a 3 recursive calls, each to a subarray of size $\frac{2n}{3}$ (and in addition take some constant number of operations).
For simplicity, we assume that $n = 3^k$ for some $k > 0$. In this case, $\lceil \frac{2n}{3} \rceil = \frac{2n}{3}$. Thus the recurrence relation can be written as $T(n) = 3T(\frac{2n}{3}) + c$. As $c \in \Theta(1) = \Theta(n^0)$ and $0 < \log_{3/2}(3)$ then case (1) of Master Theorem applies and we get $T(n) \in \Theta(n^{\log_{3/2} 3})$.

(d) (1 pts) Compare the WC-runtime `SomeSort` to that of the other sorting algorithms we've learned and determine when (if) one would prefer `SomeSort` to those algorithms.

Answer. Since $\log_{3/2} 3 > 2.7$ the running time of `SomeSort` is $\Omega(n^{2.7})$ which is worse than all of insertion sort, merge sort, heapsort and quicksort. We basically never want to use it. So this algorithm is not so elegant!

Problem 5. (15 pts) Recall the definition of Fibonacci numbers where $F(0) = 0$, $F(1) = 1$, and for $n \geq 2$: $F(n) = F(n-1) + F(n-2)$.

(a) (5 pts) Consider the following matrix: $M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$. Prove by induction that for each $n \geq 1$:

$$M^n = \begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix}$$

Answer. We prove by induction on n. The base case $n = 1$ is obvious. Induction step:Suppose the claim holds for some value of $n \geq 1$. For $n+1$ we have:

$$\begin{aligned} M^{n+1} &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} \\ &= \begin{bmatrix} F(n) & F(n+1) \\ F(n-1)+F(n) & F(n)+F(n+1) \end{bmatrix} = \begin{bmatrix} F(n) & F(n+1) \\ F(n+1) & F(n+2) \end{bmatrix} \blacksquare \end{aligned}$$

(b) (10 pts) Design an $O(\log n)$-time algorithm to compute $F(n)$. Justify the running time of your algorithm.

Answer. We have already seen the idea of taking exponent in $O(\log n)$ time. So to compute M^n we first compute $M^{n/2}$ and then multiply the result by itself if n is even, or compute first M^{n-1} and multiply the results by M if n is odd. So each time, by at most two recursive calls the size of the problem is reduced from n to $\lfloor \frac{n}{2} \rfloor$. So it only needs $O(\log n)$ matrix multiplications to compute M^n. Since for each of these matrix multiplications we only need a constant number of scalar multiplications the running time is still $O(\log n)$.

(In particular, there's no need to use Strassen's algorithm, as each matrix multiplication is of a (2×2)-matrix with another (2×2)-matrix, which requires $8 = O(1)$ scalar multiplications. If you truly want to be nit-picking, observe that since M is symmetric then for any i it holds that M^i is symmetric; so the multiplication only requires 6 scalar multiplications since coordinate $(2,1)$ will always be the same as coordinate $(1,2)$.)

Once we have computed M^n, by part (a) we have found $F(n)$ as it is the element in coordinate $(2,1)$ of M^n.

```
procedure Fib($n$)
$M \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$
$A \leftarrow$FastExp($M, n$)
return $A_{1,2}$
```

$**$ while there's **no need** to write the code for fast-exponentiation of matrices, we bring it here
$**$ just so you everything will be perfectly clear.

```
procedure FastExp($M, n$)
if ($n = 0$) then
   return $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
else if ($n$ is odd) then
   $X \leftarrow$FastExp($M, n-1$)
   return $\begin{bmatrix} M_{1,1}X_{1,1} + M_{1,2}X_{2,1} & M_{1,1}X_{2,1} + M_{1,2}X_{2,2} \\ M_{2,1}X_{1,1} + M_{2,2}X_{2,1} & M_{2,1}X_{2,1} + M_{2,2}X_{2,2} \end{bmatrix}$   $**$ returning $M \cdot X$
else
   $X \leftarrow$FastExp($M, n/2$)
   return $\begin{bmatrix} X_{1,1}X_{1,1} + X_{1,2}X_{2,1} & X_{1,1}X_{2,1} + X_{1,2}X_{2,2} \\ X_{2,1}X_{1,1} + X_{2,2}X_{2,1} & X_{2,1}X_{2,1} + X_{2,2}X_{2,2} \end{bmatrix}$   $**$ returning $X \cdot X$
```

Problem 6. (20 pts) In this question, we will discuss multiplying a $(n \times n)$-matrix M with a n-dimensional vector $\bar{v}$.
(a) (5 pts) Describe the naïve loop-based algorithm for computing $M\bar{v}$ and argue it runs in time $\Theta(n^2)$.
Answer.

```
procedure Multiply($M, \bar{v}, n$)
** precondition: $M$ is a matrix of size $n \times n$ and $\bar{v}$ is a vector of size $n$
$res \leftarrow$ a vector of size $n$
for ($i$ from 1 to $n$) do
    $res_i \leftarrow 0$
    for ($j$ from 1 to $n$) do
        $res_i \leftarrow M_{i,j} \times v_j$
return $res$
```

For each of the n coordinates of the output, the naïve algorithm makes n multiplications: the n scalars in the suitable row of n with the n coordinates of $\bar{v}$. It is therefore evident that the runtime of the algorithm is in $O(n) \cdot O(n) = O(n^2)$.

(b) (15 pts) The Hadamard matrix is $(2^k \times 2^k)$-matrix defined for any integer $k \geq 0$ by the following recursive definition: $H_0 = [1]$ which is a (1×1)-matrix, and $H_{k+1} = \begin{bmatrix} H_k & H_k \\ H_k & -H_k \end{bmatrix}$ for $k \geq 0$.

So, for example, $H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and $H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$, and H_3 is a (8×8)-matrix.

Given n which is a power of 2 (i.e., $n = 2^k$ for some k), describe an algorithm whose input is a n-dimensional vector $\bar{v}$, and returns the product of the Hadmard matrix with the given vector, i.e. $H_k \cdot \bar{v}$.
Explain why your algorithm is correct and prove that its runtime is $O(n\log(n))$.
You may use the fact (without proving it) that adding/subtracting two n-dimensional vectors takes $O(n)$ time.

Answer. Our algorithm is a divide and conquer algorithm.

For inputs of dimension 1, the algorithm simply returns $\bar{v}$. (Note, $H_0 \cdot \bar{v} = 1 \cdot \bar{v} = \bar{v}$).
For inputs of dimension 2^k for some $k \geq 1$, the algorithm partitions v into two parts:

$$\bar{v} = \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix}$$

Observe that this implies that:

$$H_k \cdot \bar{v} = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \cdot \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} H_{k-1} \cdot \bar{x} + H_{k-1} \cdot \bar{y} \\ H_{k-1} \cdot \bar{x} - H_{k-1} \cdot \bar{y} \end{bmatrix}$$

So our algorithm finds $\bar{a} = H_{k-1} \cdot \bar{x}$ and $\bar{b} = H_{k-1} \cdot \bar{y}$ using two recursive calls, and then — using one addition and one subtraction — returns the vector $\begin{bmatrix} \bar{a} + \bar{b} \\ \bar{a} - \bar{b} \end{bmatrix}$. Let $T(n)$ denote the algorithm's runtime over inputs of dimension n, then due to the two recursive calls on $\frac{n}{2}$-dimensional instances, and the $O(n)$ time operations we need to add and subtract (and copy into the right place) the vectors $\bar{a} + \bar{b}$ and $\bar{a} - \bar{b}$ we have that

$$T(n) = \begin{cases} 1 & , \text{ if } n = 1 \\ 2T(\frac{n}{2}) + n & , \text{ if } n \geq 2 \end{cases}$$

Master Theorem (case 2) is applicable to this recursion (in fact, this is the same recursion we get in `MergeSort`), and so we get that $T(n) = \Theta(n\log(n))$.