Homework Assignment #2

Due: Noon, Oct 22, 2018
Submit **printed** solutions via eClass
Submit **handwritten** solutions at class, by noon

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 $H(n) = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$.

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

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

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) $T(n) = T(n-1) + \frac{7}{n}$ with $T(0) = 0$.
Answer. Using iterated substitution:

$$\begin{aligned} T(n) &= \tfrac{7}{n} + T(n-1) \\ &= \tfrac{7}{n} + \tfrac{7}{n-1} + T(n-2) \\ &= \tfrac{7}{n} + \tfrac{7}{n-1} + \tfrac{7}{n-2} + T(n-3) \\ &= \tfrac{7}{n} + \tfrac{7}{n-1} + \tfrac{7}{n-2} + \tfrac{7}{n-3} + T(n-4) \\ &= \ldots = \tfrac{7}{n} + \tfrac{7}{n-1} + \tfrac{7}{n-2} + \tfrac{7}{n-3} + \ldots + \tfrac{7}{2} + \tfrac{7}{1} + 0 = \sum_{k=1}^{n} \tfrac{7}{k} = 7H(n) \end{aligned}$$

We prove that for all n we have $T(n) = 7H(n)$ via induction.
$T(1) = \frac{7}{1} + T(0) = 7$ and indeed $7 \cdot H(1) = 7 \cdot 1 = 7$.

Fix n. Assuming $T(n) = 7H(n)$ we show $T(n+1) = 7H(n+1)$. Indeed, $T(n+1) = \frac{7}{n+1} + T(n) \overset{\text{I.H}}{=} 7H(n) + \frac{7}{n+1} = \sum_{k=1}^{n+1} \frac{7}{k} = 7H(k+1)$. ■.

Thus $T(n) = 7H(n) = 7\ln(n) + O(1) \in \Theta(\log(n))$.

(b) (4 pts) $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log(\log(n))} + n$.
Answer. We use Master Theorem: $a = 2$, $b = 2$, and so $n^{\log_b(a)} = n^1 = n$, vs. $f(n) = n + \frac{n}{\log(\log(n))}$. 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) $T(n) = T(\lfloor\sqrt{n}\rfloor) + 4n$.
Answer. Using iterated substitution (we assume $n^{1/2^i}$ is well-defined for any i):

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

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) $T(n) = 8T(\frac{n}{2}) + (n\log n)^3$.
Answer. Using Master Theorem: $a = 8$, $b = 2$, $\log_b a = 3$, and $(n\log n)^3 \in \Theta(n^3\log^3 n)$, by case (2) we have $T(n) \in \Theta(n^3\log^4 n)$.

(e) (4 pts) $T(n) = T(n-1) + a \cdot n^c$ for some constants $a, c > 0$.
Answer. Iterative substitution gives that:

$$\begin{aligned}
T(n) &= T(n-1) + a\cdot n^c \\
&= T(n-2) + a\cdot n^c + a\cdot(n-1)^c \\
&= T(n-3) + a\cdot n^c + a\cdot(n-1)^c + a\cdot(n-2)^c \\
&\vdots \\
&= T(0) + a\cdot n^c + a\cdot(n-1)^c + a\cdot(n-2)^c + ... + a\cdot 1^c = a\left(1^c + 2^c + ... + (n-1)^c + n^c\right)
\end{aligned}$$

In HW1 we've solved the latter sum and showed it is $\Theta(n^{c+1})$, which the positive constant a doesn't change. Hence, $T(n) \in \Theta(n^{c+1})$.

Problem 2. (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 333 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 333 arithmetic operations

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 $T(n) = 2T(\frac{n}{2}) + 333$. Master Theorem applies here with $a = b = 2$ and $f(n) = 333$. 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. Take 999 basic operations
  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 $O(2^{n^{0.999}})$ operations per each element.
- Algorithm 3:
  Iterate over all possible *subsets* of elements of A and do 999 operations 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 $T(n) = 2T(n-1)+999$. This is similar to the Towers of Hanoi recurrence relation. Assuming $T(0) = 0$ this solves to $T(n) = 999(2^n - 1)$. Indeed, by induction we can see that $T(0) = 999 \cdot (2^0 - 1) = 0$; and for a given n where $T(n) = 999(2^n - 1)$ then $T(n+1) = 2T(n) + 999 = 2 \cdot 999(2^n - 1) + 999 = 999\left(2^{n+1} - 2 + 1\right) = 999\left(2^{n+1} - 1\right)$. 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 $O(n \cdot 2^{n^{0.999}})$ steps ($O(2^{n^{0.999}})$ per each of the n elements in A). Thus its runtime is given by the recurrence relation:

$R(n) = R(n-1) + n \cdot 2^{n^{0.999}}$. Iterated substitution gives that

$$\begin{aligned} R(n) &= R(n-1) + n \cdot 2^{n^{0.999}} \\ &= R(n-2) + n \cdot 2^{n^{0.999}} + (n-1) \cdot 2^{(n-1)^{0.999}} \\ &= R(n-3) + n \cdot 2^{n^{0.999}} + (n-1) \cdot 2^{(n-1)^{0.999}} + (n-2) \cdot 2^{(n-2)^{0.999}} \\ &\vdots \\ &= R(0) + \sum_{i=1}^{n} i \cdot 2^{i^{0.999}} \end{aligned}$$

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 $n \cdot 2^{n^{0.999}}$. Hence, the overall runtime is at most $O(n^2 \cdot 2^{n^{0.999}})$. Seeing as $n^2 \cdot 2^{n^{0.999}} = 2^{n^{0.999}+2\log(n)} \leq 2^{n^{0.999} \cdot 2\log(n)}$, and since $\log(n) \in o(n^{0.0009})$, then for large enough values of n we have $\log(n) \leq \frac{1}{2} n^{0.0009}$, which means that for large enough values of n, $n^2 \cdot 2^{n^{0.999}} \leq 2^{n^{0.999} \cdot 2\log(n)} \leq 2^{n^{0.999} \cdot n^{0.0009}} = 2^{n^{0.9999}}$. We get that the runtime of Algorithm 2 is in $o(2^{n^{0.9999}})$ 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 3. (15 pts) The following are all examples of recurrence-relations for a function T that takes on nonnegative values and is defined on the natural numbers.
Prove, using induction, that for each example $T(n) \in \Theta(n)$. You may ignore any rounding issues that might arise.
(a) (5 pts)

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 10n, & \text{if } 1 \le n \le 9 \\ T(\frac{7}{10}n) + T(\frac{1}{5}n) + 3n, & \text{if } n > 9 \end{cases}$$

(b) (5 pts)

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 9999n, & \text{if } 1 \le n \le 99 \\ T(0.9n) + T(0.09n) + 999n, & \text{if } n > 99 \end{cases}$$

Answer. We answer both articles together. Take any $n_0 \in \mathbb{N}$ and any c_1, c_2, c_3, c_4 positive reals such that $c_1 + c_2 < 1$. Let $T(n)$ be a recurrence relation defined as

$$T(n) = \begin{cases} c_4 n & , \text{ if } n \le n_0 \\ T(c_1 n) + T(c_2 n) + c_3 n & , \text{ if } n > n_0 \end{cases}$$

Note how this template covers both cases: in the former: $n_0 = 9, c_1 = 0.7, c_2 = 0.2, c_3 = 3, c_4 = 10$ and indeed $c_1 + c_2 = 0.9 < 1$; in the latter $n_0 = 99$, $c_1 = 0.9$, $c_2 = 0.09$, $c_3 = 999$ and $c_4 = 9999$, where indeed $c_1 + c_2 = 0.99 < 1$.

We argue that for every natural n it holds that $T(n) \le cn$ for $c = \max\{c_4, \frac{c_3}{1-(c_1+c_2)}\}$. Note how both terms in the max are > 0 seeing as $c_1 + c_2 < 1$, and in particular we have $c_3 \le c(1 - c_1 - c_2)$.

Proof: By induction on n. Base case is $n = 0$ where $T(0) \le c_4 \cdot 0 = 0 \le c \cdot 0$.

Induction step: Fix any natural n. Assuming that for all $i < n$ we have that $T(i) \le ci$, we now argue that $T(n) \le cn$.

$$\begin{aligned} T(n) &\overset{\text{definition}}{\le} T(c_1 n) + T(c_2 n) + c_3 n \overset{\text{I.H.}}{\le} c \cdot c_1 n + c \cdot c_2 n + c_3 n = n\Big(c(c_1 + c_2) + c_3\Big) \\ &= n\Big(c - c + c(c_1 + c_2) + c_3\Big) = cn + n\Big(c(-1 + c_1 + c_2) + c_3\Big) \\ &\overset{\text{def of } c}{\le} cn + \Big(c(-1 + c_1 + c_2) + c(1 - c_1 - c_2)\Big) = cn \end{aligned}$$

The claim gives that $T(n) \in O(n)$. However, note that for any $n \ge n_0 + 1$ it must hold that $T(n) \ge c_3 n$, implying also that $T(n) \in \Omega(n)$. The result is that $T(n) \in \Theta(n)$.

(c) (5 pts) Let $(m_0, m_1, m_2, ..., m_n, ...)$ be an arbitrary series such that for every $n \ge 2$ we have that m_n is some natural number $\le n - 1$.

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ T(m_n) + T(n - m_n - 1) + 1, & \text{if } n \ge 2 \end{cases}$$

Answer. We argue that for all n we have that $T(n) = n$. We use induction to prove the required. The base cases are simple, as $T(0) = 0$ and $T(1) = 1$.
Induction Step: Fix any $n > 1$. Assuming the requires holds for any integer $i < n$, we show it also holds for n. Note that by definition $0 \le m_n \le n - 1$, hence $m_n, n - 1 - m_n \le n - 1$. We can thus apply the IH to each part and have

$$T(n) = T(m_n) + T(n - m_n - 1) + 1 \overset{\text{IH}}{=} m_n + (n - m_n - 1) + 1 = n$$

It follows that $\forall n, T(n) = n$, thus $T(n) \in \Theta(n)$.

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

```
procedure SomeSort($A, b, e$)        ** Sorts the subarray $A[b,..,e]$
if ($e - b + 1 \leq 3$) then
    Sort $A[b..e]$ using any sorting algorithm
        ** Even a brute-force comparison checking all 6 possible ways of sorting, seeing as it takes only $O(1)$-time
else
    $p \longleftarrow \lfloor \frac{e-b+1}{4} \rfloor$
    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 $A = [44, 33, 22, 11]$ with $b = 1$ and $e = 4$.
Answer. The algorithm sets $p = 1$ (thus $b + p = 1 + 1 = 2$ and $e - p = 4 - 1 = 3$) and then invokes three recursive calls:
First call for `SomeSort(`$A, 1, 3$`)` which sorts $[44, 33, 22]$ and sets $A = [22, 33, 44, 11]$.
Second call for `SomeSort(`$A, 2, 4$`)` which sorts $[33, 44, 11]$ and sets $A = [22, 11, 33, 44]$.
Third call for `SomeSort(`$A, 1, 3$`)` which sorts $[22, 11, 33]$ and sets $A = [11, 22, 33, 44]$.

(b) (6 pts) Prove that for any natural $n \geq 1$, `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 $n \leq 3$ as for such sizes of inputs `SomeSort` uses a (correct) brute-force algorithm. So now fix any $n \geq$ and assume that `SomeSort` works for arrays of size strictly smaller than n and assume that A is an arbitrary array of size n. To make the arguments easier we assume that $n = 4p$, i.e. $e - b + 1 = 4p$. By then induction hypothesis and because $e - p - b + 1 < n$, the first recursive call to `SomeSort` sorts the first $3p$ 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 $A[b + 2p, b + 2p + 1, ..., e - p]$.
(ii) Otherwise, the original position of a_j was at $A[e - p + 1, ..., e]$ and those have not changed.
So regardless of which case holds, a_j must be an element in $A[b + 2p, b + 2p + 1, ..., e]$.
Therefore, all the top p largest elements of A are now at $A[b + 2p, ..., e]$. 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 the 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`.
You may assume n is a power of 4 if it helps.
Answer. The running time function $T(n)$ is constant for small values of n, and for $n \geq 4$ we have that $T(n) = 3T(\lceil \frac{3n}{4} \rceil) + c$ for some constant c. The reason is that `SomeSort` makes 3 recursive calls, each to a subarray of size $\frac{3n}{4}$ (and in addition take some constant number of operations).
For simplicity, we assume that $n = 4^k$ for some $k > 0$. In this case, $\lceil \frac{3n}{4} \rceil = \frac{3n}{4}$. Thus the recurrence relation can be written as $T(n) = 3T(\frac{3n}{4}) + c$. As $c \in \Theta(1) = \Theta(n^0)$ and $0 < \log_{4/3}(3)$ then case (1) of Master Theorem applies and we get $T(n) \in \Theta(n^{\log_{4/3} 3})$.

(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 $\log_{4/3} 3 > \log_{4/3}(\frac{16}{9}) = 2$ the running time of `SomeSort` is $\omega(n^2)$ which is worse than all of insertion sort, merge sort, heapsort and quicksort. We basically never want to use it. In fact, the runtime is on the order of $\Theta(n^{3.818\cdots})$ which is far worse than even the quadratic runtime sort algorithms! Turns out this algorithm is not so elegant...

Problem 5. (15 pts) Here is another sorting algorithm, known as Bubble-Sort.

```
procedure BubbleSort($A, n$) ** precondition: $A$ is an array containing $n$ pairwise comparable elements
$i \leftarrow 1$
while ($i \leq n-1$)
    for ($j$ from 1 to $n-1$)
        if ($A[j] > A[j+1]$)
            exchange $A[j] \leftrightarrow A[j+1]$
    end for
    $i \leftarrow i+1$
end while
```

(a) (1 pts) Exhibit how `BubbleSort` modifies the array $[15, 5, 12, 3]$. Include all intermediate stages.
Answer.

Originally:	$[15, 5, 12, 3]$	
After iteration: $i = 1, j = 1$	$[5, 15, 12, 3]$	Exchanging key as $15 > 5$
After iteration: $i = 1, j = 2$	$[5, 12, 15, 3]$	Exchanging keys as $15 > 12$
After iteration: $i = 1, j = 3$	$[5, 12, 3, 15]$	Exchanging keys as $15 > 3$
After iteration: $i = 2, j = 1$	$[5, 12, 3, 15]$	No change as $5 \leq 12$
After iteration: $i = 2, j = 2$	$[5, 3, 12, 15]$	Exchanging keys as $12 > 3$
After iteration: $i = 2, j = 3$	$[5, 3, 12, 15]$	No change as $12 \leq 15$
After iteration: $i = 3, j = 1$	$[3, 5, 12, 15]$	Exchanging keys as $5 > 3$
After iteration: $i = 3, j = 2$	$[3, 5, 12, 15]$	No change as $5 \leq 12$
After iteration: $i = 3, j = 3$	$[3, 5, 12, 15]$	No change as $12 \leq 15$

(b) (5 pts) Prove that `BubbleSort` is correct.
Your answer should include the formal loop-invariant of the while-loop and a proof of said LI.
Hint: Look at the keys 15 and 12 in the above-mentioned example. At what iteration does each of those keys take its correct position?
Answer. The reason why `BubbleSort` correctly sorts the input is that with each while-loop iteration we are guaranteed to move the ith-largest element to its correct position in A (position $n - i + 1$).

The loop-invariant of the while-loop is that at the beginning of each iteration, the $i-1$ largest elements of A are the last $i - 1$ positions in A *in order.*

Let's prove this loop invariant:
Initially: $i = 1$ so the claim vacuously hold.
Maintenance: Assuming the loop invariant holds at the beginning of some loop iteration, we show it also holds in the beginning of the next iteration. That is, the top $i-1$ elements are already in the last $i-1$ places of A and in order, we now show that we move the i-th largest element to its right position: $A[n - i + 1]$. (To formally prove the maintenance property we need a loop invariant for the for-loop, but we don't formally state it here and just gloss over it informally.)
Consider the i-th largest element. At the beginning of the iteration it is in some position $j \leq n - i + 1$ (because it cannot be in any of the last $i-1$ places). Therefore, when the for-loop reaches this j and starts comparing it with the next elements, it must be larger than any element in positions $1, ..., n - i + 1$ so the loop keeps iteratively moving its position to $j + 1$, then to $j + 2$, then $j + 3$ and so on until we compare this item to the $i-1$-largest item which is in position $n-i+2$ — from that point on the while loop makes no more exchanges.
Therefore, once the for-loop terminates, the i-th largest element is now in position $n-i+1$ and all elements in $A[n - i + 2, ...n]$ weren't exchanges as they are already in order.

Termination: i's value is only incremented and is never changed in the body of the for-loop.
Termination 2: When the for-loop terminates, $i = n$. Thus the largest $n-1$ elements are in positions $2, 3, ..., n$ and are in order. **And this is the *must-have* point:** This leaves the smallest element, which must be in position $A[1]$. Therefore, all n elements of A are sorted.

(c) (2 pts) What is, asymptotically, the number of key-comparisons `BubbleSort` makes on any input of n elements?
Answer. For each i and j we make one key-comparison, and there are two nested loops: for each value of i, j iterates through $n-1$ values; and i itself iterates through $n-1$ different values. Therefore, for any array of size n, the number of key comparisons we make overall is $(n-1) \cdot (n-1) \cdot 1 = \Theta(n^2)$.

(d) (3 pts) Based on the correctness proof, Haozhou has revised `BubbleSort` to the following version.

```
procedure Haozhou_BubbleSort(A, n)  ** A is an array that contains n pairwise comparable elements
i ← 1
while (i ≤ n − 1)
    for (j from 1 to n − i)          ** The revision is ONLY in this line
        if (A[j] > A[j + 1])
            exchange A[j] ↔ A[j + 1]
    end for
    i ← i + 1
end while
```

Does Haozhou's version make *significantly* fewer key-comparisons than the original `BubbleSort`? **Answer.**

Observe that now, for every i, the variable j only traverses on $n-i$ different elements. Therefore, the number of key comparisons we make is $\sum_{i=1}^{n-1} n - i = (n-1) + (n-2) + (n-3) + ... + 1$. Each of the first $\frac{n}{2}$ elements in this sum is $\geq \frac{n}{2}$ so overall we make at least $\frac{n}{2} \cdot \frac{n}{2} = \frac{1}{4}n^2$ key-comparisons, which is still $\Omega(n^2)$.

So while Haozhou's version of the algorithm does make fewer key-comparisons, the improvement is only a constant-factor improvement and not asymptotically significant. The number of key-comparisons remains $\Theta(n^2)$.

(e) (4 pts) Sepehr also revised `BubbleSort`, by cleverly using a Boolean variable $flag$.

```
procedure Sepehr_BubbleSort(A, n)  ** A is an array containing n pairwise comparable elements
flag ← TRUE
i ← 1
while (i ≤ n − 1 and  flag =TRUE)
            ** Now, in order to make another iteration of the while-loop it must hold that j isn't too large
            ** and that flag is set to TRUE
    flag ← FALSE
    for (j from 1 to n − 1)
        if (A[j] > A[j + 1])
            exchange A[j] ↔ A[j + 1]
            flag ← TRUE
    end for
    i ← i + 1
end while
```

Does Sepehr's version make significantly fewer key-comparisons than the original `BubbleSort`?
Discuss both the worst-case and the best-case.

Answer. The new version of the algorithm can, sometimes, make significantly fewer comparisons. Observe, we only update $flag$ to be `TRUE` when inside the for-loop we made an exchange between two adjacent elements. Therefore, if we are lucky enough and the for-loop traverses all elements without making any exchanges, then $flag$ remains `FALSE` and we do not make any more iterations in the while-loop. In fact, if we are really lucky, this happens on the very first iteration of the while-loop, where $i = 1$ — when the input is already sorted initially. In that case, we only make $O(n)$ key-comparisons, making no exchanges, and halt after the first iteration of the while-loop. Also note the for $i = 1$ the `for`-loop iterates over $n - 1 = \Omega(n)$ values, so the best-case runtime is $\Omega(n)$. Thus, the best-case runtime of the new version of Bubble-sort is $\Theta(n)$.

However, if we are unlucky, then in each iteration of the while-loop we make at least one exchange, and so we set $flag$ to `TRUE`. This can happen either on an input which appears in the inverse order $[n, n-1, ..., 2, 1]$ or even on any input whose last element is the smallest (consider the case of $[2, 3, 4..., n, 1]$).

And so, Sepehr's version has a significant improvement in the best case (from $\Theta(n^2)$ to $\Theta(n)$), but not in the worst case as we still make $\Omega(n^2)$ key-comparisons in the worst-case. In the worst-case our runtime remains $\Theta(n^2)$.

Problem 6. (15 pts) Both Alice and Beth were given the task of picking m *random* items out of n possible items. However, Alice picks the m items *with* repetitions (so she repeats m times the loop "pick a u.a.r chosen item from the n possible items); whereas Beth picks the m items *without* repetitions (so she initially sets S as the set of all possible items, and repeats m times the loop "pick a u.a.r chosen item from S and remove it from S"). And so, in Alice's set of m items *it could be* that some item appears multiple times; whereas in Beth's set no item can appear more than once.

In this question, your goal is to formalize the probability of each of the two of picking up a particular item i. So fix some item i and denote E as the event $E =$"i is among the m chosen items."

(a) (1 pts) If $m = 1$ (both pick a single item), what is the probability of E holding under Alice's sampling technique, i.e. $\Pr_{\text{Alice}}[E]$; and what is the probability of E holding under Beth's technique, i.e. $\Pr_{\text{Beth}}[E]$?
Answer. When both pick a single item ($m = 1$) it means that both follow the same procedure — in both settings i is a uniformly at randomly chosen element out of n possible options. In this case $\Pr_{\text{Alice}}[E] = \Pr_{\text{Barbard}}[E] = \frac{1}{n}$.

(b) (1 pts) In a sentence: what is the meaning of the complimentary event $\neg E$?
And so, if $m = 1$, what is $\Pr_{\text{Alice}}[\neg E]$ and what is $\Pr_{\text{Beth}}[\neg E]$?
Answer. $\neg E$ is the event: "i isn't any of the m randomly picked items".
Since for every event E we have $\Pr[\neg E] = 1 - \Pr[E]$ then in the case where $m = 1$ we have $\Pr_{\text{Alice}}[\neg E] = \Pr_{\text{Beth}}[\neg E] = 1 - \frac{1}{n}$.

(c) (10 pts) Assume $m > 1$. Whose technique, with or without repetitions, has a higher chance of picking a particular item i? Namely, who has the higher probability of E holding — Alice or Beth?
Hint#1: Start with the special case of $m = n$, this should guide you as to the answer in the general case.
Hint#2: Leverage on the the previous article. Instead of arguing about $\Pr[E]$, reason about $\Pr[\neg E]$.
Answer. The event E is more likely under sampling without repetitions, provided $m > 1$.

To see this, it would be easier to argue about the negation of E: $\neg E$ where i is not sampled into the set. We argue that the event $\neg E$ is more likely with repetitions.

In order for $\neg E$ to hold, we need that in *each* iterations of either algorithm we don't sample i.

In the first iteration, both Alice and Beth have the same probability of not sampling i: $\frac{n-1}{n}$.

However, starting from the second iteration, $\neg E$ becomes slightly more likely under Alice than under Beth. In the 2nd iteration, the probability that Alice doesn't sample i is just like in the first round: $\frac{n-1}{n}$. In contrast, the probability that Beth doesn't sample i is $\frac{n-2}{n-1}$, since now Beth has $n-1$ options in general and $n-2$ of are not i. It is easy to see that

$$\frac{n-2}{n-1} < \frac{n-1}{n} \quad \text{as} \quad n^2 - 2n < n^2 - 2n + 1$$

In fact, in any of the following iterations, Beth has smaller probability of not sampling i than Alice. Indeed, at the jth iteration, Beth doesn't sample item i with probability $\frac{n-j}{n-j+1}$ (since $j-1$ item have already been taken out, and out of the $n-(j-1)$ remaining items she shouldn't pick 1, leaving her with $n-(j-1)-1$ valid choices). Again, Alice's probability of not picking item i at the jth iteration remains the same: $\frac{n-1}{n}$. It is simple to argue that

$$\frac{n-j}{n-j+1} < \frac{n-1}{n} \quad \Leftrightarrow \quad n^2 - n\cdot j < n^2 - n - n\cdot j + j + n - 1 \quad \Leftrightarrow \quad 0 < j-1 \text{ for } j \geq 2$$

Thus,

$$\Pr_{\text{Beth}}[\neg E] = \frac{n-1}{n}\cdot\frac{n-2}{n-1}\cdot\frac{n-3}{n-2}\cdot\ldots\cdot\frac{n-m}{n-m+1} < \frac{n-1}{n}\cdot\frac{n-1}{n}\cdot\frac{n-1}{n}\cdot\ldots\cdot\frac{n-1}{n} = \Pr_{\text{Alice}}[\neg E]$$

Hence $\Pr_{\text{Beth}}[E] > \Pr_{\text{Alice}}[E]$.

(d) (3 pts) Assume $n \geq 100$. Show that when both techniques are used for picking half of the items, namely when $m = \frac{n}{2}$, then:
–In the leading of the two techniques (the one where E is more likely to hold) we have that $\Pr[E] = 0.5$
–And in the other technique (the one where E is less likely to hold) we have that $\Pr[E] < 0.4$.
You are welcome to use the following inequality

$$\text{for any } x \geq 2 \text{ we have:} \quad 1 - \tfrac{1}{x} \geq e^{-\frac{1}{x} - \frac{1}{x^2}}$$

and the fact that $e^{-\left(\frac{1}{2}+\frac{1}{200}\right)} > 0.6$. (Instead of you using a calculator to check the value of $e^{-\left(\frac{1}{2}+\frac{1}{200}\right)}$, we have done this for you.)

Answer. As we have shown $\Pr_{\text{Beth}}[\neg E] = \frac{m}{n} = \frac{n/2}{n} = 1/2$ implying that $\Pr_{\text{Beth}}[E] = 1 - \frac{1}{2} = 0.5$.

We have also shown that

$$\Pr_{\text{Alice}}[\neg E] = \left(1 - \tfrac{1}{n}\right)^m \overset{\text{hint}}{\leq} \left(e^{-\frac{1}{n} - \frac{1}{n^2}}\right)^m = e^{-\frac{m}{n} - \frac{m}{n^2}} = e^{-\frac{1}{2} - \frac{1}{2}\cdot\frac{1}{100}} > 0.6$$

Thus $\Pr_{\text{Alice}}[E] < 1 - \Pr_{\text{Alice}}[\neg E] = 0.4$.