COMP 3711 – Design and Analysis of Algorithms
2022 Fall Semester – Written Assignment Solution # 1

Solution 1: [10 points]

(a) **Input**: An array $A[1 \ldots n]$ of n distinct integers for $n \geq 2$.
Output: An array $A[1 \ldots n]$ such that the n items of the input array are sorted in decreasing order.
for $k \leftarrow 1$ to $n-1$ **do**: // the outer loop
  for $i \leftarrow n$ downto 2 **do**: // the inner loop
    if $A[i] > A[i-1]$ **then**
      swap $A[i]$ and $A[i-1]$

(b) Suppose B is the sorted decreasing array of $A[1 \ldots n]$, which is what we expect the algorithm to return. We will prove that after the first k passes, $A[1 \ldots k]$ become $B[1 \ldots k]$. Then it follows naturally that after $n-1$ passes, $A[1 \ldots n-1] = B[1 \ldots n-1]$, which means $A[1 \ldots n] = B[1 \ldots n]$ since the n items are distinct integers. That is, A[1…n] are sorted in decreasing order after $n-1$ passes.

Claim. *By the algorithm described in (a), after the first k passes, $A[1 \ldots k] = B[1 \ldots k]$.*

Proof. By induction on k.
When $k = 1$, that is in the first pass. Suppose initially $B[1]$ is in $A[j]$. Then when the inner loop runs to $i = j$, the algorithm finds that $A[j] > A[j-1]$ because $B[1]$ is the largest item in A. Then $B[1]$ is swapped to $A[j-1]$.
Since $B[1]$ is the largest item in A, the inner loop swaps $B[1]$ with its left neighbor until $i = 2$, then $B[1]$ reaches $A[1]$. So $A[1] = B[1]$ after the first pass.
When $k > 1$, assume that after $k-1$ passes, $A[1 \ldots k-1] = B[1 \ldots k-1]$. Suppose at this time $B[k]$ is in $A[j]$, where $k \leq j \leq n$.
Then in the k-th pass, when the inner loop runs to $i = j$, there are 2 cases:

(1) If $j = k$, the algorithm does not swap any more in this pass because $A[1 \ldots k]$ are in decreasing order.

(2) If $j > k$, the algorithm finds that $A[j] > A[j-1]$ since $B[k]$ is the largest item in $A[k \ldots n]$. Then $B[k]$ is swapped to $A[j-1]$.
Since $B[k]$ is the largest item in $A[k \ldots n]$ and $B[k] < B[k-1] < \cdots < B[1]$, the inner loop swaps $B[k]$ with its left neighbor until $i = k$, then $B[k]$ reaches $A[k]$.

Therefore, after the first k passes, $A[1 \ldots k] = B[1 \ldots k]$. □

(c) The worst case happens when $A[1 \ldots n]$ are sorted in increasing order initially. Then in the k-th iteration, the algorithm costs $n-1$ units of running time.
The worst-case running time of the algorithm is $\Theta(\sum_{k=1}^{n-1}(n-1)) = \Theta(n^2)$.

Solution 2: [10 points]

(a) **Input**: An array $A[1 \ldots n]$ of n distinct integers for $n \geq 1$.
Output: A list of arrays such that each array is a permutation of $A[1 \ldots n]$ and the list contains all permutations of $A[1 \ldots n]$.
Permutate(A, i):

```
if i = 1 then
    return [A[1]] // return a list which contains the only one permutation of A[1]
𝒫 ← [] // an empty list
ℬ ← Permutate(A, i − 1) // ℬ is the list containing all permutations of A[1 ... i − 1]
for B ∈ ℬ do // B is one permutation of A[1 ... i − 1], B = B[1 ... i − 1]
    for k ← 1 to i do
        for j ← 1 to i do
            if j < k then
                P[j] ← B[j]
            else if j = k then
                P[j] ← A[i]
            else
                P[j] ← B[j − 1]
        append P to 𝒫 // P = B[1] ... B[k−1]A[i]B[k] ... B[i−1], one permutation of A[1 ... i]
return 𝒫
First call: Permutate(A, n)
```

(b) When $n = 1$, we have $T(1) = 1$,

When $n > 1$, there are 3 loops. Since $A[1 \dots n-1]$ has $(n-1)!$ permutations, we have

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

The running time is $O(2n \cdot n!)$. We prove this by induction.

Proof. When $n = 1$, $T(1) = 1 < 2 \cdot 1 \cdot 1! = 2$.
When $n > 2$, suppose $T(n-1) < 2(n-1) \cdot (n-1)!$. Then

$$T(n) = T(n-1) + n \cdot n! < 2(n-1) \cdot (n-1)! + n \cdot n!.$$

If we can prove that $2(n-1) \cdot (n-1)! + n \cdot n! < 2n \cdot n!$, then we can obtain $T(n) < 2n \cdot n!$.
So next we prove that $2(n-1) \cdot (n-1)! + n \cdot n! < 2n \cdot n!$.
That is equivalent to proving

$$2(n-1) \cdot (n-1)! < n \cdot n! \iff 2(n-1) < n^2 \iff n(n-2) > -2.$$

Because $n(n-2)$ is an increasing function for $n \geq 2$,
then for $n \geq 2$, $n(n-2) \geq 0 > -2$.
Therefore, we prove that $T(n) < 2n \cdot n!$. □

Solution 3: [5 points]

(a) $n^2 / \log n = O(n^2)$

(b) $2^{\log n} = O(n^3)$

(c) $\log \log n = O(\log n)$

(d) $4^{\log_2 n} = \Theta(n^2)$

(e) $n^{1/3} = \Omega((\log n)^2)$

Solution 4: [10 points]

(a) By master theorem, $c = \log_2 4 = 2$, $f(n) = n = O(n^{2-\epsilon})$ for some $\epsilon > 0$, so $T(n) = \Theta(n^2)$.

(b) By master theorem, $c = \log_2 4 = 2$, $f(n) = n^3 = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$, so $T(n) = \Theta(n^3)$.

(c) By master theorem, $c = \log_3 1 = 0$, $f(n) = \sqrt{n} = \Omega(n^{\epsilon})$ for some $\epsilon > 0$, so $T(n) = \Theta(n^{1/2})$.

(d) By master theorem, $c = \log_4 5$, $f(n) = n \log n = O(n^{\log_4 5-\epsilon})$ for some $\epsilon > 0$, so $T(n) = \Theta(n^{\log_4 5})$.

(e) Suppose $n = 2^{2^k}$, then

$$\begin{aligned}
T(n) &= T(2^{2^k}) = T(\sqrt{2^{2^k}}) + 1 \\
&= T(2^{2^{k-1}}) + 1 \\
&= T(2^{2^{k-2}}) + 1 + 1 \\
&= \dots \\
&= T(2) + k \\
&= T(1) + 1 + \log \log(n) \\
&= \Theta(\log \log(n))
\end{aligned}$$

Solution 5: [10 points]

Find-min-for-convex(A, p, r):
if $p = r$ **then**
  return $A[p]$
if $r = p + 1$ **then**
  if $A[p] < A[r]$ **then**
    return $A[p]$
  else
    return $A[r]$
$q \leftarrow \lfloor (p + r)/2 \rfloor$
if $A[q - 1] > A[q]$ and $A[q + 1] > A[q]$ **then**
  return $A[q]$
if $A[q] < A[q + 1]$ **then**
  return Find-min-for-convex(A, p, q)
if $A[q] > A[q + 1]$ **then**
  return Find-min-for-convex$(A, q + 1, r)$
First call: **Find-min-for-convex**$(A, 1, n)$

Next we prove the correctness of the above algorithm.

Proof. When $p = r$ and $r = p + 1$, the algorithm is obviously correct.
When $r \geq p+2$, suppose the minimum of $A[p \ldots r]$ is $A[j]$, then by the properties of the strictly convex function, we have $A[p] > \cdots > A[j] < \ldots A[r]$.
Let $q = \lfloor (p+r)/2 \rfloor$. We have the following 3 cases:

(1) If $A[q-1] > A[q]$ and $A[q+1] > A[q]$, then $j = q$ must hold. Because there does not exist $q' \neq j$ such that $A[q'-1] > A[q']$ and $A[q'+1] > A[q']$. So the algorithm returns $A[q]$, that is $A[j]$.

(2) If $A[q] < A[q+1]$, then $j \leq q$ must hold. The algorithm calls **Find-min-for-convex**(A, p, q) next, which runs correctly because $p \leq j \leq q$.

(3) If $A[q] > A[q+1]$, then $j > q$ must hold. The algorithm calls **Find-min-for-convex**$(A, q+1, r)$ next, which runs correctly because $q + 1 \leq j \leq r$.

By induction, the algorithm will correctly find the minimum of $A[1 \ldots n]$. □

Running time:
When $n = 1, 2$, $T(n) = O(1)$.
When $n > 2$, the algorithm computes $q \leftarrow \lfloor (p+r)/2 \rfloor$, which costs one unit of running time. Then the algorithm divides the array $A[1 \ldots n]$into two parts and only runs recursively on one part. So $T(n) = T(n/2) + O(1)$.
By the master theorem, $c = \log_2 1 = 0$, $f(n) = O(1) = O(n^0)$, so $T(n) = O(\log n)$.

Solution 6: [10 points]

Polynomial-product$(P[0 \ldots n], Q[0 \ldots n])$:
if $n = 0$ **then**
  $R[0] \leftarrow P[0] \cdot Q[0]$
  return $R[0]$
$m \leftarrow \lfloor n/2 \rfloor$
$U \leftarrow$ **Polynomial-product**$(P[m+1 \ldots n], Q[m+1 \ldots n])$
$Z \leftarrow$ **Polynomial-product**$(P[0 \ldots m], Q[0 \ldots m])$
$P' \leftarrow P[m+1 \ldots n] \oplus P[0 \ldots m]$
$Q' \leftarrow Q[m+1 \ldots n] \oplus Q[0 \ldots m]$
$Y \leftarrow$ **Polynomial-product**$(P'[0 \ldots m], Q'[0 \ldots m])$
$R[0 \ldots 2n] \leftarrow 0$
$R[0 \ldots 2m] \leftarrow R[0 \ldots 2m] \oplus Z$
$R[m+1 \ldots m+n] \leftarrow R[m+1 \ldots m+n] \oplus Y \ominus U \ominus Z$
$R[2m+2 \ldots 2n] \leftarrow R[2m+2 \ldots 2n] \oplus U$
return R

In the above algorithm, $\oplus$ and $\ominus$ denote element-wise addition and subtraction of arrays respectively. $A[0 \ldots n] \oplus B[0 \ldots n]$ is realized by computing $C[i] = A[i] + B[i]$ for $0 \leq i \leq n$ and outputs the array $C[0 \ldots n]$, which costs $O(n)$ running time. Similar with $\ominus$.
Next we prove the correctness of the above algorithm.

Proof. Let $m = \lfloor n/2 \rfloor$, then $p(x) \cdot q(x) =$

$$
\begin{aligned}
&(P[0]+P[1]x+\cdots+P[m]x^m)(Q[0]+Q[1]x+\cdots+Q[m]x^m)+\\
&(P[0]+P[1]x+\cdots+P[m]x^m)(Q[m+1]x^{m+1}+\cdots+Q[n]x^n)+\\
&(P[m+1]x^{m+1}+\cdots+P[n]x^n)(Q[0]+Q[1]x+\cdots+Q[m]x^m)+\\
&(P[m+1]x^{m+1}+\cdots+P[n]x^n)(Q[m+1]x^{m+1}+\cdots+Q[n]x^n).
\end{aligned}
$$

Let $U[0\ldots 2(n-m-1)]$ represents a polynomial product of $P[m+1\ldots n]$ and $Q[m+1\ldots n]$, i.e., $U[0]+U[1]x+\cdots+U[2(n-m-1)]x^{2(n-m-1)} = (P[m+1]+\cdots+P[n]x^{n-m-1})(Q[m+1]+\cdots+Q[n]x^{n-m-1})$,
so $U[0]x^{2m+2}+U[1]x^{2m+3}+\cdots+U[2(n-m-1)]x^{2n} = (P[m+1]x^{m+1}+\cdots+P[n]x^n)(Q[m+1]x^{m+1}+\cdots+Q[n]x^n)$.
Let $Z[0\ldots 2m]$ represents a polynomial product of $P[0\ldots m]$ and $Q[0\ldots m]$, i.e., $Z[0]+Z[1]x+\cdots+Z[2m]x^{2m} = (P[0]+\cdots+P[m]x^m)(Q[0]+\cdots+Q[m]x^m)$.
Let $P' = P[m+1\ldots n]\oplus P[0\ldots m]$, $Q' = Q[m+1\ldots n]\oplus Q[0\ldots m]$.
Let $Y[0\ldots 2m]$ represents a polynomial product of P' and Q' in x, then

$$
\begin{aligned}
&(P[0]+P[1]x+\cdots+P[m]x^m)(Q[m+1]x^{m+1}+\cdots+Q[n]x^n)+\\
&(P[m+1]x^{m+1}+\cdots+P[n]x^n)(Q[0]+Q[1]x+\cdots+Q[m]x^m)\\
&=(Y[0]-U[0]-Z[0])x^{m+1}+\cdots+(Y[2m]-U[2m]-Z[2m])x^{m+n}.
\end{aligned}
$$

Since $p(x)\cdot q(x) = (P[0]+\cdots+P[n]x^n)(Q[0]+\cdots+Q[n]x^n) = R[0]+R[1]x+\cdots+R[2n]x^{2n}$, then from the above derivations, we can allocate the elements in R as the algorithm runs:

$$
\begin{aligned}
&R[0\ldots 2m] \leftarrow R[0\ldots 2m]\oplus Z\\
&R[m+1\ldots m+n] \leftarrow R[m+1\ldots m+n]\oplus Y\ominus U\ominus Z\\
&R[2m+2\ldots 2n] \leftarrow R[2m+2\ldots 2n]\oplus U
\end{aligned}
$$

□

Worst-case running time:
When $n=0$, $T(0)=1$.
When $n>0$, the algorithm divides the problem into 3 polynomial product subproblems, each having size $n/2$. So we have $T(n)=3T(n/2)+O(n)$.
By the master theorem, $c=\log_2 3$, $f(n)=O(n)=O(n^{\log_2 3-\epsilon})$ for some $\epsilon>0$, so $T(n)=O(n^{\log_2 3})$.