ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2017-2018

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudo-code without explanations. A good guideline is that a description of an algorithm should be such that a fellow student can easily implement the algorithm following the description.
- You are allowed to refer to algorithms covered in class without reproving their properties.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 27 points	Problem 2 / 18 points	Problem 3 / 28 points	Problem 4 / 27 points

Total / 100

1 *(27 pts)* **Basic questions.** This problem consists of three subproblems.

1a *(8 pts)* Give tight asymptotic bounds for the following recurrences (assuming that $T(1) = \Theta(1)$):

(i) $T(n) = 2T(n/4) + \Theta(\sqrt{n})$
Follows by the Master Theorem: $a = 2, b = 4$ and $f(n) = \sqrt{n} \Rightarrow T(n) = \Theta(\sqrt{n}\log n)$

(ii) $T(n) = 4T(n^{1/8}) + \Theta(\log n)$
Let $m = \log n$, and let $S(m) = T(n)$. After this substitution the recurrence becomes $S(m) = 4S(\frac{m}{8}) + \Theta(m)$, and thus $S(m) = \Theta(m)$ by the Master theorem. Thus, $T(n) = \Theta(m) = \Theta(\log n)$

(iii) $T(n) = 2T(n-2) + \Theta(1)$
Let $m = n/2$, and let $S(m) = T(n)$. Then we have $S(m) = 2S(n-1) + \Theta(1)$, and thus $S(m) = \Theta(2^m)$. Thus, $T(n) = \Theta(2^{n/2})$.

(iv) $T(n) = 16T(n/4) + \Theta(n^2)$
Follows by the Master Theorem: $a = 16, b = 4$ and $f(n) = n^2$, so $T(n) = \Theta(n^2 \log n)$

1b *(9 pts)* Answer true/false questions below (each question worth 1 point):

A binary tree of height $h \geq 1$ has at most 2^h nodes (recall that a tree with a single node has height 0). True or False?
FALSE, a complete binary tree of height h has $2^{h+1} - 1$ nodes, and this is larger than 2^h for all $h \geq 1$.

The worst-case complexity for searching in a binary search tree is $O(\log n)$. True or False?
FALSE, it's $O(n)$ since a binary tree is not necessarily balanced

A max-heap can be built from an unsorted array $A[1...n]$ in time $O(n)$. True or False?
TRUE, we showed this in the lecture

Extracting the maximum element from a max-heap has worst-case runtime $\Omega(n)$. True or False?
FALSE, it's $O(\log n)$

If $f(n) = n^{2.1}$ and $g(n) = n^2 \log n$, then $f(n) = \omega(g(n))$. True or False?
TRUE, $n^{2.1} = n^2 n^{0.1}$ and $n^{0.1}$ grows faster than $\log n$

If $f(n) = 2^{\sqrt{\log n}}$ and $g(n) = \log^2 n$, then $f(n) = o(g(n))$. True or False?
FALSE, $g(n) = \log^2 n = 2^{\log(\log^2 n)} = 2^{2\log\log n}$. Since $\log\log n = o(\sqrt{\log n})$, $f(n)$ grows faster than $g(n)$.

An array of size n which contains only zeros and ones can be sorted in linear time using a constant amount of additional memory. True or False?
TRUE, one only needs to count 0s and 1s in linear time and constant memory. Say there are u 0s and t 1s – then we output $\underbrace{0, ..., 0}_{u \text{ times}}, \underbrace{1, ..., 1}_{t \text{ times}}$.

If every node in a binary tree has either 0 or 2 children, then the tree has height $O(\log n)$. True or False?
FALSE, restricting the number of children gives no guarantees on the height of the tree – e.g., it can still only grow to the left

Running merge sort on a sorted array takes $O(n)$ time. True or False?
FALSE, the asymptotic running time of merge sort only depends on the size of input, and thus the runtime is still $\Omega(n \log n)$ even on a sorted array

1c *(10 pts)* In this problem you are given the code of a function UNKNOWN(str) that takes as input a string and outputs **true** or **false**.

```
UNKNOWN(str)

1. Initialize an empty stack S
2. n=str.length
3. for i=1 to n
4.       if str[i]=='A' or str[i]=='C'
5.            PUSH(S, str[i])
6.       else if str[i]=='B'
7.             if STACK-EMPTY(S) or POP(S)!='A'
8.                  return false
9.       else if str[i]=='D'
10.              if STACK-EMPTY(S) or POP(S)!='C'
11.                   return false
12. if STACK-EMPTY(S)
13.       return true
14. else
15.       return false
```

What does UNKNOWN output on inputs below?

1. UNKNOWN("ABBA")= FALSE, returns through line 7 at $i = 3$ with $S = \{\emptyset\}$

2. UNKNOWN("ACBD")= FALSE, returns through line 7 at $i = 3$ with $S = \{A\}$

3. UNKNOWN("ABCD")= TRUE, $S = \{\emptyset\}$

4. UNKNOWN("AAAABBBBAAAA")= FALSE, $S = \{A, A, A, A\}$

5. UNKNOWN("ACDBABCDCDCCDD")= TRUE, $S = \{\emptyset\}$

2 *(18 pts)* **Recurrences.** Consider the following algorithm UNKNOWN that takes as input an integer n:

```
UNKNOWN(n):
1. if n < 10
2.       return
3. UNKNOWN(⌊4n/5⌋)
4. for i = 1 to n
5.       for j = 1 to i
6.             print "Almost done!"
7. UNKNOWN(⌊3n/5⌋)
8. return
```

2a *(4 pts)* Let $T(n)$ be the time it takes to execute UNKNOWN(n). **Give the recurrence relation** for $T(n)$. To simplify notation, you may assume that $n/5$ always evaluates to an integer.

Solution: If $n < 50$, the algorithm UNKNOWN(n) takes $\Theta(1)$ time. Otherwise, it calls one instance of UNKNOWN with argument $4n/5$, and one instance with argument $3n/5$. In addition, in lines 5 and 6 it runs two nested loops such that the number of times **print** is called is exactly $\sum_{i=1}^{n} i = \Theta(n^2)$. Thus, the loops take time $\Theta(n^2)$.

This gives us the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < 50, \\ T(4n/5) + T(3n/5) + \Theta(n^2) & \text{otherwise.} \end{cases}$$

2b *(14 pts)* **Prove** tight asymptotic bounds on $T(n)$. Specifically, show that $T(n) = \Theta(n^a \log n)$ for some integer $a \geq 0$. You may simplify your calculations by assuming that $\lfloor n/5 \rfloor = n/5$.

Solution: Let us first make an (educated) guess about a. In the inductive proof, we will have an argument roughly like:

$$\begin{aligned} T(n) &= T(3n/5) + T(4n/5) + \Theta(n^2) \\ &\leq (3n/5)^a \log(3n/5) + (4n/5)^a \log(4n/5) + \Theta(n^2) \\ &\leq n^a((3/5)^a + (4/5)^a) \log n + \Theta(n^2) \end{aligned}$$

and for the inductive proof to work, we will at least need that

$$(3/5)^a + (4/5)^a = 1$$

(with equality because we need a tight bound). Equivalently, this is

$$3^a + 4^a = 5^a.$$

After a hopefully short round of trial and error (or binary-searching on paper), we find that $a = 2$ is the solution to this equation. We now formally prove that $T(n) = \Theta(n^2 \log n)$.

We prove $T(n) = \Theta(n^2 \log n)$:

Claim 1 *There exists a positive constant d such that $T(n) \leq dn^2 \log n$ for all $n \geq 2$.*

Proof. The base case is trivial by selecting d sufficiently large.

Now consider the inductive step (with the induction hypothesis that $T(m) \leq dm^2 \log m$ for all $m < n$):

$$\begin{aligned} T(n) &= T(3n/5) + T(4n/5) + cn^2 \\ &\leq d((3n/5)^2 \log(3n/5) + (4n/5)^2 \log(4n/5)) + cn^2 \\ &= d((3n/5)^2 \log n + (4n/5)^2 \log n) - dn^2((3/5)^2 \log(5/3) + (4/5)^2 \log(5/4)) + cn^2 \\ &\leq d((3n/5)^2 + (4n/5)^2) \log n \quad \text{(by selecting } d \geq c/(((3/5)^2 \log(5/3) + (4/5)^2 \log(5/4)))) \\ &= dn^2 \log n \end{aligned}$$

as required.

□

Claim 1 proves $T(n) = O(n^2 \log n)$. For the lower bound, i.e. for $T(n) = \Omega(n^2 \log n)$, we show the following:

Claim 2 *There exists a positive constant d such that $T(n) \geq dn^2 \log n$ for all $n \geq 2$.*

Proof. Base case is trivial by selecting d small enough.

Now the inductive step (with the induction hypothesis that $T(m) \geq dm^2 \log m$ for all $m < n$):

$$\begin{aligned} T(n) &= T(3n/5) + T(4n/5) + cn^2 \\ &\geq d((3n/5)^2 \log(3n/5) + (4n/5)^2 \log(4n/5)) + cn^2 \\ &= dn^2 \log n - dn^2((3/5)^2 \log(5/3) + (4/5)^2 \log(5/4)) + cn^2 \\ &\geq dn^2 \log n \end{aligned}$$

as long as $d < c/(((3/5)^2 \log(5/3) + (4/5)^2 \log(5/4)))$. □

We have shown that $T(n) = O(n^2 \log n)$ and $T(n) = \Omega(n^2 \log n)$. Thus, $T(n) = \Theta(n^2 \log n)$.

3 *(28 pts)* **Crater crossing.** As you may (or may not) have heard in the news, the famous Fiery Crater in the beautiful Swiss Alps has just been opened to the public, and naturally a number of companies are now trying to establish Tyrolean routes across the crater. Each of the n companies designated a pair of climbers to set up the route: for every $i = 1, \ldots, n$, the i-th pair of climbers occupies distinct positions s_i, t_i along the rim of the crater. The rim of the crater is a perfect circle, and $s_i, t_i \in [0, 2\pi)$ correspond to the angle that the climbers in the i-th pair are positioned at (see Fig. 1). Every pair of climbers is connected by a tight rope (basically a straight line), which is the candidate route. There is a major problem, however: the routes intersect! Since nobody wants to be part of a mid-air collision above a sea of lava, the Swiss Alpine Guides decided to open a subset of routes that do not intersect, and are hence considered safe. Each route also has a non-negative fun parameter f_i, $i = 1, \ldots, n$, and the Mountain Guides would like to open a non-intersecting subset of routes that maximizes the total fun. They need your help.

Input: A collection of n pairs $s_i, t_i \in [0, 2\pi)$ specifying positions of pairs of climbers on the rim of the crater, for $i = 1, \ldots, n$. The fun parameters f_i, $i = 1, \ldots, n$, for each of the n routes. You can assume that no two climbers occupy the same position on the rim of the crater.

Output: The maximum possible total fun (sum of fun parameters) achievable by a non-intersecting subset of routes.

An example problem instance is given in Fig. 1 below. In this instance $n = 4$, and the fun parameters of the 4 routes are $f_1 = f_2 = 7$, $f_3 = 1$ and $f_4 = 10$. The optimal solution opens routes 1, 2 and 3, and the total fun is $7 + 7 + 1 = 15$.

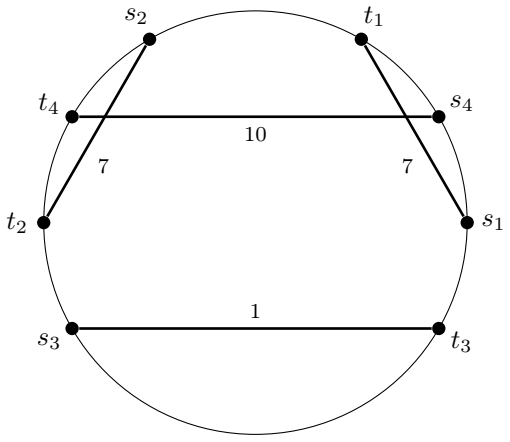


Figure 1. Illustration of set of candidate routes $s_i, t_i, i = 1, \ldots, 4$, where $s_1 = 0$, $t_1 = \pi/3$, $s_2 = 2\pi/3$, $t_2 = \pi$, $s_3 = 7\pi/6$, $t_3 = 11\pi/6$, $s_4 = \pi/6$, $t_4 = 5\pi/6$. The fun parameters are $f_1 = f_2 = 7$, $f_3 = 1$, $f_4 = 10$. The optimal solution opens routes 1, 2 and 3, and the total fun is $7 + 7 + 1 = 15$.

In the following we will design and analyze an efficient algorithm that finds the largest total fun achievable by a non-intersecting subset of routes.

Let $p_1, \ldots, p_{2n}$ denote the $2n$ distinct positions that the climbers occupy along the rim of the crater, in counterclockwise order starting from an arbitrary climber. In the example in Fig. 1, if we start with s_1 and traverse the positions of the climbers in counterclockwise order, we get $p_1 = 0$, $p_2 = \pi/6$, $p_3 = \pi/3$, $p_4 = 2\pi/3$, $p_5 = 5\pi/6$, $p_6 = \pi$, $p_7 = 7\pi/6$, $p_8 = 11\pi/6$. For every $1 \le i \le j \le 2n$ let $c[i, j]$ denote the maximum total amount of fun that can be achieved by opening a non-intersecting set of routes whose endpoints belong to the set $\{p_i, p_{i+1}, \ldots, p_j\}$. Note that $c[1, 2n]$ is the solution that you are asked to find.

3a *(23 pts)* Explain how to express $c[i, j]$ recursively in terms of values $c[a, b]$ for $i < a \le b \le j$. Write down the recurrence relation together with the base case. You may assume that you have access to a function $\textsc{Pair}(i)$ that, given an index $i \in \{1, 2, \ldots, 2n\}$, in $O(1)$ time outputs the index in p of the position of the climber that the climber in position p_i is paired to. In the example above $\textsc{Pair}(2)$=5 and $\textsc{Pair}(5)$=2, since the climber in position 2 is paired to the climber in position 5 (since $p_2 = \pi/6 = s_4$ and $p_5 = 5\pi/6 = t_4$).

Solution: For $1 \le i, j \le 2n$, let f_{ij} equal the fun of the route connecting i and j if such a route exists, and 0 otherwise. Such an array can be easily constructed in $O(n^2)$ time, not affecting the runtime of our implementation – see solutions to part b below. Also, for simplicity we assume that $c[i, j] = 0$ if $i > j$.

We now derive the recurrence relation. There are two cases to consider. If $\textsc{Pair}(i) \notin \{i+1, \ldots j\}$, then $c[i, j] = c[i+1, j]$. Otherwise there are two options: either the route $(i, \textsc{Pair}(i))$ is part of the optimum, in which case we get $f_{i,\textsc{Pair}(i)} + c[i + 1, \textsc{Pair}(i) - 1] + c[\textsc{Pair}(i) + 1, j]$, or it is not, in which case we get $c[i + 1, j]$.

To summarize:

$$c[i, j] = \begin{cases} c[i + 1, j] & \text{if } \textsc{Pair}(i) \notin \{i, \ldots, j\} \\ \max\left\{ f_{i,\textsc{Pair}(i)} + c[i + 1, \textsc{Pair}(i) - 1] + c[\textsc{Pair}(i) + 1, j], c[i + 1, j] \right\} & \text{otherwise} \end{cases}$$

The base case is $c[i, j] = 0$ for $i \ge j$.

3b *(5 pts)* What is the runtime of the bottom-up implementation of the dynamic programming solution to the problem that uses your recurrence from **3a**? Justify your answer.

Solution:

We now show how to implement the recurrence in $O(n^2)$ time. First, sorting the points s_i, t_i and thus constructing the values $p_1, \ldots, p_{2n}$ takes $O(n \log n)$ time. The array f_{ij} as well as an array that implements the function PAIR can now be constructed in $O(n^2)$ time.

Algorithm 1 Initializing arrays fun and $pair$

```
procedure InitializeArrays(s, t, f, n)
    Allocate 2n by 2n array fun of zeroes
    Allocate 2n array pair of zeroes
    for r = 1 to n do                          ▷ Loop over all candidate routes
        idxS = 0, idxT = 0
        for i = 1 to 2n do
            if p_i = s_r then
                idxS=i
            if p_i = t_r then
                idxT=i
        pair[idxS] = idxT
        pair[idxT] = idxS
        fun[idxS, idxT] = f_r
        fun[idxT, idxS] = f_r
    return (pair, fun)
```

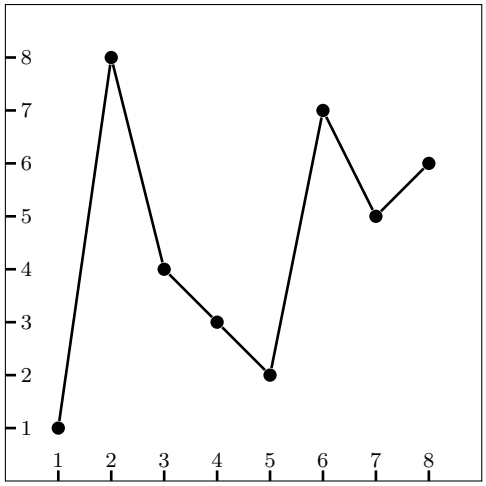
Now the actual code for solving the DP works in $O(n^2)$ time. Note that the array c needs to be filled in the order of increasing $j - i$.

Algorithm 2 Initializing arrays fun and $pair$

```
procedure SolveDP(s, t, f, n)
    Allocate 2n by 2n array c of zeroes
    (pair, fun) ← InitializeArrays(s, t, f, n)
    for ℓ = 1 to 2n − 1 do
        for i = 1 to 2n − ℓ do
            j = i + ℓ
            if pair[i] < i or pair[i] > j then
                c[i, j] = c[i + 1, j]
            else
                c[i, j] = max(fun[i, pair[i]] + c[i + 1, pair[i] − 1] + c[pair[i] + 1, j], c[i + 1, j])
    return c[1, 2n]
```

4 *(27 pts)* **Tallest mountains.** You are planning a hike in the beautiful Swiss Alps again, and are facing a difficult choice: which mountain range should you go to to maximize opportunities for fun hikes? In this problem you will design an algorithm for this challenging task.

A mountain range with n mountains can be represented by an array of mountain heights A of length n, where $A[i]$ for $i = 1, \ldots, n$ is the height of the i-th mountain on the horizon from left to right – see Fig. 2 for an illustration.



Array $A[1 \ldots 8] =$

1	8	4	3	2	7	5	6

Figure 2. Representation of a mountain range as an array A of mountain heights.

A mountain range with n mountains offers $n(n+1)/2$ different hikes: for every $1 \le i \le j \le n$ you can start at the i-th mountain and then visit all mountains $i, i+1, \ldots, j$ in a single trip. The height h_{ij} of a hike from mountain i to mountain $j \ge i$ is the height of the tallest mountain that you visit along the way, i.e. for $1 \le i \le j \le n$ we define

$$h_{ij} := \max_{i \le k \le j} A[k].$$

The total height of a mountain range is sum of heights of all hikes $1 \le i \le j \le n$, i.e. $\sum_{i=1}^{n} \sum_{j=i}^{n} h_{ij}$. Your task is to **design** and **analyze** an efficient algorithm for computing the total height of a mountain range.

> **Input:** An array A of integers of length n. You can assume that all elements of A are distinct.
>
> **Output:** The total height of A: $\sum_{i=1}^{n} \sum_{j=i}^{n} h_{ij}$, where $h_{ij} = \max_{i \le k \le j} A[k]$.

A solution that runs in $O(n \log n)$ time suffices for full credit (e.g. there exists a divide and conquer approach similar to what is used for the maximum subarray problem). $O(n)$ time solutions also exist.

4a *(22 pts)* Design an efficient algorithm for computing the total height of an array A of n integers.

Solution:

For this problem, we propose two solutions. The first one uses stacks and runs in $O(n)$ time, and the second one is based on a modification of the $O(n \log n)$ time solution to the maximum subarray problem presented in class.

$O(n)$ time solution. First note that

$$\sum_{i=1}^{n} \sum_{j=i}^{n} h_{ij} = \sum_{k=1}^{n} A[k] \cdot (next[k] - k) * (prev[k] - k),$$

where $next[k]$ is the smallest $j > k$ such that $A[j] > A[k]$ (we assume that $A[0] = A[n+1] = \infty$ for simplicity), and $prev[k]$ is the largest $j < k$ such that $A[j] > A[k]$.

How can we calculate, for example, the array $prev$? Initialize an empty stack, and push 0 onto the stack (recall we are assuming $A[0] = \infty$ for simplicity). The loop over i from 1 to n. At each iteration first pop elements off the stack while the element at the top of the stack is smaller than i. When we stop popping elements off the stack, set $prev[i]$ to the index at the top of the stack. Push i onto the stack and proceed. To compute $next[i]$, use the same procedure, but start by pushing $n+1$ onto the stack (recall we are assuming $A[n+1] = \infty$ for simplicity), and loop over i from n down to 1. We provide the code below.

Algorithm 3 Solve(A, n)

```
procedure SOLVE(A, n)
    Array prev[n], next[n]
    Stack S1, S2. Push 0 onto S1, n+1 onto S2.
    for i = 1 to n do
        while S1.top() ≠ 0 and A[S1.top()] < A[i] do S1.pop()
        prev[i] = S1.top()
        S1.push(i)
    for i = n downto 1 do
        while S2.top() ≠ n+1 and A[S2.top()] < A[i] do S.pop()
        next[i] = S2.top()
        S2.push(i)
    Answer = 0
    for i = 1 to n do
        Answer = Answer + A[i] · (i − prev[i]) · (next[i] − i)
    return Answer
```

Since every element is pushed onto the stack or popped off the stack exactly once in every pass, runtime is $O(n)$.

$O(n \log n)$ time solution.

We design a Divide-and-Conquer algorithm for this problem. First split the array into two halves, and find the solution in each of them using recursive calls. Then we need to compute the sum of heights h_{ij} of subarrays $A[i...j]$ that cross the midpoint. The recursive solution is similar to the maximum subarray problem, but we will need to design a new version of the merge procedure that takes care of subarrays $A[i...j]$ that cross the midpoint.

We now design the function CROSSING($A, start, end, mid$) that computes $\sum_{1 \le i \le mid} \sum_{mid+1 \le j \le n} h_{ij}$. How do we compute this? Note that for every k between 1 and n the element $A[k]$ contributes to the sum above if it is the maximum for some subarray $A[i...j]$ with i<=mid and j>=mid+1. Which subarrays does a given index k contribute to? To find this, it suffices to start with a left pointer leftIdx equal to $k-1$ and a right pointer rightIdx equal to $k+1$, and then keep moving the left pointer left and the right pointer right while the elements that they are pointing

Algorithm 4 Solve($A, start, end$)

```
1: procedure SOLVE(A, start, end)
2:     if start = end then
3:         return A[start]
4:     mid = ⌊(start+end)/2⌋
5:     answer = Solve(A, start, mid) + Solve(A, mid + 1, end)
6:     answer = answer + COMPUTECROSSING(A, start, end, mid)
7: return answer
```

to are smaller than $A[k]$. Then once both pointers stop, we know that $A[k]$ is the maximum for all arrays whose left endpoint is in [leftIdx+1...k] and the right endpoint is in [k...rightIdx-1]. There are exactly (rightIdx-k)*(k-leftIdx) such subarrays. Not all of these subarrays cross the midpoint. If $k \geq mid + 1$ and lefIdx$\leq$ mid, then the number of crossing subarrays that k is the maximum for is (mid-leftIdx)*(rightIdx-k). Similarly, if $k \leq mid$ and rightIdx$\geq$mid+1, the number of crossing subarrays that k is the maximum for is (k-leftIdx)*(rightIdx-(mid+1)).

How do we find all k's that are maximum for at least one crossing subarray? Start with k=mid if A[mid]>A[mid+1] and k=mid+1 otherwise. We then move the left and the right pointers as above, and update the answer. Then move k to leftIdx or RightIdx, depending on which of A[leftIdx] and A[rightIdx] was the smallest, and repeat. Suppose A[leftIdx] was the smallest. If we were to repeat from scratch, we would need to set rightIdx to k+1, but we know that all elements between k+1 and the current value of rightIdx are smaller than A[k], and in particular smaller than A[leftIdx], so it suffices to only reset the left pointer. This is exactly what leads to linear runtime for COMPUTECROSSING, and $O(n \log n)$ runtime overall.

Algorithm 5 ComputeCrossing($A, start, end, mid$)

```
1:  procedure ComputeCrossing(A, start, end, mid)
2:      if A[mid]>A[mid+1] then
3:          maxIdx=mid
4:      else
5:          maxIdx=mid+1
6:      leftIdx=maxIdx-1, rightIdx=maxIdx+1
7:      ans=0
8:      while leftIdx>=start and rightIdx<=end do
9:          while leftIdx>=start and A[leftIdx]<A[maxIdx] do
10:             leftIdx=leftIdx-1                    ▷ Move left pointer while possible
11:         while rightIdx<=end and A[rightIdx]<A[maxIdx] do
12:             rightIdx=rightIdx+1                  ▷ Move right pointer while possible
13:         if maxIdx>=mid+1 then
14:             ans=ans+A[maxIdx]*(rightIdx-maxIdx)*(mid-leftIdx)
15:         else
16:             ans=ans+A[maxIdx]*(rightIdx-(mid+1))*(maxIdx-leftIdx)
17:         if leftIdx<start and rightIdx<=end then
18:             maxIdx=rightIdx, rightIdx=maxIdx+1
19:         if leftIdx>=start and rightIdx>end then
20:             maxIdx=leftIdx, leftIdx=maxIdx-1
21:         if leftIdx>=start and rightIdx<=end then
22:             if A[leftIdx]>A[rightIdx] then
23:                 maxIdx=rightIdx, rightIdx=maxIdx+1
24:             else
25:                 maxIdx=leftIdx, leftIdx=maxIdx-1
        return ans
```

4b *(5 pts)* Give a tight asymptotic bound on the runtime of your algorithm.

Solution: The running time of both solutions are described in the previous sub-problem.