Data Structures

Tutorial 1:
Solutions for Assignment 1 & 2

Assignment 1

Question 1

Your friend, John, has given you an array *A*[1..n] of n numbers. He told you that there is some i such that *A*[1..i] is straightly increasing, and *A*[i..n] is straightly decreasing. This implies that *A*[i] is the maximum entry in the array.

1. Design an O(log n)-time algorithm to find this maximum.
2. Explain why it runs in O(log n) time.
3. Briefly show the correctness of your algorithm.

Algorithm

```
Finding_Max(Array A, Left-boundary L, Right-
   boundary R)
{
   If(L==R) return A[L];

   Compare A[(L+R)/2] with  A[(L+R)/2 +1])
      If A[(L+R)/2] is bigger then
            Finding_Max(A, L, (L+R)/2) ;
      Else
            Finding_Max(A,  (L+R)/2 +1, R);
}
```

Time complexity analysis

According to our algorithm, after a round, the size of the array would become half.

➔Suppose $n=2^k$, then our algorithm would run $k=\log n$ rounds.

In our algorithm, each round just costs constant time to do the comparison.

Therefore the time complexity of our algorithm is $c\log n=O(\log n)$, where c is a constant.

Correctness Proof

In each round of our algorithm, we compare A[(L+R)/2] with A[(L+R)/2 +1]).

If A[(L+R)/2] is bigger, that means A[(L+R)/2] and A[(L+R)/2 +1]) are in the straightly decreasing section.(why?)

Thus, we are sure that the numbers in the right-hand side of A[(L+R)/2] is impossible to be the maximum.(why?)

Therefore, the maximum must be in the $A[L...(L+R)/2]$, then we continue to find the maximum in it.

By the same reason, if $A[(L+R)/2]$ is smaller, the maximum must be in the $A[(L+R)/2 +1...R]$, then we continue to find the maximum in it.

Finally, we could find the maximum correctly by using our algorithm.

Question 2 – Bubble Sort

```
for (round j = 1, 2, ..., n – 1) {
    for (position i = 1, 2, ..., n – j) {
        if (A[i] > A[i + 1])
            Swap A[i] with A[i + 1];
    }
    if (there is no swapping in a round)
        Break the for-loop;
}
```

Question 2 – Bubble Sort

- Correctness of bubble sort
- Running time = $O(n^2)$
- Worst-case running time = $\Omega(n^2)$
- Running time ≠ $\Theta(n^2)$

Question 2 – Bubble Sort

- Prove by induction
- Induction statement

  At ith round, the last i numbers are at the correct positions and are sorted

- After n rounds, the numbers are sorted
- If the algorithm stops before n rounds, then the numbers are also sorted

Question 2 – Bubble Sort

- Base case

  The largest number must be at the rightmost position after the first round

- Inductive hypothesis

  If after ith round, the statement holds, then it must hold for (i + 1)th round

Question 2 – Bubble Sort

- Wrong proof
- Base case

  When input size is 1, the algorithm is correct
- Inductive hypothesis

  If the algorithm is correct when input size is k, then it must be correct when input size is (k + 1)

Question 2 – Bubble Sort

- Correctness of bubble sort
- Running time = $O(n^2)$
- Worst-case running time = $\Omega(n^2)$
- Running time ≠ $\Theta(n^2)$

Question 2 – Bubble Sort

```
for (round j = 1, 2, ..., n – 1) {                      O(n)
    for (position i = 1, 2, ..., n – j) {               O(n)
        if (A[i] > A[i + 1])                            O(1)
            Swap A[i] with A[i + 1];                    O(1)
    }
    if (there is no swapping in a round)                O(1)
        Break the for-loop;                             O(1)
}
```

$O(n) * O(n) * [O(1) + O(1)] + [O(1) + O(1)] = O(n^2)$

Question 2 – Bubble Sort

- Correctness of bubble sort
- Running time = $O(n^2)$
- Worst-case running time = $\Omega(n^2)$
- Running time ≠ $\Theta(n^2)$

Question 2 – Bubble Sort

- Worst-case

  A[1] > A[2] > … > A[n]

Running time $= (n - 1) + (n - 2) + \ldots + 1$

$= n\,(n - 1) / 2 = \Omega(n^2)$

Question 2 – Bubble Sort

- Worst-case

  $A[1] > A[2] > \ldots > A[n]$

Running time $= (n - 1) + (n - 2) + \ldots + 1$

$= c_1 n^2 + c_2 n + c_3 = \Omega(n^2)$

(c_1, c_2 and c_3 are constants)

Question 2 – Bubble Sort

- Correctness of bubble sort
- Running time = $O(n^2)$
- Worst-case running time = $\Omega(n^2)$
- Running time $\neq \Theta(n^2)$

Question 2 – Bubble Sort

- Running time = $O(n^2)$

  For every input

- Worst-case running time = $\Omega(n^2)$

  Only for worst-case

- Running time ≠ $\Theta(n^2)$
- Prove by contradiction

Question 2 – Bubble Sort

- Suppose running time = $\Theta(n^2)$

  Running time = $\Omega(n^2)$

- Best-case

  $A[1] < A[2] < \ldots < A[n]$

  Running time = $O(n)$

- Contradiction!

Question 3

- Given an array B[1..n] and number Y, find the portion B[i..j] such that B[i] + B[i+1] + … + B[j] = Y

	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]	B[10]
B :	5	3	8	2	6	1	5	8	1	9

Y = 22 B[3..7]

Y = 26 No answer

Question 3 Example

	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]	B[10]
B :	**5**	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**

Y = 22   B[3..7]

Y = 26   No answer

Intuitive Solution

- List all possible combinations and compute their summations

Intuitive Solution

- List all possible combinations and compute their summations
  - There are $O(n^2)$ combinations
  - Computing the summation of each combination needs $O(n)$ time
  - Total time complexity is $O(n^3)$

Intuitive Solution

- List all possible combinations and compute their summations
  - There are $O(n^2)$ combinations
  - Computing the summation of each combination needs $O(n)$ time
  - Total time complexity is $O(n^3)$
- Correctness?

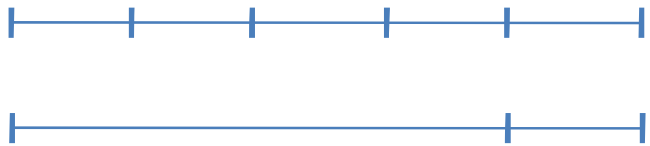
Observation

- There are some redundant computations

Observation

- There are some redundant computations

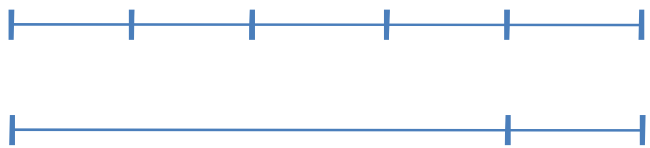
5	3	8	2	6	1	5	8	1	9



Observation

- There are some redundant computations

5	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**

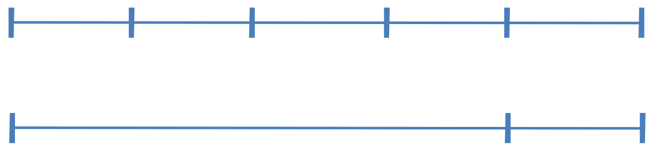


- We can spend only $O(1)$ time to compute the summation of each combination

Observation

- There are some redundant computations

5	3	8	2	6	1	5	8	1	9



- We can spend only $O(1)$ time to compute the summation of each combination
- Total time complexity: $O(n^2)$

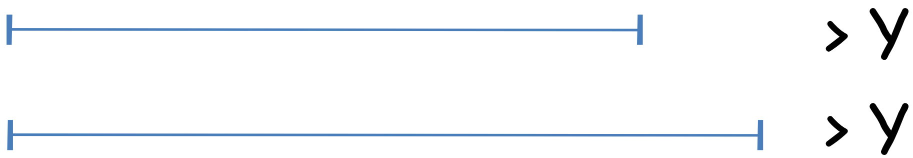
Another Observation

- There are still some redundant computations

Another Observation

- There are still some redundant computations

5	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**



Another Observation

- There are still some redundant computations

5	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**

$> y$

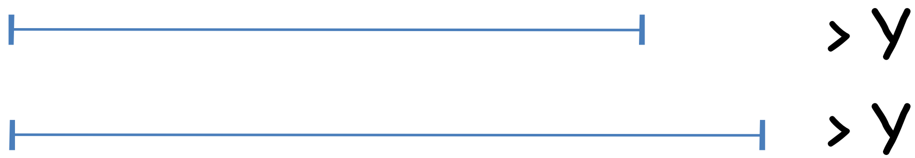
$> y$

- We can drop the first number (why?)

Another Observation

- There are still some redundant computations

5	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**



- We can drop the first number (why?)

Another Observation

- There are still some redundant computations

5	3	8	2	6	1	5	8	1	9

> y

> y

- We can drop the first number (why?)
  - The portion B[1..i] won't be the desired one for any i

Clever Linear Time Algorithm

- Look at the entries from left to right
- At each time we have a candidate portion B[i..j]
  - If the summation of this portion is smaller than Y, pick up the next entry and add it to the previous summation, and the candidate portion becomes B[i..j+1]
  - If the summation of this portion is bigger than Y, drop the first entry of this portion and minus it from the previous summation, and the candidate portion becomes B[i+1..j]
- Repeat the procedure until the summation of B[i..j] is equal to Y, or j = n and the summation of B[i..j] is smaller than Y

Clever Linear Time Algorithm

- Time complexity:
  - Each entry is at most picked up once and dropped once, so the time complexity is $O(n)$

Clever Linear Time Algorithm

- Time complexity:
  - Each entry is at most picked up once and dropped once, so the time complexity is $O(n)$
- Correctness?

Clever Linear Time Algorithm

- Time complexity:
  - Each entry is at most picked up once and dropped once, so the time complexity is O(n)
- Correctness?
- It needs to be prove that we can drop the first number

Assignment 2

Question 1

Recurrences:

- $T(n) = T(n/5) + T(3n/4) + n$
  - Hint: substitution and recursion tree method
- $T(n) = \text{٨}\mathrm{T}(n/2) + n^3$
  - Hint: master theorem
- $T(n) = 8\mathrm{T}(\sqrt{n}) + (\log n)^3$
  - Hint: changing variable

$T(\text{١}) = 1$

Question 1 (a)

- Use substitution method
- Assume $T(n) = cn$ (c is a constant)
- Then $T(n) = (19c/20 + 1)n$
- By solving $(19c/20 + 1)n \leqq cn$
- We get $c \geqq 20$
- Thus, $T(n) = \Theta(n)$

Question 1 (b)

- By master theorem, case 2

- We get $T(n) = \Theta(n^3 \log n)$

Question 1 (c)

- Let $m = \log n$ $(n = 2^m)$
- $T(n) = T(2^m) = 8T(2^{m/2}) + m^3$
- Let $S(m) = T(2^m) = 8S(m / 2) + m^3$
- $S(m) = \Theta(m^3 \log m)$, by Question 1 (b)
- $T(n) = S(m) = \Theta((\log n)^3 \log \log n)$

Question 2

Quick Sort is a very practical algorithm that can sort an array $A[1..n]$ of n distinct numbers. It works recursively as follows:

QuickSort(Array A, Length n)

{

if ($n \leq 1$) return A;

Pick an arbitrary element x from A;

Partition the other elements of A into 2 groups, A_{small} and A_{large}, such that A_{small} = all elements with value smaller than x ;

A_{large} = all elements with value larger than x ;

Use QuickSort to sort A_{small} ;

Use QuickSort to sort A_{large} ;

return sorted A_{small}, followed by x, followed by sorted A_{large};

}

1. Show that QuickSort is correct.

2. Show that in the worst case, the running time of QuickSort is $O(n^2)$.

3. The above algorithm assumes that all the numbers in A are distinct. What will happen if they may be non-distinct?

4. Briefly explain how to modify the above algorithm so that it can handle the case where numbers may not be distinct.

Correctness Proof

For each round in the QuickSort algorithm, we will pick an arbitrary element x from A, then we will put x in the correct position in A. (why?)

So, after we put every element of A into its correct position, the array A would be sorted.

Worst Case

Suppose $A[1\ldots n]=\{x_1,x_2,\ldots,x_n\}$, where $x_1<x_2<\ldots<x_n$.

Assume that, for each round, we always pick the first x in *A* and then we partition the other elements of *A* into 2 groups, A_{small} and $A_{large.}$

➔How many number of comparison does the algorithm have?

sol: $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 = O(n^2)$.

3. What will happen if the elements may be non-distinct in A?

Sol: We would not know how to partition the elements.

4. How to modify the above algorithm so that it can handle the case where numbers may not be distinct?

Sol: We could just distribute the non-distinct numbers into A_{small} group (or A_{large} group) when we partition the elements of A.

Question 3

- Given array B[1..n], list the smallest k numbers in sorted order in $O(n + k \log n)$ time

Question 3 Example

- Given array B[1..n], list the smallest k numbers in sorted order in $O(n + k \log n)$ time

5	**3**	**8**	**2**	**6**	**1**	**5**	**8**	**1**	**9**

K = 5

Ans: 1 1 2 3 5

Question 3

- Meaning of $O(n + k \log n)$ time:
  - It means that the time we spend is $O(\text{Max}\{n, k \log n\})$

Question 3

- Meaning of $O(n + k \log n)$ time:
  - It means that the time we spend is $O(\text{Max}\{n, k \log n\})$
  - When k is small, the time complexity is $O(n)$
  - When k is large, the time complexity is $O(k \log n)$

Question 3

- Meaning of $O(n + k \log n)$ time:
  - It means that the time we spend is $O(Max\{n, k \log n\})$
  - When k is small, the time complexity is $O(n)$
  - When k is large, the time complexity is $O(k \log n)$
  - Hence, we can't simply use the comparison sorting algorithms (why?)

Question 3

- Make the elements in array B to a min heap by heapify
- Do k extract-min operations

Question 3

- Make the elements in array B to a min heap by heapify
- Do k extract-min operations

- What is the time complexity?