Faculty of Computer &
Information Sciences
Ain Shams University

Algorithms Design and Analysis
D&C Sheet

3rd Year 2nd Semester - 2024

TABLE OF CONTENTS

Algorithms Design and Analysis
D&C Sheet

FIRST: General Questions

QUESTION1: T/F

1. Overall complexity of the selection sort is θ(N^2)
2. Overall complexity of the insertion sort is θ(N^2)
3. Merge sort takes θ(N) when the array is sorted
4. The cost of partitioning in Quick sort is θ(N) while that of Merge sort is θ(1)
5. Binary insertion sorting (insertion sort that uses binary search to find each insertion point) requires O(n log n) total operations
6. In the merge-sort execution tree, roughly the same amount of work is done at each level of the tree
7. Best case complexity of selection sort is Ω(N)?

Answer:

1. T
2. F
3. F
4. T
5. F
6. T
7. F

QUESTION2: Tracing

1. For the **Quick Sort** algorithm (Ascending) with **first element as a pivot**, Trace the algorithm on the array {29, 6, -2, -3, -8, -11, 50, 0} and answer the following questions:

a) Write down the **entire array** after **finishing** the **first level** (i.e. after placing 1st pivot in its correct location)?

b) Write down the **entire array** after **finishing** the **third level** (i.e. after placing 3rd level pivot in its correct location)?

c) What's the value of the **LAST** selected pivot?

Answer:

a) [0, 6, -2, -3, -8, -11, 29, 50]

b) [-11, -8, -2, -3, 0, 6, 29, 50]

c) -2

2. For the **Merge Sort** algorithm, Trace the algorithm on the array {38, 41, 52, 26, 35, 57, 9, 49} and answer the following questions:

a) After finishing all the partitioning steps, how many levels are there (including the first level)?

b) Write down the array after finishing the merge step of the last (bottom) level?

c) Write down the array after finishing the merge step of the level before last?

Answer:

a) [4]

b) [38, 41, 26, 52, 35, 57, 9, 49]

c) [26, 38, 41, 52, 9, 35, 49, 57]

3. If you apply the **Insertion sort** to this array: (12, 69, 97, 92, 83, 77, 56, 57, 26, 73). Which of the following sequences can be occurred during the execution?

a.	12,26,56,57,69,73,83,92,97,77
b.	12,69,77,83,92,97,56,57,26,73
c.	12,26,56,57,69,77,83,92,73,97
d.	12,56,57,69,73,77,83,92,97,26
e.	12,26,56,57,83,77,97,92,69,73

Answer:

b

4. For the **Select Kth Element algorithm#1** with expected linear time and **first element as a pivot**, answer the following questions on the array {29, 6, -2, -3, -8, -11, 50, 0}:

a) How many pivots are selected until finding the **2nd smallest** element? What's this element?

b) How many pivots are selected until finding the **4th largest** element? What's this element?

Answer:

a) 3, -8

b) 2, 0

5. For the **worst case linear time** algorithm (ALG#2) with **groups of 5**, what's the value of the **first selected pivot**: {5, -1, 7, -4, 9, -3, 8, 0, 2, -6, 1, 2, 3, 4, 5, 9, 9, 9, 9, 9, -7, -8, 6, 6, 4}

Answer:

4

QUESTION3: Others

1. Which sort is better (Insertion, Quick, Merge, Bubble, Selection)

You are running a library catalog. You know that the books in your collection are almost in sorted ascending order by title, with the exception of one book which is in the wrong place. You want the catalog to be completely sorted in ascending order.

Answer:

Since it's almost sorted except 1 book, the INSERTION sort is the most suitable one as it'll takes Θ(N) for checking all elements and a max of O(N) to place the 1 book into its correct place.

2. If an $O(n^2)$ algorithm (e.g., the bubble sort, the selection sort, or the insertion sort) takes 3.1 milliseconds to run on an array of 200 elements, how long would you expect it to take to run on a similar array of:

a. 400 elements?
b. 40,000 elements?

Answer:

Since the algorithm is N^2, so
200^2 steps ➔ 3.1 milliseconds
400^2 steps ➔ ?! milliseconds
Solve to get the ?!

3. In the **Quicksort with first element as pivot**, if all elements are **identical** and either left or right iterators will **skip over** equal keys. What's the complexity of the algorithm in this case?

a.	$\Theta(1)$
b.	$\Theta(N)$
c.	$\Theta(N \log(N))$
d.	$\Theta(N^2)$
e.	None of the choices

Answer:
d

4. Suppose you have invented a new version of Quicksort, and you find, much to your surprise, that your new version always selects the second largest element of the list as the pivot point. What is the worst and average case of your new algorithm?

SOLUTION

Since the behavior of the algorithm is independent of the input, average and worst case are the same.

Worst case is given by the following recurrence.

$T(n) = T(n-2) + (n-1)$, Solve by iteration method

⇨ **$\Theta(N^2)$**

5. Suppose your smart friend gives you a technique for merging two sorted arrays into one sorted array in constant time. Analyze the complexity of merge sort using your friend's technique?

SOLUTION

Since the work done by non-recursive part becomes **$\Theta(1)$**, the recurrence becomes:

$T(n) = 2\,T(n/2) + \Theta(1)$,

Solve by Master method (case 1)

- ⇨ **$\Theta(N)$**

SECOND: DESIGN PROBLEMS [with solution]

QUESTION1: Two Items with Difference X

- **Given**
  - An array **A** of ***N*** integers.
  - An integer value **X**.
- **Required**
  - Check is there TWO items whom difference equal X?
- **Examples**
  - Input: A = {−7, −2, 5, −3, 8}, X = 4
  - Output: yes (-3 – (-7))

  - Input: A = {−10, 4, −5, 9, 8}, X = 2
  - Output: No

Naive Solution

- We shall try all possibilities (each number with the others) → **$O(N^2)$**

Efficient Solution

1. **Idea**
   - Goal: Check if two items have difference = X
   - The solution here to avoid **O(N^2)** based on 2 steps:
     1. Sort the array, for example using merge sort →Θ(N log N)
     2. Then start the D&C solution by:
        - Looping for each element in the array -> O(N)
        - for each element A[i], Calculate the remaining R = abs(X-A[i])-> Θ(1)
        - Then, binary search for R in the array -> O(log N)
2. **D&C solution:**

   For each element ***i*** : calculate remaining R = X – A[i]
   1. Divide:
      – Compare this value (R) with the middle of the array (if found, terminate!)
   2. Conquer:
      – By searching for R in one of the TWO halves based on if greater or smaller than the middle
   3. Combine:

      Trivial

```
TwoItemsWithDifference(array A, int N, int X)
{
      MergeSort(A,N);
      For i = 0 to N
            R = abs(A[i] - X)
            ret = BinarySearch(A, 0, N-1, R)
            if ret != -1
                  return True       //this means that the 2 numbers
are exist that their difference equals to X, otherwise the loop will
continue to check the next element in the array.

}
BinarySearch(array A, int start, int end, int R)    //T(N)=T(N/2)+O(1)
{
       if (low > high)
            return -1;
      mid = floor((low + high)/2);
      if (A[mid] == R)        return mid;
      else if (A[mid] > R)    return BinarySearch(A, low, mid-1, R);
      else                    return BinarySearch(A, mid+1, high, R);
}
```

- **Analysis**
  1. Pre-SORT = **Θ(Nlog(N))**
  2. Binary search remain of each element = **N×O(Log(N))**
  3. **TOTAL = Θ(Nlog(N))**

QUESTION2: Powering a Number

Write an efficient algorithm to compute $A^N, N\ is\ positive\ integer$ without using "Power" function?

SOLUTION

Powering a Number (Normal Solution)

```
POWER-A-NUMBER(A, N)
    Pow = 1
    For I = 1 to N
        Pow = Pow × A
    End for
    Return Pow
```

$\Theta(N)$

Powering a Number (D&C)

D&C Steps:

1. **Divide:** Trivial.
2. **Conquer:** Recursively solve 1 part only with N/2.
3. **Combine:** Mul. the solution of subproblem by itself (and by A if odd).

Pseudocode:

```
POWER-A-NUMBER(A, N)
    If N = 1 then
        Return A
    pow = POWER-A-NUMBER(A, floor(N/2))
    if N % 2 ≠ 0 then
        return pow × pow × A
    else
        return pow × pow
    end if
```

Analysis: similar to previous problem

1. Calculate running time (recurrence relation)

   $T(N) = T(N/2) + O(1)$

2. Solve it using master method:
   - $a = 1$, $b = 2$, $f(N) = 1$
   - Compare $f(N)$ vs. $N^{\log_b a}$
   - $N^{\log_b a} = N^{\log_2 1} = N^0 = 1$ vs. $f(N) = 1$
   - $N^{\log_b a} = \boldsymbol{\theta}(f(N))$ ➔ It's **case (2)**,

$$T(N) = \Theta(Log(N))$$

QUESTION3: Even Odd Checking

Given an array with a set of integer numbers, each number value can be between 1 and 231-1, you have to determine if the sum of all the numbers is odd (return true) or even (return false)

Note: 1-Adding all the items' values might cause an overflow
2-Even + Even = Even, Odd + Odd = Even, Odd + Even = Odd

Sample Input:
1 30 4 300000000 5 8 7 4 5
Sample Output:
false
(Note: summing all the 9 number will lead to an even number so we returned false)

Naiive IDEA

Summing all numbers then get the total and check its modulus if 0 then then sum is even and otherwise it is odd.
Issue in this idea is that this is not an acceptable solution, since the summation at a specific number will result in overflow.
Time complexity: O(N)

SOLUTION

Idea:

Since summing the numbers can lead to an overflow, so we can utilize the divide & conquer design to get another acceptable solution that doesn't add the numbers.

This can be done by utilizing the following given note in the problem, which is:

Even + Even = Even, Odd + Odd = Even, Odd + Even = Odd

D&C Steps

1. **Divide:** Trivial (Divide the array into 2 subarrays) -> O(1)
2. **Conquer:** Recursively check the 2 subarrays -> 2T(N/2)
3. **Combine:** Check the return status (i.e. even/odd) from the two subproblems as follows: -> O(1)
    a. Even + Even -> return Even (False)
    b. Odd + Odd -> return Even (False)
    c. Even + Odd -> return Odd (True)

```
private static bool Calc(int first, int Last, int[] arr)
{
    if (first == Last)
    {
        if (arr[first] % 2 == 0)
            return false;
        else return true;
    }
    int mid = (first+Last)/2;
    bool a = Calc(first, mid, arr);
    bool b = Calc(mid + 1, Last, arr);

    if (a == b)
        return false;
    return true;
}
```

Analysis:
From this code we can deduce that T (N) =2T(N/2) +O(1)
Using Master method ➔ **Θ(N)**

Note:
The point here is the D&C solution does not enhance the time complexity to be better than O(N), it consumes O(N) as the naive summation idea. While it is used here to solve the problem to obtain a successful solution instead of the failed summation idea that can cause overflow.

QUESTION4: Unimodal Search

An array A[1…N] is unimodal if it consists of an increasing sequence followed by a decreasing sequence, or more precisely, if there is an index $m \in \{1,2,\dots,N\}$such that

- $A[i] < A[i+1]\ \forall\ 1 \le i < m, and$
- $A[i] > A[i+1]\ \forall\ m \le i < N$

In particular, $A[m]$ is the maximum element, and it is the unique "locally maximum" element surrounded by smaller elements ($A[m-1]\ and\ A[m+1]$)
Give an algorithm to compute the maximum element of a unimodal input array in an efficient way?

SOLUTION

D&C Steps

1. **Divide:** Check middle element with its adjacent.
2. **Conquer:** Recursively search 1 sub-array.
3. **Combine:** Trivial.

Pseudocode

```
UNIMODAL-SEARCH(A, low, high)
    If low == high then
        return low

    mid = low + (high - low)/2
    if A[mid] > A[mid-1] and A[mid] > A[mid+1] then
        return mid
    if A[mid] > A[mid-1] then
        return UNIMODAL-SEARCH(A, mid + 1, high)
    else
        return UNIMODAL-SEARCH(A, low, mid - 1)
    end if
```

Analysis

1. Calculate running time (recurrence relation)

   $T(N) = T(N/2) + O(1)$

2. Solve it using master method:

   - $a = 1, b = 2, f(N) = 1$
   - Compare $f(N)$ vs. $N^{\log_b a}$
   - $N^{\log_b a} = N^{\log_2 1} = N^0 = 1$ vs. $f(N) = 1$
   - $N^{\log_b a} = \theta(f(N))$ ➔ It's **case (2)**,

$$T(N) = \Theta(Log(N))$$

QUESTION5: K-Piles

A volunteer selects 3 cards from a deck containing N distinct cards. We can then show cards in k piles to the volunteer, who will point to all the piles containing one or more of the three cards. The computational model we will use is that it takes 1 time unit to show 1 pile of cards to the volunteer. How can we determine which three cards the volunteer picked in an efficient way?

SOLUTION

If we show the volunteer 2 piles, and they both have mystery cards, then we are no further ahead. But, if we show 4 equaled piles, then at least one of them must not contain one of the cards, since the 3 cards can be found in at most 3 piles. All the cards in the remaining pile not pointed to are no longer candidates for the 3 cards to guess, so they can be put aside. Thus, by showing 4 piles, we are sure to eliminate at least 1/4 of the remaining cards.

D&C Steps:

1. **Divide:** split the cards into K piles and ask about each pile.
2. **Conquer:** Recursively splitting & asking on the selected piles.
3. **Combine:** Trivial.

Analysis:

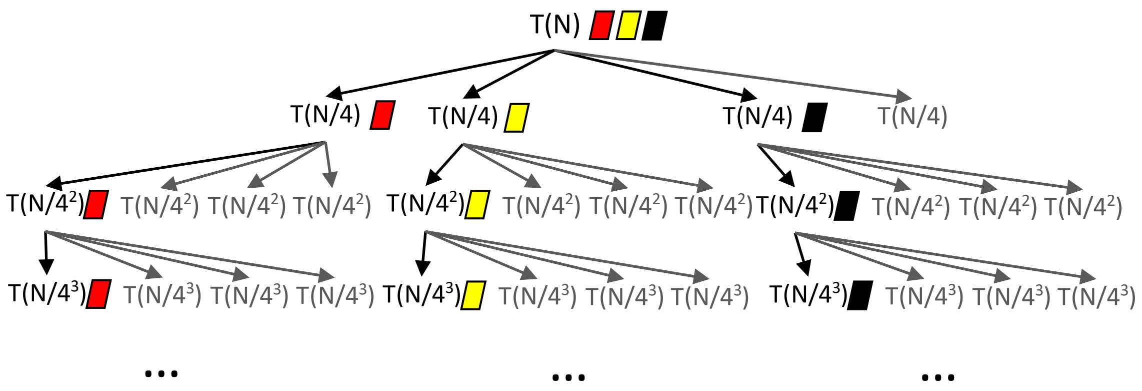
So, an algorithm for determining the 3 mystery cards is to repeat showing 4 piles and eliminate the cards in piles not pointed to, reducing the size of the problem by at least 1/4 each time, until each pile contains only 1 card. The recurrence for this algorithm will be

$$T(N) = 3T(N/4)+c$$

Using Master Method: this will give $o(N^{\log_4 3}) = o(N^{0.8})$.

It's upper bound!! do you know why…?!

In this solution, we assume that we'll continue **working on 3 piles at each level**… which is not true except for 1st level in the **worst case!!** The case in which each card will appear in separate pile at the first level, then when we divide any of these piles into 4 quarters again, the card will be in one of them… leaving 3 quarters away! Look to the following tree:



It's clear from the figure that we have, in the worst case, **3 sub-trees** each with the following recurrence:

$$T(N) = T(N/4) + c$$

Which, by master method case 2, takes

$$\Theta(\log(N))$$

Thus, the final worst case order of all three sub-trees is

$$\Theta(3 \times \log(N)) = \Theta(\log(N))$$

Note that it's Big- Θ now!!

Can you deduce the best case?

QUESTION6: Celebrity Person

Adel is a person whom everybody knows but he knows nobody. You have gone to a party. There are total *n* persons in the party. Your job is to find Adel in the party. You can ask questions of the form "*Does person X know person Y?*" How many questions you should ask to find Adel. *(O(n)* ,O(n^2). *O(nlogn),….)*.

SOLUTION

D&C Steps:

1. **Divide:** ask the question between two persons.
2. **Conquer:** Recursively asking the question between the selected person and the remaining ones.
3. **Combine:** Trivial.

Pseudocode:

`Persons`: is an Array Containing Persons

In main function:

```
X= Persons[1]            ; Initialization X to be first person.
Persons = Persons – {X} ; Remove it from the Persons List.
Adel = GetAdel(Persons, X)
```

```
GetAdel(Persons, Idx, X) returns Adel   //function that returns Adel
{
      If (Idx > N)
      {
            //No more persons to check against them→ X is Adel
            Return X
      }
      If (X knows Persons[Idx])
      {
            //Since Adel knows NOBODY→ X is not Adel, continue with
            //Persons[Idx] after skipping it from the list
            GetAdel(Persons, Idx+1, Persons[Idx])
      }
      Else
      {
            //Persons[Idx] is not Adel since Adel is known by
            everybody. Skip it from the list and continue with X
            GetAdel(Persons, Idx+1, X)
      }
}
```

This code keeps track of person X such that

1. Must **knows nobody**
2. Is **known by everybody**

➔So, by the end, when the list is finished, X must be Adel!!

Analysis: T (N) =T(N – 1) + c ➔ Using Iteration/Recursion tree method ➔ **Θ(N)**

QUESTION7: A[i] = i?

Suppose you are given an increasing sequence of n distinct integers in an array A[1..n]. Design an efficient algorithm to determine whether there exists an index i such that A[i] = i. Analyze the running time of your algorithm.

SOLUTION

Idea of the algorithm

The algorithm starts from the middle of the array A and check if we have found a match (i.e. $A[mid]$ = mid). If not, we check to see if what we've got is greater or lesser than the required value. If it is greater (i.e. $A[mid] > mid$), then we can confidently ignore all the values greater than and including the *mid*point. If it is smaller (i.e. $A[mid] < mid$), then we can ignore all the values smaller than and including the *mid*point. This goes on until we either and a match or when the *low* point is greater than

Details

D&C Steps:

1. **Divide:** Check middle element against its index.
2. **Conquer:** Recursively checking in one of the two halves
3. **Combine:** Trivial.

Pseudocode:

```
Check_Existence(array A, int low, int high)
{
       if (low > high)
            return false;
      mid = floor((low + high)/2);
      if (A[mid] == mid)
            return true;
      else if (A[mid] > mid)
            return Check_Existence(A, low, mid-1);
      else
            return Check_Existence(A, mid+1, high);
}
```

It is like binary search except the condition. In binary search we are searching for A[mid] =key. But here we are searching for A[mid]= mid.

From this code we can deduce that T (N) = T(N/2) + c;

Master Method:

F(N) vs $N^{\log_b a}$

1 vs $N^{\log_2 1}$

1 vs 1 => it is Θ(Log N) (**case 2**)

QUESTION8: Guess a Number!

Your friend guesses an integer between A and B: You can ask questions like is the number less than 100? He will give YES NO answers. How many questions can your friend force you to ask, if you are a smart person?

SOLUTION

From the problem statement, our guess will be in the form " **is the number less than X**".

(Hint:This is a game that most of us play it ..Beta3et "اخترلك رقم")

Details

D&C Steps:

4. **Divide:** Ask the question on the middle value of the range.
5. **Conquer:** Recursively asking in one of the two halves of the range
6. **Combine:** Trivial.

Pseudocode:

```
int FindNumber (int low, int high)
{
      if (low == high)
            return low;
      mid = floor((low + high)/2);
      if (Is the number less than mid? == true)
            return FindNumber(low, mid-1);
      else
            return FindNumber(mid, high);
}
```

Analysis:

As shown it also has same behavior as binary search except the conditions so.

Let N = B – A

T(N) = T(N/2) + c which is Θ(Log N) BY Master Method

QUESTION9: Integer Multiplication

It's required to multiply two N-digit large integer numbers in an efficient way?

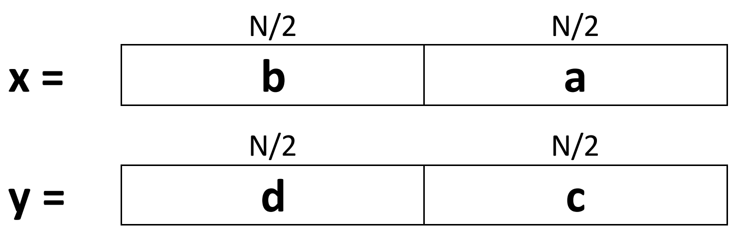
Solution:

refer to its video in [D&C Lecture]

D&C Steps:

4. **Divide:**
   a. split x into two halves: a & b
   b. split y into c & d.
   c. Calculate a+c and b+d
5. **Conquer:** Recursively calculate 3 multiplications: ac, bd and (a+c)(b+d).
6. **Combine:**
   a. subtract the first two multiplications from the third one.
   b. pad the second multiplication (bd) with N zeros and the subtraction result with N/2 zeros.
   c. add them together with the first multiplication (ac).

Explanation:



We can write x and y as follows:

$$x = 10^{\frac{N}{2}} \times b + a$$

$$y = 10^{\frac{N}{2}} \times d + c$$

Multiplying x by y will give the following:

$$result = 10^{N} \times (\boldsymbol{b} \times \boldsymbol{d}) + 10^{\frac{N}{2}} \times (\boldsymbol{b} \times \boldsymbol{c} + \boldsymbol{a} \times \boldsymbol{d}) + (\boldsymbol{a} \times \boldsymbol{c})$$

Now, we have four multiplications, each of size N/2…

However, the two multiplications $(\boldsymbol{b} \times \boldsymbol{c} + \boldsymbol{a} \times \boldsymbol{d})$ can be reduced to one using the **Gauss trick**, as follows:

$1.\, calculate\ z = (\boldsymbol{a} + \boldsymbol{b}) \times (\boldsymbol{c} + \boldsymbol{d}) = ac + bd + bc + ad$

$2.\, subtract\ ac\ \&\ bd\ from\ z\text{:}\ \boldsymbol{w} = z - (\boldsymbol{a} \times \boldsymbol{c}) - (\boldsymbol{b} \times \boldsymbol{d}) = bc + ad$

$Now, \boldsymbol{w}\ contains\ the\ required\ multiplications\text{:}\ (\boldsymbol{b} \times \boldsymbol{c} + \boldsymbol{a} \times \boldsymbol{d})$

We can rewrite the multiplication of x and y using three multiplications only instead of four:

$$\boldsymbol{m1} = \boldsymbol{a} \times \boldsymbol{c}, \qquad \boldsymbol{m2} = \boldsymbol{b} \times \boldsymbol{d}, \qquad \boldsymbol{z} = (\boldsymbol{a} + \boldsymbol{b}) \times (\boldsymbol{c} + \boldsymbol{d})$$

$$result = 10^{N} \times \boldsymbol{m2} + 10^{\frac{N}{2}} \times (\boldsymbol{z} - \boldsymbol{m1} - \boldsymbol{m2}) + \boldsymbol{m1}$$

Analysis:

1. Calculate running time (recurrence relation)

   Non recursive part:

   1. Divide step:
      a. it takes $\theta(N)$ to divide the N-digit number into two halves.
      b. it takes $\theta(N)$ to add the two halves together.
   2. Combine step:
      a. it takes $\theta(N)$ to subtract the first two multiplications from the third one.
      b. it takes $\theta(N)$ to pad the subtraction results with N/2 zeros.
      c. it takes $\theta(N)$ to pad the second multiplication (bd) with N zeros.
      d. it takes $\theta(N)$ to add them together with the first multiplication (ac).

   So, non-recursive part of the code takes $\boldsymbol{\theta(N)}$

   Recursive part:

   1. Conquer step:
      a. Since there're three multiplications that are recursively calculated, each of size $N/2$. So, it takes $\boldsymbol{3 \times T(N/2)}$.

   So, $T(N) = \boldsymbol{3 \times T\left(\frac{N}{2}\right)} + \boldsymbol{\theta(N)}$

2. Solve it using master method:

   a = 3, b = 2, f(N) = N

   Compare f(N) vs. $N^{\log_b a}$

   $N^{\log_b a} = N^{\log_2 3} = N^{1.58}$ vs. f(N) = N

   $f(N) = O\left(N^{\log_b(a)-\epsilon}\right)$ ➔ It's <u>**case (1)**</u>,

$$N = O(N^{1.58-\epsilon})\text{, } any\ 0 < \epsilon \le 0.58\ will\ satisfy\ the\ relation \xrightarrow{yields} master\ succeed$$

$$\boldsymbol{T(N) = \Theta(N^{1.58})}$$

$\boldsymbol{Only\ enhanced\ by\ 0.42!!!\ \ Is\ it\ Worth?!}$

N	Naïve Solution $\boldsymbol{\Theta(N^2)}$	D&C Solution $\boldsymbol{\Theta(N^{1.58})}$	**Speedup factor**
100	10,000	1,446	**7x**
1000	1,000,000	54,954	**18x**
10,000	100,000,000	2,089,296	**48x**

QUESTION10: Majority Element

Assume you have an array $A[1::n]$ of n elements. A *majority element* of A is any element occurring in more than $n/2$ positions (so if $n = 6$ or $n = 7$, any majority element will occur in at least 4 positions). Assume that elements *cannot* be ordered or sorted, but can be compared for equality. (You might think of the elements as chips, and there is a tester that can be used to determine whether or not two chips are identical.) Design an efficient divide and conquer algorithm to find a majority element in A (or determine that no majority element exists).

Aim for an algorithm that does $O(n \lg n)$ equality comparisons between the elements. A more difficult $O(n)$ algorithm is possible, but may be difficult to find.

SOLUTION

The algorithm begins by splitting the array in half repeatedly and calling itself on each half. This is similar to what is done in the merge sort algorithm. When we get down to single elements, that single element is returned as the majority of its (1-element) array. At every other level, it will get return values from its two recursive calls.

The key to this algorithm is the fact that if there is a majority element in the combined array, then that element must be the majority element in either the left half of the array, or in the right half of the array. There are 4 scenarios:

1. Both return \no majority." Then neither half of the array has a majority element, and the combined array cannot have a majority element. Therefore, the call returns \no majority."
2. The right side is a majority, and the left isn't. The only possible majority for this level is with the value that formed a majority on the right half, therefore, just compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return \no majority."
3. c. Same as above, but with the left returning a majority, and the right returning \no majority."
4. d. Both sub-calls return a majority element. Count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array, then return it. Otherwise, return \no majority." The top level simply returns either a majority element or that no majority element exists in the same way.

ANALYSIS

To analyze the running time, we can first see that at each level, two calls are made recursively with each call having half the size of the original array. For the non-recursive costs, we can see that at each level, we have to compare each number at most twice (which only happens in the last case described above). Therefore, the non-recursive cost is at most $2n$ comparisons when the procedure is called with an array of size n. This lets us upper bound the number of comparisons done by $T(n)$ defined by the recurrence $T(1) = 0$ and $T(n) = 2T(n/2) + 2n$:
We can then determine that $T(n) = \Theta(n \log n)$ as desired using Case 2 of the Master Theorem.

QUESTION11: Max Difference

Let A[1::n] be an array of positive integers. Design a divide-and-conquer algorithm for computing the maximum value of A[j] - A[i] with j ≥ i.

Analyze your algorithm running time.

SOLUTION

If we divide A into two roughly equal size sub-arrays, each with approximately *n*=2 elements, we observe that the maximum value of *A*[*j*] - *A*[*i*] must be one of the following:

1. The maximum value of A[j] - A[i] in A[1:: $\lfloor n/2 \rfloor$] where 1 ≤ i ≤ j ≤ $\lfloor n/2 \rfloor$
2. The maximum value of A[j] - A[i] in A[$\lfloor n/2 \rfloor$+ 1::n] where $\lfloor n/2 \rfloor$+ 1 ≤ i ≤ j ≤ n
3. The maximum value of *A*[*j*] - *A*[*i*] where *A*[*i*] is the minimum value of *A*[1:: $\lfloor n/2 \rfloor$] and *A*[*j*] is the maximum value of *A*[$\lfloor n/2 \rfloor$ + 1::*n*]

Input

A is an array of positive integers

p is the start index of *A*

r is the end index of *A*

Output

An integer array [*MaxAll; MinAll; ValAll*] where *MaxAll* and *MinAll* are the maximum and minimum value of the input array respectively *ValAll* is the maximum value of *A*[*j*] - *A*[*i*] with *j* ≥ *i*

ANALYSIS

Let *T*(*n*) be the worst-case number of comparisons for problem of size *n*. When *n* = 1, FMD takes constant time. So, *T*(1) = *O*(1).

For *n* > 1, FMD performs two recursive calls on partitions of half of the total size of the array and takes constant time for the operations in the combining step. Therefore we have the following recurrence.

$T(n) = 2T(n/2) + O(1)$ ➔ Solve by Master method ➔ O(N)

THIRD: DESIGN PROBLEMS [without solution]

Problem1: Position of Largest Element

Design a **divide-and-conquer** algorithm for finding the **position of the largest element** in an array of ***N*** numbers?

Problem2: Is A[] & B[] Contain Common Element

Let *A* and *B* be two sets of integers, represented as unsorted arrays. Suppose that $|A| = n$, $|B| = m$, and $n \le m$. Write an efficient algorithm (in high-level pseudocode) to decide if *A* and *B* contain any common elements in $O(m \log n)$ time.

Problem3: Equivalent Strings

Two strings a and b of equal length are called equivalent in one of the two cases:
1. They are equal.
2. If we split string a into two halves of the same size a_1 and a_2, and string b into two halves of the same size b_1 and b_2, then one of the following is correct:
    1. a_1 is equivalent to b_1, and a_2 is equivalent to b_2
    2. a_1 is equivalent to b_2, and a_2 is equivalent to b_1

Sample Input1: aaba, abaa

Sample Output1: True

Sample Input2: aabb**,** abab

Sample Output2: false

Note

In the first sample you should split the first string into strings "aa" and "ba", the second one — into strings "ab" and "aa". "aa" is equivalent to "aa"; "ab" is equivalent to "ba" as "ab" = "a" + "b", "ba" = "b" + "a".

In the second sample the first string can be splitted into strings "aa" and "bb", that are equivalent only to themselves. That's why string "aabb" is equivalent only to itself and to string "bbaa".

Problem4: Kth element of two sorted arrays

Given two sorted arrays **A** and **B** of size **M** and **N** respectively and an element k. The task is to find the element that would be at the k'th position of the final sorted array.

Input:
First line consists of test cases T. First line of every test case consists of 3 integers N, M and

K, denoting M number of elements in A, N number of elements in B and kth position element. Second and Third line of every test case consists of elements of A and B respectively.

Output:
Print the element at the Kth position.

Constraints:
$1 <= T <= 200$
$1 <= N, M <= 10^6$
$1 <= A_i, B_i <= 10^6$
$1 <= K <= N+M$

Example:
Input:
1
5 4 5
2 3 6 7 9
1 4 8 10

Output:
6

Explanation:
Testcase 1: Element at 5th position after merging both arrays will be 6.

Problem5: Longest Common Prefix

Given a set of strings, find the longest common prefix .

Examples :

Input : {“geeksforgeeks”, “geeks”, “geek”, “geezer”}

Output : "gee"

Input : {"apple", "ape", "april"}

Output: "ap"

Problem6: Painting Fence

Bizon the Champion decided to paint his old fence his favorite color, orange. The fence is represented as n vertical planks, put in a row. Adjacent planks have no gap between them. The planks are numbered from the left to the right starting from one, the i-th plank has the width of 1 meter and the height of ai meters.

Bizon the Champion bought a brush in the shop, the brush's width is 1 meter. He can make vertical and horizontal strokes with the brush. During a stroke the brush's full surface must touch the fence at all the time (see the samples for the better understanding). What minimum number of strokes should Bizon the Champion do to fully paint the fence? Note that you are allowed to paint the same area of the fence multiple times.

Input

The first line contains integer n (1 ≤ n ≤ 5000) — the number of fence planks. The second line contains n space-separated integers a1, a2, ..., an (1 ≤ ai ≤ 109).

Output

Print a single integer — the minimum number of strokes needed to paint the whole fence.

Examples

input

```
5
2 2 1 2 1
```

output

```
3
```

input

```
2
2 2
```

output

```
2
```

input

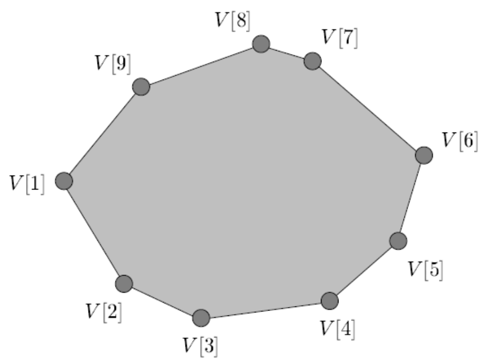
```
1
5
```

output

```
1
```

Problem7: Bounding Box of Convex Polygon

A polygon is convex if all of its internal angles are less than 180° (and none of the edges cross each other). Figure 1 shows an example.



We represent a convex polygon as an array V[1…N] where each element of the array represents a vertex of the polygon in the form of a coordinate pair (x,y).

We are told that V[1] is the vertex with the minimum x-coordinate and that the vertices V[1…N] are ordered counterclockwise, as in the figure. You may also assume that the x-coordinates of the vertices are all distinct, as are the y-coordinates of the vertices.

It's required to find the bounding box of such polygon in an efficient way? (i.e. minx, maxX, minY, maxY)

Summary:

1- All angles are less than 180
2- None of edges cross each other
3- V[1…N] contains the vertices sorted in counter clockwise direction
4- V[1] is the vertex with minimum X coordinate

Requirements:

1. Search for the vertex with maximum X coordinate in O(lg(N)) ?
2. Search for the vertex with minimum Y coordinate in O(lg(N)) ?
3. Search for the vertex with maximum Y coordinate in O(lg(N)) ?