Finding the Maximum & Minimum

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

The Max-Min Problem

- The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.
- The straightforward solution to this problem is to scan the elements searching for the **maximum**, which would require n−1 comparisons, then scan the elements one more time to search for the **minimum**.
- In this Naïve method, the maximum and minimum number can be found separately.
- The number of comparison in Naive method is **2n - 2**.

.

METHOD 1: Naïve method (Simple Linear Search)

- The straightforward solution to this problem is to scan the elements searching for the **maximum**, which would require n−1 comparisons, then scan the elements one more time to search for the **minimum.**

Algorithm Straightforward maximum and minimum

- Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately.
- To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

executes (n-1) times

This is an ***O(N)*** algorithm

Algorithm Straightforward maximum and minimum

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

- This algorithm is to scan the elements searching for the minimum, which would require n−1 comparisons, then scan the elements one more time to search for the maximum.
- Thus in total, there will be 2n−2 comparisons [best, average and worst case].

An immediate improvement

```
7        if (a[i] > max) then max := a[i];
8        if (a[i] < min) then min := a[i];
```

Replacing the line number 7 and 8 with the following

```
if (a[i] > max) then max := a[i];
else if (a[i] < min) then min := a[i];
```

StraightMaxMin modified in line 7 and 8

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           else if (a[i] < min) then min := a[i];
9       }
10  }
```

Count of comparisons performed by the algorithm

- Now the best case occurs when the elements are in increasing order. The number of element comparisons is **n-1**.
- The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is **2(n-1)**.
- The average number of element comparison is less than **2(n - 1).**
- On the average, a[i] is greater than ***max*** half the time, and so the average number of comparisons is **3n/2 - 1**

METHOD 2: Divide and Conquer Approach

- In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

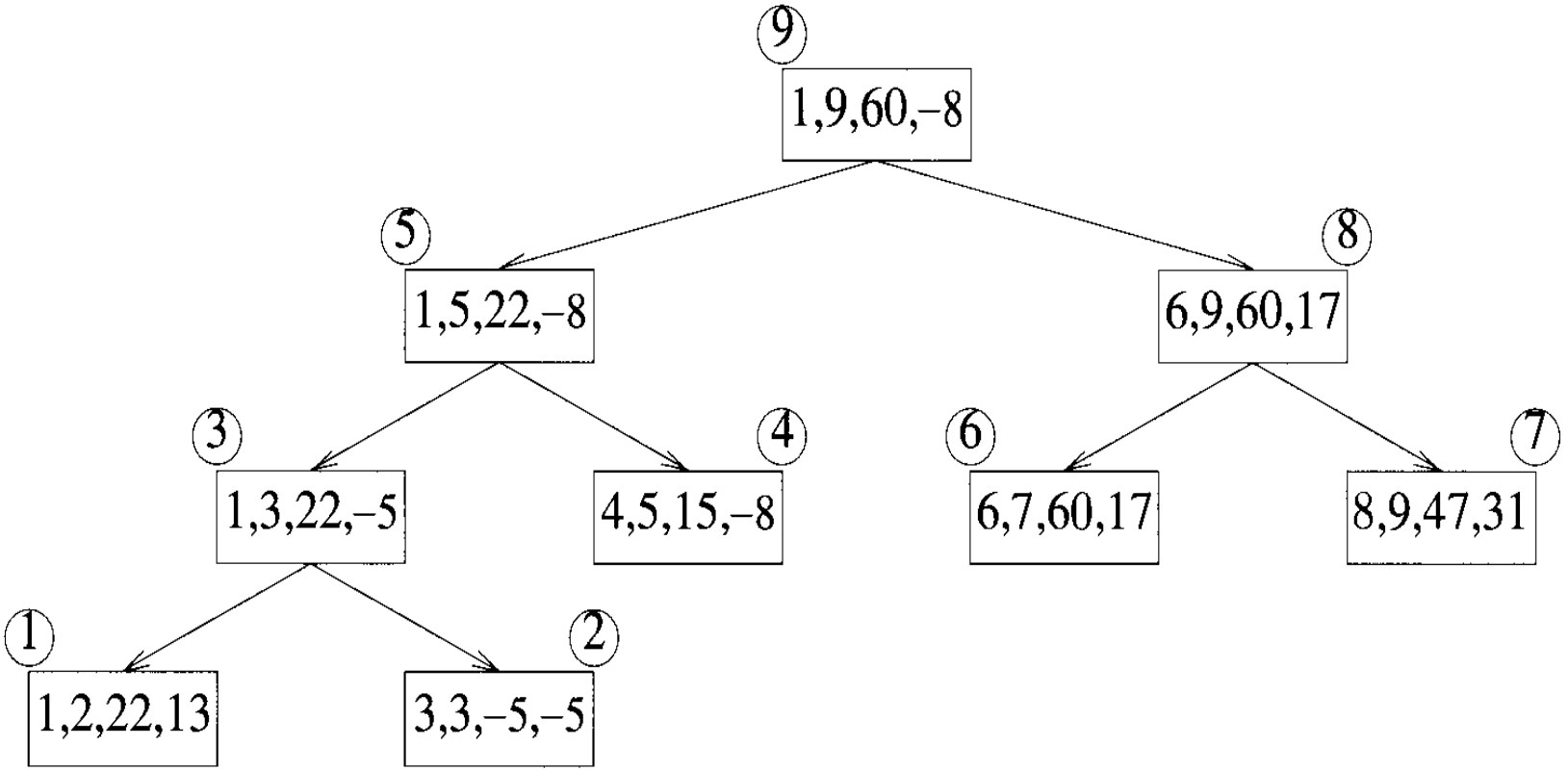
A divide-and-conquer algorithm for the Maximum & Minimum

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then  // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

The tree of recursive calls for computing MaxMin

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47



Time Complexity

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned}$$

Compared with the straight forward method (2n-2) this method saves 25% in comparisons

Time Complexity

- Let ***T(n)*** be the number of comparisons made by algorithm MaxMin

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, $\mathbf{n = 2^k}$ where **k** is height of the recursion tree.

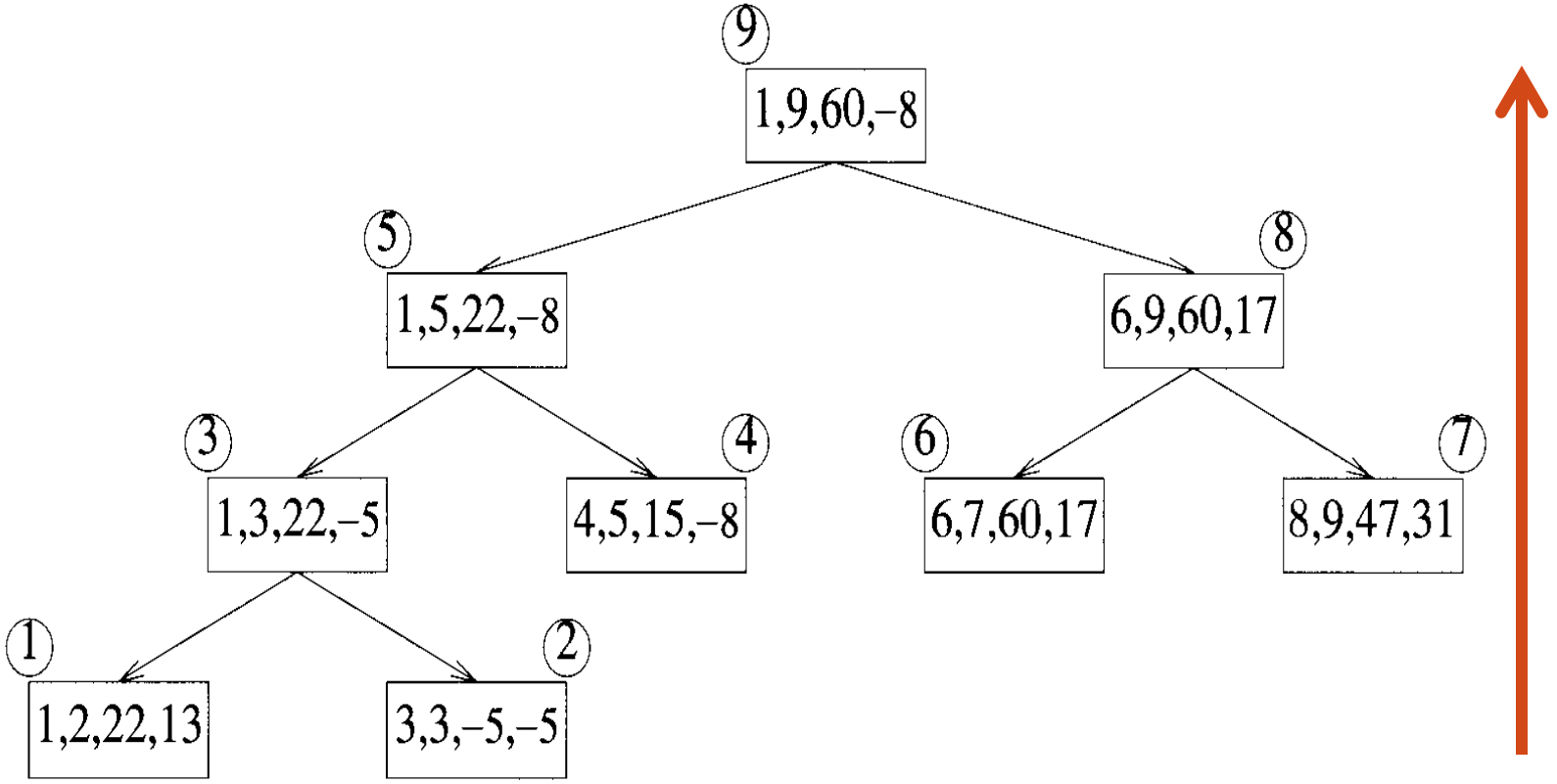
So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\tfrac{n}{4}) + 2\right) + 2..... = \frac{3n}{2} - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

The tree of recursive calls for computing MaxMin(1,16, max, min).

$n = 2^k$, with $k = 4$, for 9 elements in the array, requires a stack space O(log n)



Solving Recurrence relation

$$T(2) = 1 \qquad (\text{assume } N = 2^k)$$

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

$$T(N) = 2[2T(N/4) + 2] + 2$$

$$T(N) = 2^2 T(N/2^2) + (2^2 + 2)$$

$$T(N) = 2^r T(N/2^r) + \sum_{i=1}^{r-1} 2^i$$

Solving Recurrence relation

$$T(N) = 2^{k-1}T(N/2^{k-1}) + 2\sum_{i=0}^{k-2} 2^i$$

$$T(N) = 2^{k-1} + 2(2^{k-1} - 1)$$

$$T(N) = \frac{N}{2} + 2(\frac{N}{2} - 1)$$

$$T(N) = \frac{3N}{2} - 2$$

Note : $\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$

Note, although this is still *O(N)*, it is better than the Straight MaxMin *T(N)=2N* found earlier.

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i ≥ j − 1) { // Small(P)                    // Small(P)
7
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i ≥ j − 1) { // Small(P)
7
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

Reducing no of comparison

Assuming $n = 2^k$

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

$$\begin{aligned} C(n) &= 2C(n/2) + 3 \\ &= 4C(n/4) + 6 + 3 \\ &\vdots \\ &= 2^{k-1}C(2) + 3 \sum_0^{k-2} 2^i \\ &= 2^k + 3 * 2^{k-1} - 3 \\ &= 5n/2 - 3 \end{aligned}$$

No of comparisons in StraightMaxMin is 3(n-1)

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

In for loop compares n times

Each compares (n-1) times

Total comparison 3(n-1)

MaxMin will be slower than **StraightMaxMin** because of the overhead in stacking **i, j ,max , min** for the recursion

METHOD 3: Compare in Pairs

(Compare in Pairs)

1. If n is odd then initialize min and max as first element.
2. If n is even then initialize min and max as minimum and maximum of the first two elements respectively.
3. For rest of the elements, pick them in pairs and compare their maximum and minimum with max and min respectively.

METHOD 3: Compare in Pairs: Increment the loop by 2

Solution Steps

1. Create max and min variables.
2. Check for the size of the array
   - If odd, initialize min and max to the first element
   - If even, compare the elements and set min to the smaller value and max to the bigger value
3. Traverse the array in pairs
4. For each pair, compare the two elements and then
   - Compare the bigger element with max, update max if required.
   - Compare the smaller element with min, update min if required.
5. Return max and min.

Complexity Analysis

- Time Complexity is O(n) and Space Complexity is O(1).
- For each pair, there are a total of three comparisons, first among the elements of the pair and the other two with min and max.
- Total number of comparisons:-
- If n is odd, 3 * (n-1) / 2
- If n is even, 1 + 3*(n-2)/2 = 3n/2-2
- **Critical ideas to think!**
- Why min and max are initialized differently for even and odd sized arrays?
- Why incrementing the loop by 2 help to reduce the total number of comparison ?
- Is there any other way to solve this problem? Think.
- In which case, the number of comparisons by method 2 and 3 is equal?

Pseudo Code

```
1.  int[] findMinMax(int A[], int n)
2.   {
3.   int max, min; int i ;
4.  if ( n is odd )
5.  {
6.   max = A[0] ; min = A[0] ; i = 1
7.   }
8.  else
9.  {
10. if ( A[0] < A[1] )
11. {
12. max = A[1];  min = A[0];
13. }
14. else
15. {
16. max = A[0];  min = A[1];
17.  }
18. i = 2;
19.  }
20. while ( i < n )
21. {
22. if ( A[i] < A[i+1] )
23.  {
24. if ( A[i] < min )
25. min = A[i] ;
26. if ( A[i+1] > max )
27. max = A[i+1];
28.  }
29.  else
30. {
31. if ( A[i] > max )
32. max = A[i] ;
33. if ( A[i+1] < min )
34. min = A[i+1]; }
35.  i = i + 2; }
36. // By convention, we assume ans[0] as max and ans[1]
    as min
37. int ans[2] = {max, min}
38. return ans;
39. }
```

Comparison of different solutions

Approach	No. of Comparisons Best case	No. of Comparisons Worst Case
Linear Comparisons	2n-1	n-1
Tournament Method	3n/2 - 2	3n/2 - 1
Comparison in Pairs	3n/2 - 2	3n/2 - 2

Thanks for Your Attention!

APPENDIX
C Program to find maximum/minimum element in array

C Program to find maximum/minimum element in array

```
#include <stdio.h>
 int main()
{
    int arr[100];
    int i, max, min, size;
 /* Reads size array and elements in the array   */
    printf("Enter size of the array: ");
    scanf("%d", &size);
    printf("Enter elements in the array: ");
    for(i=0; i<size; i++)
    {     scanf("%d", &arr[i]);   }
```

C Program to find maximum/minimum element in array

```
/* Supposes the first element as maximum and minimum */
   max = arr[0];
   min = arr[0];
 /*
    * Finds maximum and minimum in all array elements.
    */
   for(i=1; i<size; i++)
   {
      /* If current element of array is greater than max */
      if(arr[i]>max)
      {
         max = arr[i];
      }
```

C Program to find maximum/minimum element in array

```
/* If current element of array is smaller than min */
      if(arr[i]<min)
      {
         min = arr[i];
      }
   }
    /*
    * Prints the maximum and minimum element
    */
   printf("Maximum element = %d\n", max);
   printf("Minimum element = %d", min);
   return 0;
}
```

Exercises

Exercises

1. How do I find the max and min value of an array in $3n/2-2$ comparisons?
2. Consider a divide and conquer approach for finding minimum and maximum elements in an array. Calculate the number of operations in a divide and conquer approach when the input is not an exact power of 2.
3. Write an algorithm to find minimum and maximum value using divide and conquer and also drive its complexity.
4. Translate algorithm MaxMin into a computationally equivalent procedure that uses no recursion.
5. Test your iterative version of MaxMin derived above against StraightMaxMin. Count all comparisions.

Exercises

6. Write an algorithm to **find minimum** and **maximum** value **using divide and conquer** and also drive its complexity.
7. Given an array A[] of size n, you need to find the maximum and minimum element present in the array. Your algorithm should make the minimum number of comparisons.
8. Find the smallest and second smallest element in the array using minimum number of comparisons
9. Find the minimum element in a sorted and rotated array
10. Find Kth largest element in an array
11. Find K largest element in an array
12. Find the middle element among three numbers
13. Find median of k sorted arrays

Finding the Majority Element

- In an election involving *n voters and k candidates, each voter casts his vote to one of the k candidates. Thus, the* outcome from voting can be represented as an *n-element sequence where an element is an integer in [1,k]. There* are then various criteria to determine the winner. One criterion could be to declare the candidate who scores *more than 50% of the votes as the winner.* Such element is known as the *majority element. Since the majority element is not always assured, an alternative* criterion is that the winner be the candidate that scores most votes. Such element is known as the *mode element*. Still another possibility is to have a rerun election limited to the candidates who score above certain threshold.

Finding the Majority Element

- **Problem:** Given a sequence of *n elements where each element is an integer in [1,k], return* the majority element (an element that appears more than *n/2 times) or zero if no majority element is found.* A sequence of size 2 has a majority only if both elements are equal. For an input of size *n=8 (and n=9), the* majority element must appear at least 5 times. For example, for the sequence 1,2,1,3,1,1,4,1 the majority element is 1.