

Design and Analysis of Algorithms

L13: MaxMin (Divide & Conquer)

Dr. Ram P Rustagi
Sem IV (2019-H1)
Dept of CSE, KSIT/KSSEM
rprustagi@ksit.edu.in

Resources

- Text book 2: Horowitz (MaxMin)
- Text book 1: Levitin (Mergesort)
- [https://www.tutorialspoint.com/
design_and_analysis_of_algorithms/
design_and_analysis_of_algorithms_max_min_prob
lem.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_max_min_problem.htm)

MaxMin

- Problem: Given a set of N elements, find the max and min of these elements
 - Assume that these elements are in an array of size N
 - Element could be complex and comparison cost is not negligible e.g. address matching, URL name etc.
- Approaches
 - Naive approach
 - Just using simple looping and iterate over all elements
 - Each iteration, compares curr min/max and update
 - Optimization of Naive approach
 - If num is less than Min, then max does not change
 - If num is greater than Max, Min does not change.
 - Divide and Conquer approach

Finding Max: Naive approach

- **Algo** Max (A [])

```
max=A[1]
```

```
for i←2 to N do
```

```
    if A[i] > Max, then
```

```
        max=A[i]
```

```
return(max)
```

- **Time complexity analysis**

- N-1 comparisons (cost contributing operation)

- N assignments.

- **Time Complexity:** $C_N=N$

Finding Min: Naive approach

- **Algo** Min (A [])

```
min=A[1]
```

```
for i←2 to N do
```

```
    if A[i] < Min, then
```

```
        min=A[i]
```

```
return(min)
```

- **Time complexity analysis**

- N-1 comparisons (cost contributing operation)

- N assignments. (non contributing)

- **Time Complexity:** $C_N=N$

MaxMin: Naive approach

- **Algo** MaxMin (A [])

```
max=A[1]
```

```
min=A[1]
```

```
for i←2 to N do
```

```
    if A[i] < min, then
```

```
        min=A[i]
```

```
    if A[i] > Max, then
```

```
        max=A[i]
```

```
return(min, max)
```

- **Time complexity analysis**

- 2 (N-1) comparisons (contributing operation)

- N+1 assignments (non contributing)

- **Time Complexity:** $C_N = 2(N-1)$

MaxMin: Optimized Naive approach

- **Algo** MaxMin (A [])

```
max=A[1]
```

```
min=A[1]
```

```
for i←2 to N do
```

```
    if A[i] < min, then
```

```
        min=A[i]
```

```
    else if A[i] > Max, then
```

```
        max=A[i]
```

```
return(min, max)
```

- **Time complexity analysis**
 - Comparisons vary for best case and worst case

MaxMin: Optimized Naive approach

- Time complexity analysis
- Best case:
 - Elements are sorted in descending order.
 - Comparison of Min always succeeds
 - Comparison of Max never occurs
 - Time complexity $C_N = (N-1)$
- Worst case:
 - Elements are sorted in ascending order.
 - Comparison of Min always fails
 - Comparison of Max invoked every time
 - Time complexity $C_N = 2(N-1)$

MaxMin

- Approach : Divide and Conquer
 - Divide the array into two halves
 - Find maximum for each half
 - Find minimum for each half
 - Compare the max values of two halves
 - Larger `max` will be desired max
 - Compare the min values of two halves
 - Smaller `min` will be the desired min

MaxMin: Divide and Conquer

- **Algo** MaxMin (i, j, A[])
 - i : lower array index on the left
 - j : higher index on the right
- ```
if (i==j) # small input array, recursion ends
 max=min=A[i]
elif (i=j-1) #small input array, recursion ends
 if A[i] < A[j]
 max = A[j]
 min = A[i]
 else
 max = A[i]
 min = A[j]
fi
fi
input array is not small, divide and conquer
```

# MaxMin: Divide and Conquer...

```
input array is not small, divide and conquer
mid = (i + j) / 2
(max1, min1) ← MaxMin(i, mid, A[])
(max2, min2) ← MaxMin(mid+1, j, A[])
if (max1 < max2) then
 max = max2
else
 max = max1
if (min1 < min2) then
 min = min1
else
 min = min2
return (max, min)
```

# MaxMin: Divide and Conquer...

- Complexity analysis
  - When input size  $N$  is 1, no comparison i.e.  $C_N=0$
  - When input size  $N$  is 2, one comparison i.e.  $C_N=1$
  - When input size  $N$  is  $>2$ ,
    - Two invocations of algo with input size  $N/2$
    - Two comparisons (one of max, one for min)
- Thus,

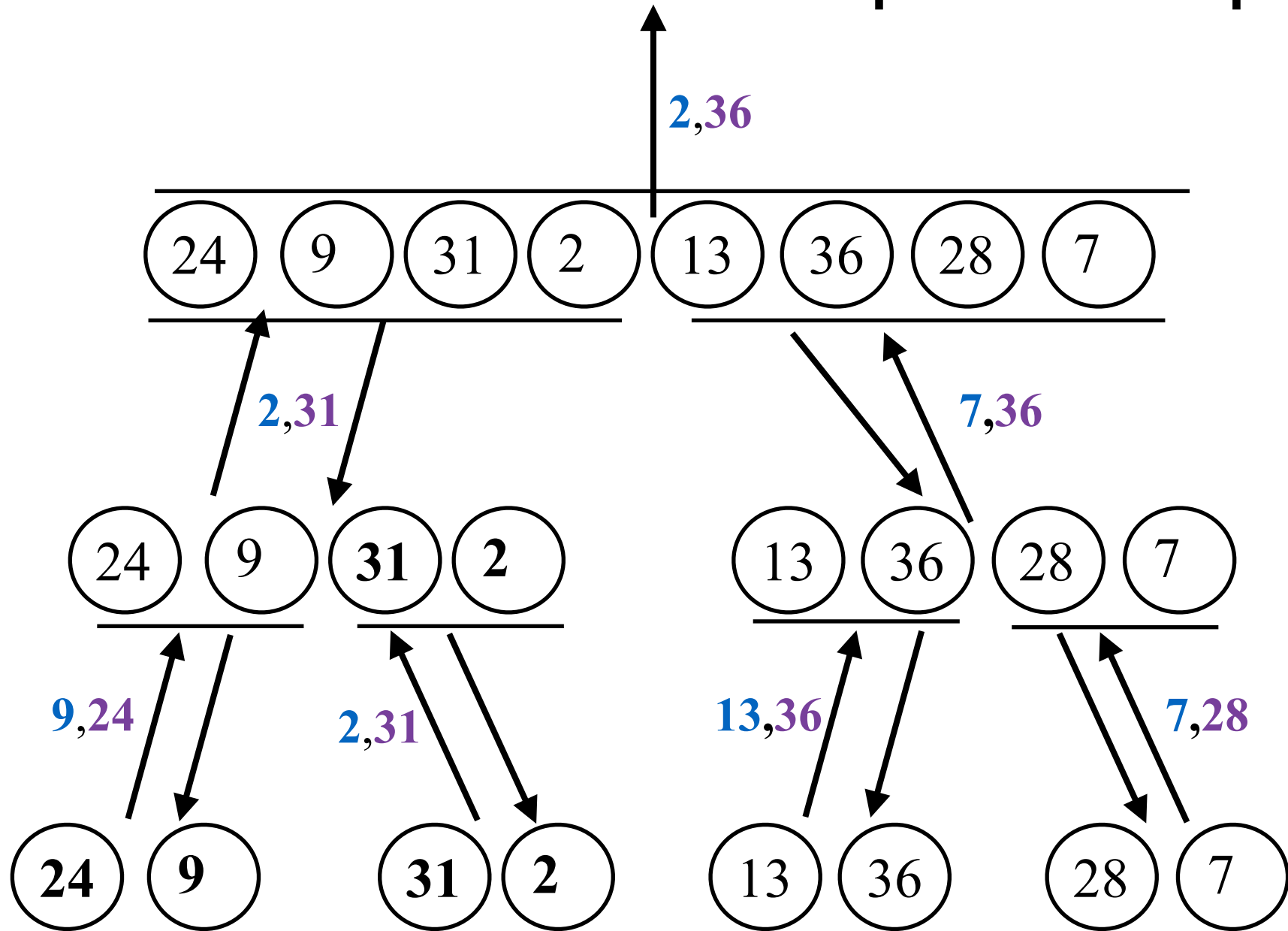
$$T(n) = \begin{cases} 0, & n = 1 \\ 1, & n = 2 \\ T(n/2) + T(n/2) + 2, & n > 2 \end{cases}$$

# MaxMin :Complexity Analysis

- Let  $n = 2^k$
- Then,

$$\begin{aligned}T(n) &= 2T(n/2) + 2 \\&= 2[2T(n/4) + 2] + 2 \\&= 2^2T(n/2^2) + 2^2 + 2 \\&= \dots \\&= 2^{k-1}T(n/2^{k-1}) + 2^{k-1} + \dots + 2^2 + 2 \\&= 2^{k-1}T(2) + 2^{k-1} + \dots + 2^2 + 2 \\&= 2^{k-1} * 1 + (2^{k-1} + \dots + 2^2 + 2 + 1) - 1 \\&= 2^{k-1} + (2^k - 1) - 1 \\&= (n/2) + n - 2 = 3n/2 - 2\end{aligned}$$

# MaxMin: Divide and Conquer Example



# Analysis: MaxMin Div & Conq

- Consider when comparison of indices  $i$  and  $j$  are of equal cost to that comparing elements.

– Then, for small input size ( $i=j$ , or  $i=j-1$ ), then

$$C_n = 2$$

– And for larger input size

$$C_n = 2C(n/2) + 3$$

$$= 2^2C(n/2^2) + 6 + 3$$

$$= 2^2C(n/2^2) + (2^1 + 2^0) 3$$

=...

$$= 2^{k-1}C(n/2^{k-1}) + 3(2^{k-2} + 2^1 + 2^0)$$

$$= 2^{k-1}C(2) + 3(2^{k-1} - 1)$$

$$= n/2 * 2 + 3(n/2 - 1)$$

$$= 5n/2 - 3$$

- For naive approach (using loop comparison):  $3(n-1)$

# Summary: MaxMin

- When elements comparison much more costly than integer comparison (loop variables)
  - Divide & Conquer is more efficient
    - Actually, it is an optimal strategy
- When elements comparisons are of similar cost, then overhead of recursion overheads (stacking of variables etc) will not yield much benefits.
- Use Divide and Conquer as a guide to better algorithm,
  - but not necessarily always.