Divide & Conquer

Smitasankhe@somaiya.edu

Algorithmic Evaluation

Strategies are evaluated along the following dimensions:

- Completeness : does it always find a solution if one exists?
- Time complexity : number of nodes generated
- Space complexity : maximum number of nodes in memory
- Optimality : does it always find a least-cost solution?

Divide-and-conquer

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size.
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to sub-problems to create a solution to the original problem.

Control Abstraction

```
Type DAndC(Problem P)
{
if small (P) return S(P);
else{
    divide P into smaller instances P1, P2, .... ,Pk, k ≥1;
    Apply DAndC to each of these sub problems;
    Return combine(DAndC(P1), DAndC(P2),....,
    DAndC(Pk));
}
}
```

General Form

$$T(n) = aT(n/b) + f(n)$$

Where,

n: size of original problem.

a: number of subproblems.

b: size of each subproblem.

f(n): time to divide and combine subproblems

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

Finding Maximum and Minimum

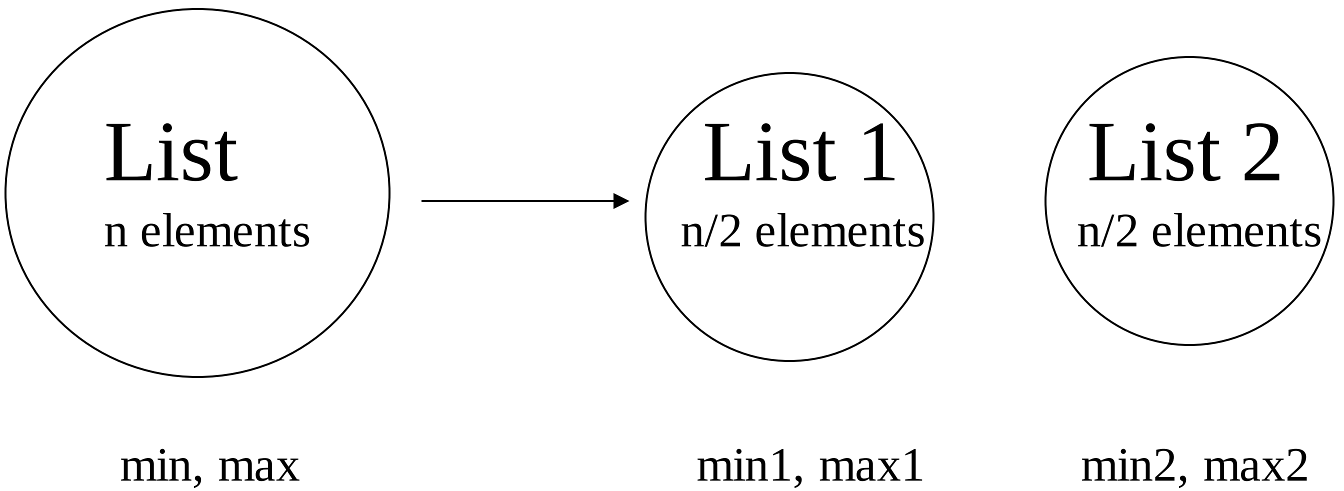
```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then max := a[i];
        if (a[i] < min) then min := a[i];
    }
}
```

1. Find the maximum and minimum

The problem: Given a list of unordered *n* elements, find max and min

The straightforward algorithm:

max ← min ← A (1);
for i ← 2 to n do
  if A (i) > max, max ← A (i);
  if A (i) < min, min ← A (i);



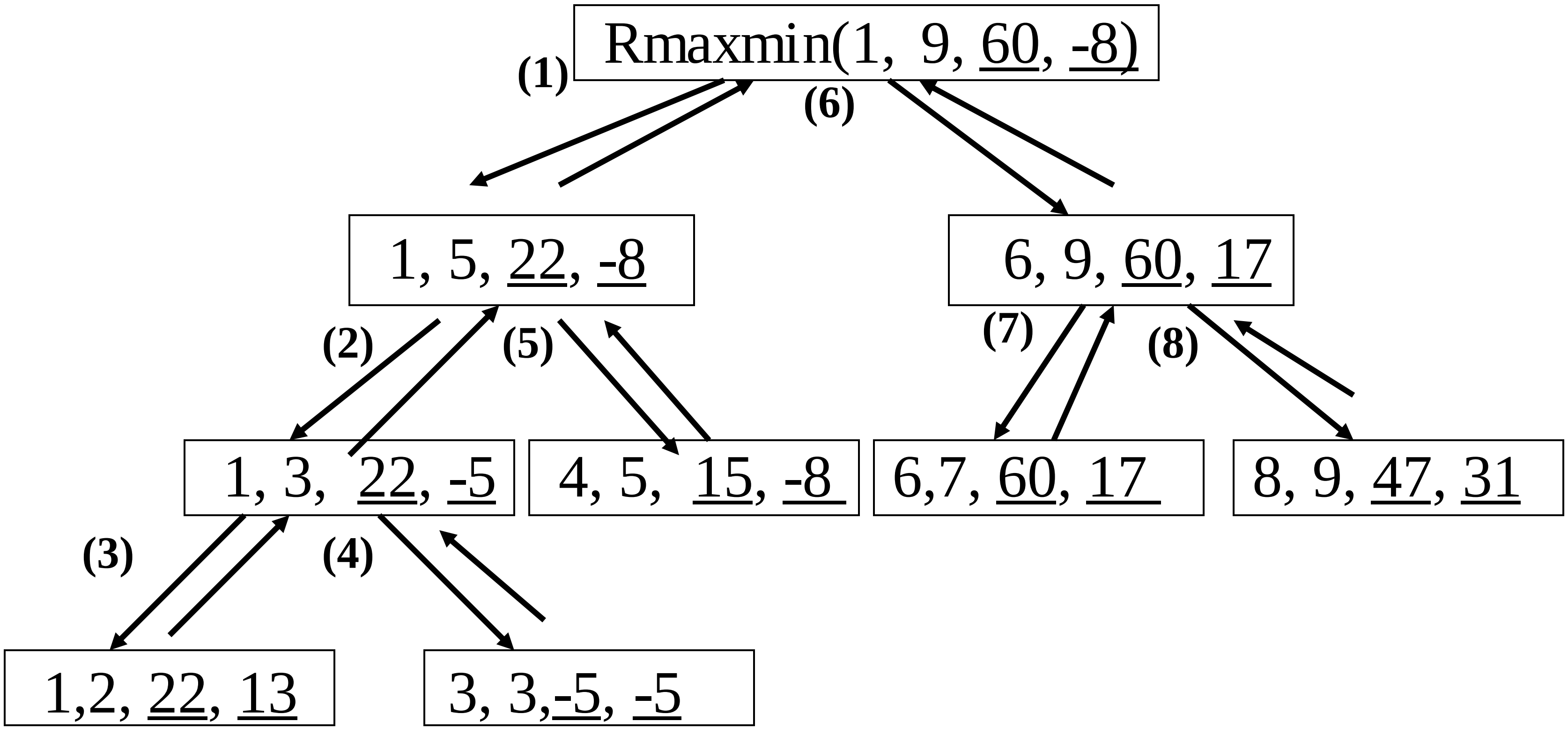
min = MIN (min1, min2)
max = MAX (max1, max2)

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i : j], respectively.
{
    if (i = j) then max := min := a[i]; // Small(P)
    else if (i = j − 1) then  // Another case of Small(P)
        {
            if (a[i] < a[j]) then
            {
                max := a[j]; min := a[i];
            }
            else
            {
                max := a[i]; min := a[j];
            }
        }
        else
        {   // If P is not small, divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(i + j)/2⌋;
            // Solve the subproblems.
                MaxMin(i, mid, max, min);
                MaxMin(mid + 1, j, max1, min1);
            // Combine the solutions.
                if (max < max1) then max := max1;
                if (min > min1) then min := min1;
        }
}
```

Example: find max and min in the array:

22, 13, -5, -8, 15, 60, 17, 31, 47 (n = 9)

Index:	1	2	3	4	5	6	7	8	9
Array:	22	13	-5	-8	15	60	17	31	47



Analysis: $\quad$ *For algorithm containing recursive calls, we can use recurrence relation to find its complexity*

T(n) - # of comparisons needed for Rmaxmin

Recurrence relation:

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

Assume $n = 2^k$ for some integer k

$$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (2^{k-1} + 2^{k-2} + \cdots + 2^1)$$

$$= 2^{k-1} \cdot T(2) + (2^k - 2) = \frac{n}{2} \cdot 1 + n - 2$$

$$= 1.5n - 2$$

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

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