Divide-and-conquer (Module 2)

By
Smita Sankhe
Email id: smitasankhe@somaiya.edu

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));
}
}
```

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

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

- Time complexity to solve "Divide & Conquer" problem is given by recurrence relations.
- Recurrence relation is derived from algorithm and solved to calculate complexity.
- The general recurrence relation for divide and conquer is given as follows:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where, T(n/b) : time required to solve each subproblem

f(n) : time required to combine the solutions of all subproblems

BINARY SEARCH

- There are two approaches:
  1. Iterative or Non-recursive
  2. Recursive
- There is a linear Array 'a' of size 'n'.
- Binary Search is one of the fastest searching algorithm.
- Binary Search can only be applied on "Sorted Arrays"- either ascending or descending order.
- We compare "key" with item in the middle position. If they are equal, search ends successfully.
- Otherwise,

  if key is less than element present in the middle position,

  then apply binary search on lower half,

  else apply BINARY SEARCH on upper half of the array.
- Same process is applied to remaining half until match is found or there are no more elements left

BINARY SEARCH

Iterative Approach:

```
Algorithm IBinaryS(arr[ ], start, end, key){
        int mid;
        while(start<=end){
                mid = (start + end)/2;
                if (arr[mid] == key)
                        return 1;
                if (arr[mid]<key)
                        start = mid+1;
                else
                        end = mid-1;
        }
        return 0;
}
```

Recursive Approach:

```
Algorithm RBinaryS(arr[ ], start, end, key){
        int mid;
        if (start > end) { return 0; }
else
        mid = (start + end)/2;
        if (key == arr[mid])
        return (mid);
        else
        if (key < arr[mid]){
        RBinaryS(arr[],key, start, mid-1)
        else
        RBinaryS(arr[],key, mid+1, end)
        }
}
```

Finding Maximum and Minimum

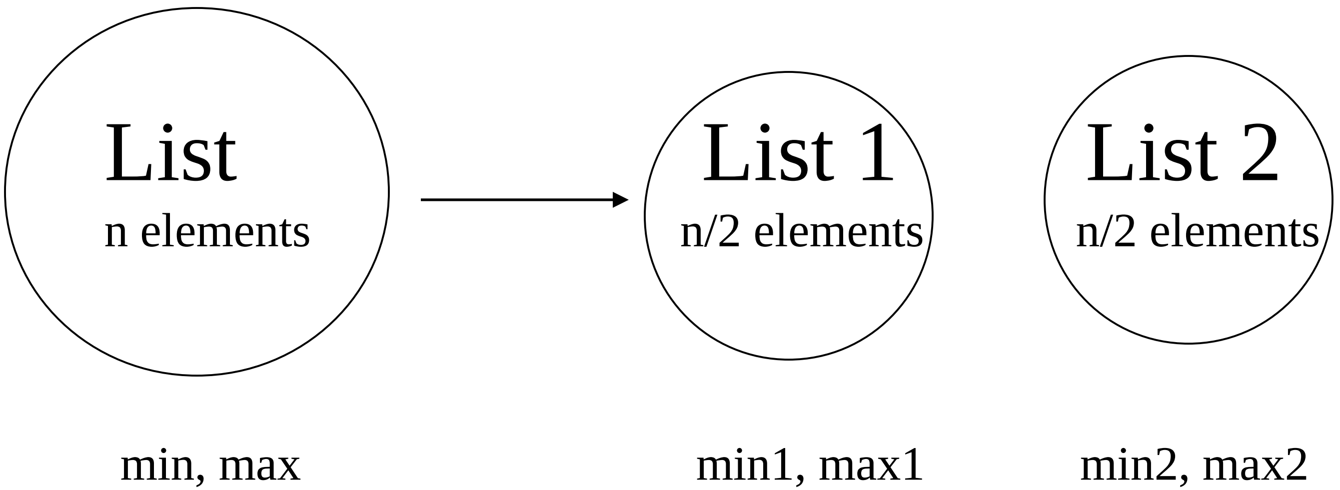
```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then max := a[i];
        if (a[i] < min) then min := a[i];
    }
}
```

1. Find the maximum and minimum

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

The straightforward algorithm:

$$\max \leftarrow \min \leftarrow A(1);$$
for $i \leftarrow 2$ to n do
$$\left\{ \begin{array}{l} \text{if } A(i) > \max,\ \max \leftarrow A(i); \\ \text{if } A(i) < \min,\ \min \leftarrow A(i); \end{array} \right.$$



min = MIN (min1, min2)

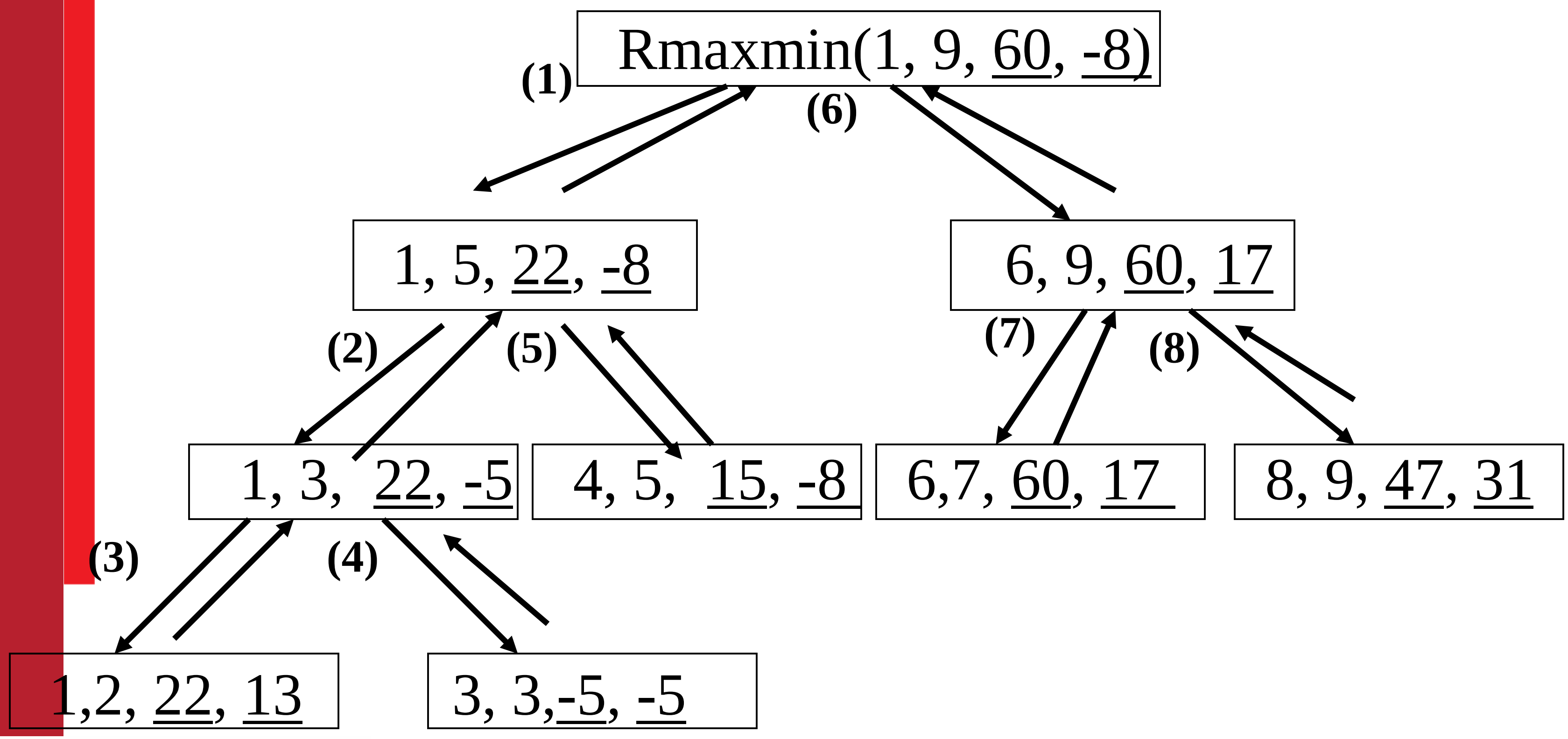
max = MAX (max1, max2)

```
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: *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

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

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

$$= 1.5n - 2$$

Divide and Conquer

- An important general technique for designing algorithms:
  - divide problem into subproblems
  - recursively solve subproblems
  - combine solutions to subproblems to get solution to original problem
- Use recurrences to analyze the running time of such algorithms

Additional D&C Algorithms

- binary search
  - divide sequence into two halves by comparing search key to midpoint
  - recursively search in one of the two halves
  - combine step is empty
- quicksort
  - divide sequence into two parts by comparing pivot to each key
  - recursively sort the two parts
  - combine step is empty

Additional D&C applications

- computational geometry
  - finding closest pair of points
  - finding convex hull
- mathematical calculations
  - converting binary to decimal
  - integer multiplication
  - matrix multiplication
  - matrix inversion
  - Fast Fourier Transform

Strassen’s Matrix Multiplication

Matrix Multiplication

- Consider two *n* by *n* matrices *A* and *B*
- Definition of *AxB* is *n* by *n* matrix *C* whose (*i*,*j*)-th entry is computed like this:
  - consider row *i* of *A* and column *j* of *B*
  - multiply together the first entries of the rown and column, the second entries, etc.
  - then add up all the products
- Number of scalar operations (multiplies and adds) in straightforward algorithm is $O(n^3)$.
- Can we do it faster?

Divide-and-Conquer

$$A \quad \times \quad B \quad = \quad C$$

$$\begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \end{bmatrix}$$

- Divide matrices A and B into four submatrices each
- We have 8 smaller matrix multiplications and 4 additions. Is it faster?

Divide-and-Conquer

Let us investigate this recursive version of the matrix multiplication.

Since we divide A, B and C into 4 submatrices each, we can compute the resulting matrix C by

- 8 matrix multiplications on the submatrices of A and B,
- plus $\Theta(n^2)$ scalar operations

Divide-and-Conquer

- Running time of recursive version of straightfoward algorithm is
  - $T(n) = 8T(n/2) + \Theta(n^2)$
  - $T(2) = \Theta(1)$

  where $T(n)$ is running time on an n x n matrix
- Master theorem gives us:

  $$T(n) = \Theta(n^3)$$
- Can we do fewer recursive calls (fewer multiplications of the $n/2$ x $n/2$ submatrices)?

Strassen's Matrix Multiplication

$$A \quad \times \quad B \quad = \quad C$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Strassen's Matrix Multiplication

- Strassen found a way to get all the required information with only 7 matrix multiplications, instead of 8.

- Recurrence for new algorithm is
  - $T(\mathrm{n}) = 7T(n/2) + \Theta(n^2)$

Solving the Recurrence Relation

Applying the Master Theorem to

$T(n) = a\,T(n/b) + f(n)$

with a=7, b=2, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case a) applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.