Advanced Data Structures and Algorithms

Divide and Conquer

Dr G.Kalyani

Department of Information Technology

Velagapudi Ramakrishna Siddhartha Engineering College

Topics

- General method
- Binary Search
- Merge Sort
- Quick Sort
- Finding the Maximum Minimum
- Strassen's Matrix Multiplication

General Method of Divide-and-Conquer

- The Divide and Conquer Technique splits n inputs into k subsets, $1 < k \le n$, yielding k sub problems.
- These sub problems will be solved and then combined by using a separate method to get a solution to a whole problem.
- If the sub problems are large, then the Divide and Conquer Technique will be reapplied.
- Often sub problems resulting from a Divide and Conquer Technique are of the same type as the original problem.

Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Solve** the sub-problems
  - Solve the sub-problems <u>recursively</u>
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

General Method of Divide-and-Conquer

```
Algorithm DandC(p)
{
    if Small(p) then return s(p);
    else
    {
        Divide p into smaller instances p1,p2,.......,pk, k>1;
        DandC(p1), DandC(p2),..., DandC(pk);
        return Combine( );
    }
}
```

General Method of Divide-and-Conquer

If the size of p is n and the sizes of the k sub problems are $n_1, n_2, \ldots, n_k$, then the computing time of DandC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1)+T(n_2)+\ldots\ldots+T(n_k)+f(n) & \text{Otherwise} \end{cases}$$

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

- Where T(n) is the time for DandC on any input of size n and
- g(n) is the time to compute the answer directly for small inputs.
- The function f(n) is the time for dividing p and combining the solutions to sub problems.

Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication

Binary Search

- Consider the problem of determining whether a given element x is present in the list.

- If x is present, we are to determine a value j such that a[j] = x.

- If x is not in the list, then j is to be set to -1.

Steps in Binary Search

- Calculate the middle position
- Compare x with the middle element.
- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half sub array after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

Binary Search

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

Search for the following values of x:151, -14, and 9

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Recursive Algorithm for Binary Search

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then   // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                  return BinSrch(a, i, mid − 1, x);
             else return BinSrch(a, mid + 1, l, x);
    }
}
```

Time Complexity of Binary Search

- **Best Case**
  - Array contains single element (or) The search element is exactly in the middle position
- **Worst Case**
  - The search element is not present in the array
- **Average Case**
  - The search element is present but not in the middle position

Time Complexity of Binary Search

- **Best Case:**
  - O(1)

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                return BinSrch(a, i, mid − 1, x);
            else return BinSrch(a, mid + 1, l, x);
    }
}
```

Time Complexity of Binary Search

- **Worst Case and Average Case:**

$$T(n)=T(n/2)+C$$
$$=[T(n/4)+C]+C$$
$$=[T(n/8)+C]+2.C$$
$$\ldots\ldots$$
$$=T(n/2^i)+ i * C$$

Assume $n=2^i \rightarrow \log n=\log 2^i \rightarrow i=\log n$

$$T(n)=T(1)+C * \log n$$
$$= O(1) +(C * \log n)$$

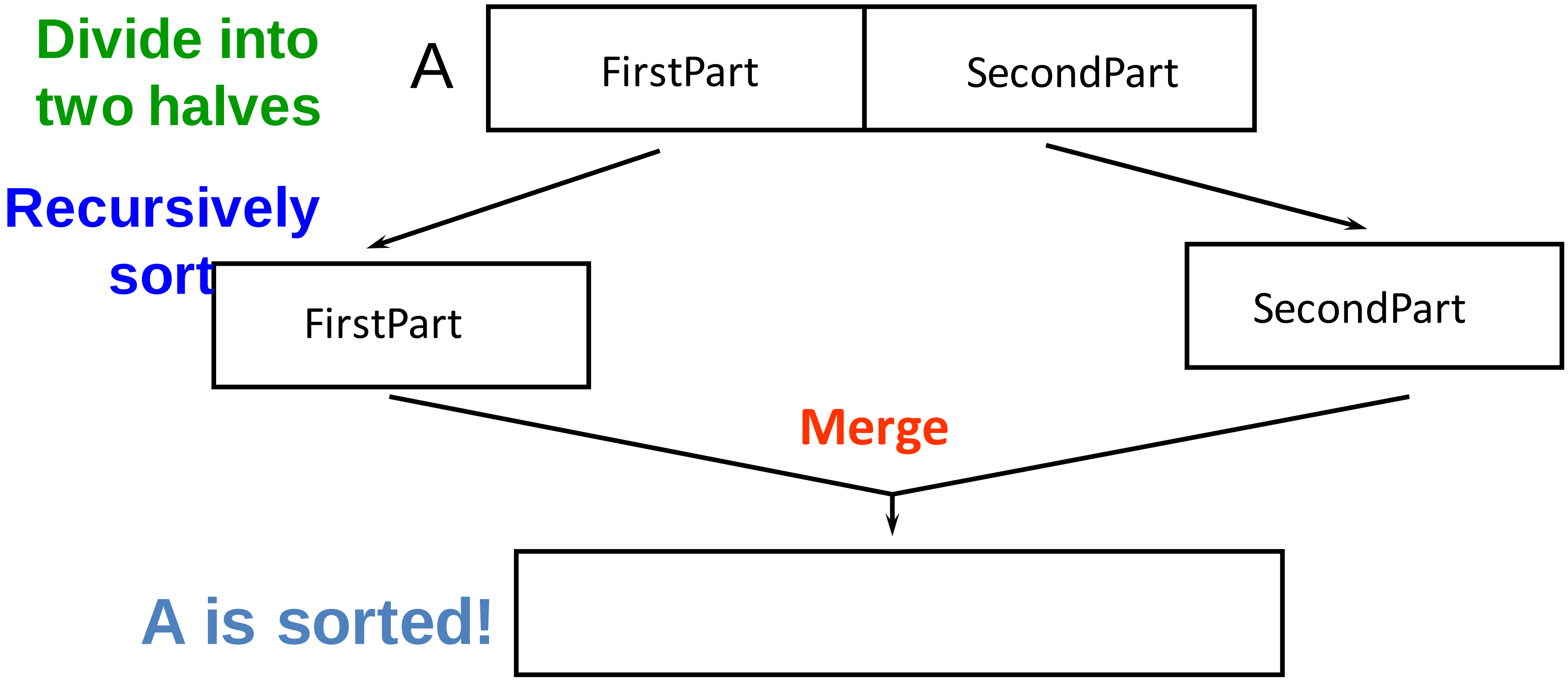
Hence $O(\log n)$

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                return BinSrch(a, i, mid − 1, x);
            else return BinSrch(a, mid + 1, l, x);
    }
}
```

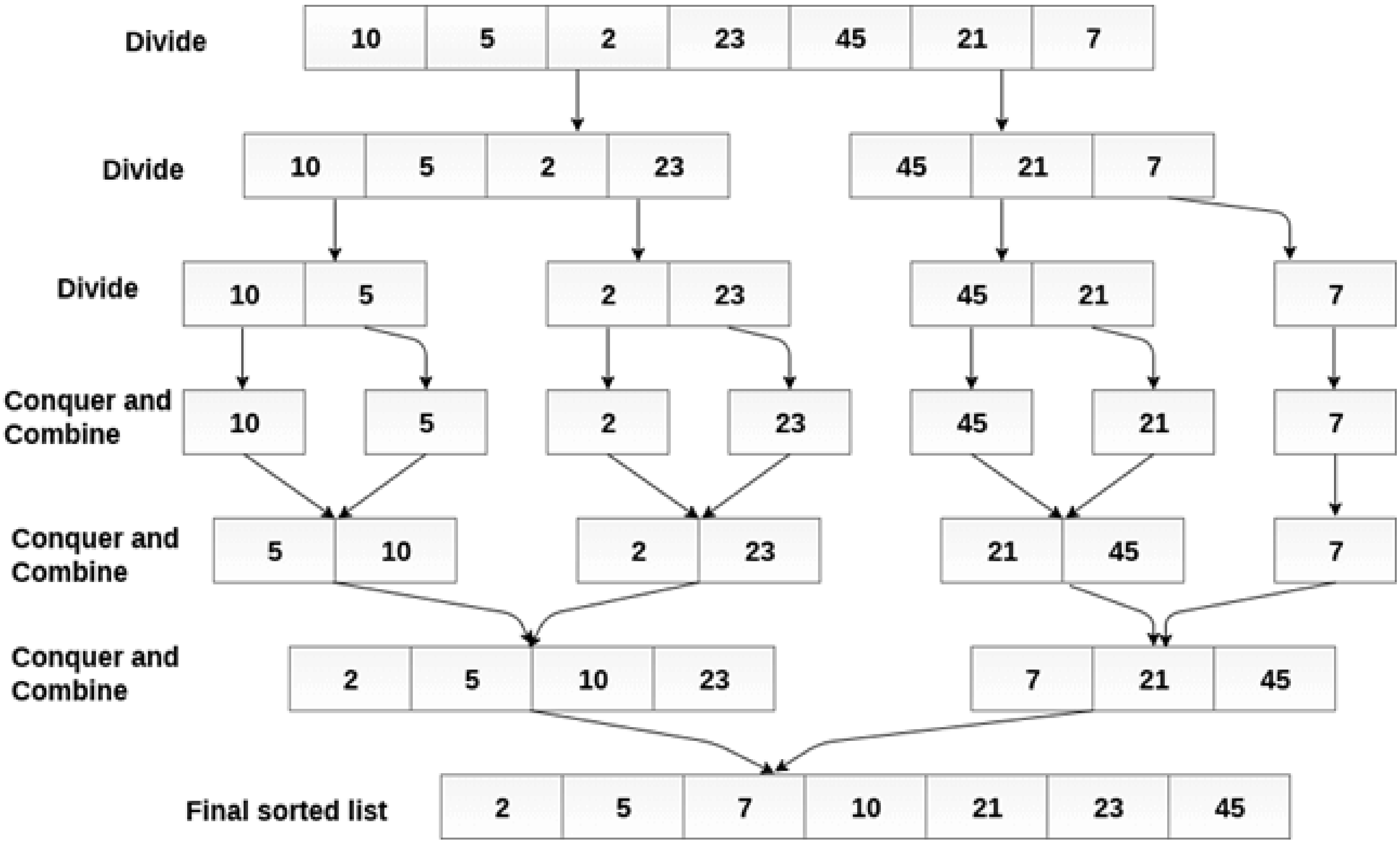
Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication

Merge Sort: Idea



Example for Merge Sort

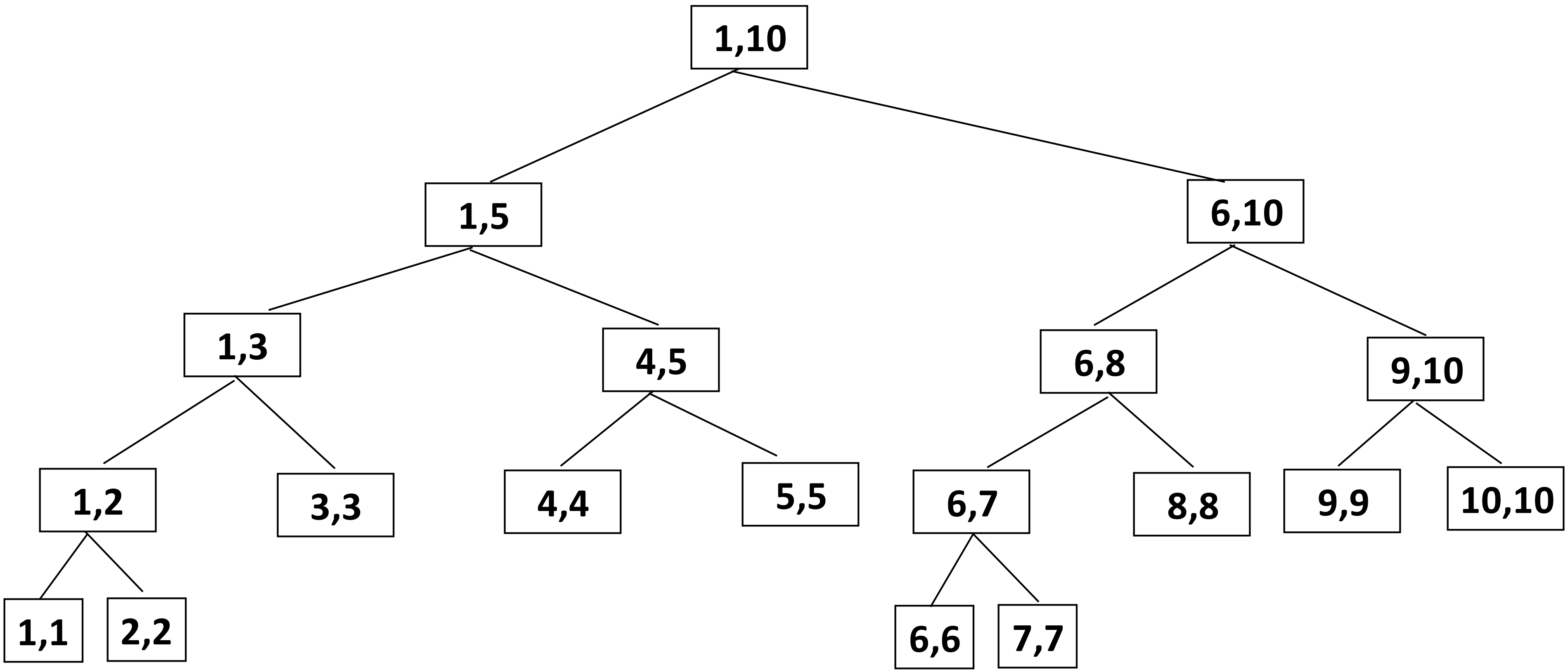


Example for Merge Sort

- Ex 2:-

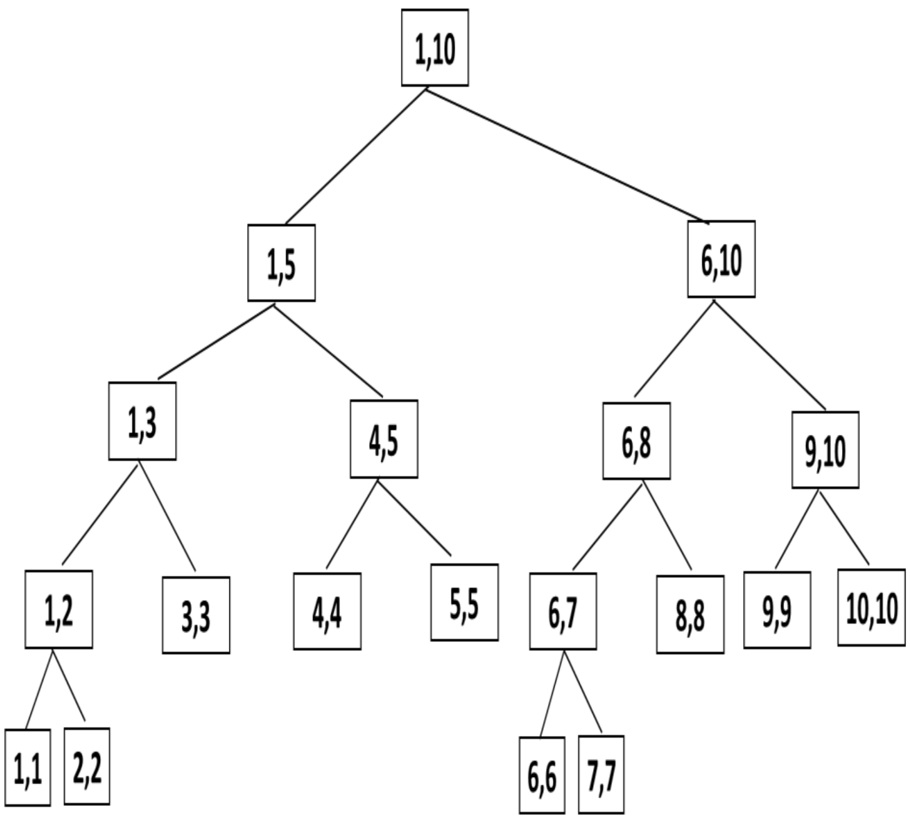
  179, 254, 285, 310, 351, 423, 450, 520, 652,861

Example for Merge Sort

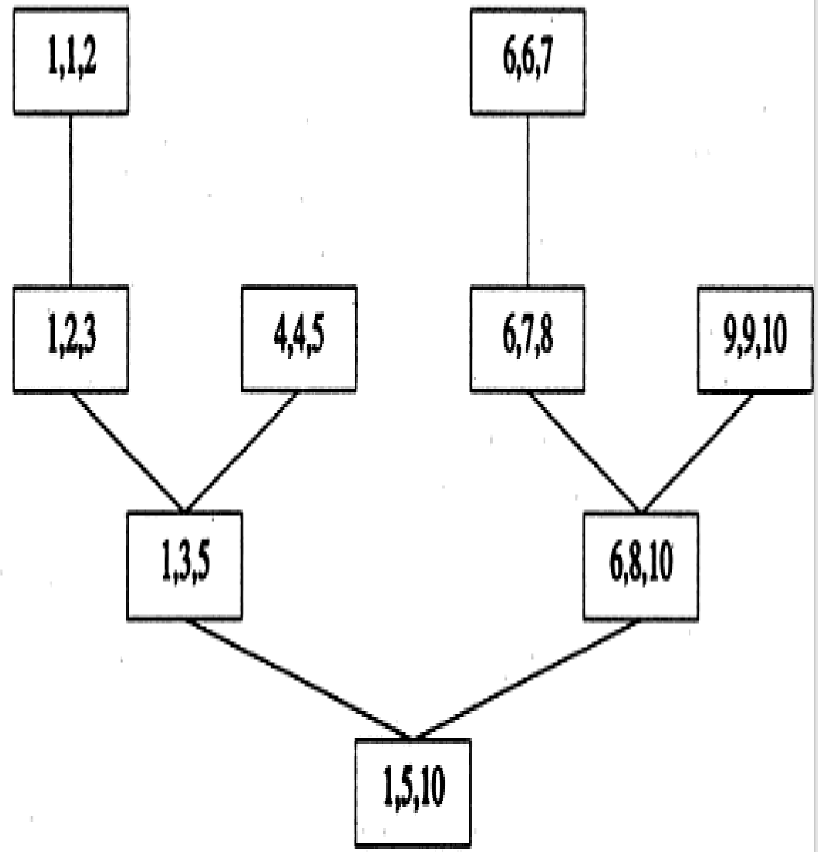


Merge-Sort: Merge Example

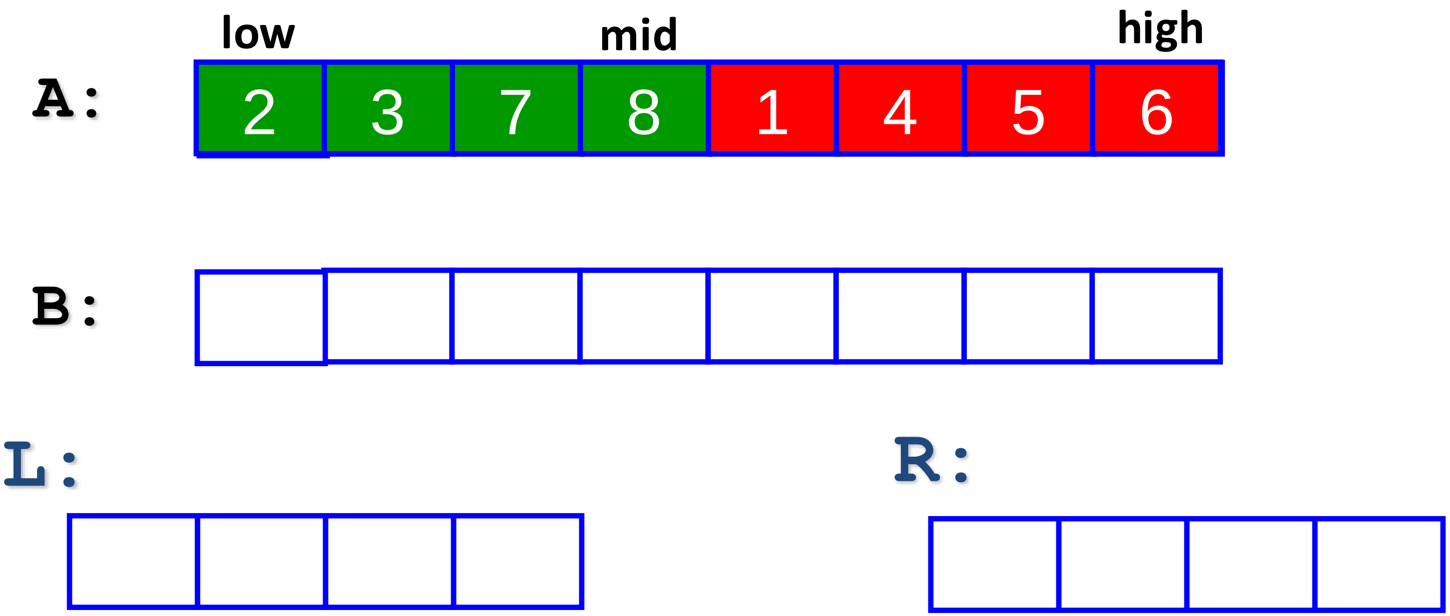
Tree calls for Recursive Merge Sort



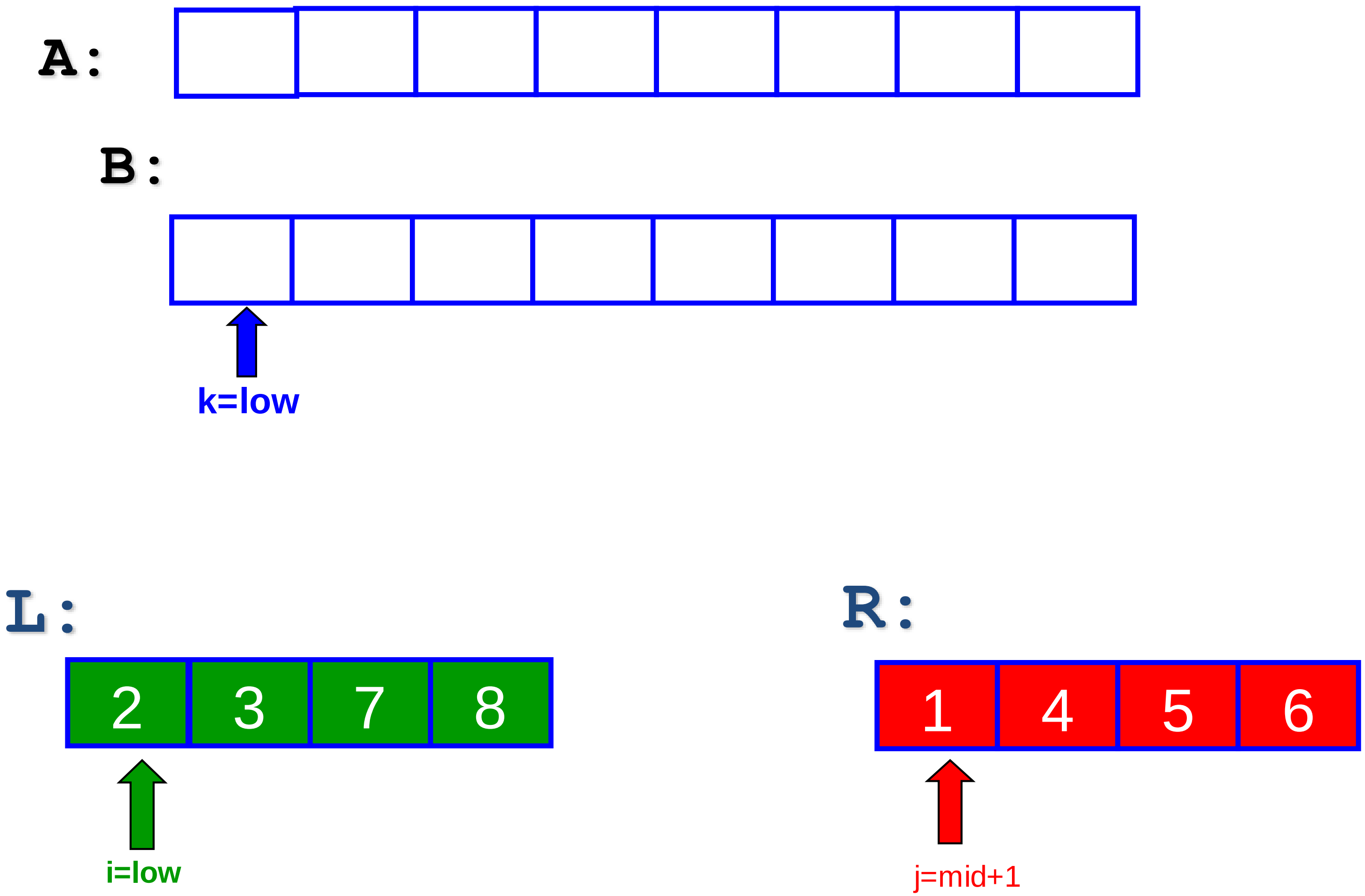
Tree calls for Merge Operation



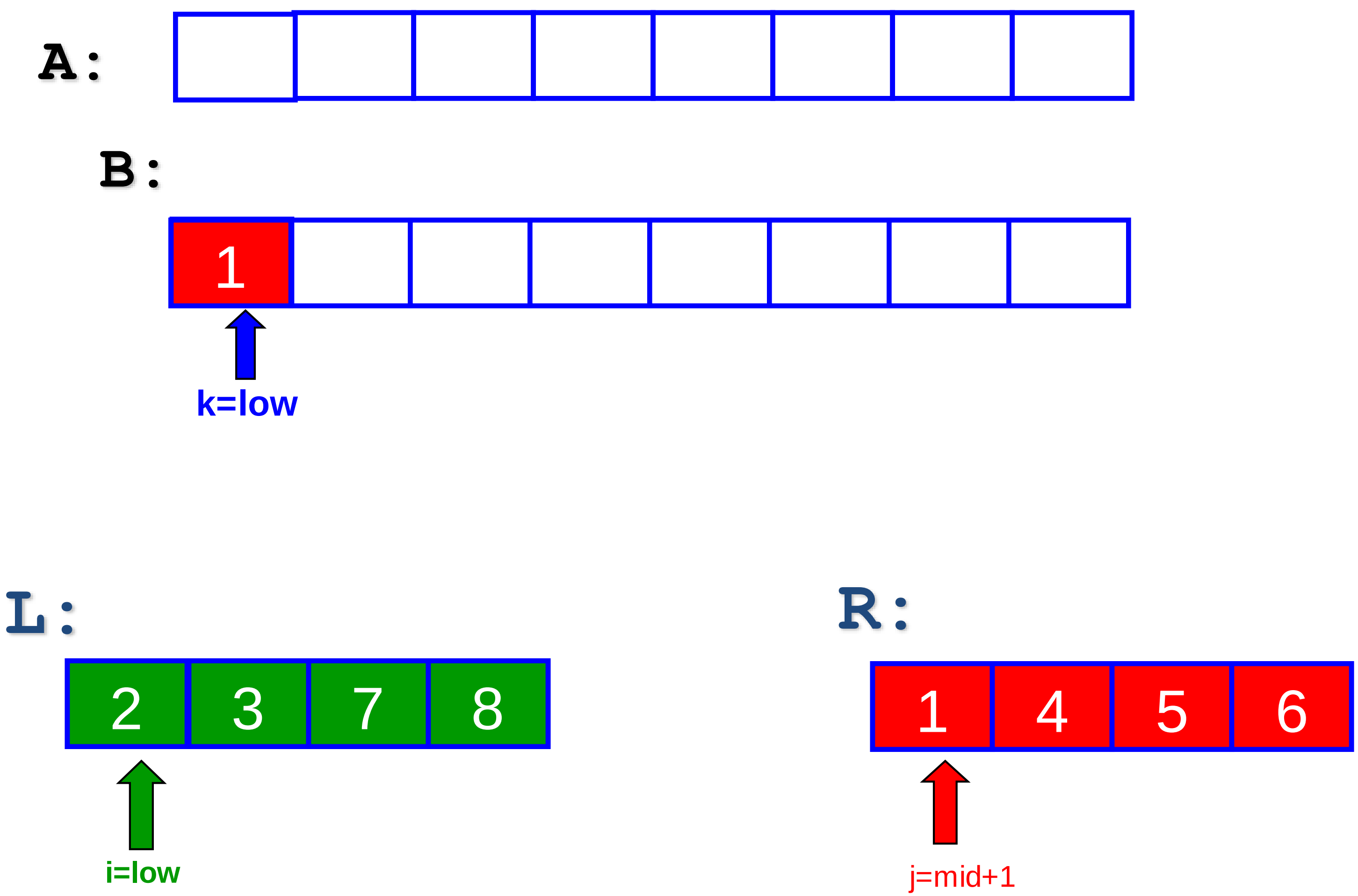
Merge-Sort: Merge Example



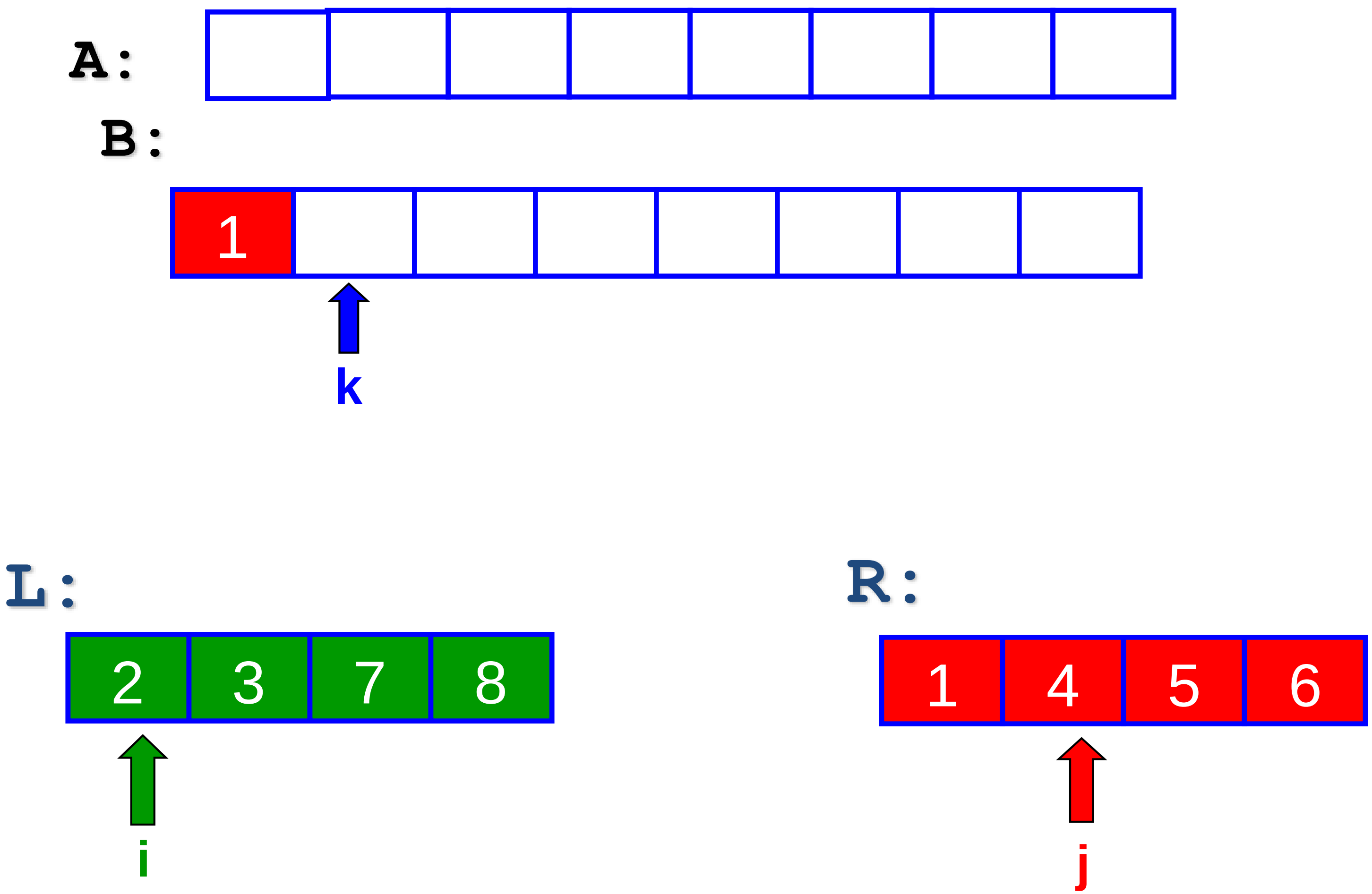
Merge-Sort: Merge Example



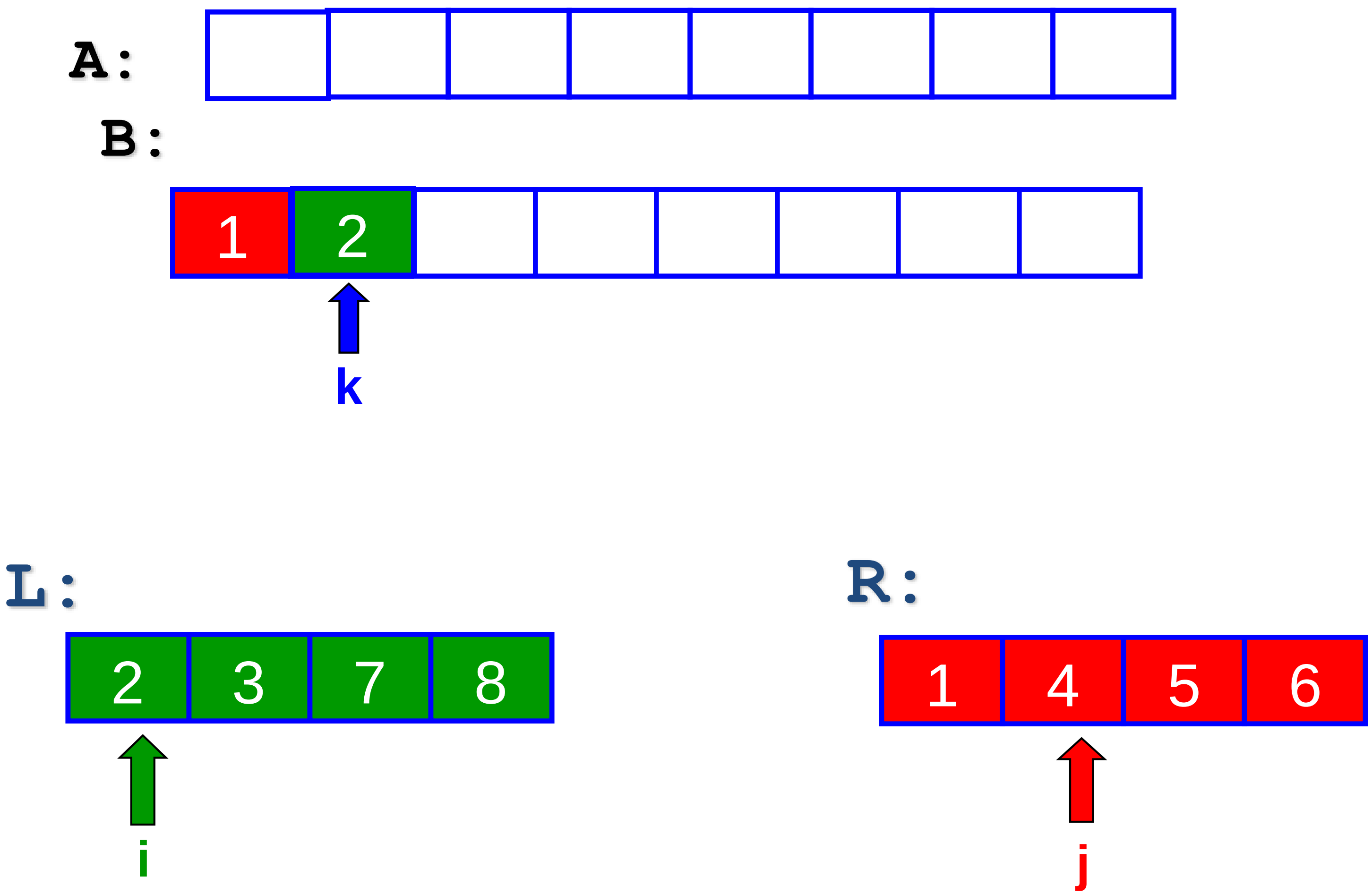
Merge-Sort: Merge Example



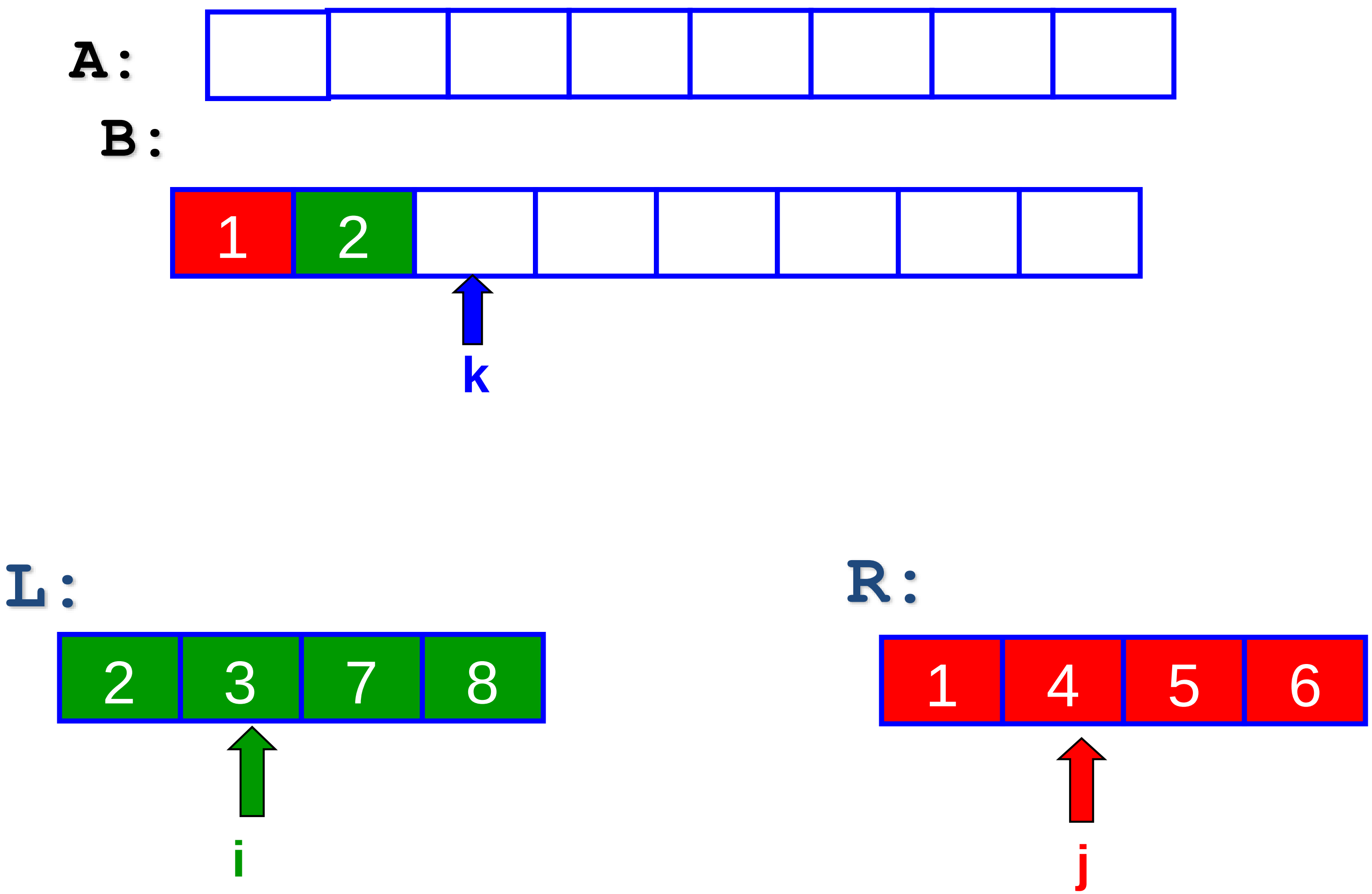
Merge-Sort: Merge Example



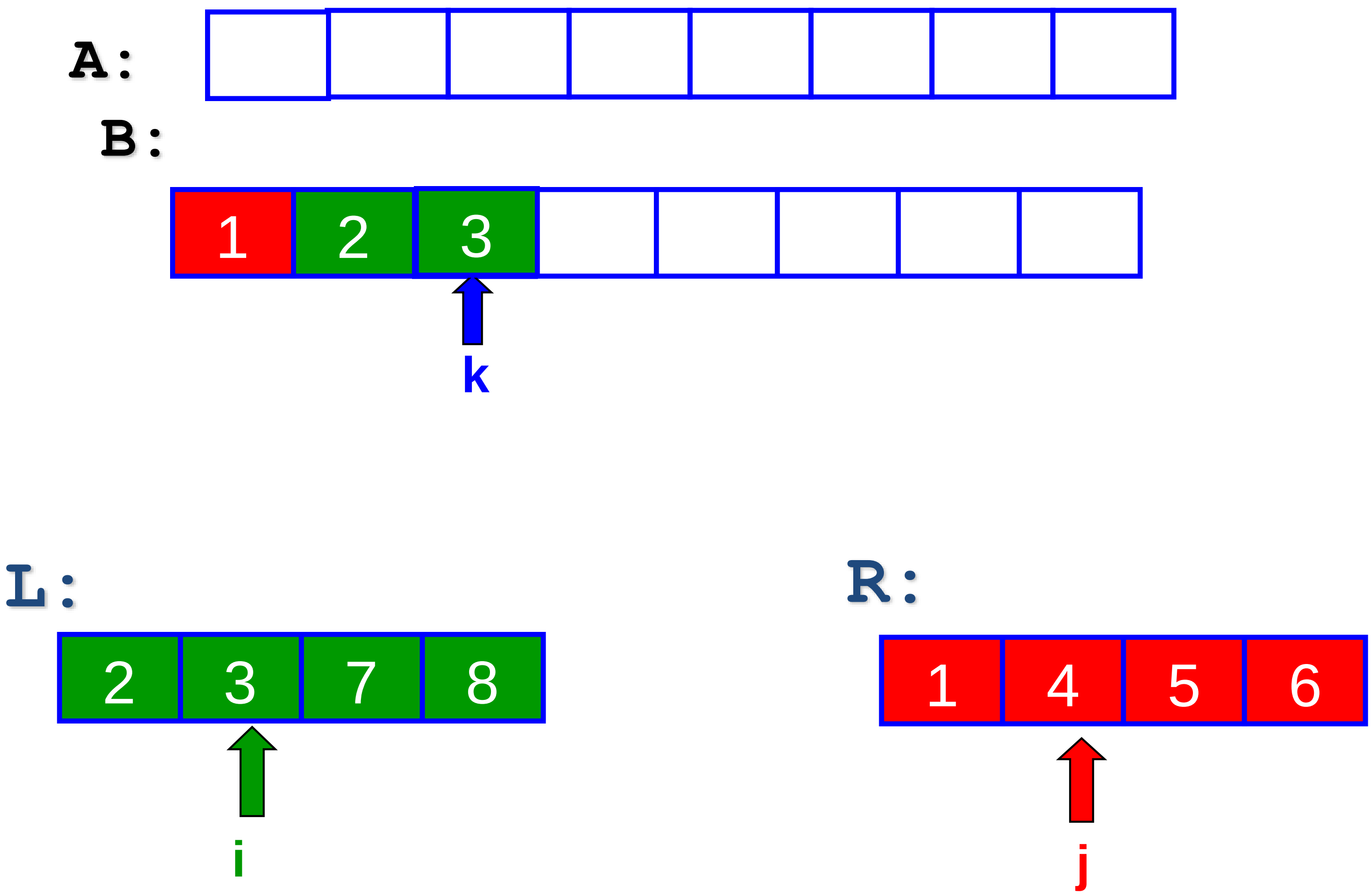
Merge-Sort: Merge Example



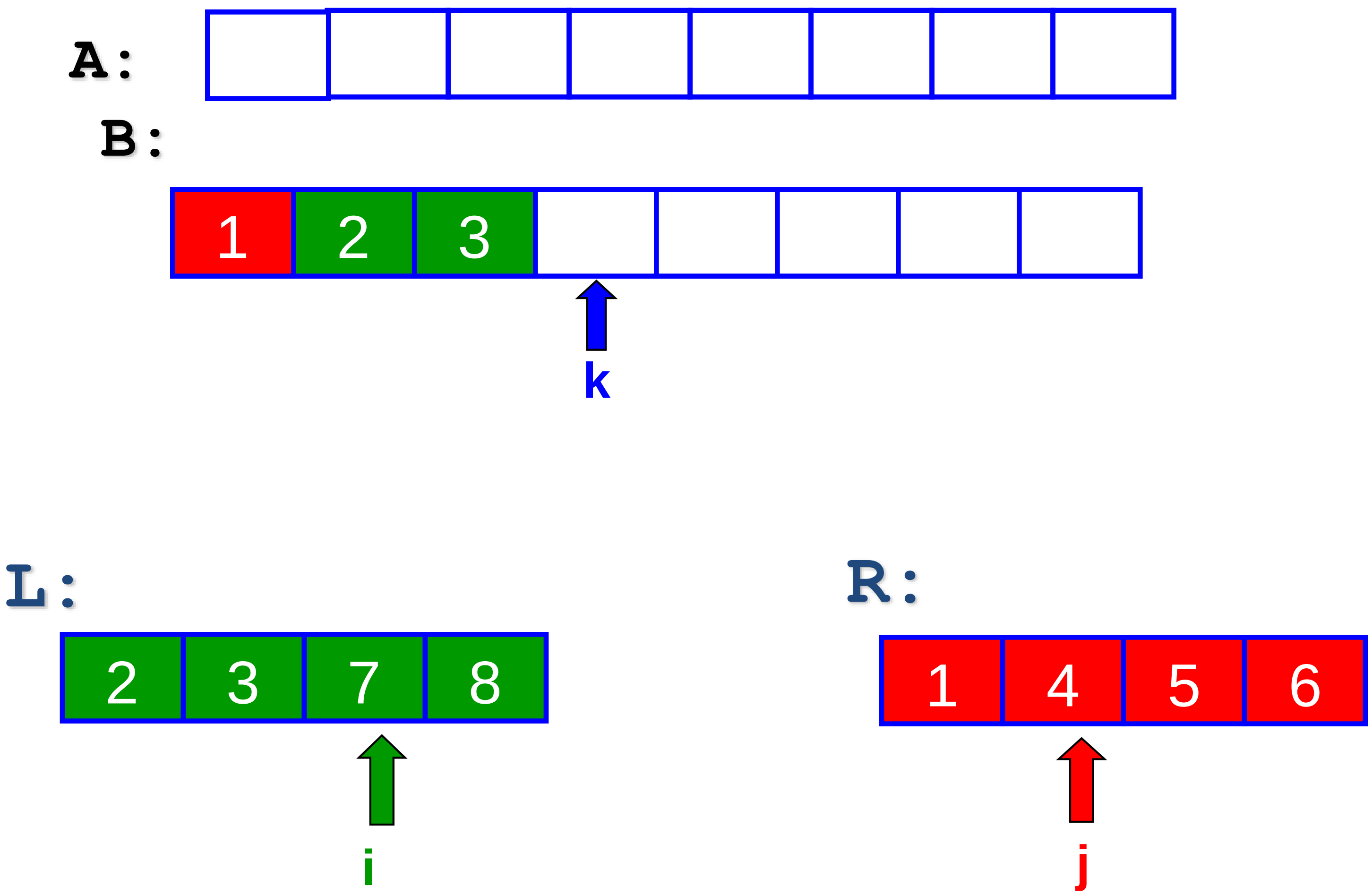
Merge-Sort: Merge Example



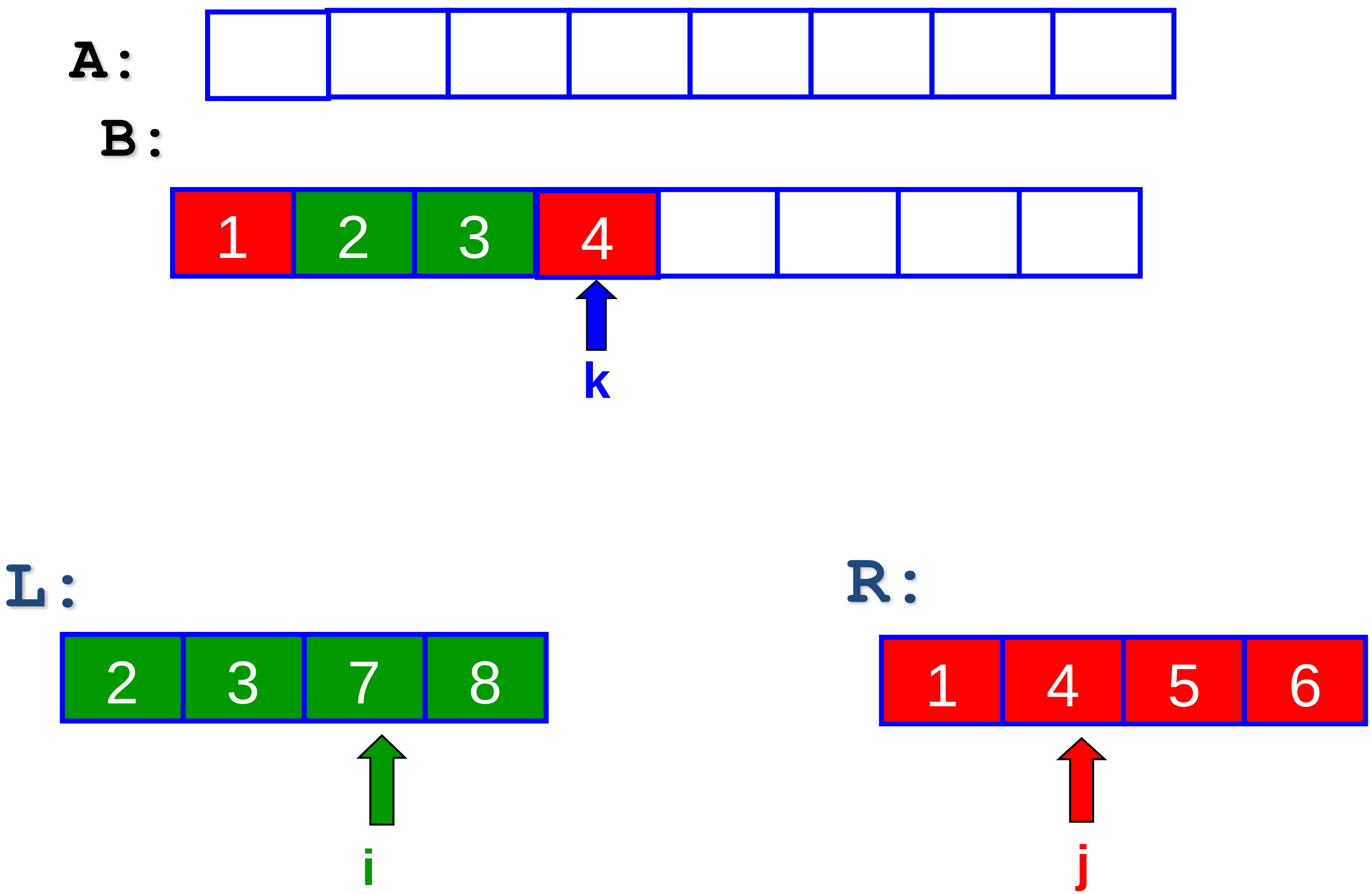
Merge-Sort: Merge Example



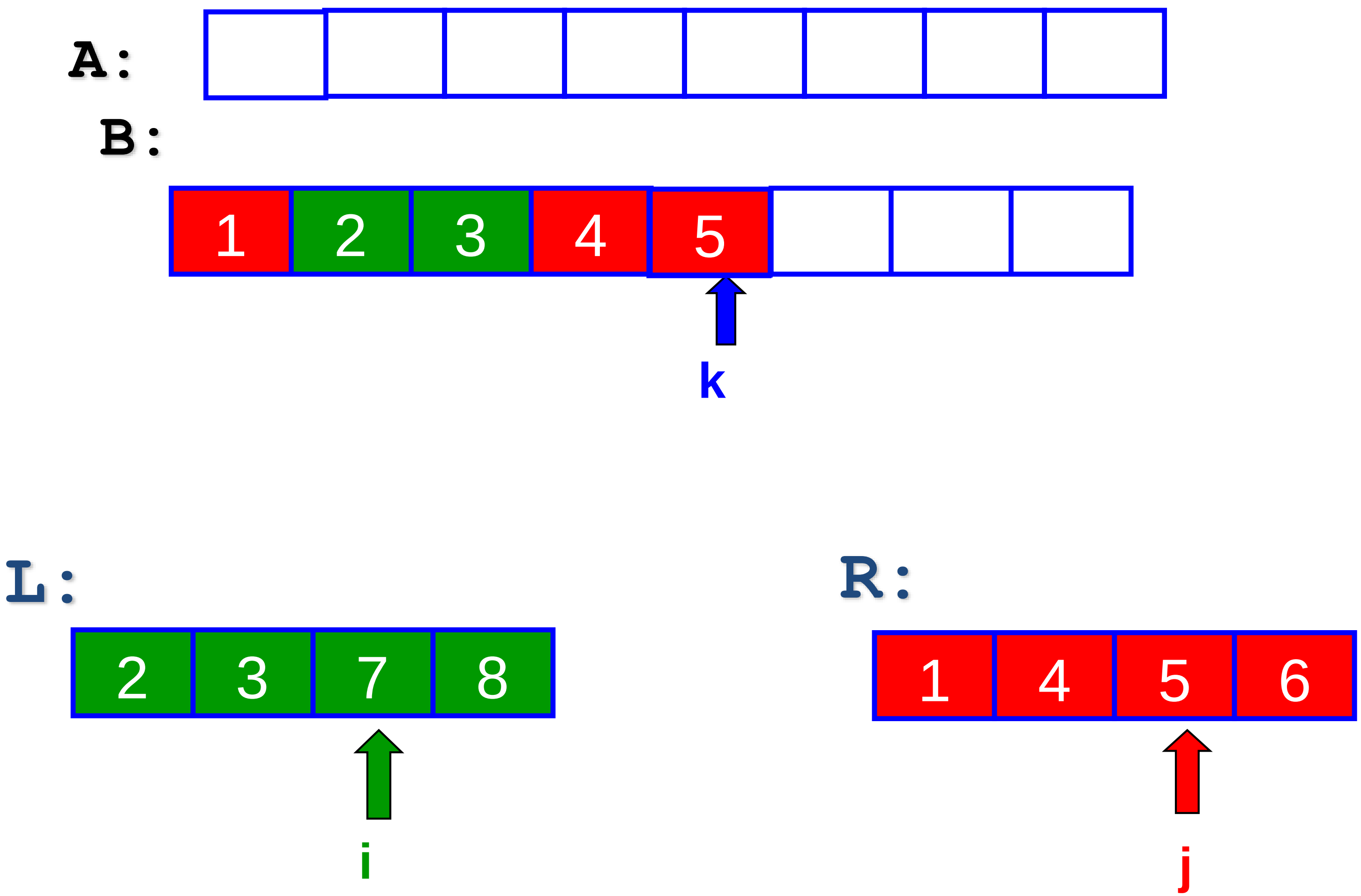
Merge-Sort: Merge Example



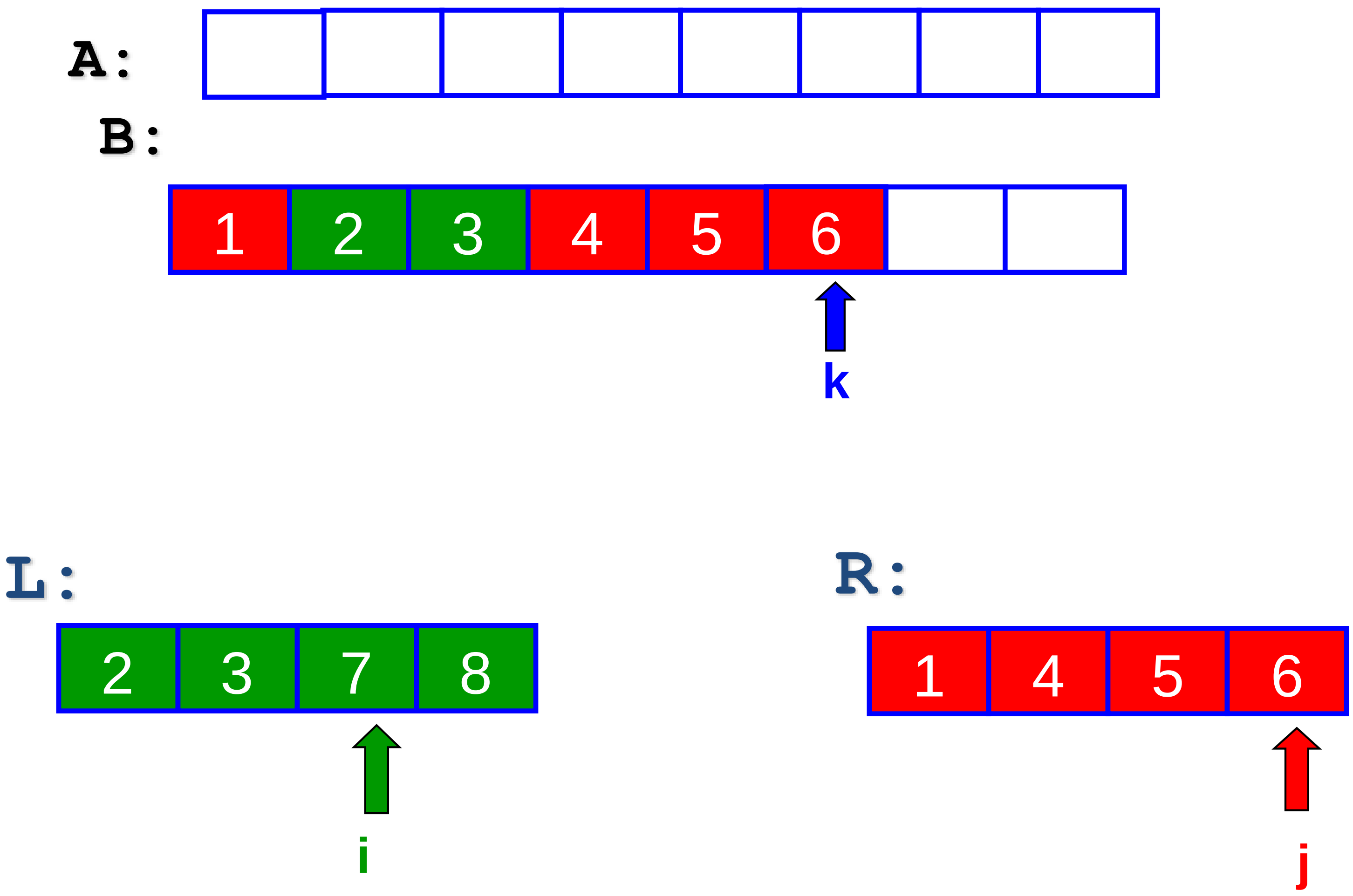
Merge-Sort: Merge Example



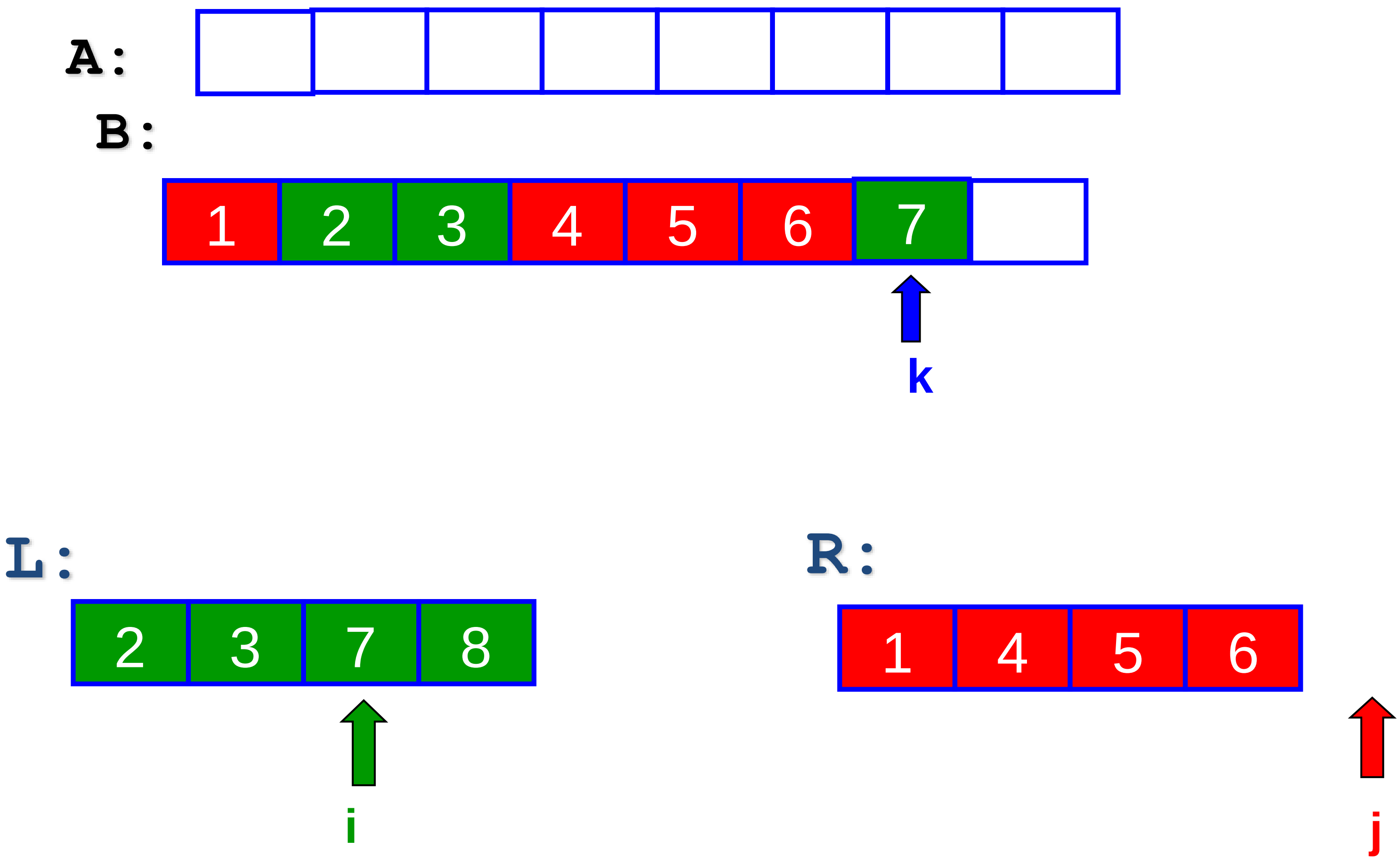
Merge-Sort: Merge Example



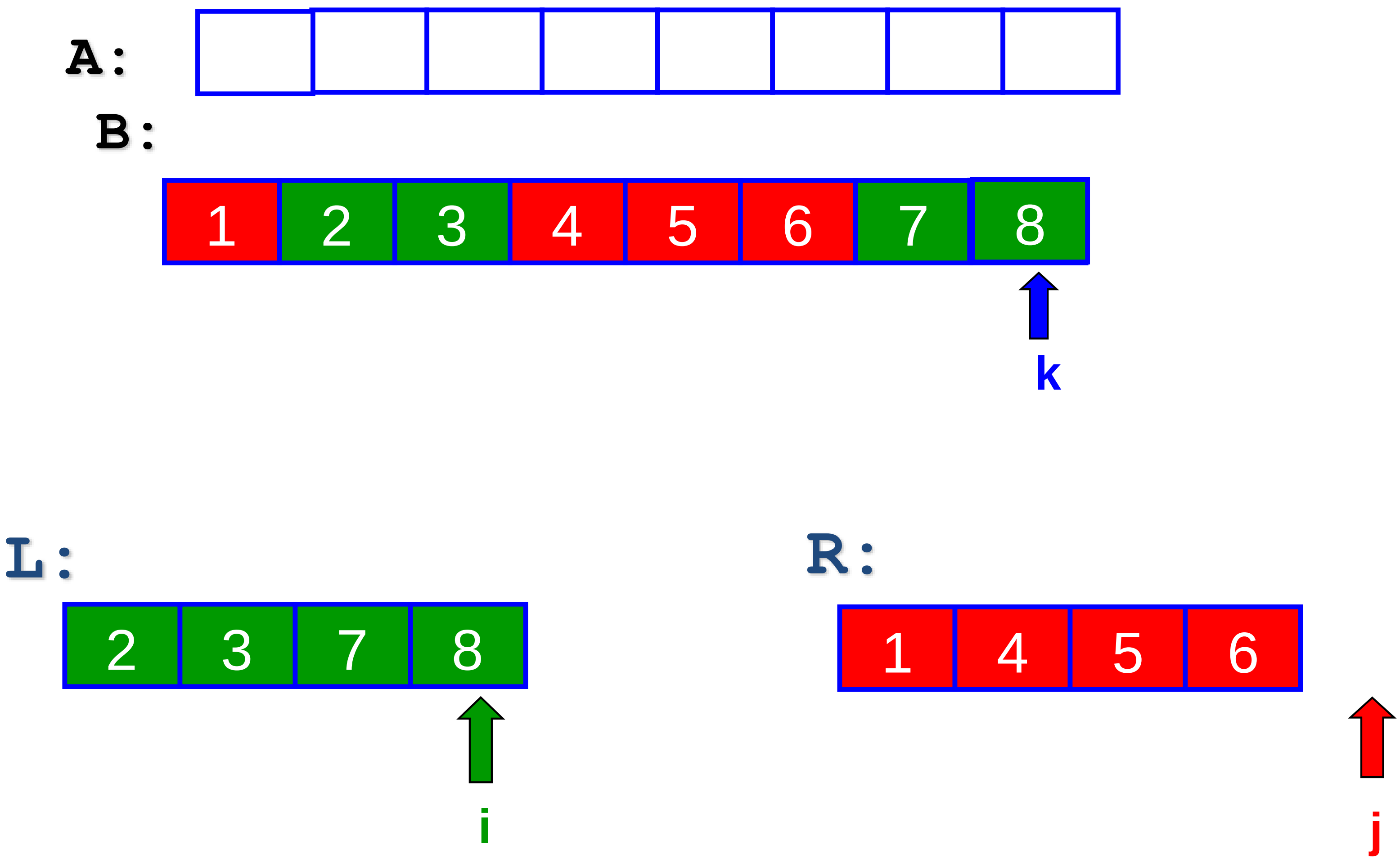
Merge-Sort: Merge Example



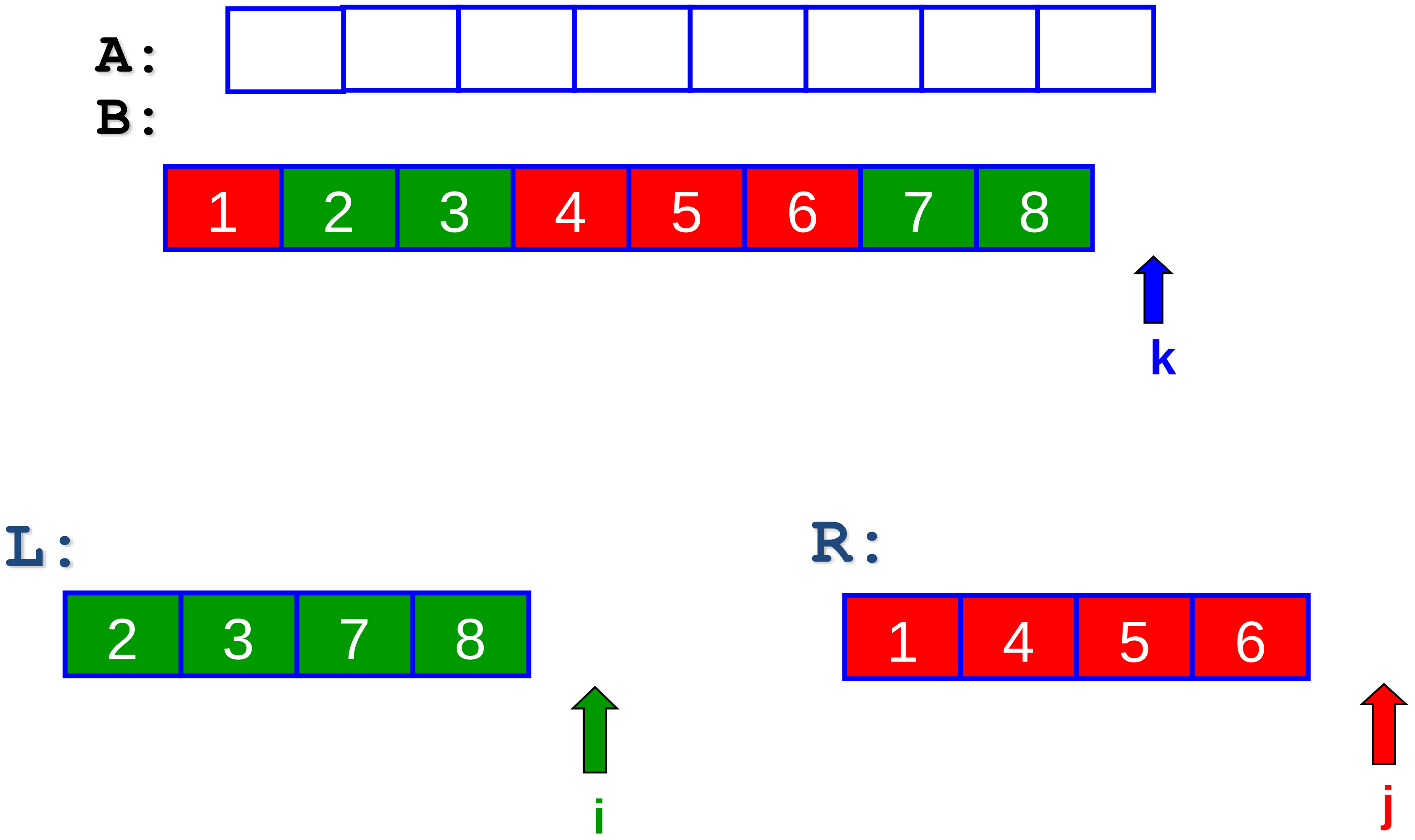
Merge-Sort: Merge Example



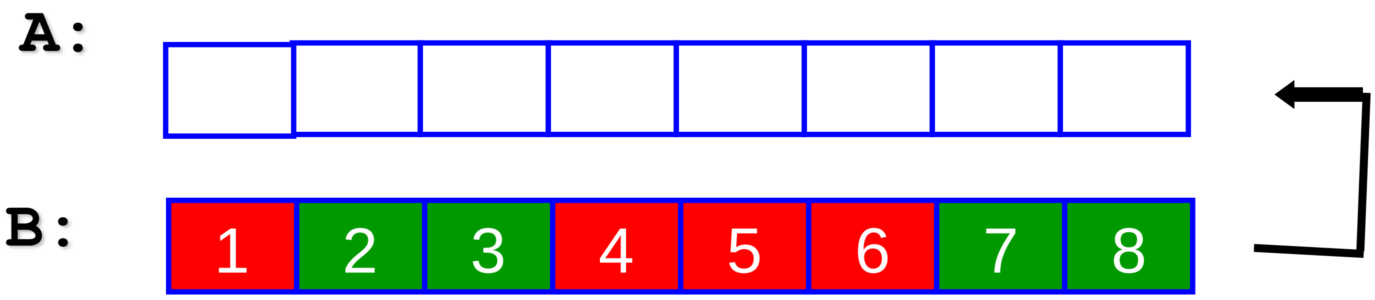
Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge Sort: Algorithm

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Merge Sort: Algorithm for Merge

```
Algorithm Merge(low, mid, high)
{
    i: = low; k: = low; j: = mid +1 ;
    while ((i ≤ mid ) and (j ≤ high)) do
    {
        if (a[i] ≤ a[j]) then
        {
            b[k]: = a[i]; i: = i + 1;
        }
        else
        {
            b[k]: = a[j]; j: = j + 1;
        }
        k: = k + 1;
    }
    if (i > mid ) then
    {
        for x: = j to high do
        {
            b[k]: = a[x]; k: = k + 1;
        }
    else
    {
        for x: = h to mid do
        {
            b[k]: = a[x]; k: = k + 1;
        }
    }
    for x: = low to high do   a[x]: = b[x];
}
```

Time Complexity of Merge Sort

- Best Case
- Worst Case
- Average Case
- All the three cases are similar irrespective of whether the given array is already sorted or unsorted.

Time Complexity of Merge Sort

$T(n)=2*T(n/2)+c*n$

$=2*[2*T(n/4)+c*n/2]+c*n = 4*T(n/4)+2*c*n$

$=4*[2*T(n/8)+c*n/4]+2*c*n = 8*T(n/8)+3*c*n$

$=16*T(n/16)+4*c*n$

...

Assume $2^i = n$ $\quad$ $=2^i*T(n/2^i)+i*c*n$

$i=\log n$ $\quad$ $=2^{\log n} * T(1)+c*n*\log n$

$=n*1+c*n*\log n$

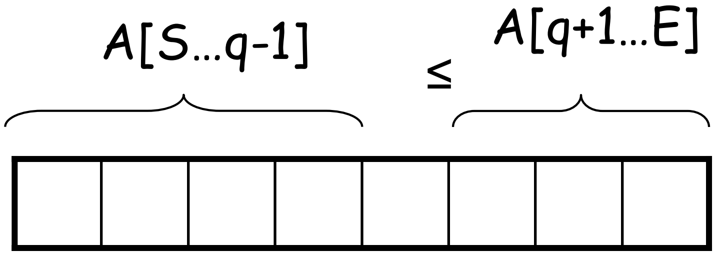
$=O(n\log n)$

Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication

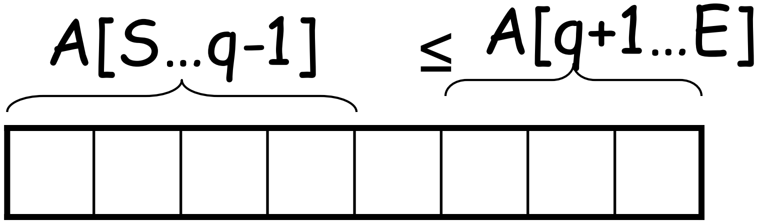
Quick Sort

- Sort an array A[S…E]



- **Divide**
  - Partition the array A into 2 subarrays A[S..q-1] and A[q+1..E], such that each element of A[S..q-1] is smaller than A[q] and each element in A[q+1..E] is greater than equal to A[q]
  - Need to **find index q to partition the array**

Quick Sort



- **Conquer**
  - Recursively sort A[S..q-1] and A[q+1..E] using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them

Quick Sort

- **Divide**:
  - Pick any element as the **pivot**, e.g, the first element
  - Partition the remaining elements into

    FirstPart, which contains all elements < pivot

    SecondPart, which contains all elements > pivot

- **Recursively sort** FirstPart and SecondPart.
- **Combine**: no work is necessary since sorting is done in place.

Quick Sort procedure

1.First element will be taken as a pivot element

2.Move "i" towards right until it satisfies a[i]>pivot

3.Move "j" towards left until it satisfies a[j]≤pivot

4.If (i<j) then swap a[i] and a[j] & continue from step 2

5.If (i≥j) then swap a[j] and pivot

6.If pivot is moved then array will be divided into 2 halfs.

7. First sub array < pivot and second sub array >pivot

8. Again apply the quick sort procedure to both halfs till the elements are sorted

Process:

Keep going from left side as long as a[i]<=pivot and

from right side as long as a[j]>pivot

pivot →	85	24	63	95	17	31	45	98
	i							j

85	24	63	95	17	31	45	98
		i					j

85	24	63	95	17	31	45	98
			i				j

85	24	63	95	17	31	45	98
			i			j	

If i<j interchange i^{th} and j^{th} elements and then Continue the process.

85	24	63	45	17	31	95	98
			i			j	

85	24	63	45	17	31	95	98
				i		j	

85	24	63	45	17	31	95	98
					i	j	

85	24	63	45	17	31	95	98
						j	
						i	

85	24	63	45	17	31	95	98
					j	i	

If $i \geq j$ interchange j^{th} and pivot elements and then divide the list into two sublists.

31 24 63 45 17 85 95 98

j

Two sublists:

31 24 63 45 17 | 85 | 95 98

Recursively sort

FirstPart and SecondPart

QickSort(low, j-1) QickSort(j+1,high)

Quick Sort-Another Example

- Sort the Elements

24 9 29 4 19 27

Algorithm for Quick Sort

```
Algorithm QuickSort(low,high)
//Sorts the elements a[low],.....,a[high] which resides in the global array a[0:n-1] into ascending order;
// a[n] is considered to be defined and must ≥ all the elements  in a[1:n].
{
        if( low< high )                 // if there are more than one element
        {                               // divide p into two subproblems.
                j :=Partition (low,high);
                QuickSort (low,j-1);
                QuickSort (j+1,high);
                                        // There is no need for combining solutions.
        }
}
```

Algorithm for Quick Sort

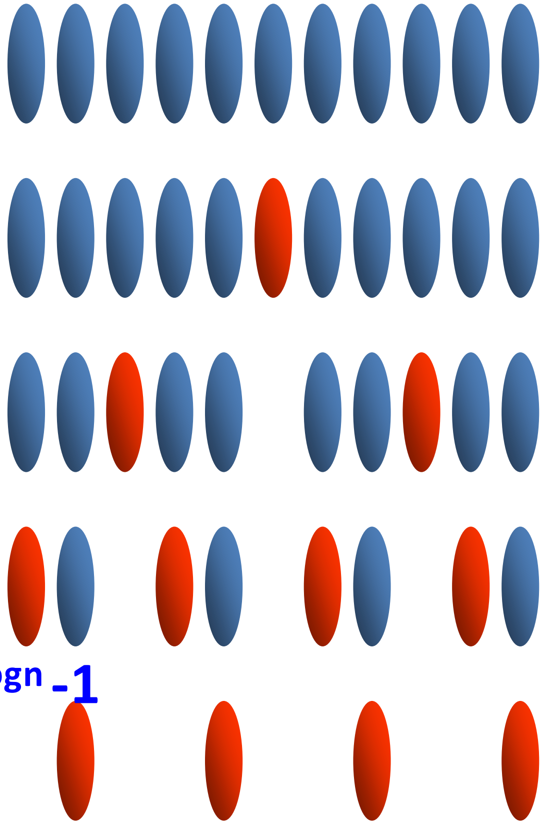
```
Algorithm Partition(l,h)
{
        pivot:= a[l] ;      i:=l;           j:= h;
        while( i < j ) do
        {
                while( a[ i ] <= pivot && i<=h ) do
                        i++;
                while( a[ j ]  > pivot && j>=l ) do
                        j--;

                if ( i < j ) then Interchange(a[i],a[j] );     // interchange ith and jth elements.
        }

        Interchange(pivot, a[j] );          // interchange pivot and jth  element.
        return j;
}
```

Time Complexity of Quick Sort

- **Best Case:**
  - Pivot element will positioned at exactly middle position
- **Worst Case:**
  - Pivot element will positioned at any one end
- **Average Case:**
  - Pivot element will be positioned at any position

Time Complexity of Quick Sort: Best Case

- It occurs only if each partition divides the list into two equal size sub lists.



$T(n)=2*T(n/2)+(n+1)$

$=2*[2*T(n/4)+(n/2+1)]+(n+1) = 4*T(n/4)+2*n+3$

$=4*[2*T(n/8)+(n/4+1)]+2*n+c = 8*T(n/8)+3*n+7$

$=16*T(n/16)+4*n+15$

...

Assume $2^i=n$ $\quad =2^i*T(n/2^i)+i*n+(2^i-1)$

$i=\log n$ $\quad =2^{\log n} * T(1)+n*\log n+ 2^{\log n} -1$

$=n*O(1)+n*\log n+n-1$

$=O(n\log n)$

Time Complexity of Quick Sort: Worst Case

It occurs only if each partition divides the list into two sub lists like one sublist is empty and other sub list contains (n-1) elements.

$$T(n)=T(n-1)+(n+1)$$

$$= [T(n-2)+(n)]+(n+1)$$

$$= T(n-2)+[(n)+(n+1)]$$

$$= [T(n-3)+(n-1)]+ [(n)+(n+1)]$$

$$=T(n-3)+[(n-1)+(n)+(n+1)]$$

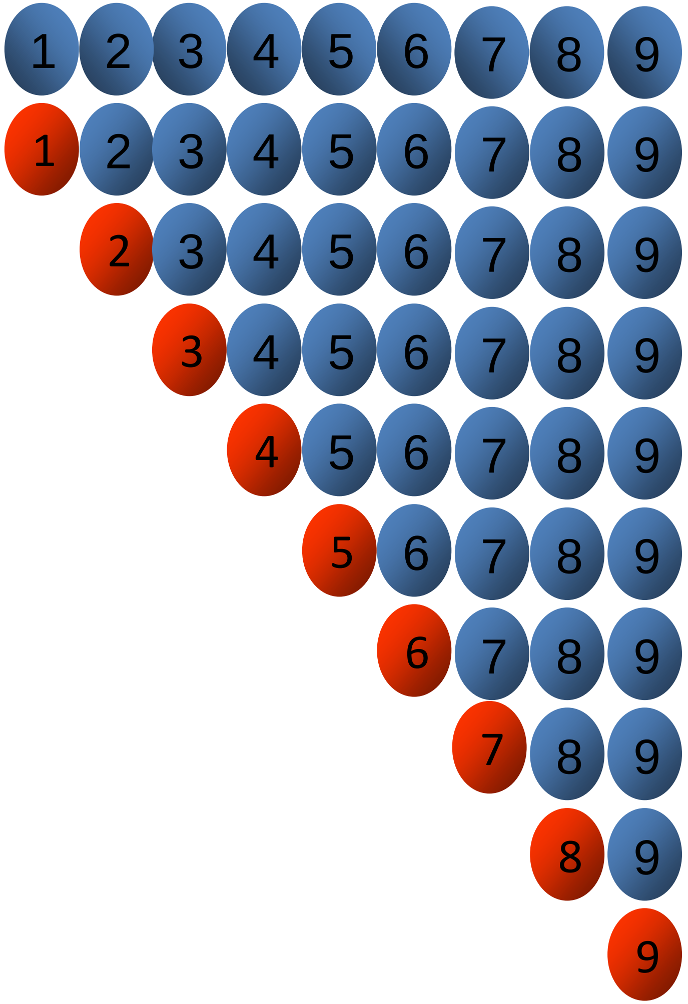
$$=T(n-4)+ [(n-2)+(n-1)+(n)+(n+1)]$$

....

$$=T(1)+ [3...+n+(n+1)]$$

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

$$= O(n^2)$$



Time Complexity of Quick Sort: Average Case

After the partition, pivot can be placed at any position in 1 to n. Partition divides the list into two sub lists such that both the sub lists are of random sizes less than n.

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1)) + C_A(n-k)] \qquad (3.5)$$

The number of element comparisons required by Partition on its first call is $n+1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.5) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \qquad (3.6)$$

Replacing n by $n-1$ in (3.6) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting this from (3.6), we get

$$nC_A(n) - (n-1)C_A(n-1) \quad = \quad 2n + 2C_A(n-1)$$

$$or$$

$$C_A(n)/(n+1) \quad = \quad C_A(n-1)/n + 2/(n+1)$$

Time Complexity of Quick Sort: Average Case

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots,$ we get

$$\begin{aligned}
\frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\ \vdots \\
&= \frac{C_A(1)}{2} + 2\sum_{3 \le k \le n+1} \frac{1}{k} \\
&= 2\sum_{3 \le k \le n+1} \frac{1}{k}
\end{aligned} \tag{3.7}$$

Since

$$\sum_{3 \le k \le n+1} \frac{1}{k} \le \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

$$C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication

Divide-and-Conquer Approach

- Let P = (n, a[i],….,a[j]) denote an arbitrary instance of the problem.

- Here n is the number of elements in the list a[i],…,a[j]and we are interested in finding the maximum and minimum of this list.

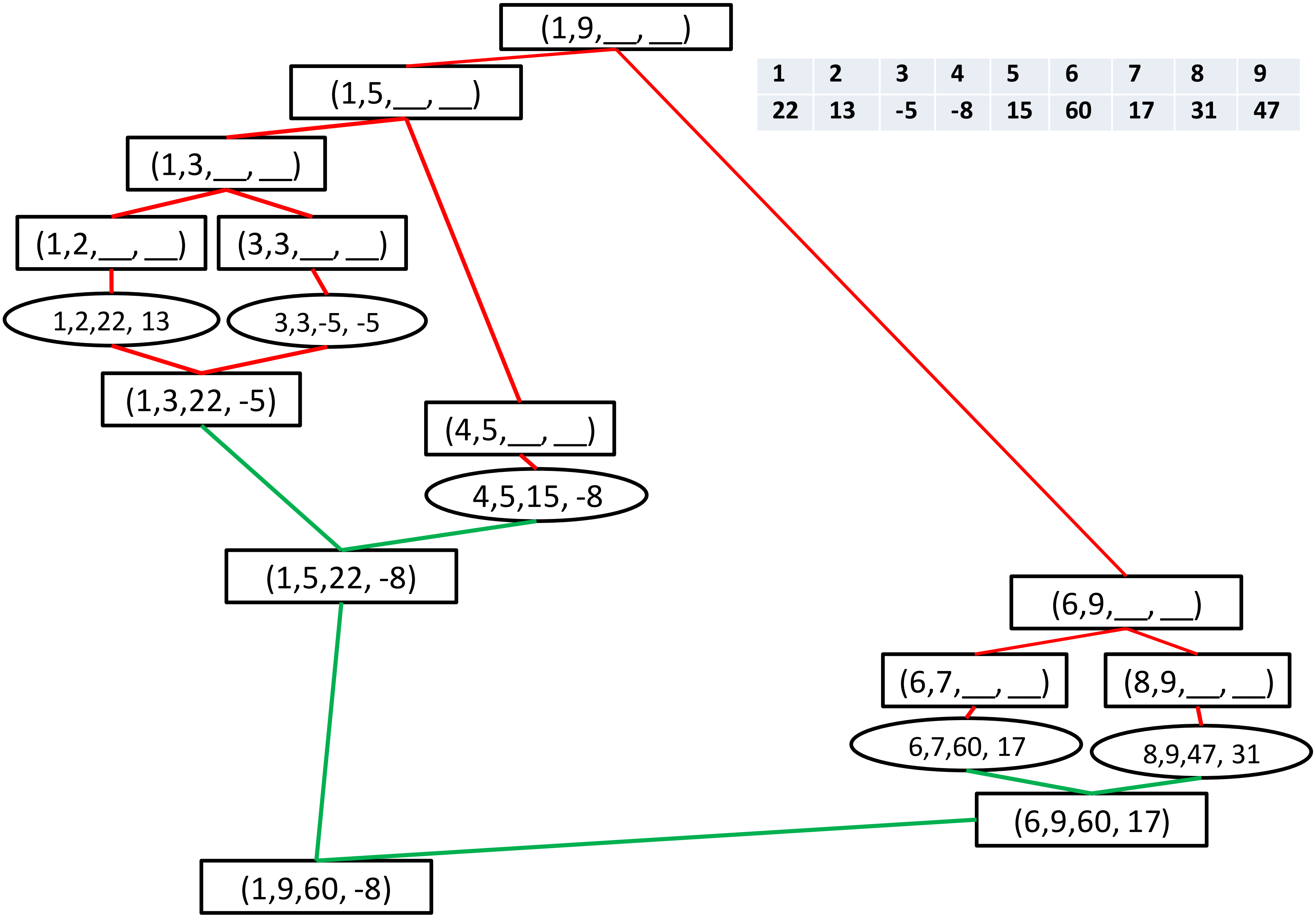
- Small(P):when n <=2.
  - If n=1, the maximum and minimum is a[i].
  - If n=2, the problem can be solved by making one comparison.

Divide-and-Conquer Approach

- If the list has more than two elements, P has to be divided into smaller instances stances.
- For example, we might **divide P into the two instances**
  - P1 = (n/2,a[l],...,a[|n/2]) and
  - P2 = (n- [n/2], a[[n/2+1],...,a..[n]).
- After having divided P into two smaller sub problems wee can **solve them by recursively** invoking the same divide-and-conquer algorithm.
- How can we **combine the solutions for P1 and P2** to obtain a solution for P?
- If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then
  - MAX(P) is the larger of MAX(P1) and MAX(P2).
  - MIN(P) is the smaller of MIN(P1) and MIN(P2).

Working Example of Finding Max Min

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



Algorithm for Finding MaxMin

```
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;
        }
}
```

Time Complexity of Finding MaxMin

- **Best Case**
- **Worst Case**
- **Average Case**
- **All the three cases are similar irrespective of elements in the array.**

Time Complexity

$T(n)=2*T(n/2)+5$

$T(n)=2*T(n/2)+c$

$=2*[2*T(n/4)+c]+c \qquad = 4*T(n/4)+3c$

$=4*[2*T(n/8)+c]+3c \qquad = 8*T(n/8)+ 7c$

$=16*T(n/16)+ 15c$

...

Assume $2^i = n$ $\qquad =2^i*T(n/2^i)+ (2^i-1)c$

$i=\log n$ $\qquad = 2^{\log n} *O(1)+ (2^{\log n} -1)c$

$= n*O(1) + (n -1)c$

$= O(n)$

Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication

Matrix Multiplication

- multiply two 2×2 matrices

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 3 & 5 \\ 1 & 4 \end{pmatrix} = \begin{pmatrix} 1*3+2*1 & 1*5+2*4 \\ 3*3+4*1 & 3*5+4*4 \end{pmatrix}$$

$$= \begin{pmatrix} 5 & 13 \\ 13 & 31 \end{pmatrix}$$

How many multiplications and additions did we need?

Basic Matrix Multiplication

Let A and B two n×n matrices. The product C=A*B is also an n×n matrix.

```
void matrix_mult (A, B, N)
{
  for (i = 1; i <= N; i++)
    {
        for (j = 1; j <= N; j++)
         {
              for(k=1; k<=N; k++)
              {
                C[i,j]=C[i,j]+A[i,k]*B[k,j];
              }
         }
     }
   return C;
}
```

Time complexity of above algorithm is **$T(n)=O(n^3)$**

Divide and Conquer technique

- We want to compute the product C=A*B, where each of A,B, and C are n×n matrices.
- Assume n is a power of 2.
- If n is not a power of 2, add enough rows and columns of zeros.
- We divide each of A,B, and C into four n/2×n/2 matrices, rewriting the equation C=A* B as follows:

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) * \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

Divide and Conquer technique

Then,

$C_{11}=A_{11}B_{11}+A_{12}B_{21}$

$C_{12}=A_{11}B_{12}+A_{12}B_{22}$

$C_{21}=A_{21}B_{11}+A_{22}B_{21}$

$C_{22}=A_{21}B_{12}+A_{22}B_{22}$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 5 & 6 & 6 \\ 5 & 5 & 6 & 6 \\ 7 & 7 & 8 & 8 \\ 7 & 7 & 8 & 8 \end{bmatrix}$$

A_{11} A_{12} B_{11} B_{12}

A_{21} A_{22} B_{21} B_{22}

- Each of these four equations specifies two multiplications of n/2×n/2 matrices and the addition of their n/2×n/2 products.
- We can derive the following recurrence relation for the time T(n) to multiply two n×n matrices:

$$T(n)= \begin{cases} c_1 & \text{if } n<=2 \\ 8T(n/2)+ cn^2 & \text{if } n>2 \end{cases}$$

$T(n) = 8T(n/2) + cn^2$

$$= 8\left[8T(n/4) + c(n/2)^2\right] + cn^2$$

$$= 8^2\,T(n/4) + c2n^2 + cn^2$$

$$= 8^2\left[8T(n/8) + c(n/4)^2\right] + c2n^2 + cn^2$$

$$= 8^3\,T(n/8) + c2^2n^2 + c2n^2 + cn^2$$

:

$$= 8^iT(n/2^i) + cn^2[2^{i-1}\ldots\ldots\ldots\ldots\ldots\ldots + 2^2 + 2 + 1]$$

$$= 8^{\log_2 n}\,1 + [(2^{i-1+1}-1)/2]cn^2$$

$$= n^{\log_2 8} * 1 + 2^i c\, n^2$$

$$= n^3 + [2^{\log_2 n}]cn^2$$

$$= n^3 + [n]cn^2 = O(n^3)$$

$$S_n = a + ar + ar^2 + \ldots + ar^{n-1}$$

$$\text{When } r > 1,\ S_n = a\frac{(r^n - 1)}{(r - 1)}$$

$T(n) = O(n^3)$

- This method is no faster than the ordinary method.

Strassen's Matrix Multiplication

- Matrix multiplications are more expensive than matrix additions or subtractions($O(n^3)$ versus $O(n^2)$).
- Strassen has discovered a way to compute the multiplication using only **7** multiplications and **18** additions or subtractions.
- His method involves computing 7 $n \times n$ matrices P,Q,R,S,T,U, and V, then Cij's are calculated using these matrices.

Formulas for Strassen's Algorithm

P =

Q =

R =

S =

T =

U =

V =

$$\begin{bmatrix} [c_{11}] & [c_{12}] \\ [c_{21}] & [c_{22}] \end{bmatrix} = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) * \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

C_{11}=

C_{12}=

C_{21}=

C_{22}=

Formulas for Strassen's Algorithm

$P = (A_{11} + A_{22}) * (B_{11} + \mathbf{B_{22}})$
$Q = (A_{21} + A_{22}) * B_{11}$
$R = A_{11} * (B_{12} - \mathbf{B_{22}})$
$S = A_{22} * (B_{21} - \mathbf{B_{11}})$
$T = (A_{11} + A_{12}) * \mathbf{B_{22}}$
$U = (A_{21} - A_{11}) * (B_{11} + \mathbf{B_{12}})$
$V = (A_{12} - A_{22}) * (B_{21} + \mathbf{B_{22}})$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) * \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

$\mathbf{C_{11} = P + S - T + V}$
$\mathbf{C_{12} = R + T}$
$\mathbf{C_{21} = Q + S}$
$\mathbf{C_{22} = P + R - Q + U}$

Example on Strassen's Algorithm

$P = (A_{11} + A_{22}) * (B_{11} + \mathbf{B_{22}})$
$Q = (A_{21} + A_{22}) * B_{11}$
$R = A_{11} * (B_{12} - \mathbf{B_{22}})$
$S = A_{22} * (B_{21} - \mathbf{B_{11}})$
$T = (A_{11} + A_{12}) * \mathbf{B_{22}}$
$U = (A_{21} - A_{11}) * (B_{11} + \mathbf{B_{12}})$
$V = (A_{12} - A_{22}) * (B_{21} + \mathbf{B_{22}})$

$$\begin{bmatrix} \begin{bmatrix} c_{11} \end{bmatrix} & \begin{bmatrix} c_{12} \end{bmatrix} \\ \begin{bmatrix} c_{21} \end{bmatrix} & \begin{bmatrix} c_{22} \end{bmatrix} \end{bmatrix} = \begin{array}{c} \begin{array}{cc} A_{11} & A_{12} \end{array} \\ \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \\ \begin{array}{cc} A_{21} & A_{22} \end{array} \end{array} \times \begin{array}{c} \begin{array}{cc} B_{11} & B_{12} \end{array} \\ \begin{bmatrix} 5 & 5 & 6 & 6 \\ 5 & 5 & 6 & 6 \\ 7 & 7 & 8 & 8 \\ 7 & 7 & 8 & 8 \end{bmatrix} \\ \begin{array}{cc} B_{21} & B_{22} \end{array} \end{array}$$

$\mathbf{C_{11} = P + S - T + V}$
$\mathbf{C_{12} = R + T}$
$\mathbf{C_{21} = Q + S}$
$\mathbf{C_{22} = P + R - Q + U}$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} c_1 & n <= 2 \\ 7T(n/2) + c_2 n^2 & n > 2 \end{cases}$$

$$T(n) = 7T(n/2) + cn^2$$

$$= 7[7T(n/4)+c*(n/2)^2] + cn^2 \quad = 7^2 * T(n/4) + cn^2[7/4+1]$$

$$= 7^2[7T(n/8)+c*(n/4)^2] + cn^2[7/4+1]$$

$$= 7^3 * T(n/8) + cn^2[(7/4)^2+(7/4)+1]$$

$$\ldots\ldots$$

$$= 7^i * T(n/2^i) + cn^2[(7/4)^{i-1}+\ldots+(7/4)+1]$$

$$= 7^{\log n} \; T(1) + cn^2 \; [(7/4)^i-1]/[7/4-1]$$

$$S_n = a + ar + ar^2 + \ldots + ar^{n-1}.$$

$$\text{When } r > 1, S_n = a\frac{(r^n - 1)}{(r - 1)}$$

$$= 7^{\log n} * 1 + c\, n^2 \, (7/4)^{\log n}$$

$$= n^{\log 7} + c\, n^{\log 4} \, (n^{\log 7 - \log 4})$$

$$= n^{\log 7} + c\, n^{\log 4 + \log 7 - \log 4}$$

$$= n^{\log 7} + c\, n^{\log 7}$$

$$= (c+1)\, n^{\log_2 7}$$

$$= O(n^{\log_2 7}) \sim O(n^{2.81})$$

Topics

- General method
- Binary search
- Merge sort
- Quick sort
- Finding the Maximum Minimum
- Strassen‘s matrix multiplication