Design and Analysis of Algorithms

UNIT-II

DIVIDE AND CONQUER

Dr G. Kalyani

Topics

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

Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** 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

- The Divide and Conquer Technique splits n inputs into k subsets, $1< k \leq n$, 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.

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;
        Apply DandC to each of these subproblems;
        return Combine(DandC(p1), DandC(p2),..., DandC(pk));
    }
}
```

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.

General Method of Divide-and-Conquer

There are 2 methods for solving an recurrence relation

- Substitution method
- Master’s theorem

General Method of Divide-and-Conquer

Substitution method:

Example: Consider the case in which a=2 and b=2. Let T(1)=2 and f(n)=n. We have

$$
\begin{aligned}
T(n) &= 2T(n/2)+n \\
&= 2[2T(n/4)+n/2]+n \\
&= 4T(n/4)+2n \\
&=4[2T(n/8)+n/4]+2n \\
&= 8T(n/8)+3n \\
&\quad \vdots \\
&=2^iT(n/2^i)+in, \text{ for any } \log_2 n>=i>=1 \\
&=2^{\log_2 n}T(n/2^{\log_2 n})+n\log_2 n
\end{aligned}
$$

Thus, $T(n)=nT(1)+n\log_2 n=n\log_2 n+2n$

Problems for Practice

Solve the recurrence relation (3.2) for the following choices of a, b, and $f(n)$ (c being a constant):

(a) $a = 1,\ b = 2$, and $f(n) = cn$

(b) $a = 5,\ b = 4$, and $f(n) = cn^2$

(c) $a = 28,\ b = 3$, and $f(n) = cn^3$

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 0 or -1.

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);
    }
}
```

Iterative Algorithm for Binary Search

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
             else return mid;
    }
    return 0;
}
```

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

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

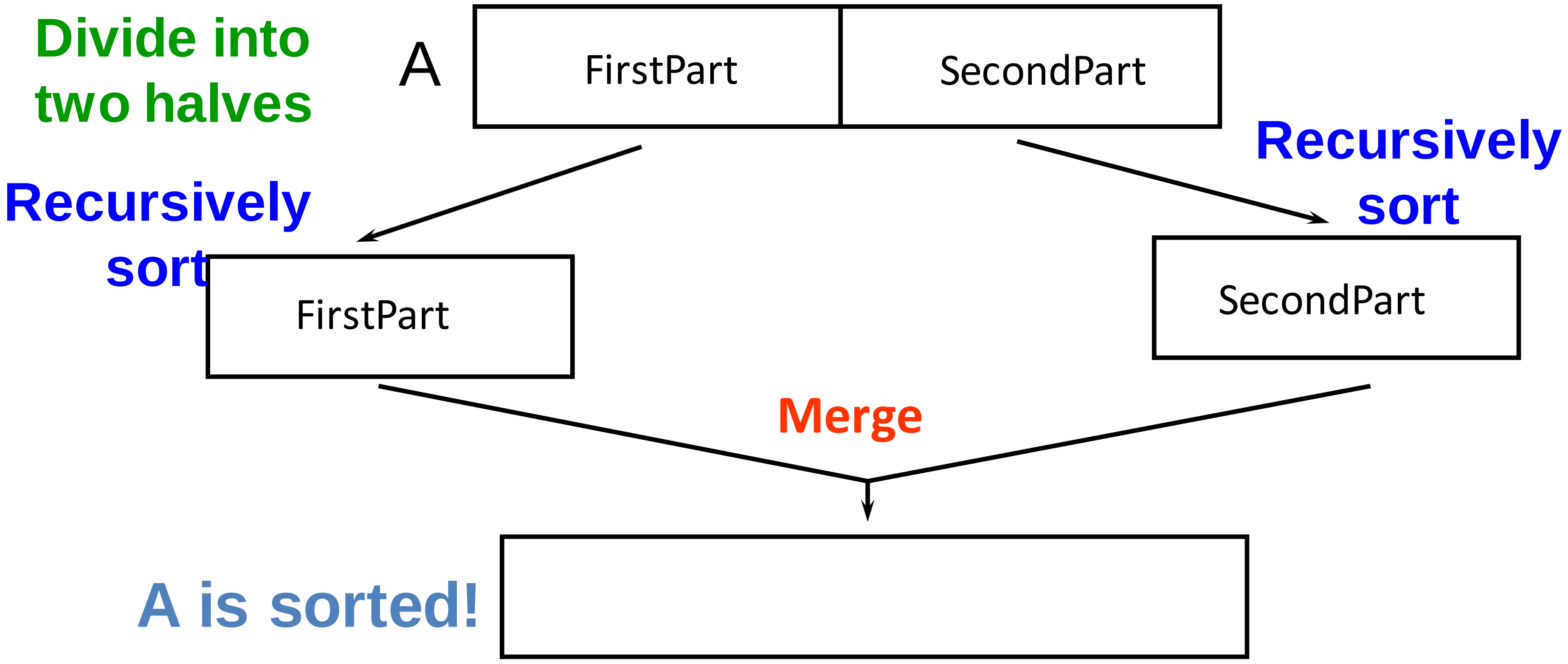
Hence $O(\log n)$

```
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);
    }
}
```

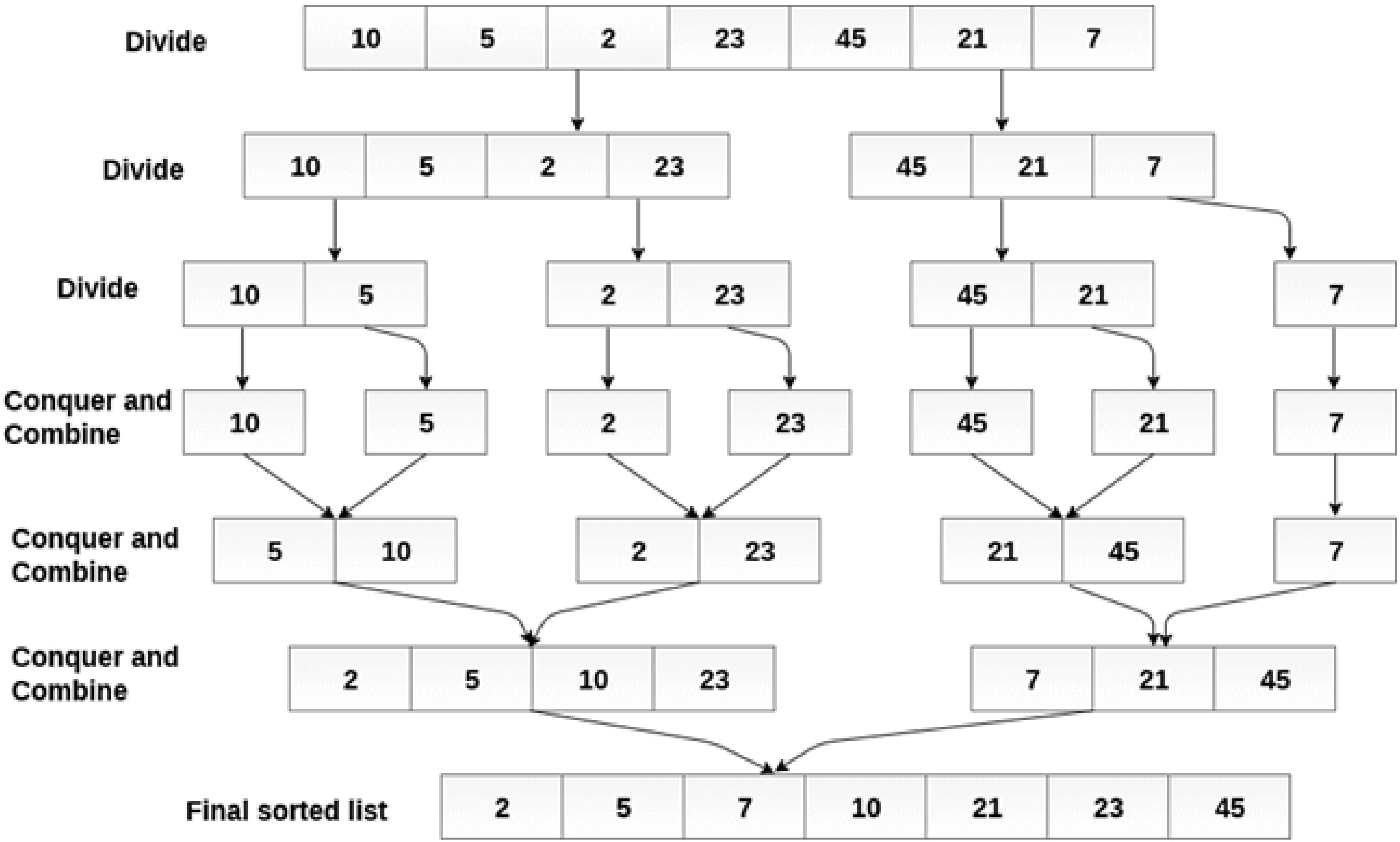
Topics

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

Merge Sort: Idea



Example for Merge Sort

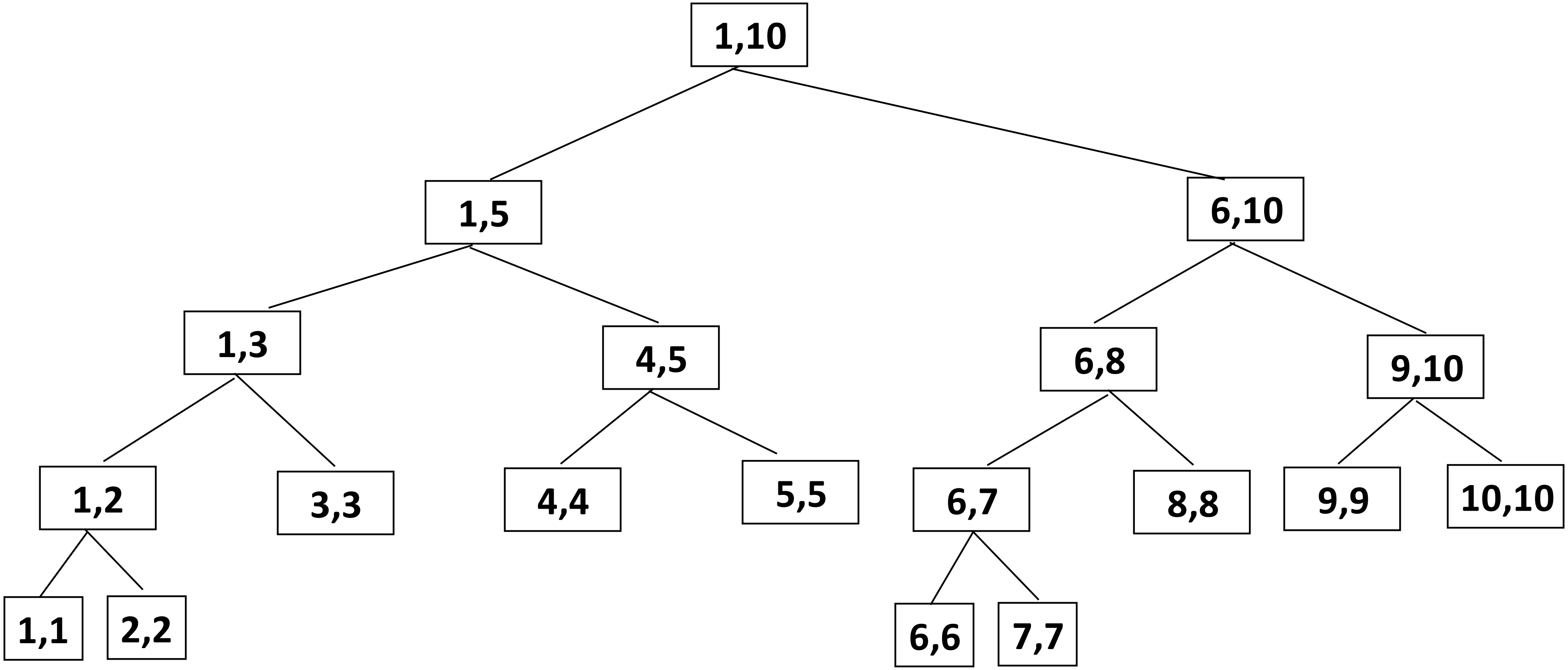


Example for Merge Sort

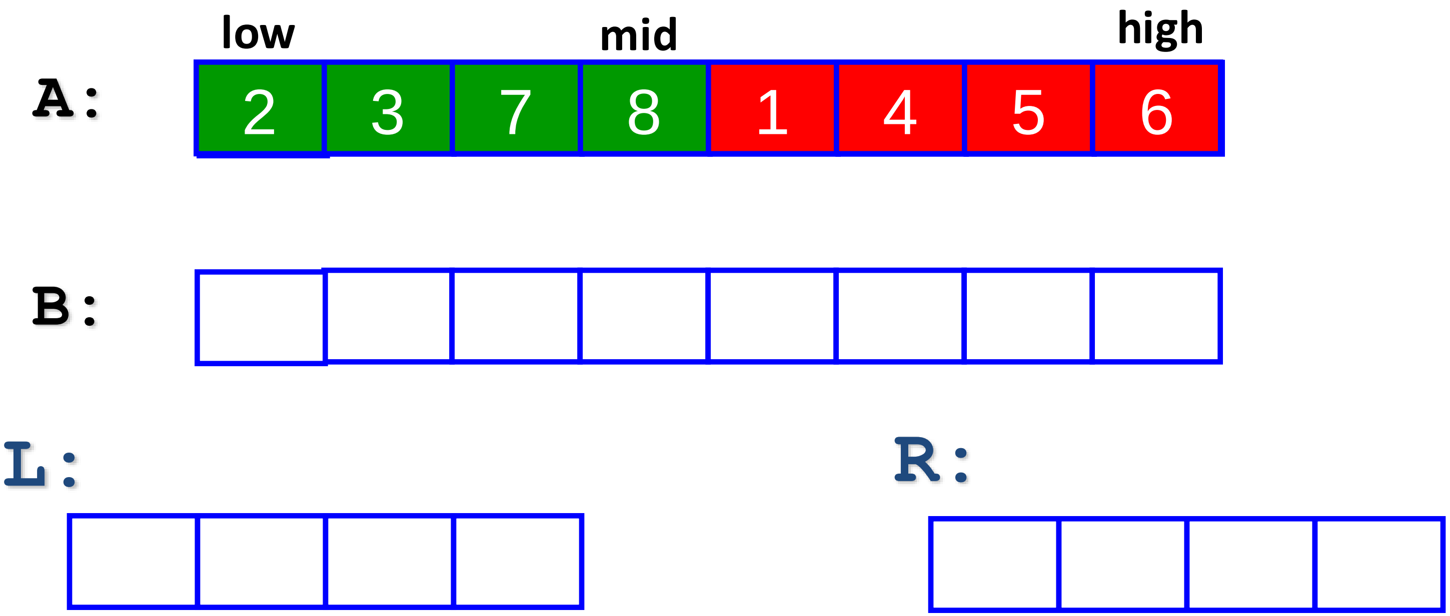
- Ex 2:-

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

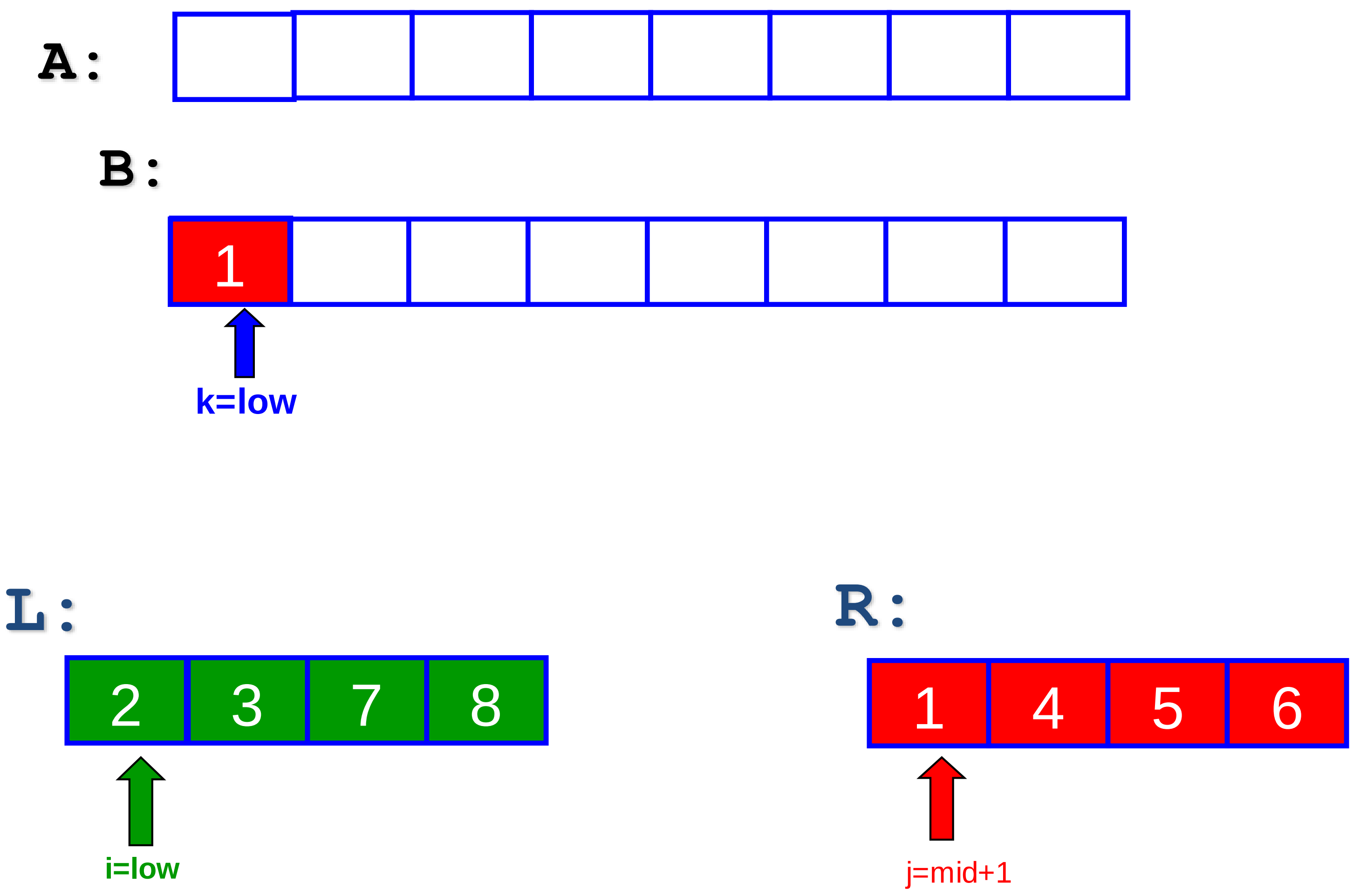
Example for Merge Sort



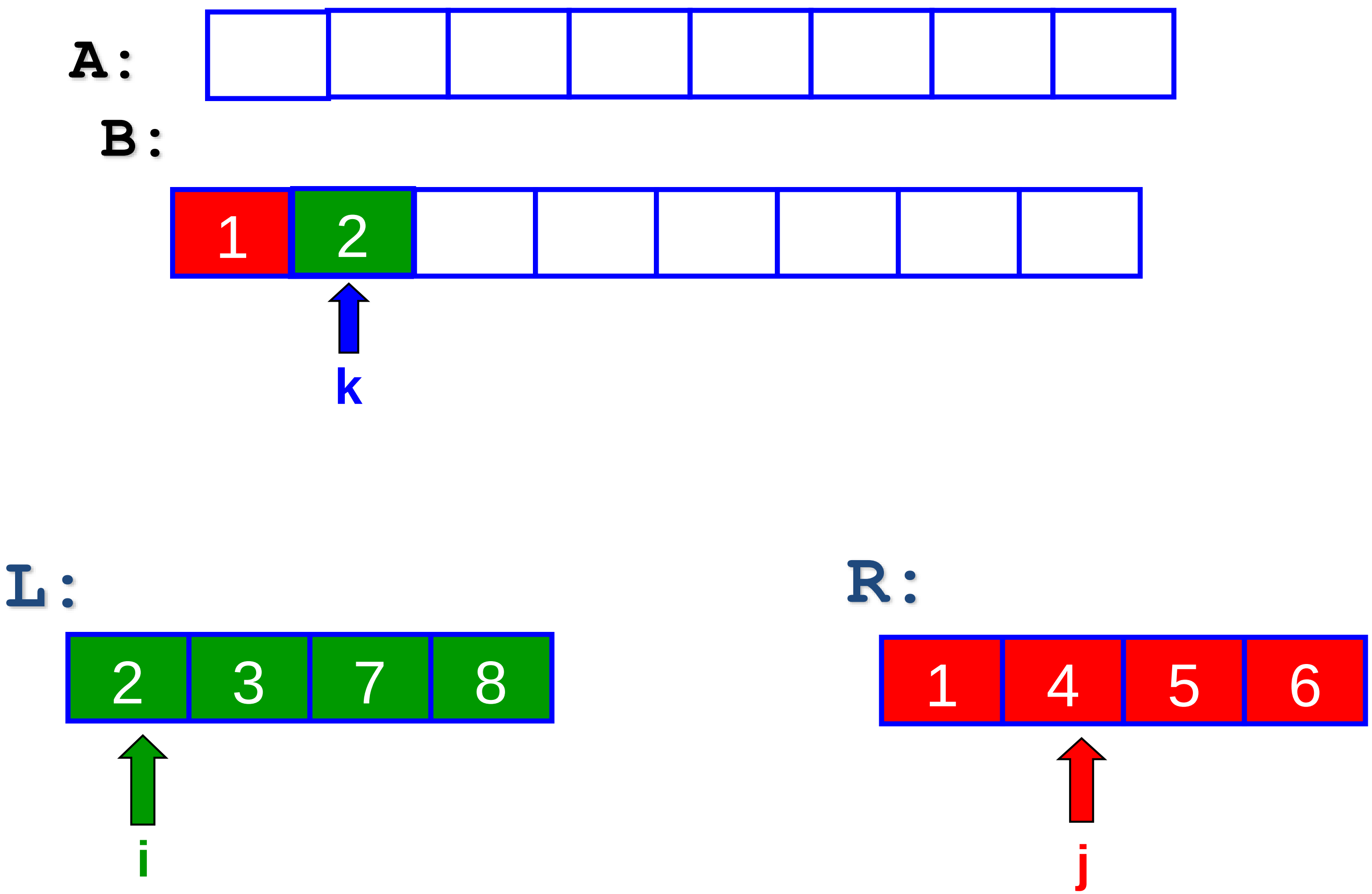
Merge-Sort: Merge Example



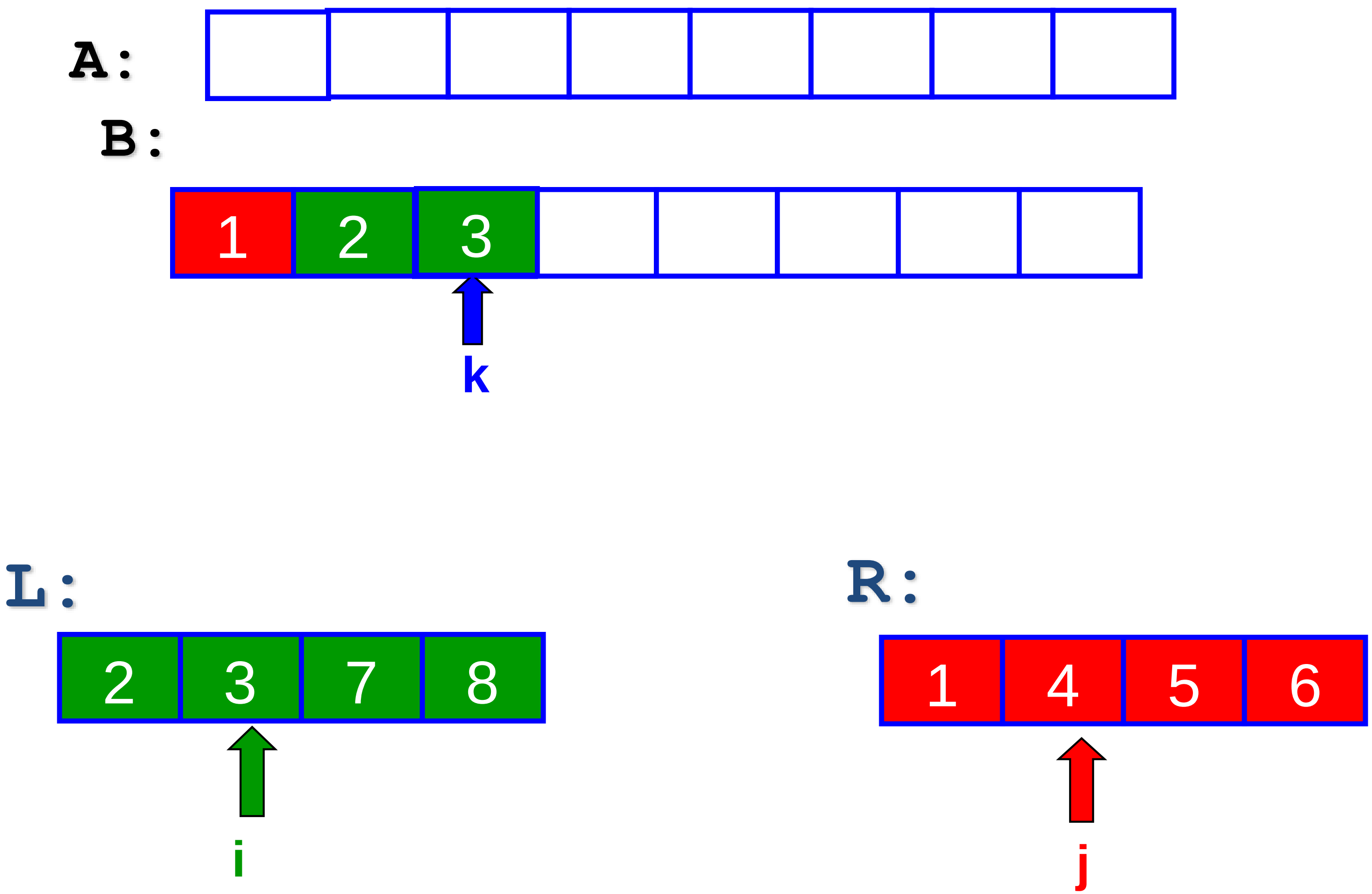
Merge-Sort: Merge Example



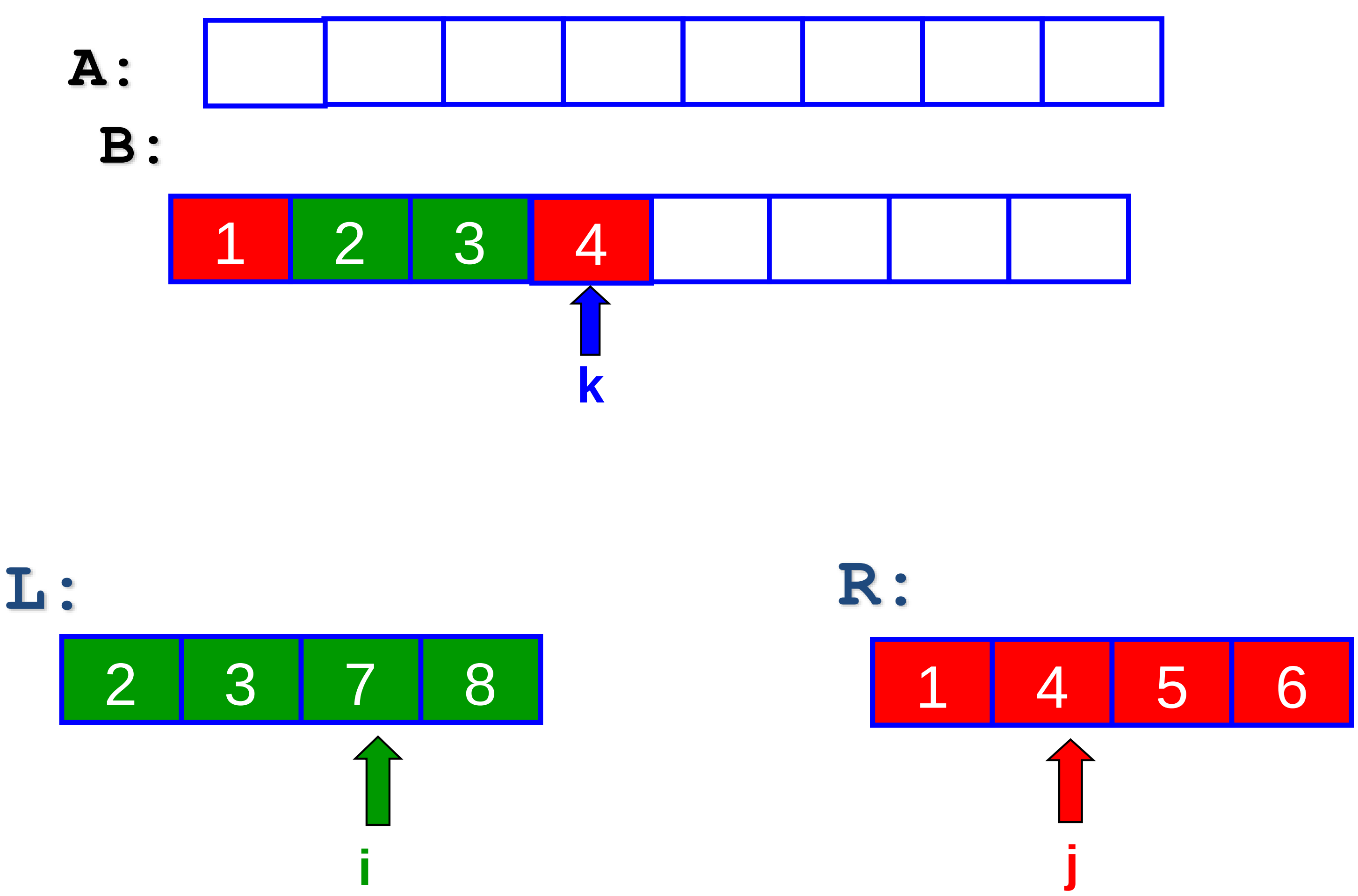
Merge-Sort: Merge Example



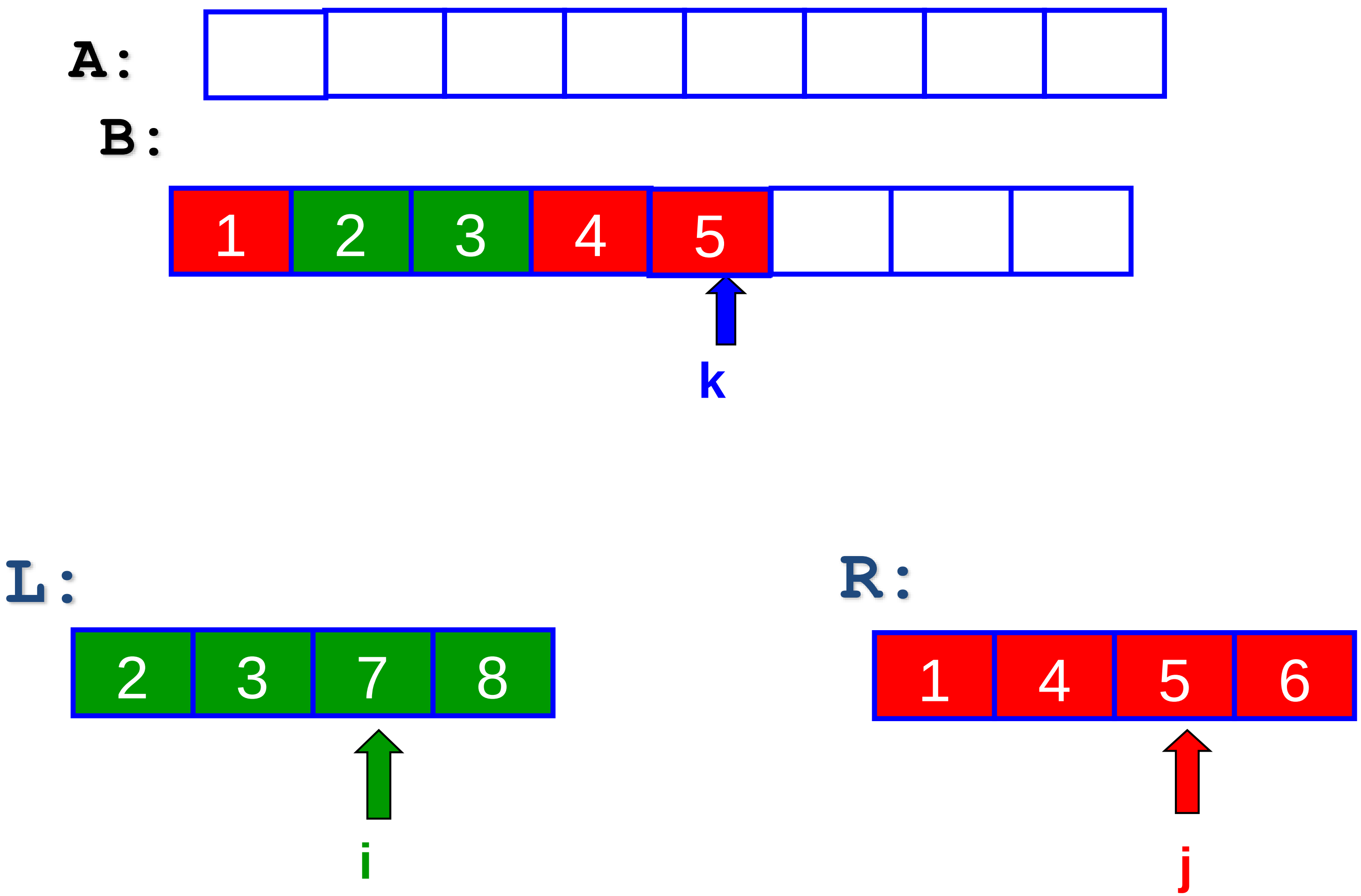
Merge-Sort: Merge Example



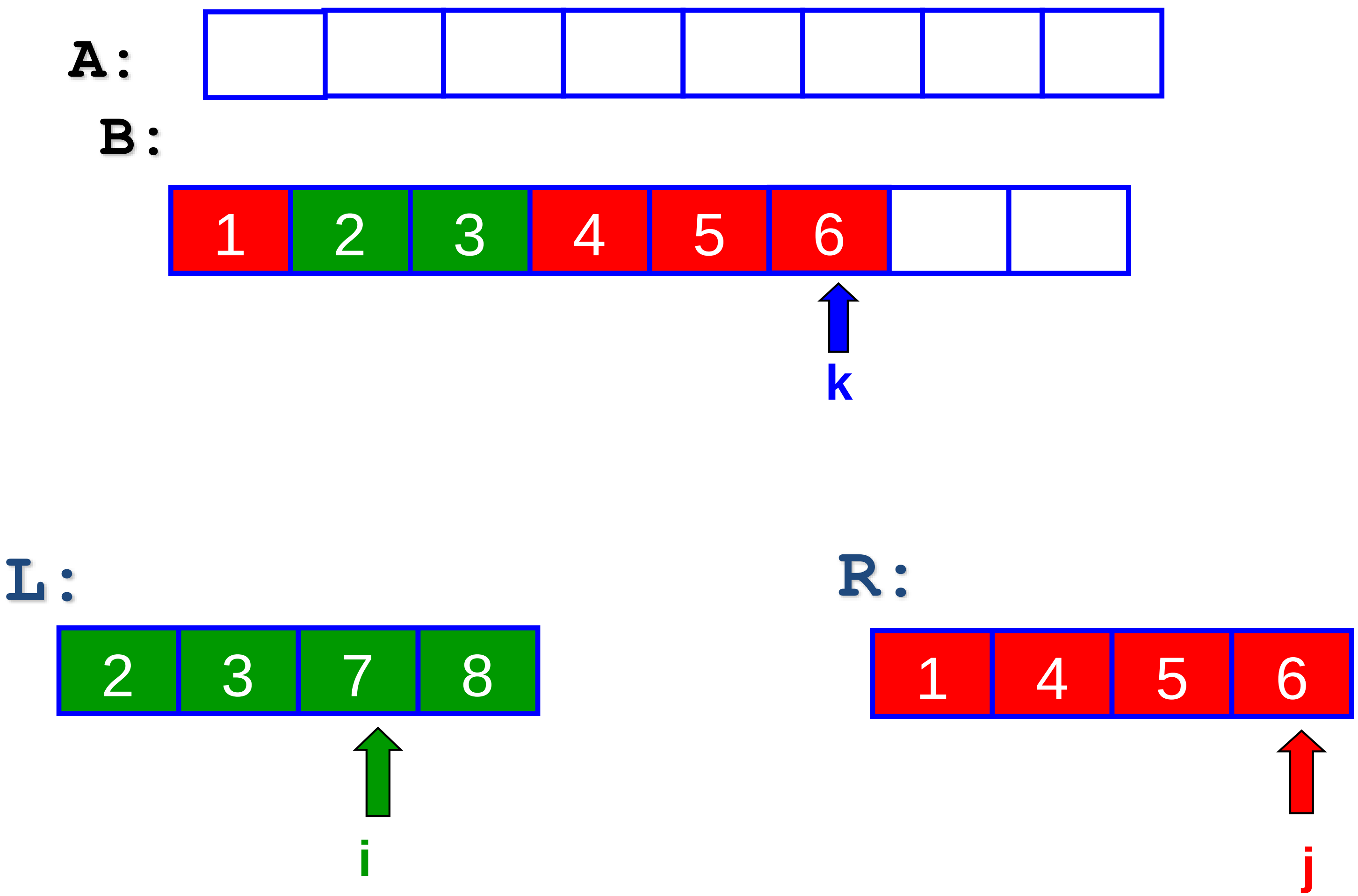
Merge-Sort: Merge Example



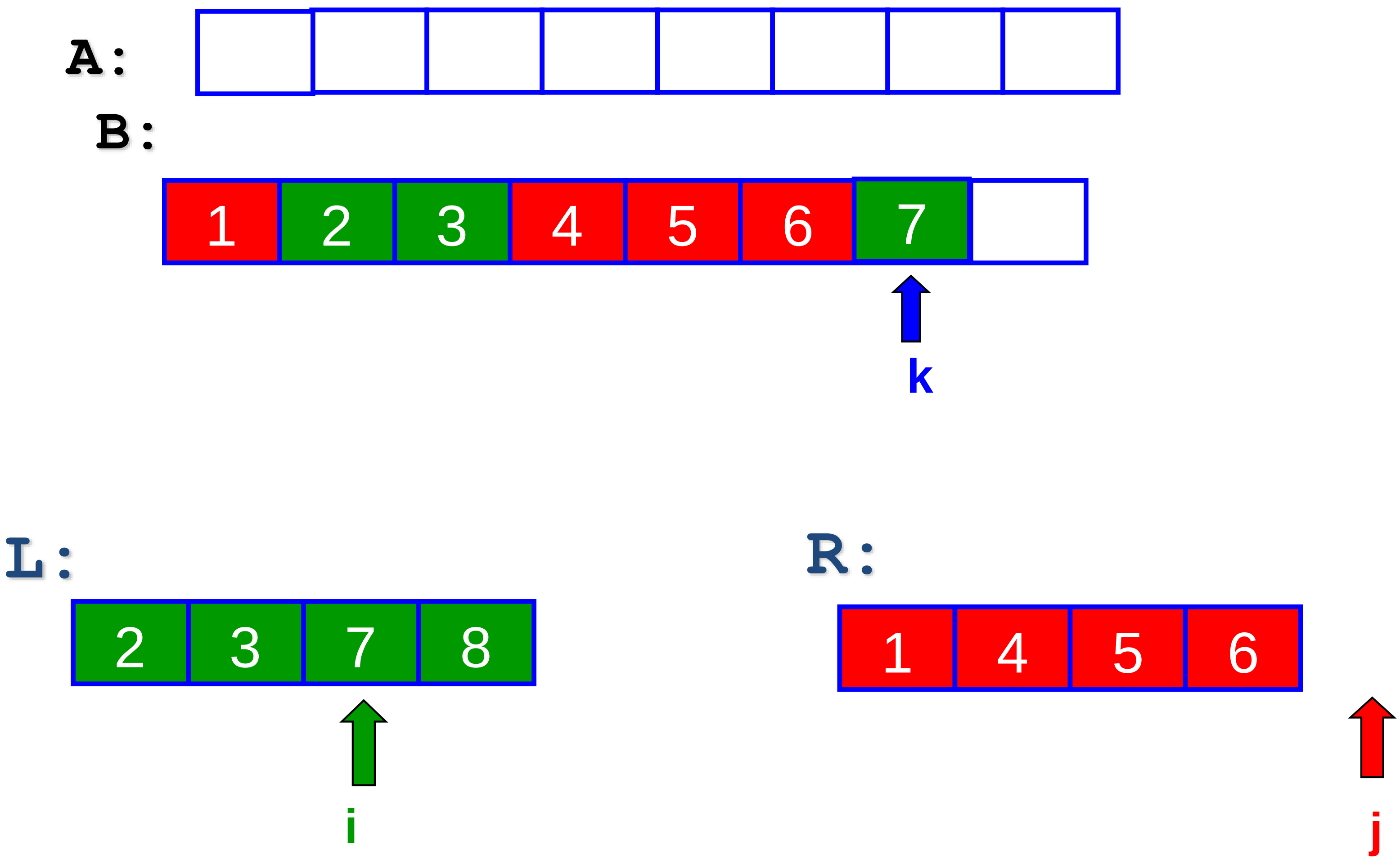
Merge-Sort: Merge Example



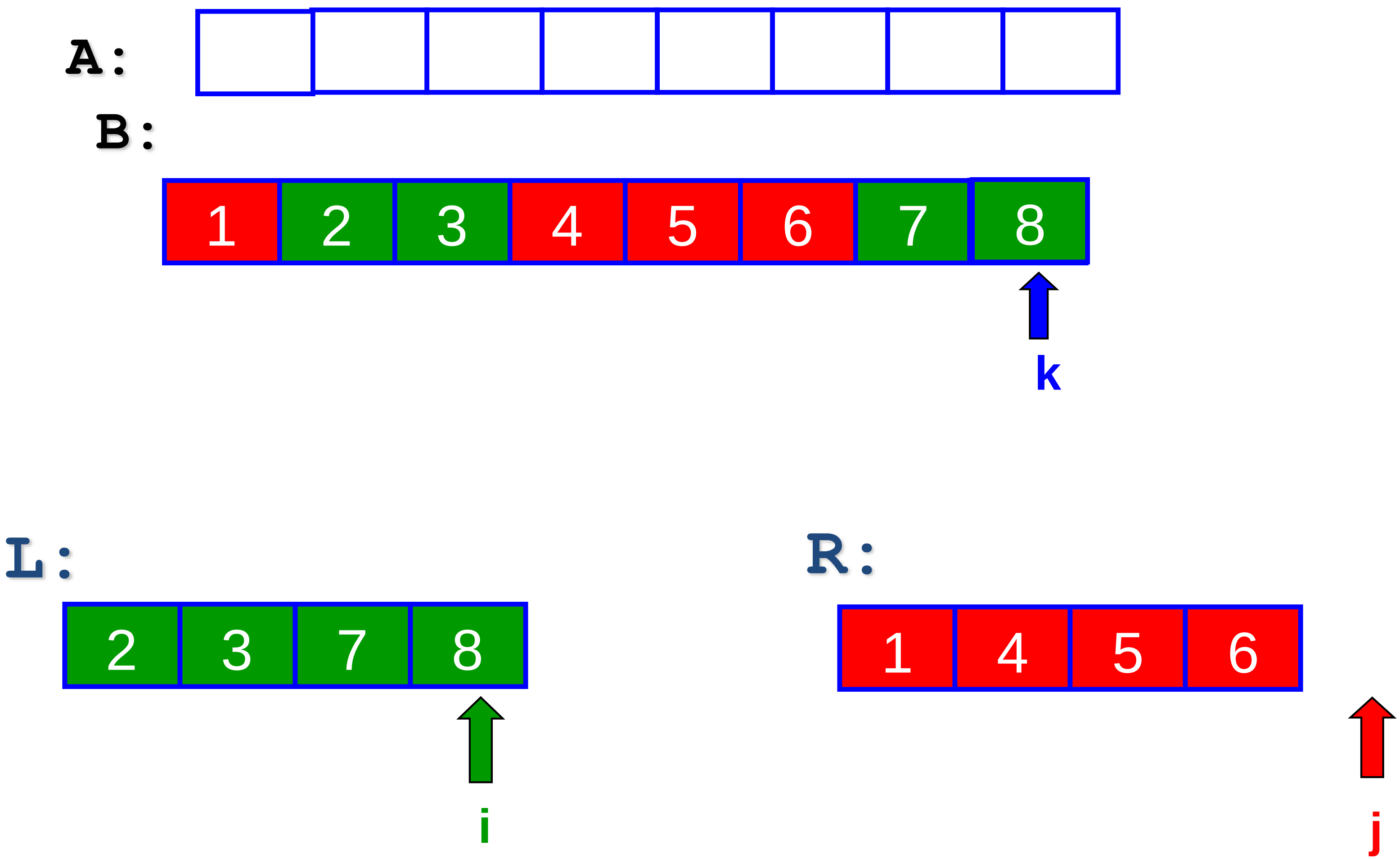
Merge-Sort: Merge Example



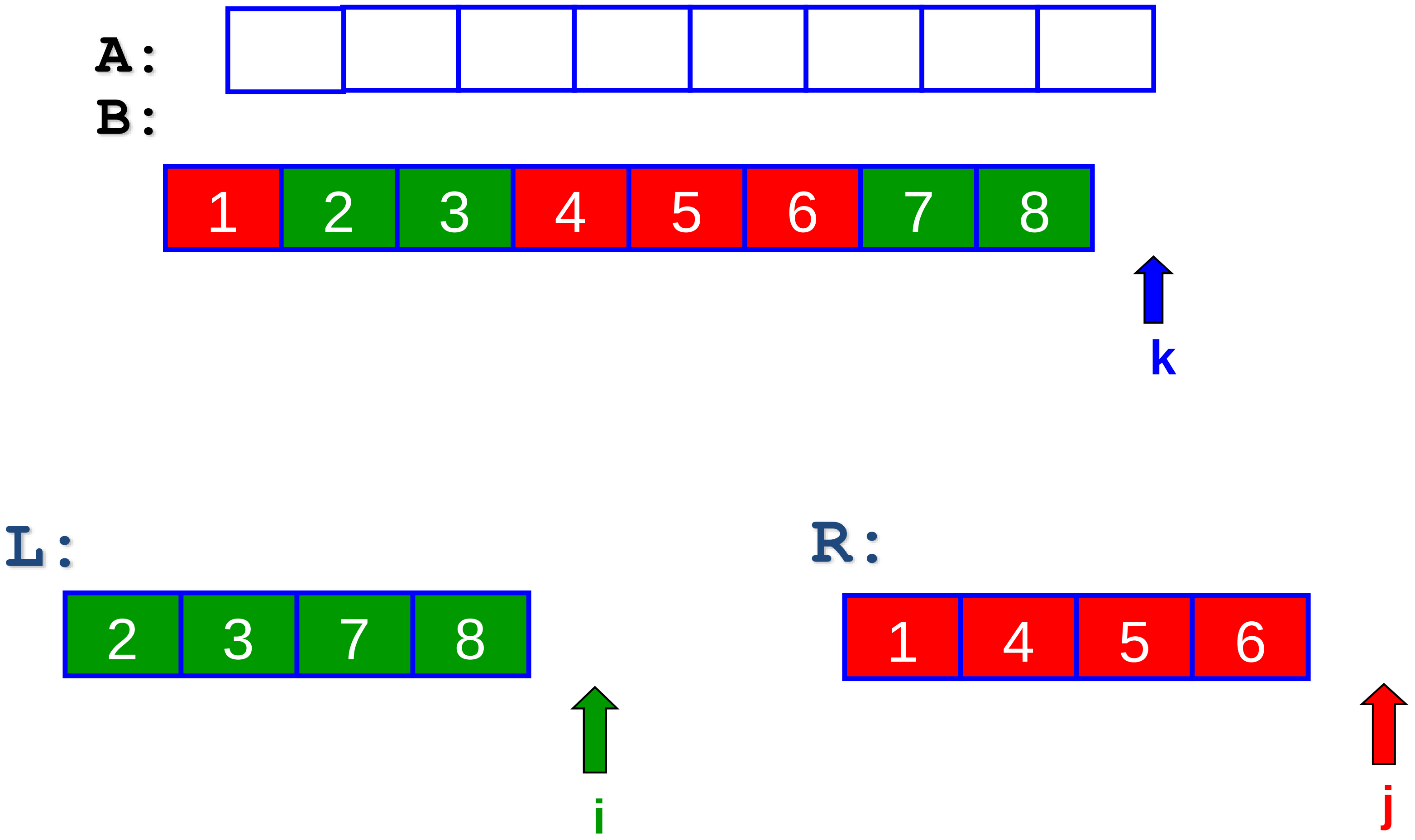
Merge-Sort: Merge Example



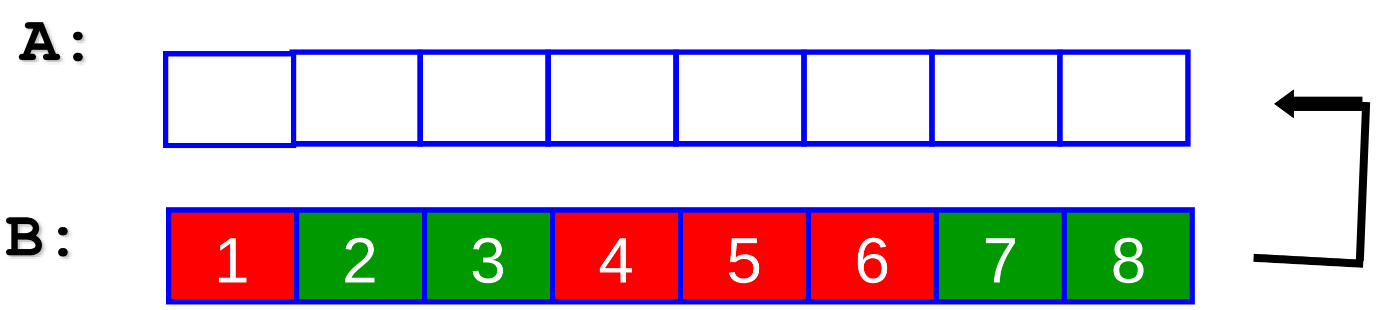
Merge-Sort: Merge Example



Merge-Sort: Merge Example

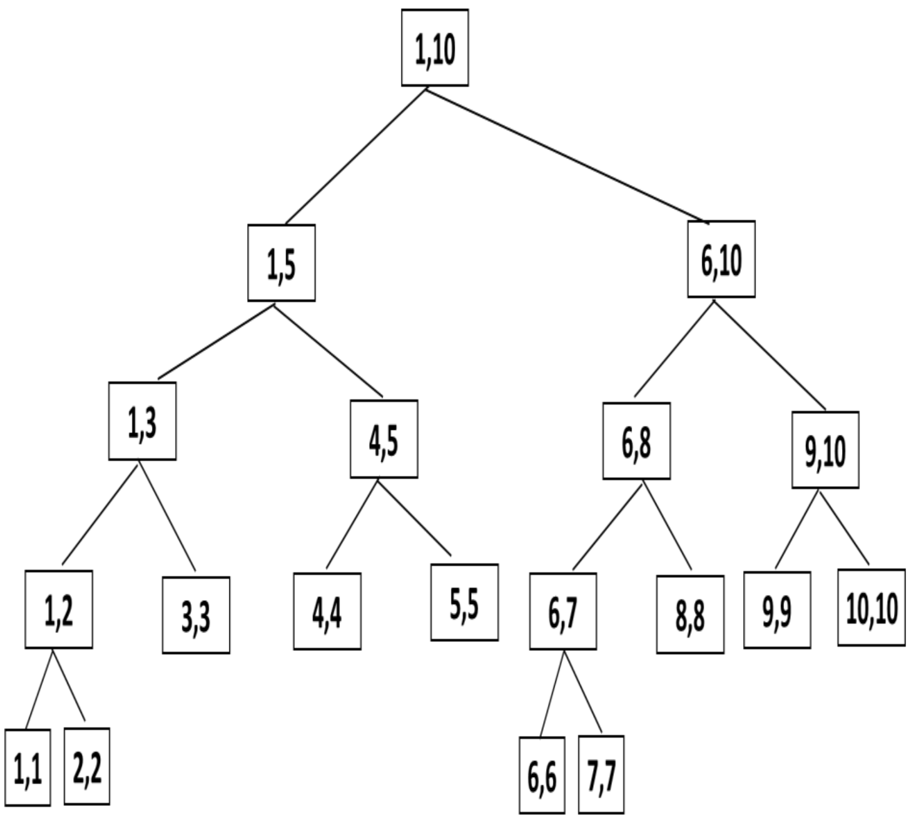


Merge-Sort: Merge Example

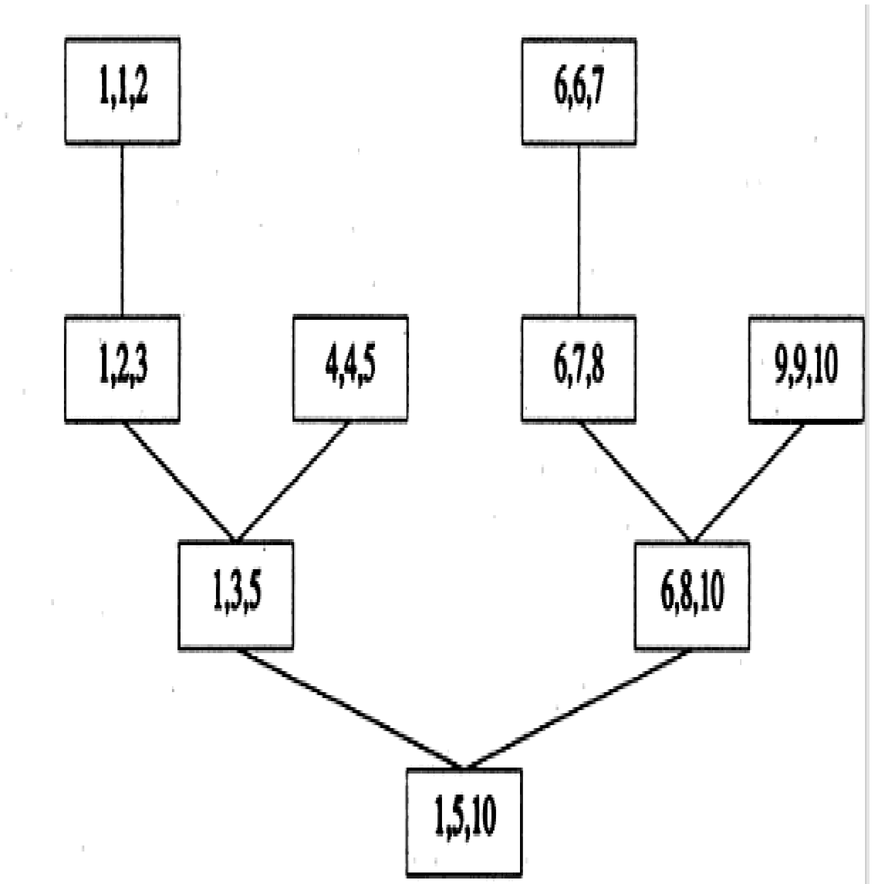


Merge-Sort: Merge Example

Tree calls for Recursive Merge Sort



Tree calls for Merge Operation



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)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

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

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

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

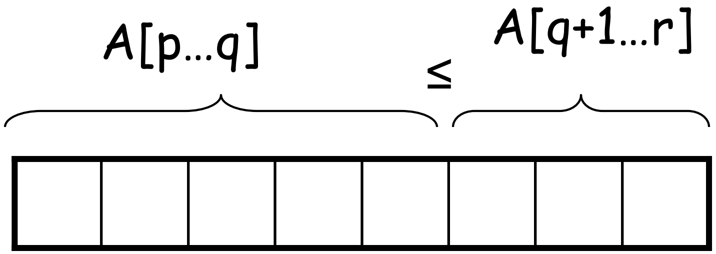
$=O(n\log n)$

Topics

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

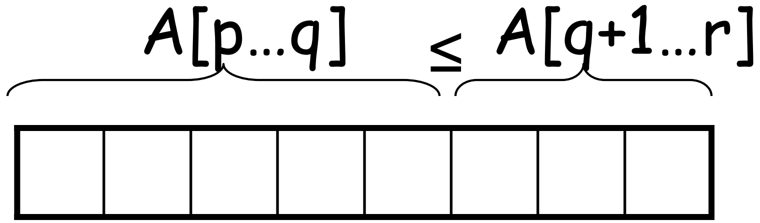
Quick Sort

- Sort an array A[p…r]



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

Quick Sort



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

Quick Sort

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

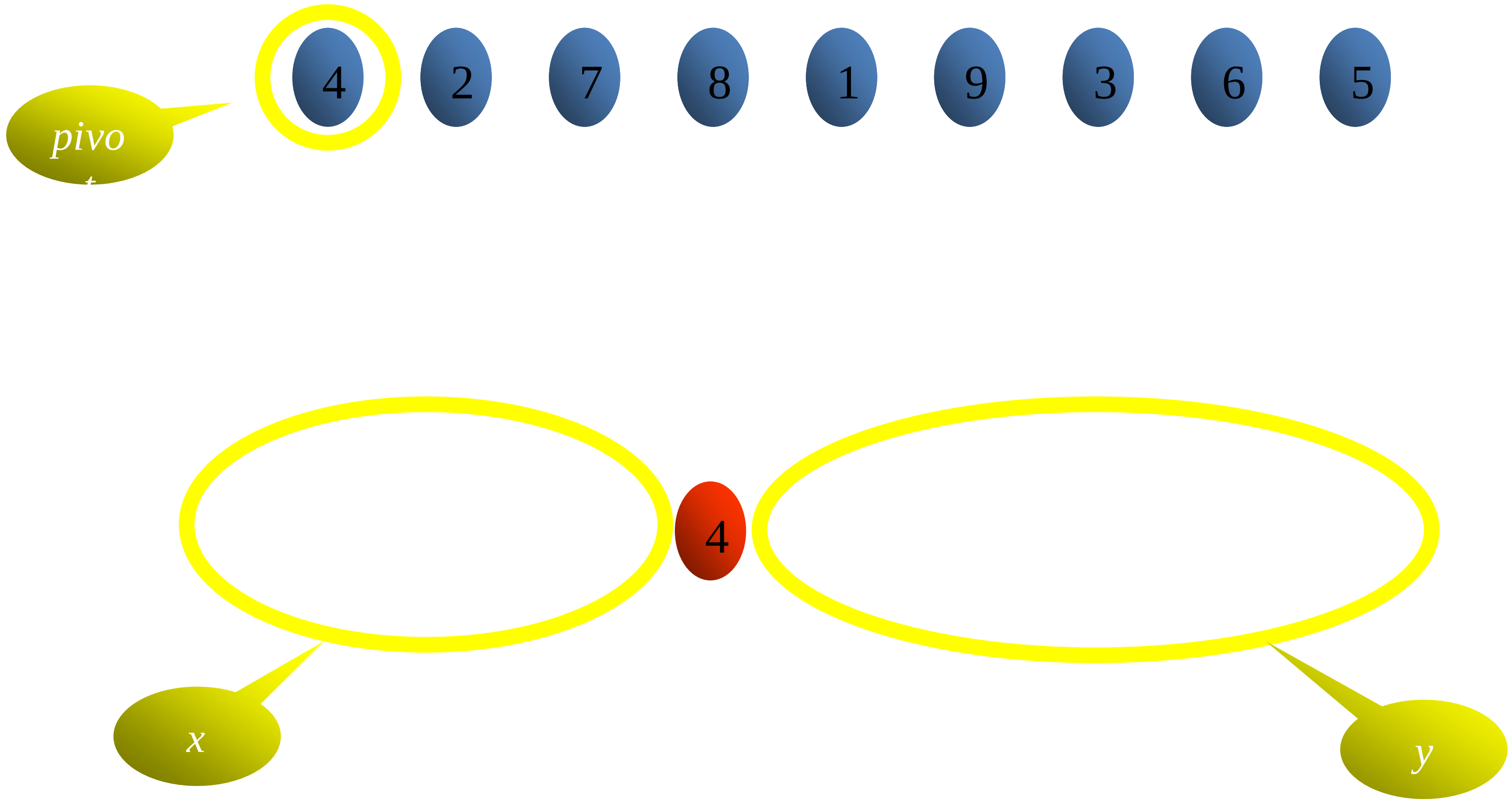
    FirstPart, which contains all elements < pivot

    SecondPart, which contains all elements > pivot

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

Quick Sort

pivot divides a into two sublists x and y.



Quick Sort

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 condition satisfied 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 the 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		

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)

Algorithm for Quick Sort

Algorithm QuickSort(low,high)

```
//Sorts the elements a[low],.....,a[high] which resides in the global array a[1:n] into
    //ascending order;
// a[n+1] 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);
                // j is the position of the partitioning element.
              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+1;  j:= h;
        while( i < j ) do
        {

                        while( a[ i ] < pivot ) do
                                i++;
                        while( a[ j ]  > pivot ) do
                                j--;

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

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

Algorithm for Quick Sort

```
Algorithm interchange (x,y )
{
        temp=a[x];
        a[x]=a[y];
        a[y]=temp;
}
```

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.

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

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

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

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

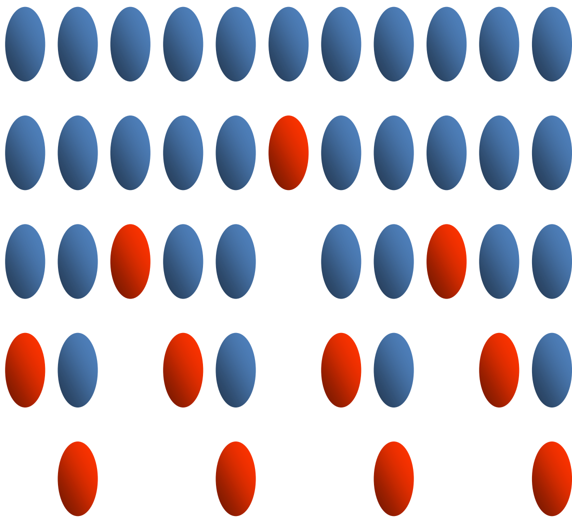
...

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

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

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

$=O(n\log n)$

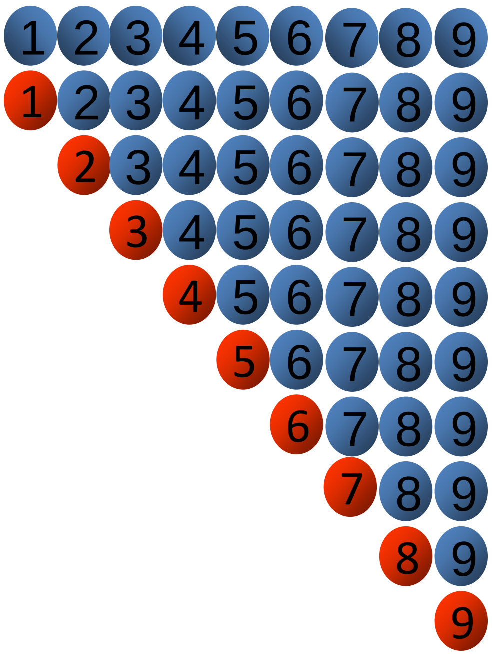


Time Complexity of Quick Sort: Worst Case

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.

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

$= O(n^2)$



Time Complexity of Quick Sort: Average Case

It occurs only if each partition divides the list into two sub lists such that both the sub lists are of random sizes less than n.

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

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

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

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

Problem and Straight forward approach

- Find the maximum and minimum items in a set of n element.

```
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];
    }
}
```

- StraightMaxMin requires 2(n - 1) element comparison ins the best, average, and worst cases.

Modified Approach

- An immediate improvement is possible by realizing that the comparison(a[i] < min) is necessary only when (a[i] > max) is false.

- Hence we can replace the contents of the for loop by

$$\begin{array}{l} \textbf{if } (a[i] > max) \textbf{ then } max := a[i]; \\ \textbf{else if } (a[i] < min) \textbf{ then } min := a[i]; \end{array}$$

Time Complexity of Modified Approach

- Best case:
  - when the elements are in increasing order.
  - The number of element comparisons is n-1.
- Worst case:
  - when the elements are in decreasing order.
  - The number of element comparisons is 2(n- 1).
- Average case:
  - Random Order
  - The number of element comparisons is less than 2(n -1).

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.

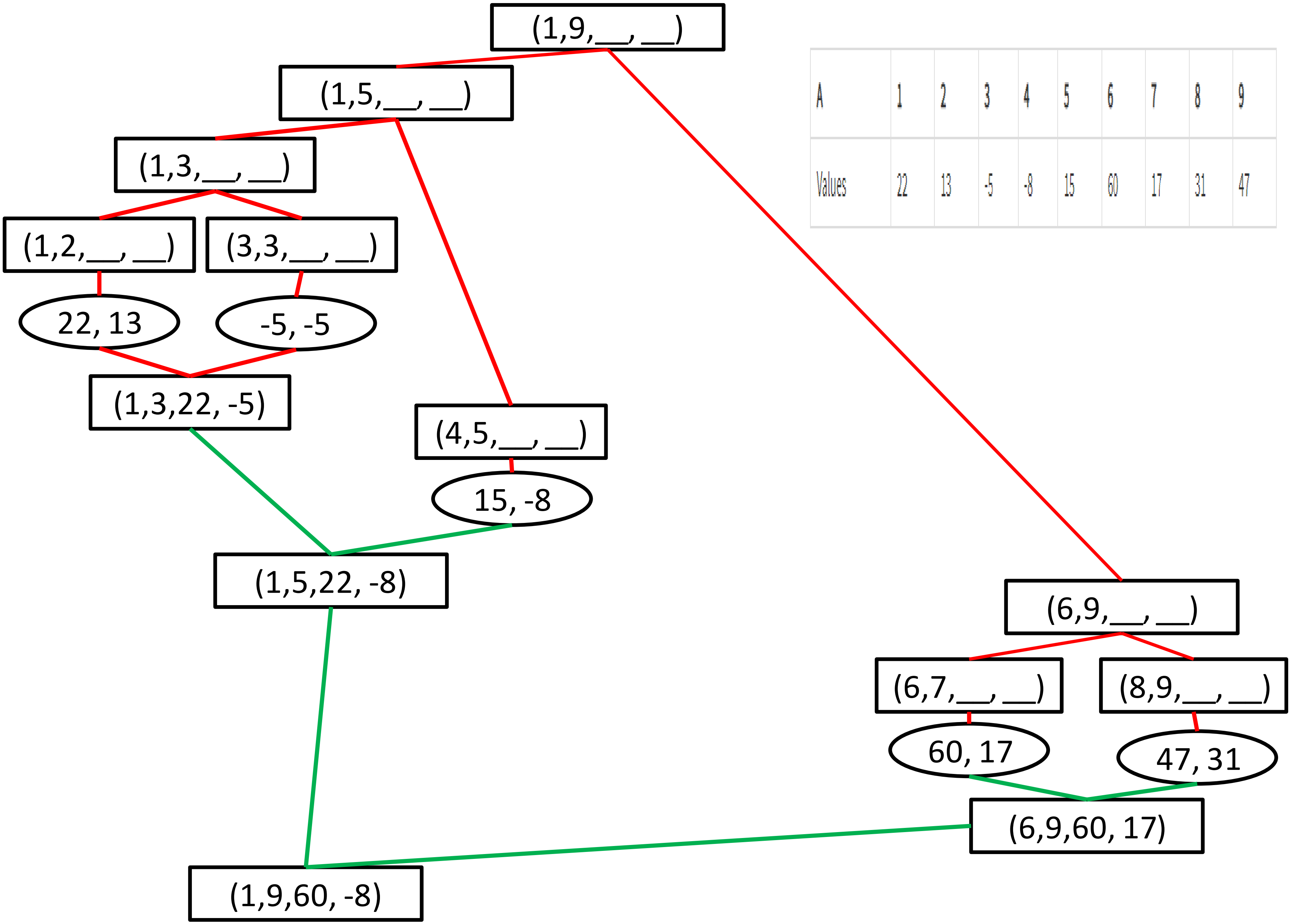
- Small(P):when n <=2.
  - If n=1, the maximum and minimum is a[i].
  - If n=2, the problem can be solved by making one comparison.

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

A	1	2	3	4	5	6	7	8	9
Values	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

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

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

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

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

$=16*T(n/16)+ 2^4+2^3 +2^2+2$

...

Assume $2^i =n$

$i=\log n$

$*x+x^2+...+x^n=(x^{n+1}-x)/(x-1)$

$=2^{i-1}*T(n/2^{i-1})+ 2^{i-1}+ 2^{i-2} ...+2^2+2$

$=2^{i-1} * T(2)+ 2^i -2$

$= 2^{\log n-1} *1+ 2^{\log n} -2$

$= n*2^{-1}+ n -2$

$=3/2* n -2$

$=O(n)$

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 (){
  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];
                                }
  }}
```

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,

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

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

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

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

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

- 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:

$$T(n)= \begin{cases} c_1 & \text{if } n<=2 \\ 8T(n/2)+ c_2n^2 & \text{if } n>2 \end{cases}$$

$$T(n) = 8T(n/2) + c_2 n^2$$

$$= 8\left[8T(n/4) + c_2(n/2)^2\right] + c_2 n^2$$

$$= 8^2\, T(n/4) + c_2 2n^2 + c_2 n^2$$

$$= 8^2\left[8T(n/8) + c_2(n/4)^2\right] + c_2 2n^2 + c_2 n^2$$

$$= 8^3\, T(n/8) + c_2 4n^2 + c_2 2n^2 + c_2 n^2$$

$$\vdots$$

$$= 8^k T(1) + \underbrace{\ldots\ldots\ldots\ldots\ldots + c_2 4n^2 + c_2 2n^2 + c_2 n^2}$$

$$= 8^{\log_2 n} c_1 + c\, n^2$$

$$= n^{\log_2 8} * c1 + c\, n^2$$

$$= n^3\, c_1 + cn^2$$

$$= O(n^3)$$

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

- 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 $M_1, M_2, M_3, M_4, M_5, M_6$, and M_7, then cij's are calculated using these matrices.

Formulas for Strassen's Algorithm

$$M_1 = (A_{11} + A_{22}) * (B_{11} + \mathbf{B_{22}})$$
$$M_2 = (A_{21} + A_{22}) * B_{11}$$
$$M_3 = A_{11} * (B_{12} - \mathbf{B_{22}})$$
$$M_4 = A_{22} * (B_{21} - \mathbf{B_{11}})$$
$$M_5 = (A_{11} + A_{12}) * \mathbf{B_{22}}$$
$$M_6 = (A_{21} - A_{11}) * (B_{11} + \mathbf{B_{12}})$$
$$M_7 = (A_{12} - A_{22}) * (B_{21} + \mathbf{B_{22}})$$

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

$$\mathbf{C_{11} = M1 + M4 - M_5 + M_7}$$
$$\mathbf{C_{12} = M_3 + M_5}$$
$$\mathbf{C_{21} = M_2 + M_4}$$
$$\mathbf{C_{22} = M_1 + M_3 - M_2 + M_6}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

The resulting recurrence relation for T(n) is

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

$T(n)= 7T(n/2) +c_2n^2$

$= 7[7T(n/4)+c_2*(n/2)^2] + c_2n^2 \quad =7^2 * T(n/4) + c_2n^2[7/4+1]$

$= 7^2[7T(n/8)+c_2*(n/4)^2] + c_2n^2[7/4+1]$

$=7^3 * T(n/8) + c_2n^2[(7/4)^2+(7/4)+1]$

.....

$= 7^i * T(n/ 2^i) + c_2n^2[(7/4)^{i-1}+...+(7/4)+1]$

$= 7^{\log n} \ T(1) + c_2n^2 \ [(7/4)^i-1]/[7/4-1]$

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

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

$= 7^{\log n} c_1 \ +c_2 n^2 (7/4)^{\log n}$

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

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

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

$=2 n^{\log_2 7}$

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

Example Strassen's Matrix Multiplication

- multiply two 2×2 matrices

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 3 & 5 \\ 1 & 4 \end{pmatrix} = \begin{pmatrix} 1*3+2*1=5 & 1*5+2*4=13 \\ 3*3+4*1=13 & 3*5+4*4=31 \end{pmatrix}$$

How many multiplications and additions did we need?

Example Strassen's Matrix Multiplication

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{pmatrix}$$

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

$$= \begin{pmatrix} 35-8-12-10 & 1+12 \\ 21-8 & 35+1-21+16 \end{pmatrix}$$

$$m_1 = (a_{00}+a_{11})*(b_{00}+b_{11}) = 35$$

$$m_2 = (a_{10}+a_{11})*b_{00} = 21$$

$$m_3 = a_{00}*(b_{01}-b_{11}) = 1$$

$$m_4 = a_{11}*(b_{10}-b_{00}) = -8$$

$$m_5 = (a_{00}+a_{01})*b_{11} = 12$$

$$m_6 = (a_{10}-a_{00})*(b_{00}+b_{01}) = 16$$

$$m_7 = (a_{01}-a_{11})*(b_{10}+b_{11}) = -10$$

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

Topics

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