Unit –I
Introduction to Algorithm

Amrita Naik

Algorithm

- An algorithm is finite set of instruction if followed accomplishes a particular task.
- Ex:

```
Algorithm  checkEven(n)
{
      If(n%2==0)
            Print “Even”
      Else
            Print”Odd”
}
```

Algorithm Example

- Step 1: Take the brush
- Step 2: Put the paste on it.
- Step 3: Start brushing
- Step 4: Rinse
- Step 5: Wash
- Step 6: Stop

Criteria to design efficient Algorithm

- 1.Input
- 2.Output
- 3.Definiteness
- 4.Finiteness
- 5.Effectiveness

Performance Analysis

- 1.Space Complexity: is the amount of memory it needs to run to completion.
  - Represented by S(P)
- 2.Time Complexity: is the amount of computer time it needs to run to completion.
  - Represented by T(P)

Space Complexity(S(P))

- The space needed by each algorithm is sum of:
- (i)Fixed part: Space for code, fixed variable and constants - C
- (II)Variable part: Space needed by reference variable ,recursion stack - Sp
- S(P)= C+Sp

Example1 for space complexity

```
Algorithm abc(a,b,c)
{
 return a+b+b*c+(a+b-c)/(a+b)
}
```

- C=3
- Sp=0
- S(P)=C+Sp=3+0=3

Example2 for space complexity

```
Algorithm sum(a,n)
{
 s=0
for i=1 to n do
    s=s+a[i]
Return s
}
```

- C=3
- Sp=n
- S(P)=C+Sp=3+n

Example3 for space complexity

```
Algorithm  RSum(a,n)
{
If(n<=0) then
    return 0;
Else
    return Rsum(a,n-1)+a[n];
}
```

- $C=0$
- $S_p=3(n+1)$
- $S(P)=0+S_p=0+3(n+1)$

Time Complexity

- 2 methods:
- 1)using Global variable
- 2)using Tabular method

1)Calculate Time complexity using Count variable

```
Algorithm sum(a,n)
{
 s=0;
 count=count+1
for i=1 to n do
{
    count=count+1
    s=s+a[i]
    count=count+1
}
count=count+1
count=count+1
Return s
}
```

T(n)=No of times count is incremented=1+n n+1+1

T(n)=2n+3

2)Using tabular method

```
Algorithm sum(a,n)
{
 s=0
 for i=1 to n do
     s=s+a[i]
 return s
}
```

s/e	frequency	Total steps
0	-	0
0	-	0
1	1	1
1	n+1	n+1
1	n	n
1	1	1
0	-	0
		2n+3

Another Example

```
Algorithm  Rsum(a,n)
{
If (n<=0) then
  return 0
Else
  return Rsum(a,n-10)+a[n]
}
```

s/e	F(n=0)	F(n>0)	TS(n=0)	TS(n>0)
0	-	-	0	0
0	-	-	0	0
1	1	1	1	1
1	1	0	1	0
0	-	-	-	-
1+X	0	1	0	1+X
0	-		0	
			2	2+X

T(n)=2+2+2…n+1 times=2(n+1)

Another Example

```
Algorithm  Add(a,b,c,m,n)
{
for i=1 to m do
   for j=1 to n do
      c[i,j]=a[I,j]+b[I,j]
}
```

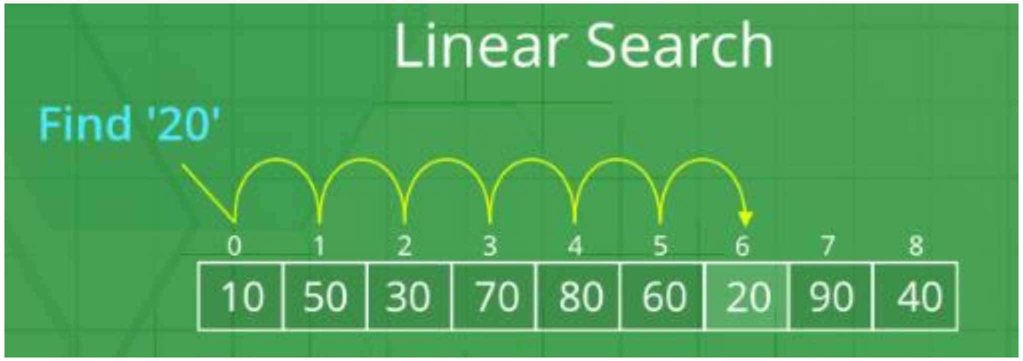
s/e	Frequency	Total count
0	-	0
0	-	0
1	m+1	m+1
1	m(n+1)	mn+m
1	mn	mn
0	-	0
		2mn+2m +1

T(n)=2mn+2m+1

An Example of Linear Search Algorithm

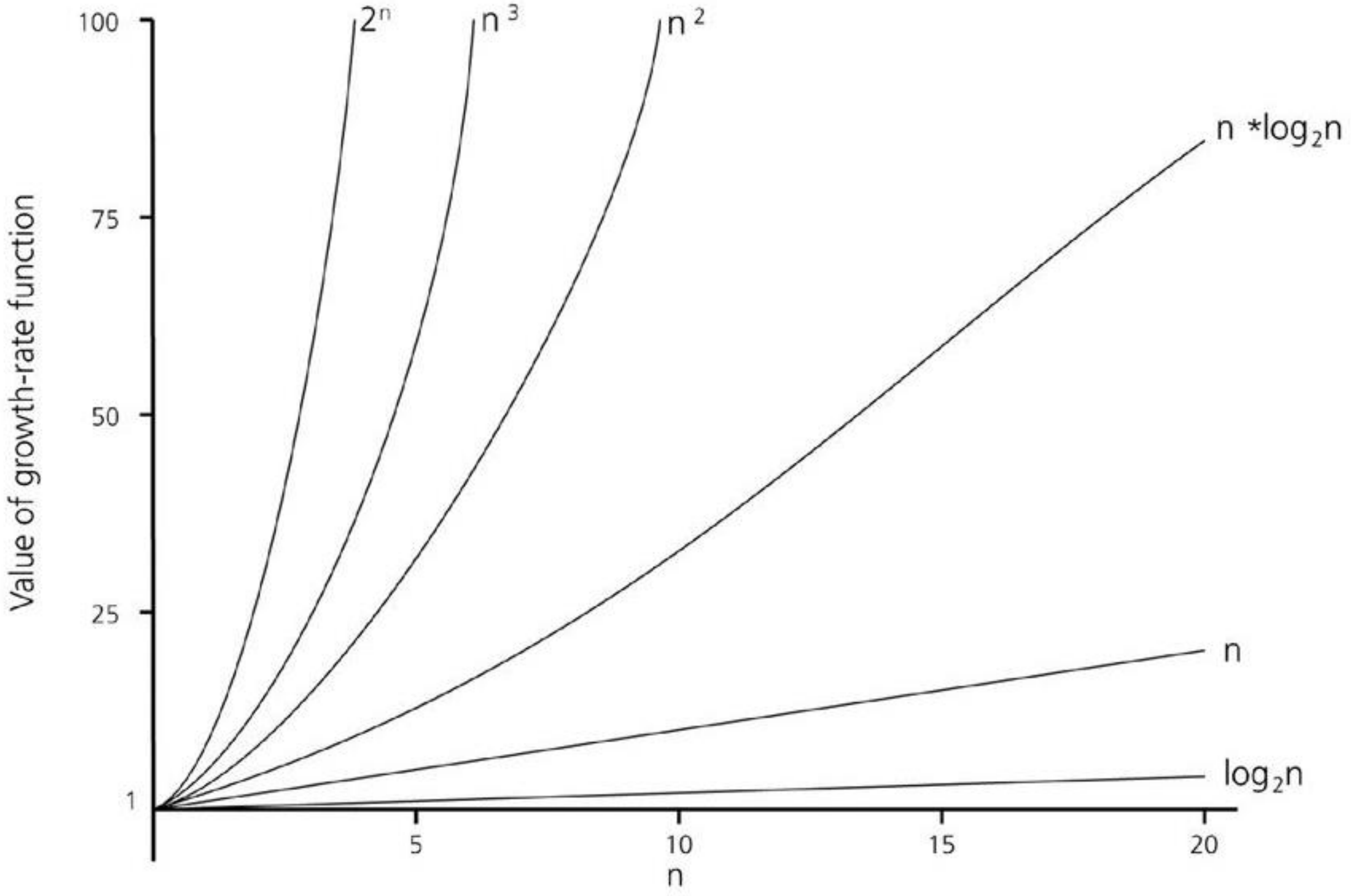
- Start from the leftmost element of array and one by one compare x with each element of array
- If x matches with an element, return the index.
- If x doesn’t match with any of elements, return -1.
- Ex: Consider an array A={10,50,30,70,80,60,20,90,40}

Linear Search



The best time complexity of linear search is O(1)
The average time complexity of the above algorithm is O(n)
The worst time complexity of the above algorithm is O(n)

Comparison of growth function



Example of Time Complexity: $T(n)=n^2+2n+3$

Asymptotic Notations

- 1.Big Oh(O) : "f(n) is O(g(n))" iff for some constants c and N_0, $f(n) \leq cg(n)$ for all $N > N_0$
- 2.Big Omega(Ω):f(n)= Ω (g(n)), iff for some constants c and N_0, $f(n) \geq cg(n)$ for all $N > N_0$
- 3.Big Theta(ϴ):iff f(n) is O(g(n)) and f(n) is Ω(g(n))

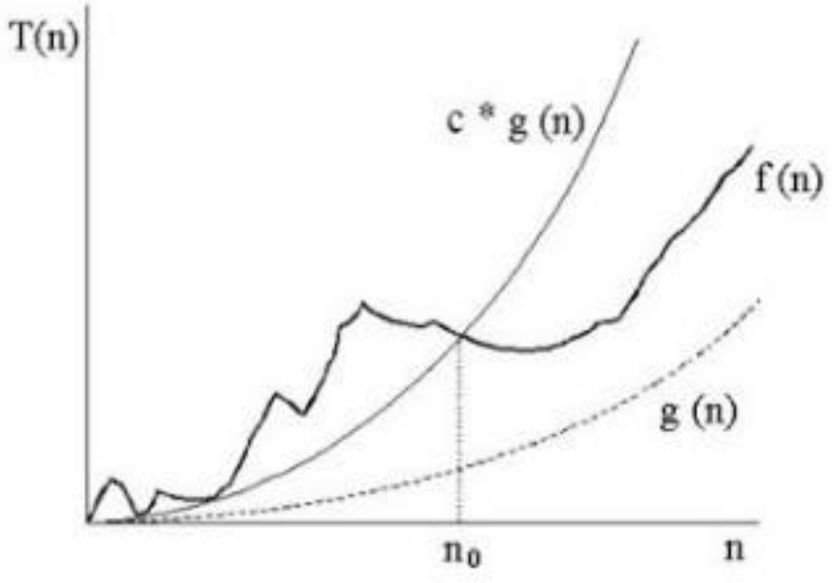
1.Big Oh(O)

- Big-Oh is about finding an *asymptotic upper bound*.
- Formal definition of Big-Oh:

  $f(N) = O(g(N))$, if there exists positive constants c , N_0 such that

  $f(N) \le c \cdot g(N)$ for all $N \ge N_0$.

  - We are concerned with how f grows when N is large.
    - not concerned with small N or constant factors
  - Lingo: "$f(N)$ grows no faster than $g(N)$."

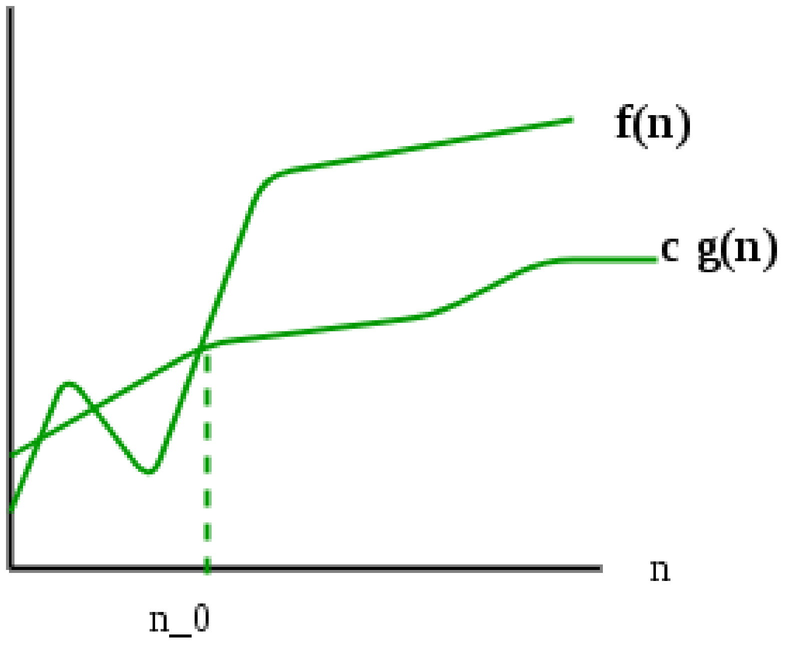


Big Oh Example

- Example: P.T 2n + 10 is O(n)
  - $2n + 10 \leq cn$
  - $2+10/n \leq c$ [Divide both sides by n]
  - Pick c >=12 and n0 = 1

Big-Omega

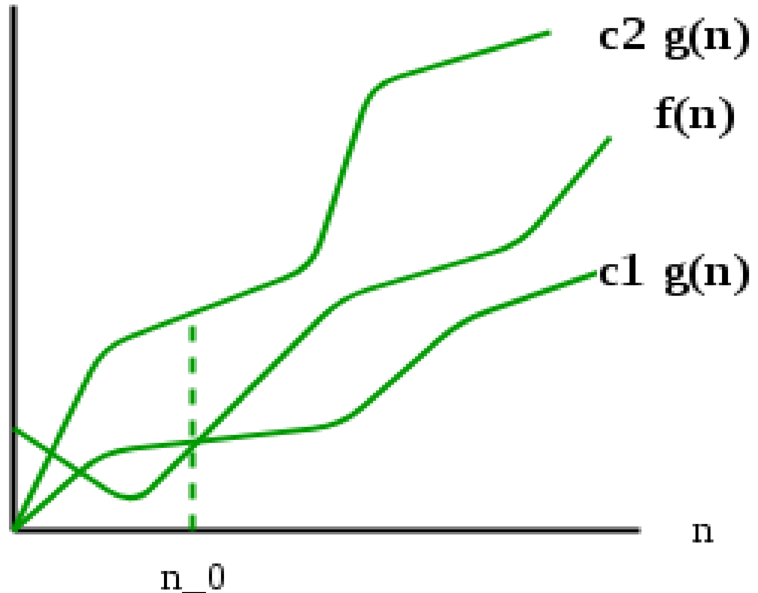
- f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$



f(n) = Omega(g(n))

Big-Theta

- ❑f(n) is $\Theta(g(n))$ if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$



f(n) = theta(g(n))

Different Type of Algorithm

- 1.Divide and Conquer
- 2.Greedy
- 3.Dynamic
- 4.Backtracking
- 5.Branch and Bound

1.Divide and Conquer

- 1. Divide the problem into two or more smaller subproblems.
- 2. Conquer the subproblems by solving them recursively.
- 3. Combine the solutions to the subproblems into the solutions for the original problem.

General Method

```
Divide_Conquer(problem  P)
{
        if Small(P)
            return S(P);
        else
        {
            Divide P into smaller instances P1, P2, ...,
            Pk;
            Apply Divide_Conquer  to each of these
            subproblems;
            Return Combine(Divide_Conque(P1),
            Divide_Conque(P2),...,  Divide_Conque(Pk));
        }
}
```

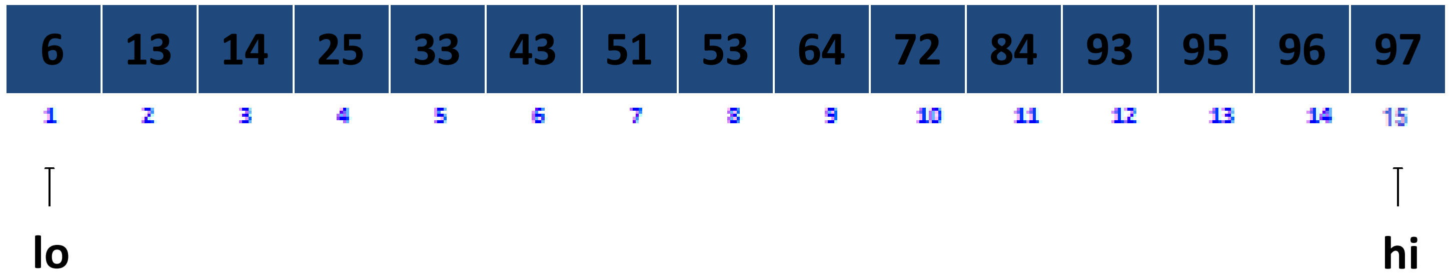
1.Binary Search

- Fastest searching algorithm which uses Divide and Conquer approach.
- Search a sorted array by repeatedly dividing the search interval in half.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise narrow it to the upper half.
- Repeatedly check until the value is found or the interval is empty.

```
Algorithm BinarySearch(a, n,x)
{
low=1, high=n
while(low<=high){
    mid=L(low+high)/2 ˩
    if(x<a[mid])
        high=mid-1
    else
    {
        If(x>a[mid])
        low=mid+1
        else
        return mid
    }
  }
return "Not Found"
}
```

Binary Search

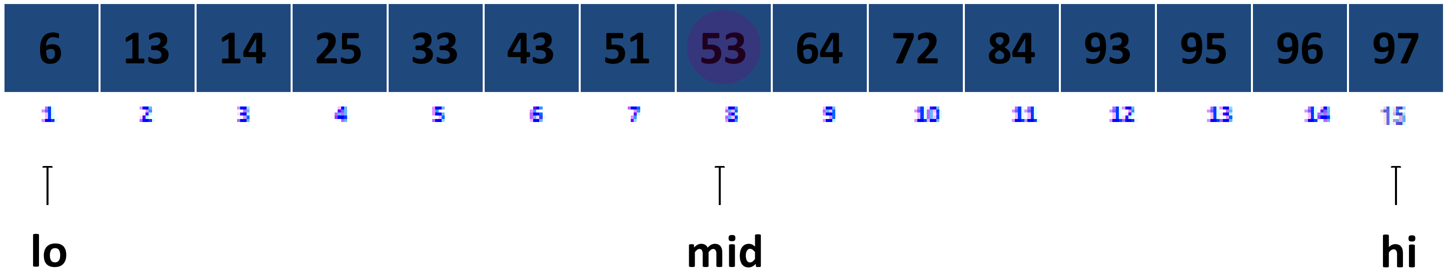
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



- Ex. Binary search for 33 in an array A={6,13,14,25,33,43,53,64,72,84,93,95,96,97}

Binary Search

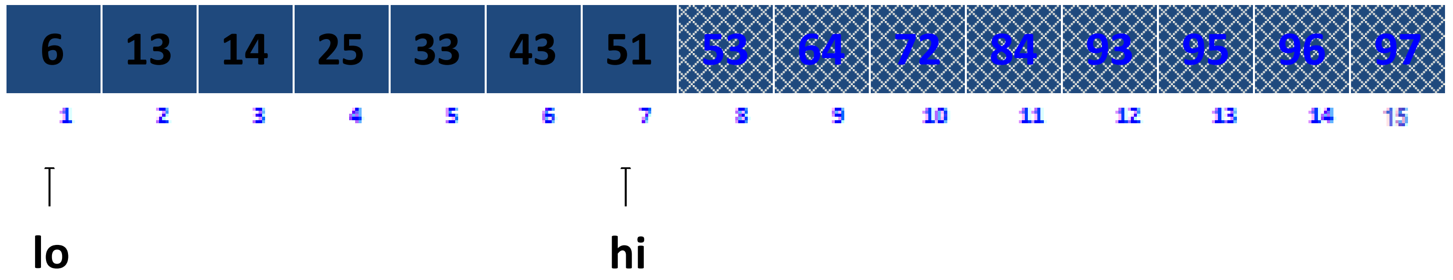
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

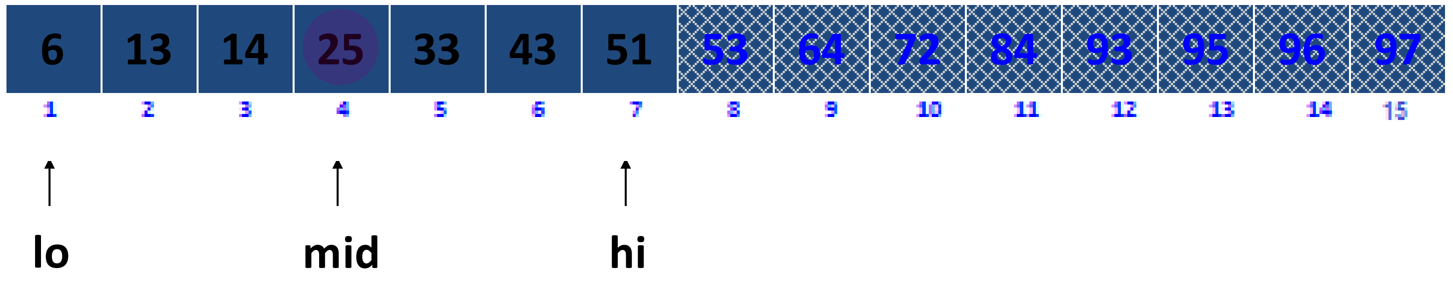
- Binary search. Given `value` and sorted array `a[]`, find index `I` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

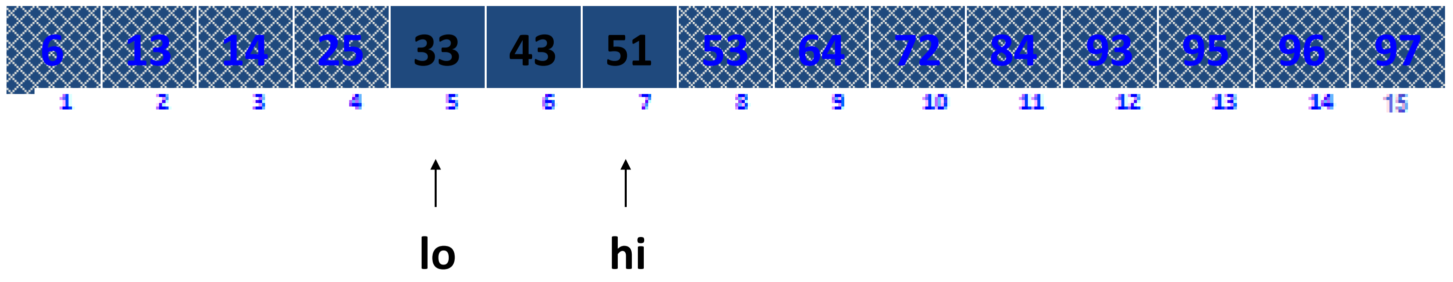
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

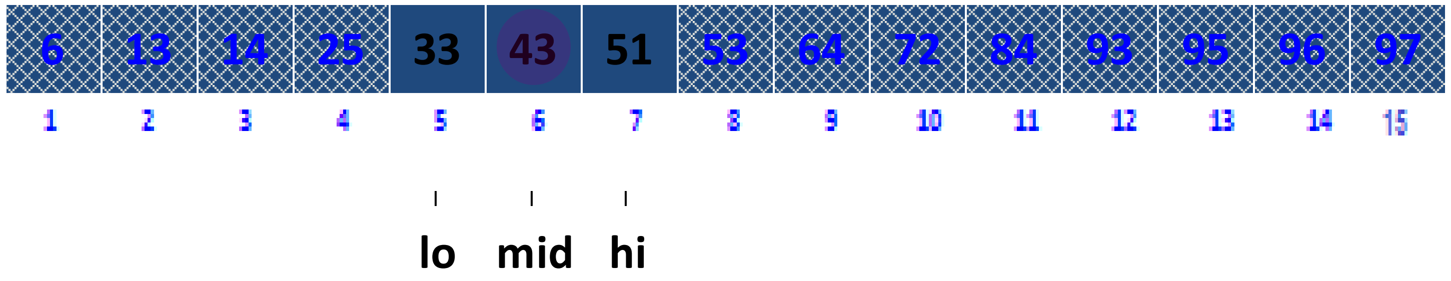
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

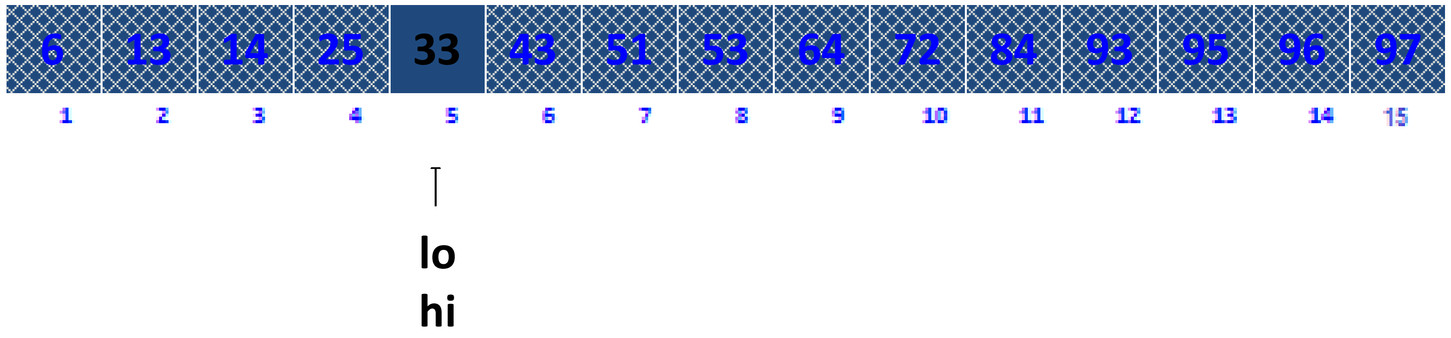
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

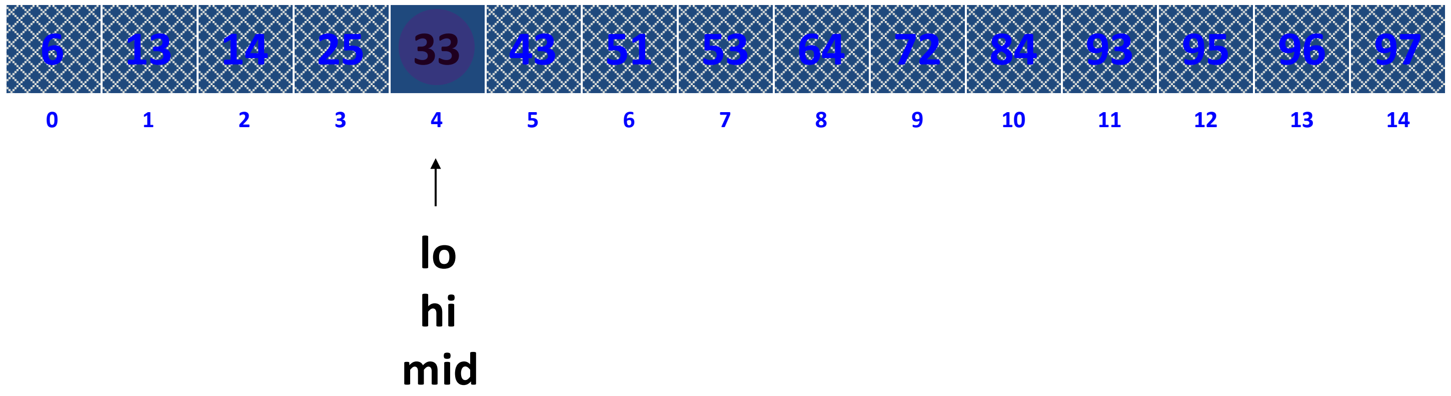
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

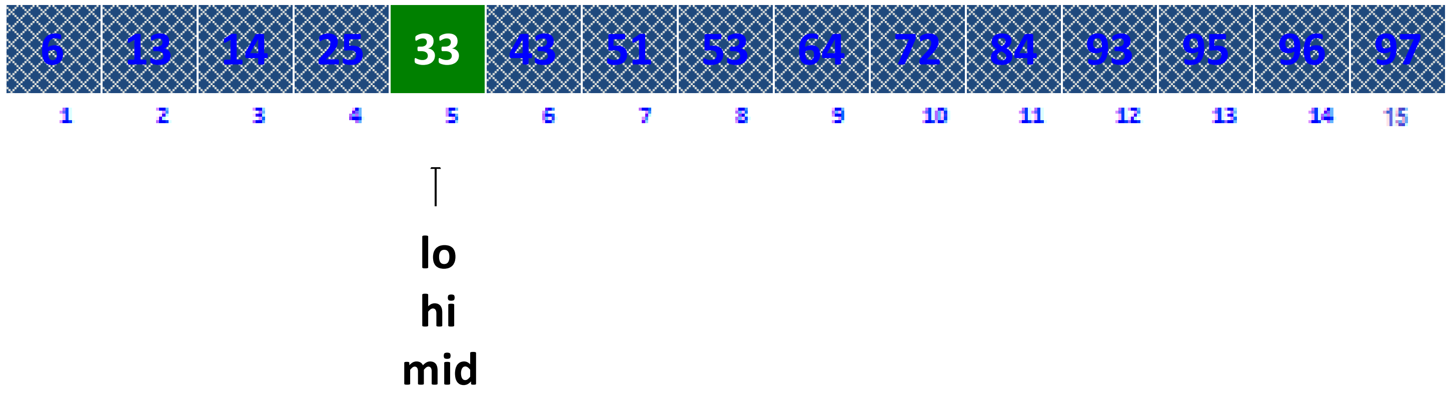
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



- Ex. Binary search for 33.

Time Complexity –Binary Search

- $T(n)=T(n/2)+f(n)$
- $\quad\quad =T(n/2)+c$
- [Base step: $T(1)=c$]

Recurrence Relation

- Is an equation that recursively describes a time complexity function for an algorithm given a base step.
- 4 ways to solve recurrence relation:
  - 1)Substitution
  - 2)Iteration
  - 3)Recursion tree
  - 4)Master's Theorem

1)Substitution

- 2 Steps:
- 1)We make a guess for the solution
- 2)We use mathematical induction to prove the guess is correct or incorrect.

Time Complexity of Binary Search using Substitution

- $T(n) = T(n/2) + c$
- $T(n)=aT(n/b)+f(n)$
  - We guess the solution as $T(n) = O(\text{Logn})$.
  - Now we use induction to prove our guess.
  - We need to prove that $T(n) <= c\text{Logn}$.
  - We can assume that it is true for values smaller than n.

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

$= T(n/2^i)+ ic = c\text{Log}(n/2^i) + ic$

$= c\text{Logn} - c\text{Log}2^i + ic$

$= c\text{Logn} - ci + ic = c\text{logn}$

$<= c\text{Logn}$

Example of Substitution

- $T(n) = 2T(n/2) + n$
  - We guess the solution as $T(n) = O(n\mathrm{Log}n)$.
  - Now we use induction to prove our guess.
  - We need to prove that $T(n) <= cn\mathrm{Log}n$.
  - We can assume that it is true for values smaller than n.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2c(n/2)\mathrm{Log}(n/2) + n \\
&= cn\mathrm{Log}n - cn\mathrm{Log}2 + n \\
&= cn\mathrm{Log}n - cn + n = cn\mathrm{Log}n - n(c-1) \\
&<= cn\mathrm{Log}n
\end{aligned}
$$

2)Iteration Method

Time Complexity for Binary Search

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

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

$=T(n/8)+c+c+c$

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

$=T(n/32)+5c$

$=T(n/2^5)+5c$

$=T(n/2^i)+ic$

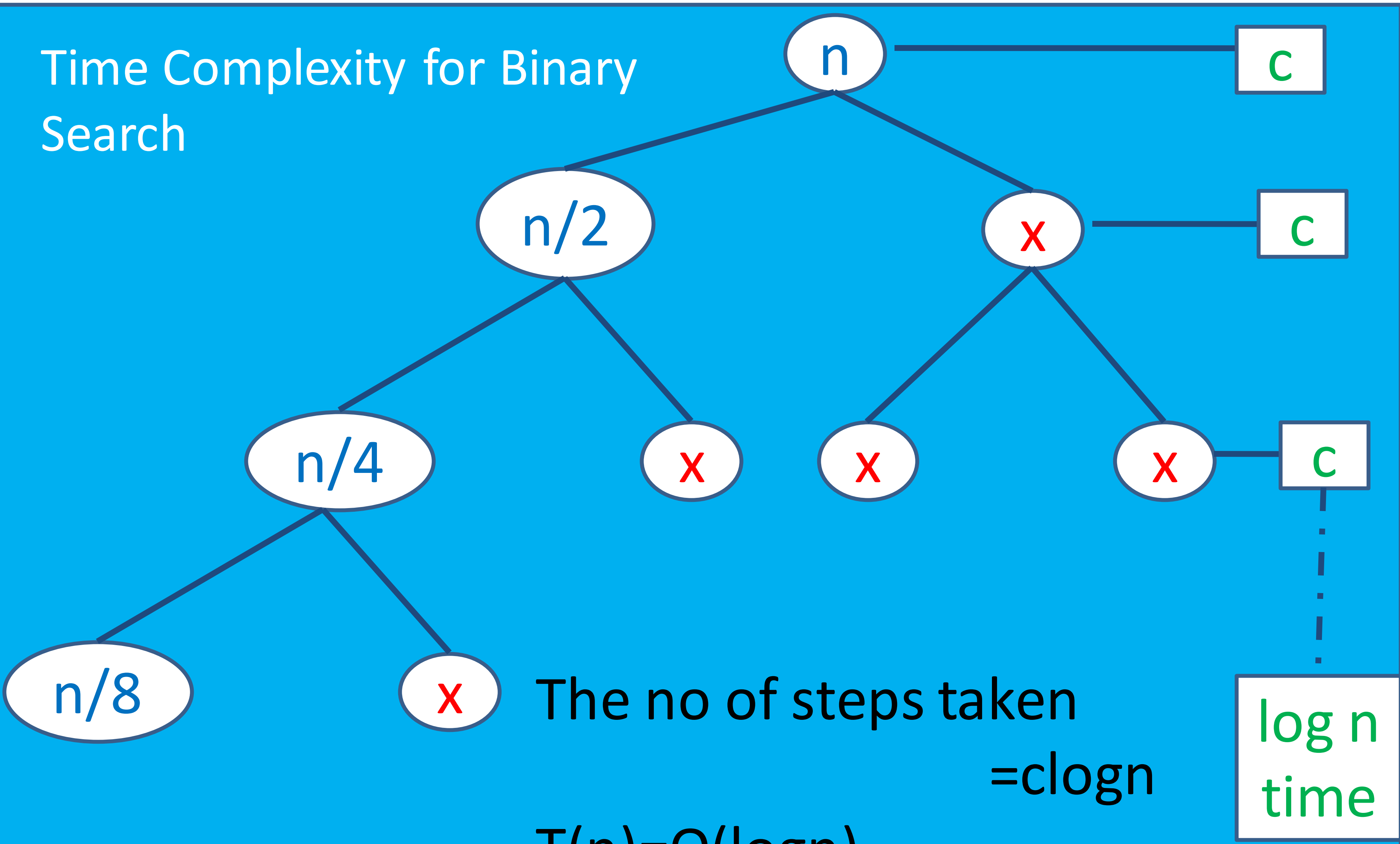
$=(c+c\log n)$ [when $n=2^i$,then $\log n=i$]

$T(n)>=c\log n$

$T(n)=\Omega(\log n)$

3)Recursion Method

Time Complexity for Binary Search



The no of steps taken

=clogn

T(n)=O(logn)

4)Master's Theorem

- Direct way of finding the solution , when the time complexity is following form:

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$
Here $f(n)$ is of the form n^d

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Time Complexity for Binary Search using Master's Theorem

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

  Here $a=1$, $b=2$, $d=0$

  $a=b^d$

  Therefore $T(n)=O(n^d \log n)$

  $=O(\log n)$

2)Another Example of Iteration

- T (n) = 1 **if** n=1

  = 2 T (n-1) **if** n>1

$T(n)=2 *(2*T(n-2))$

$=2*(2*(2*T(n-3)))$

$=2^{K}T(n-k)$

$n-k=1 \Rightarrow k=n-1$

$=2^{n-1}\ T(1)= 2^{n-1}$

2)Another Example of Iteration

- $T(n) = 2T(n-1)$
  $= 2[2T(n-2)]$
  $= 2^2T(n-2)$
  $= 4[2T(n-3)]$
  $= 2^3T(n-3)$
  $= 8[2T(n-4)]$
  $= 2^4T(n-4)$ ---------------------------- (Eq.1)

Repeat the procedure for i times $T(n) = 2^i T(n-i)$
Put $n-i=1$ or $i= n-1$ in (Eq.1)

$T(n) = 2^{n-1} T(1) = 2^{n-1} .1$ {T (1) =1 .....given}

$= 2^{n-1}$

2)Another Example of Iteration

- $T(n) = T(n-1) + 1$ and $T(1) = \theta(1)$.
- $= T(n-2)+1+1$
- $= T(n-3)+3$
- $= T(n-k)+k$
- Substitute $n-k=1$, $k=n-1$,
- $= T(1)+n-1=1+n-1$
- $= n$

2)Another Example of Iteration

- $T(n) = T(n-1) + 1$ and $T(1) = \theta(1)$.
- Solution: $T(n) = T(n-1) + 1$

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

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

$= T(n-4) + 4$

$= T(n-5) + 1 + 4$

$= T(n-5) + 5$

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

$T(n-k) = T(1) = \theta(1)$

$T(n) = \theta(1) + (n-1) = 1 + n - 1 = n = \theta(n)$.

3)Another Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$\begin{tikzcd}
 & n^2 \arrow[dl, dash] \arrow[dr, dash] & \\
T(n/4) & & T(n/2)
\end{tikzcd}$$

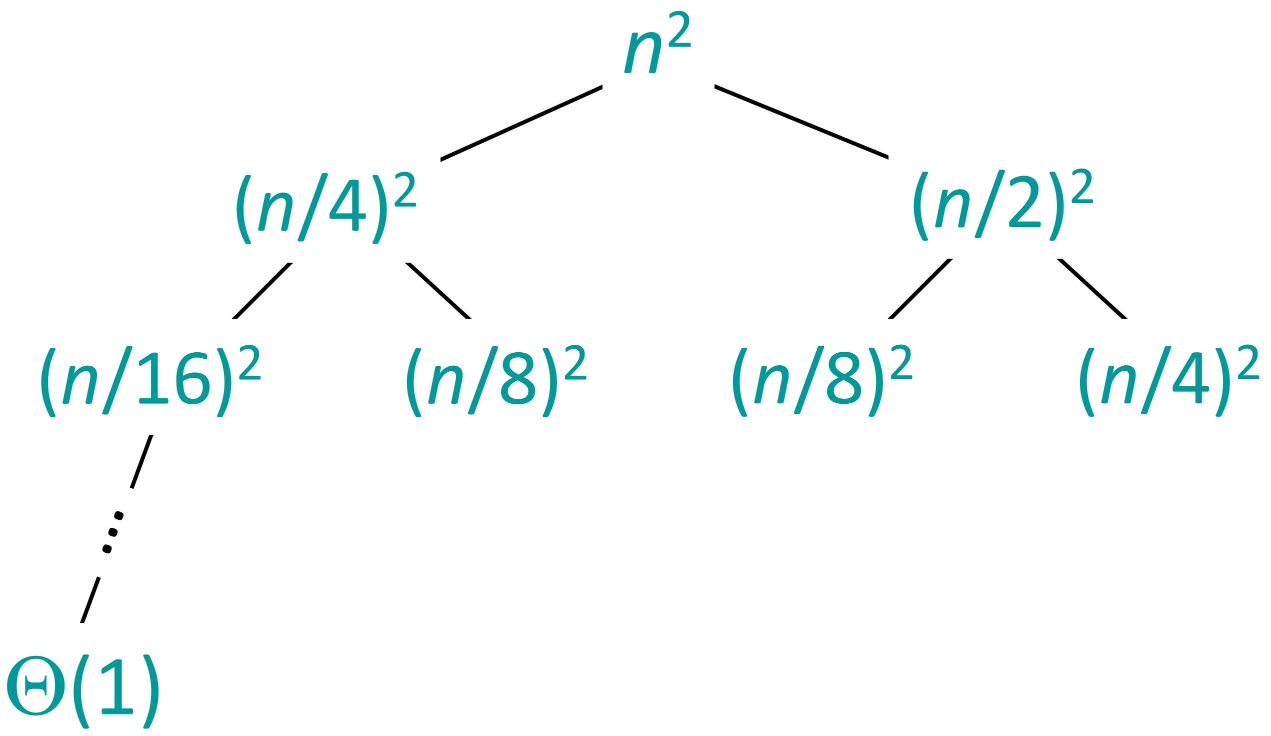
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$\begin{tikzcd}
 & & & n^2 \arrow[dll, no head] \arrow[drr, no head] & & & \\
 & (n/4)^2 \arrow[dl, no head] \arrow[dr, no head] & & & & (n/2)^2 \arrow[dl, no head] \arrow[dr, no head] & \\
T(n/16) & & T(n/8) & & T(n/8) & & T(n/4)
\end{tikzcd}$$

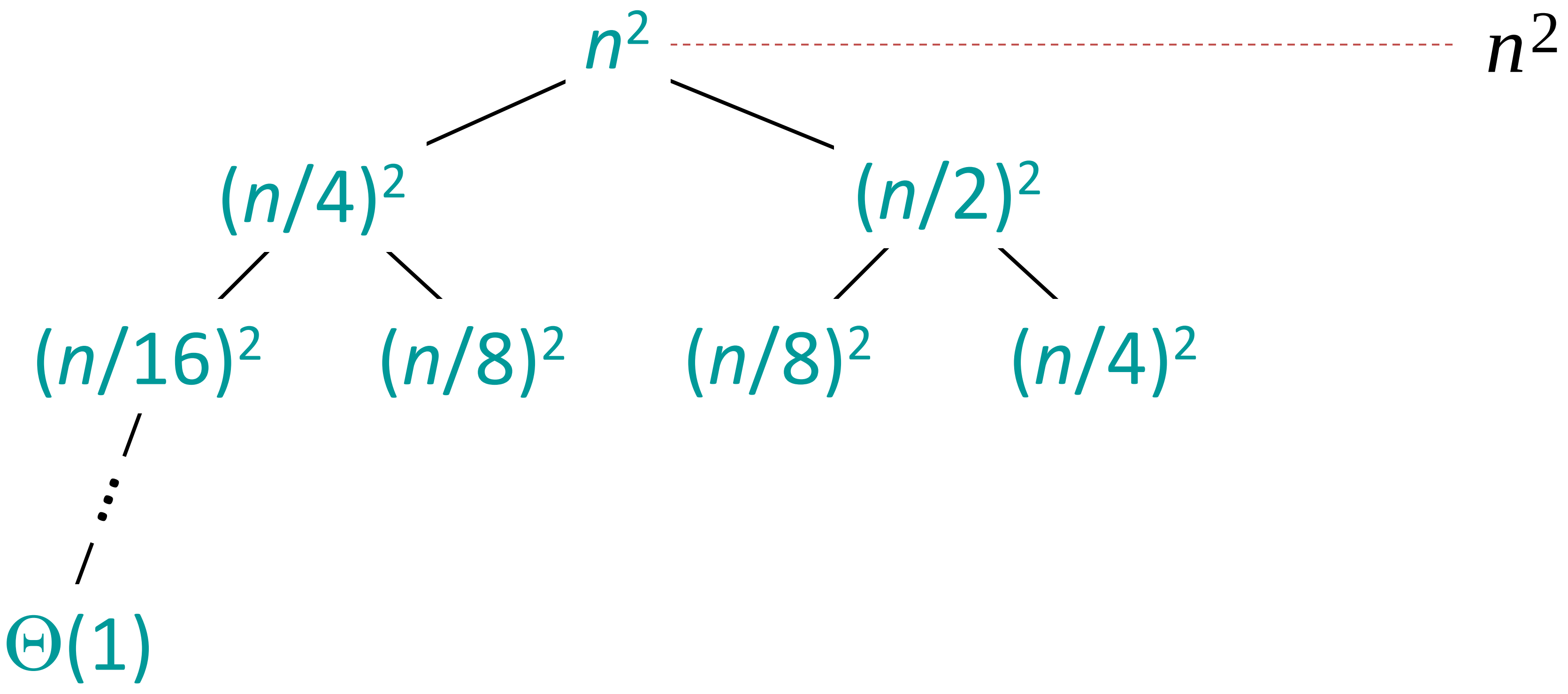
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



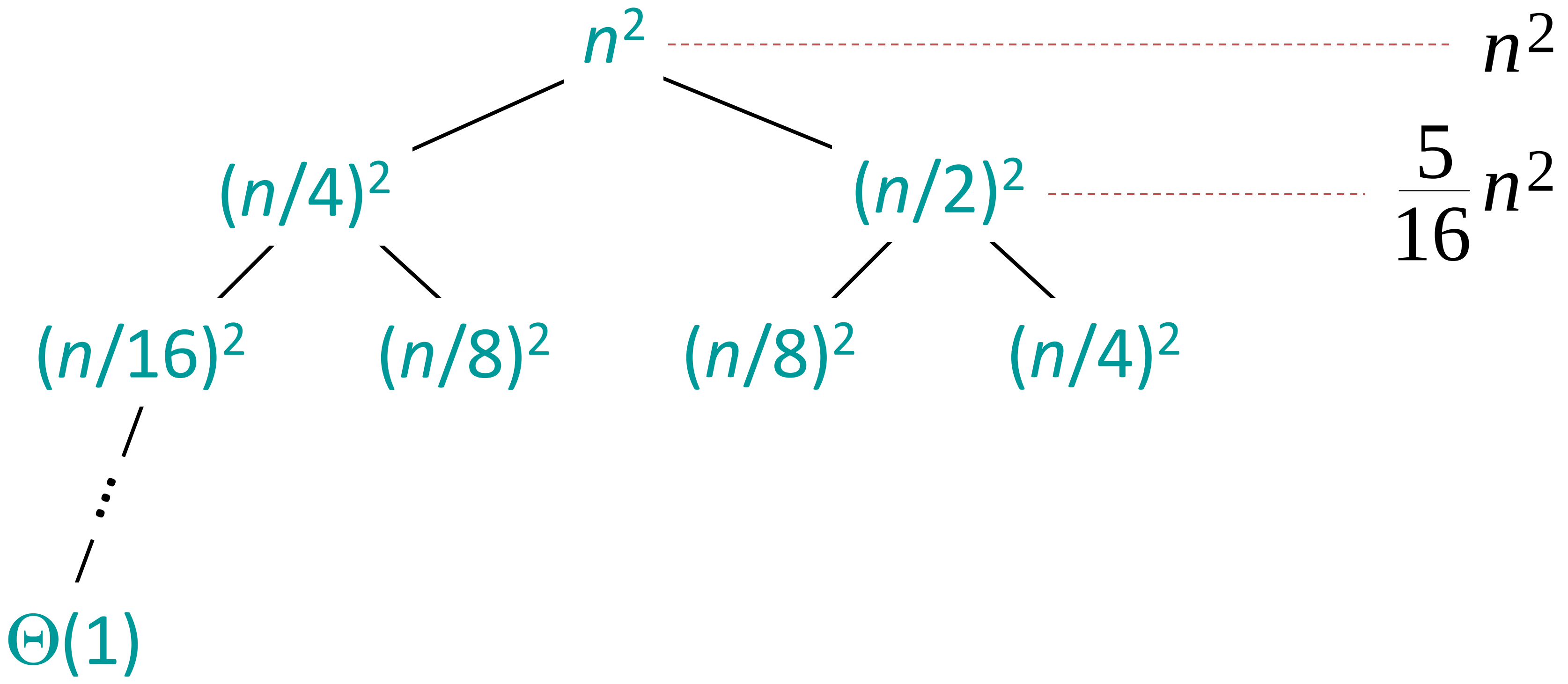
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



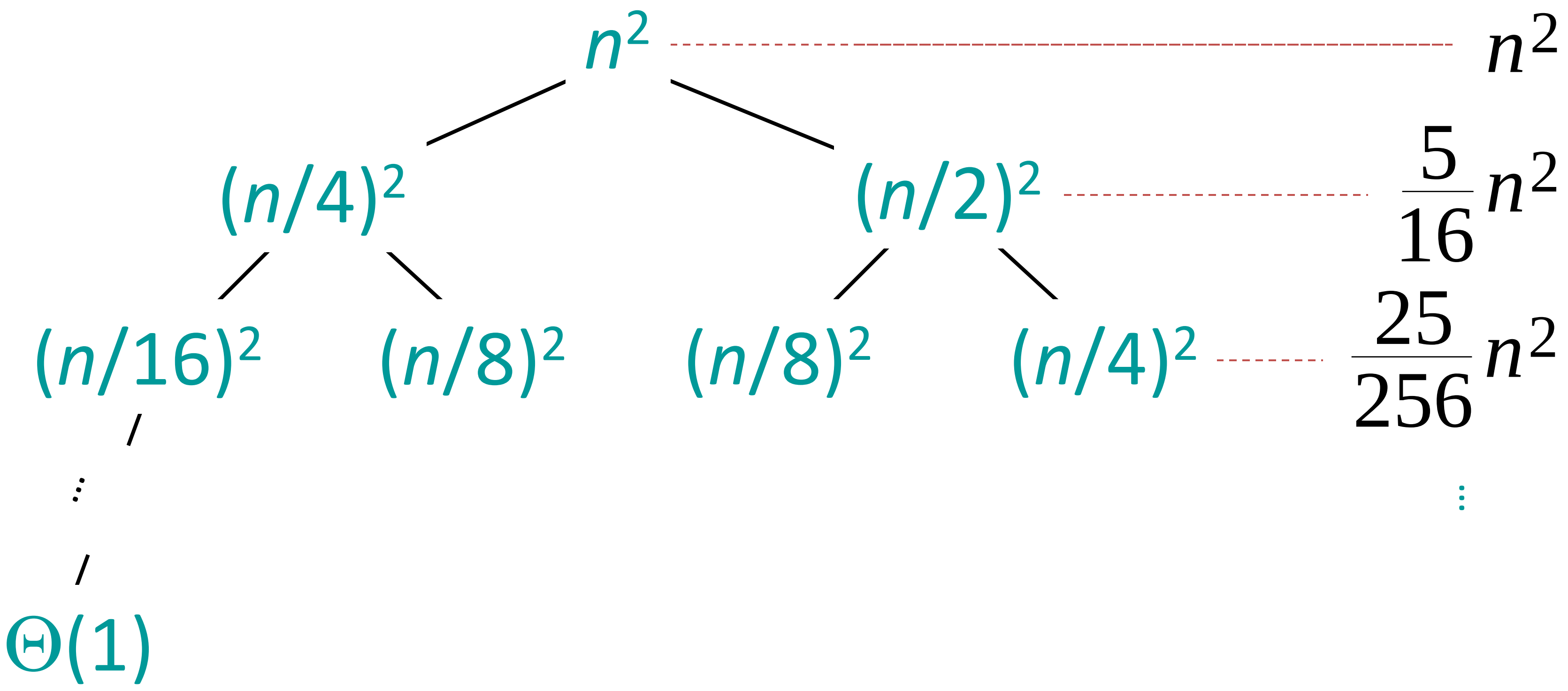
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



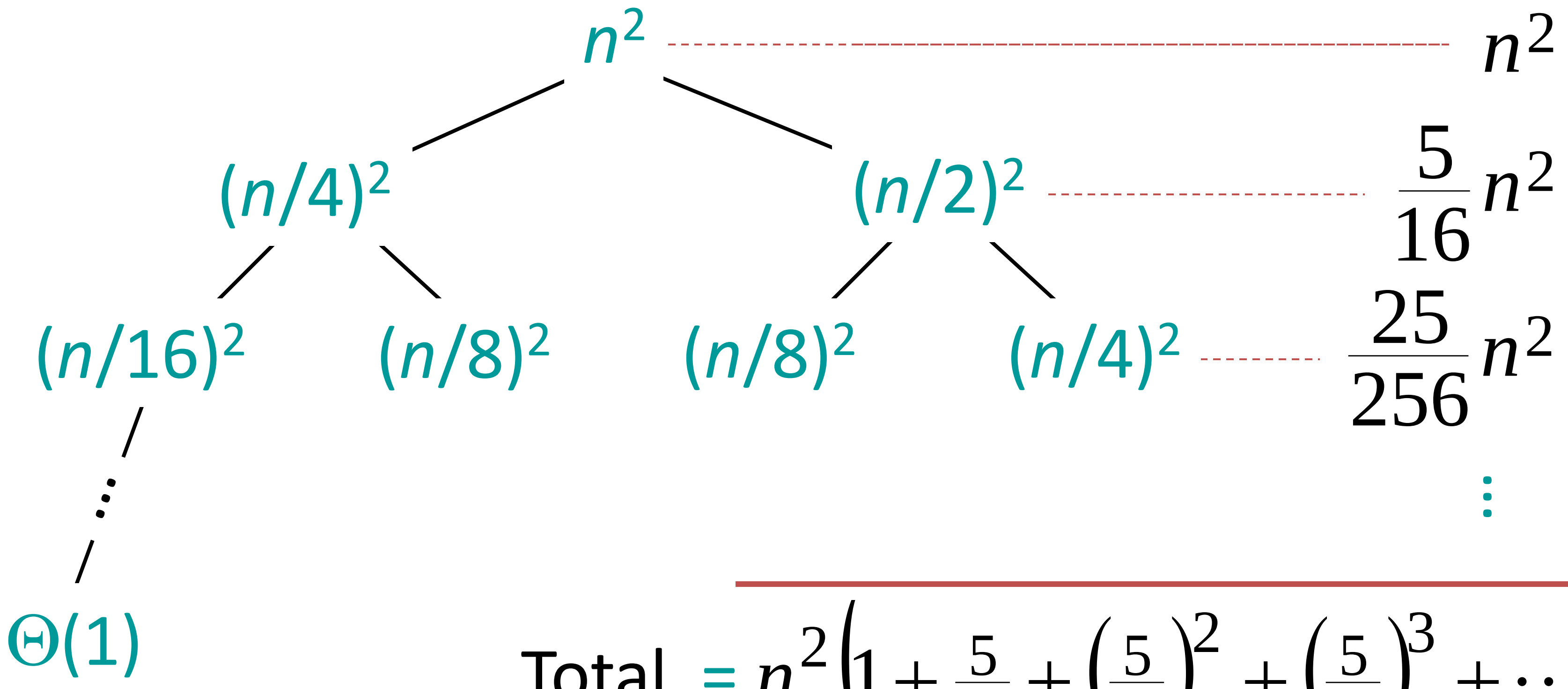
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$$= \Theta(n^2)$$

geometric series

4)Another Example of Master's Theorem

- (1)$T(n) = 3T(n/2) + n^2$
- (2) $T(n)=4T(n/2) + n$

4)Another Example of Master's Theorem

1)T(n) = 3T(n/2) + n2

- Solution: a=3, b=2, d=2

$3<2^2 \quad => \quad a<b^d$

Therefore $T(n)=O(n^d)$

$=O(n^2)$

2) T(n)=4T (n/2) + n

- Solution:a=4, b=2, d=1

$4>2^1 \quad => \quad a>b^d$

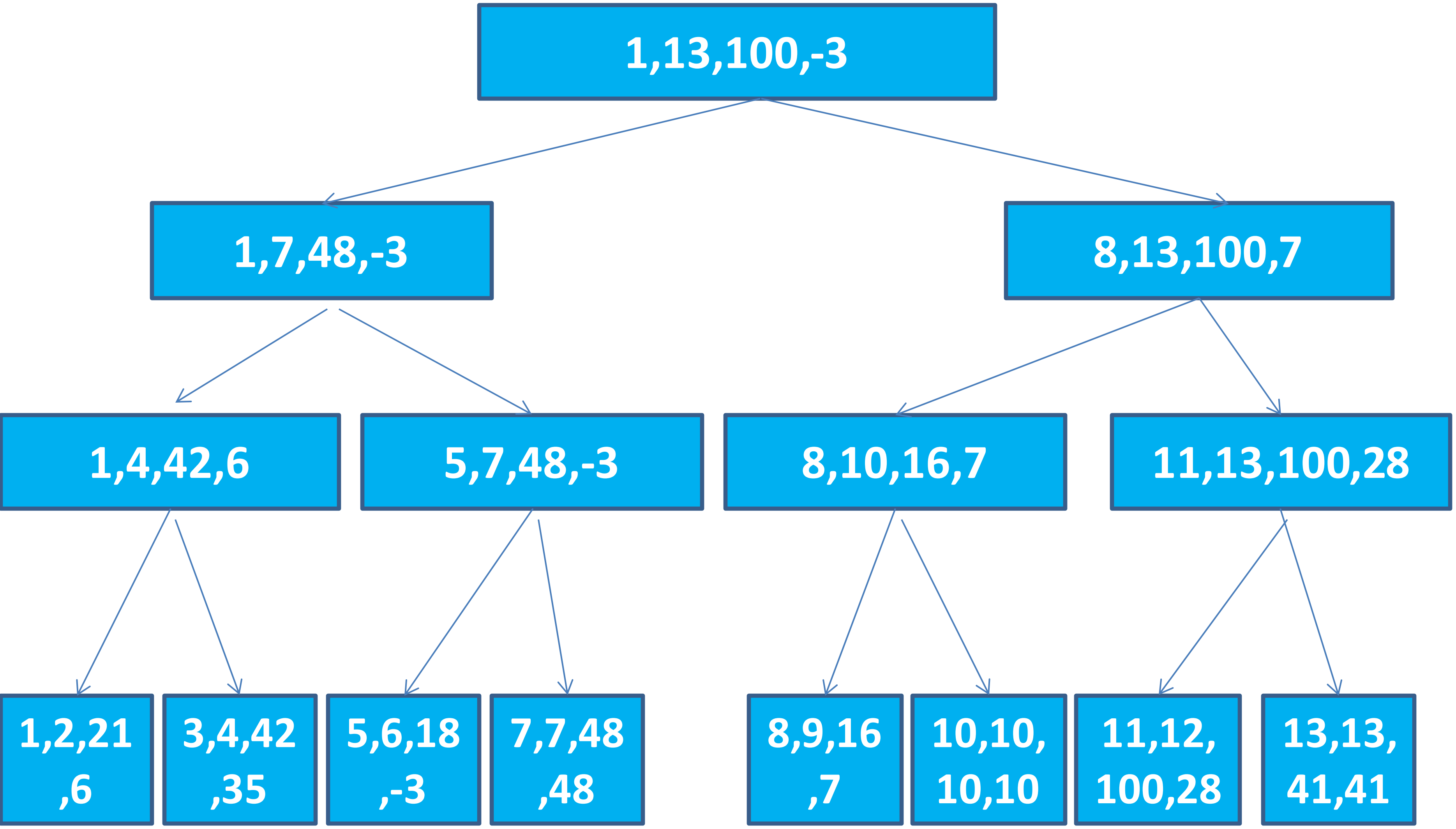
Therefore $T(n)=O(n^{(\log b\ a)})$

$=O(n^2)$

Min -Max

```
Algorithm MinMax(i,j,max,min)
{
If(i=j)
return max=min=a[i]
else if(i=j-1)
{
    If(a[i]<a[j])
    {
        max=a[j]
        min=a[i]
    }
    else
    {
        max=a[i]
        min=a[j]
    }
}
    Else
    {
    mid=L(i+j)/2 ⌋
    MinMax(i,mid,max,
    min)
    MinMax(mid+1,j,ma
    x1,min1)
    If(max<max1)
        max=max1
    If(min>min1)
        min=min1
    }
}
```

Ex:{21,6,35,42,-3,18,48,16,7,10,28,100,41}



Min Max Time Complexity

- $T(n) = \begin{cases} T(n/2)+T(n/2)+2 & n>2 \\ 1 & n=2 \\ 0 & n=1 \end{cases}$
- $T(n)=2T(n/2)+2$
- $\quad =2(2T(n/4)+2)+2$
- $\quad =4T(n/4)+4+2$
- $\quad =2^{k-1}T(2)+2^k-2$
- $\quad =(2^k/2)+2^k-2 \qquad [n=2^k]$
- $\quad =n/2+n-2$
- $\quad =3n/2-2=O(n)$

Merge Sort

- Uses Divide and Conquer strategy

1. Divide the array into two parts

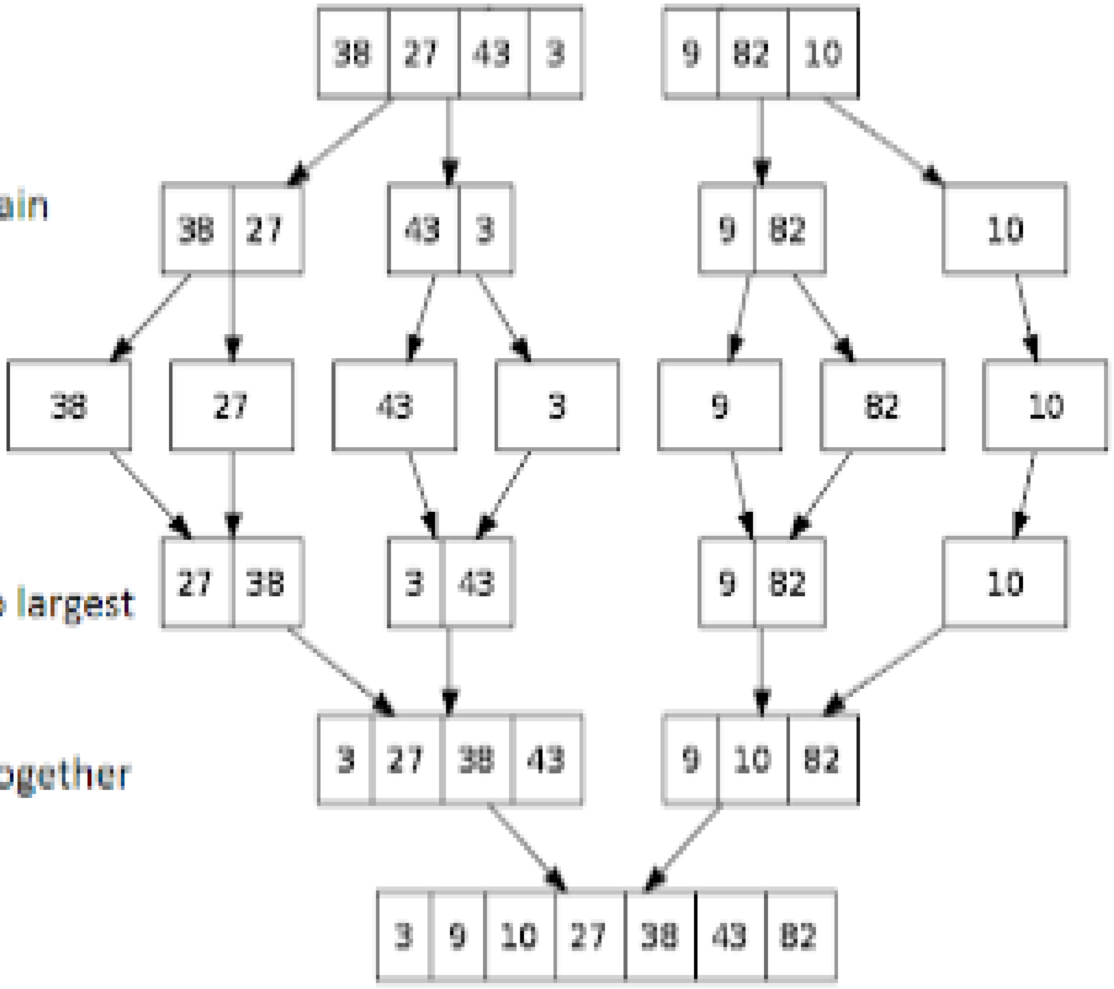
2. Divide the array into two parts again

3. Break each element into single parts

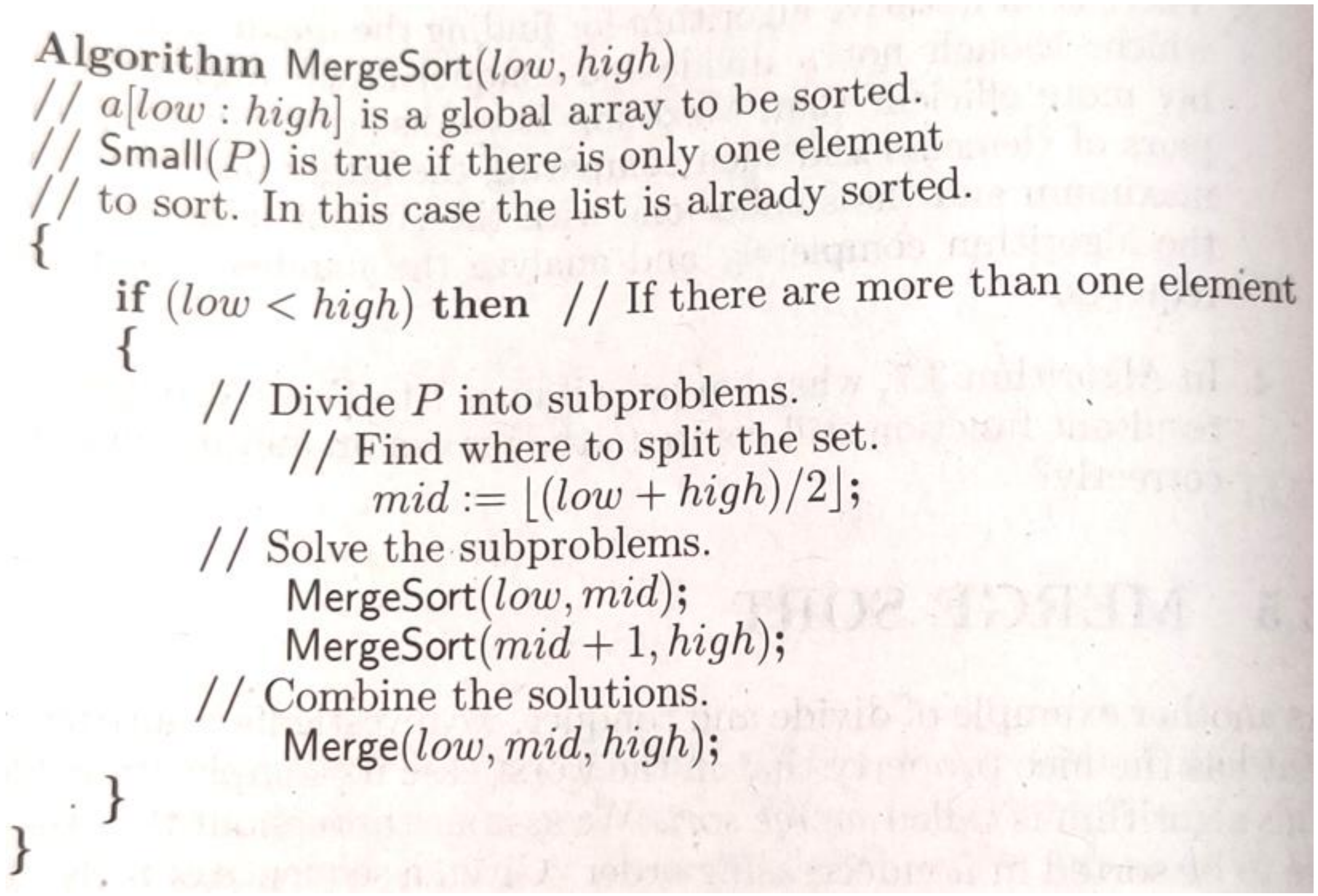
4. Sort the elements from smallest to largest

5. Merge the divided sorted arrays together

6. The array has been sorted

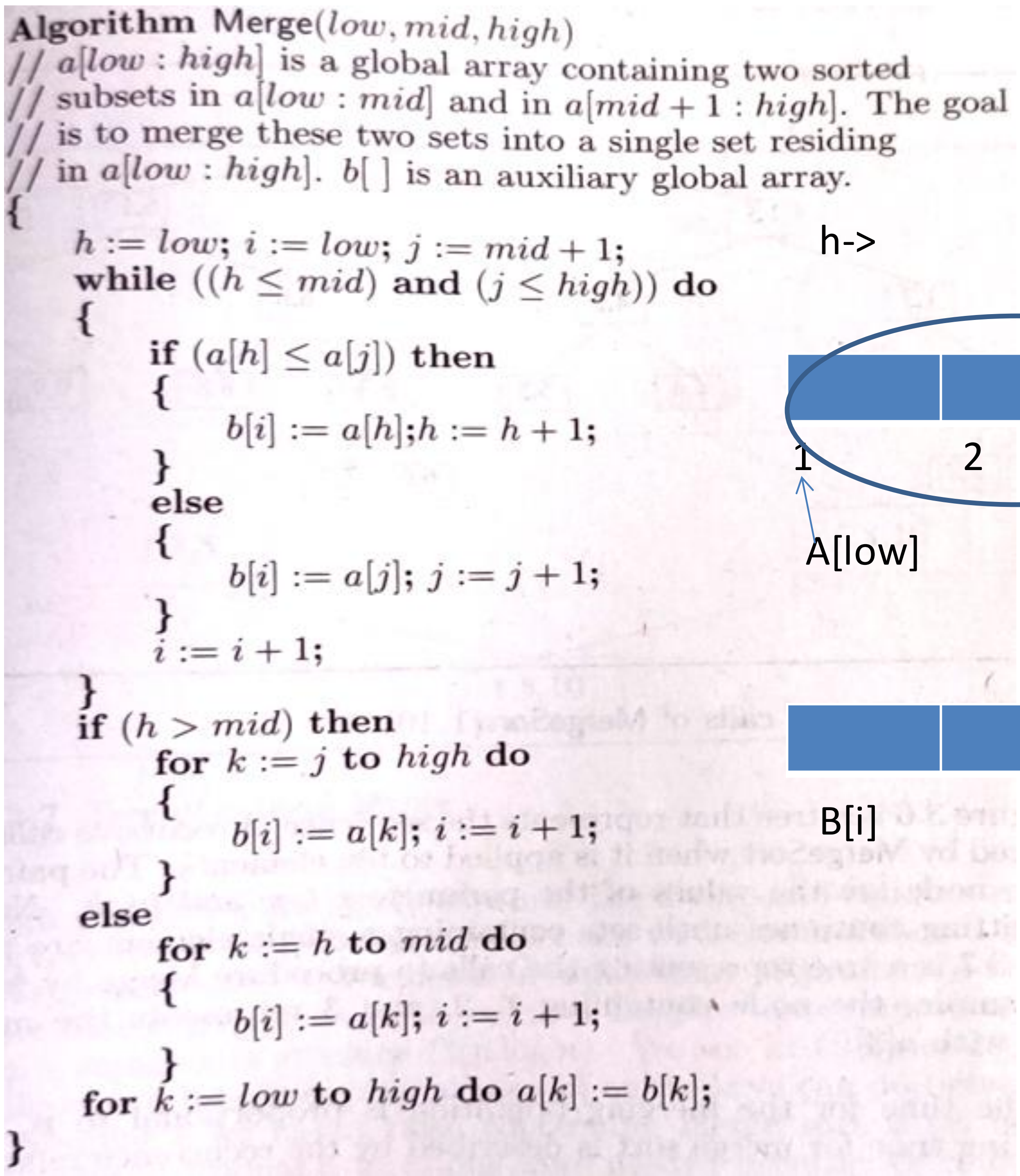


Merge Sort

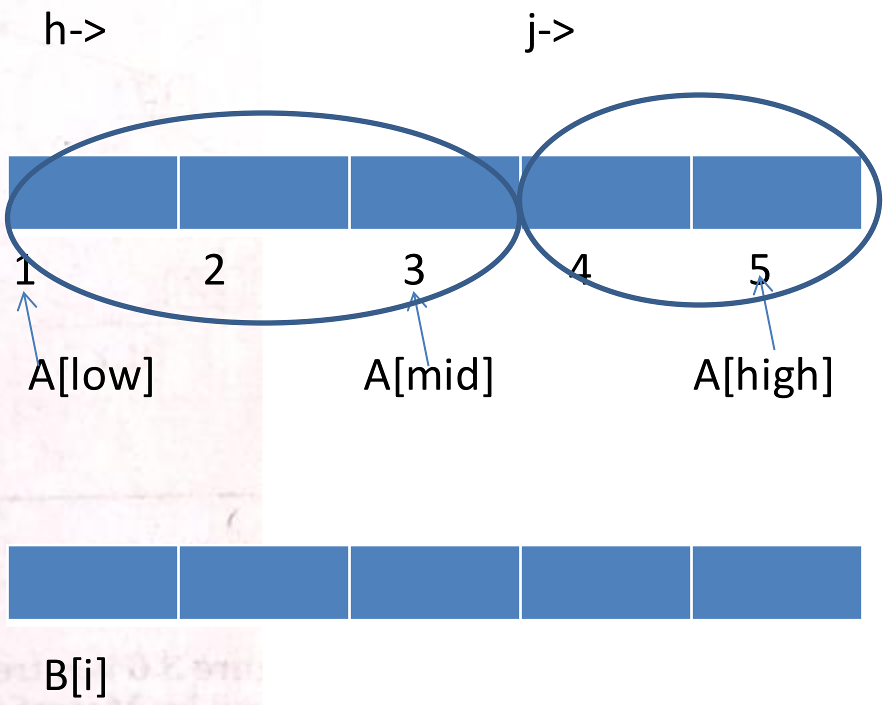


```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then // If there are more than one element
    {
        // Divide P into subproblems.
     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

Merge Sort



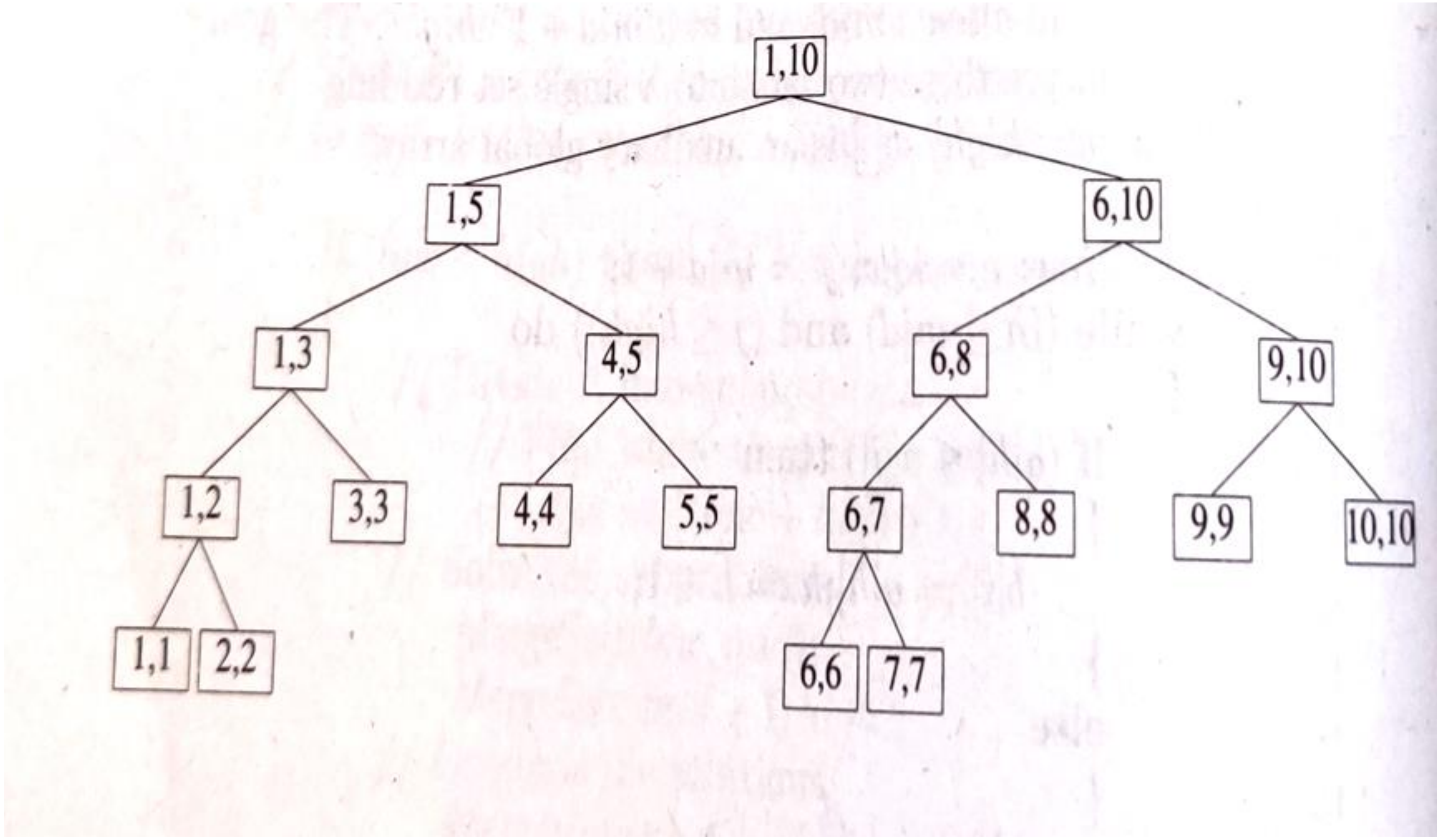
```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\ ]$ is an auxiliary global array.
{
    $h := low$; $i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



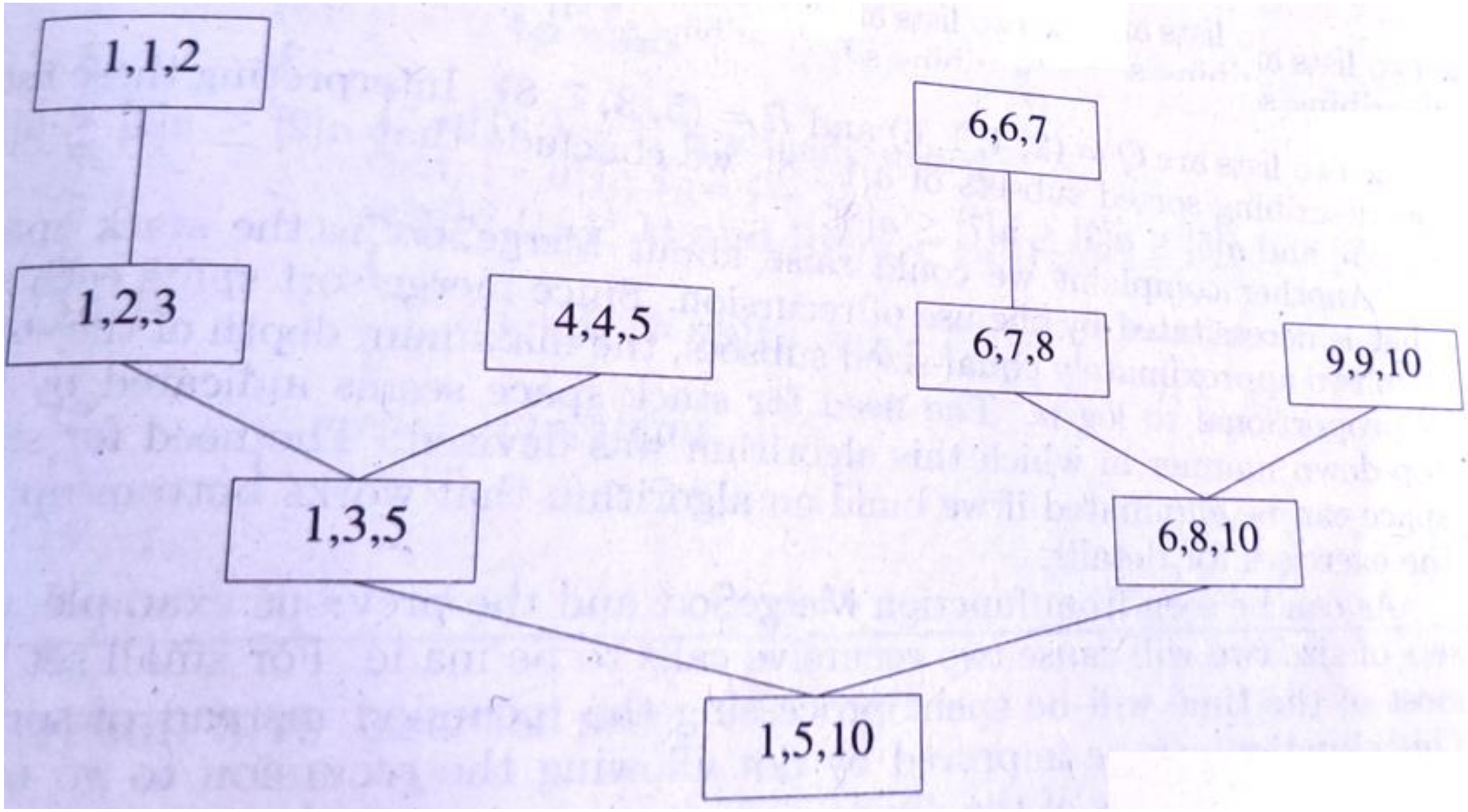
Merge sort Example

- Ex:{310,285,179,652,351,423,861,254,450,520}

Merge Sort Example



Merge Sort Example



Merge Sort Example

- 310,285,179,652,351,423,861,254,450,520
- $\underline{285,310,}$179,652,351,423,861,254,450,520
- $\underline{179,285,310}$,652,351,423,861,254,450,520
- 179,285,310,$\underline{351,652}$,423,861,254,450,520
- $\underline{179,285,310,351,652}$,423,861,254,450,520
- 179,285,310,351,652,$\underline{423,861}$,254,450,520
- 179,285,310,351,652,$\underline{254,423,861}$,450,520
- 179,285,310,351,652,254,423,861,$\underline{450,520}$
- 179,285,310,351,652,254,$\underline{423,450,520,861}$
- 179,254,285,310,351,423,450,520,652,861

Merge Sort Time Complexity

- $T(n) = \begin{cases} 2T(n/2)+cn & n>1,\ c\text{-constant} \\ a & n=1,\ a\text{-constant} \end{cases}$
- $T(n)= 2T(n/2)+cn$

  $=2(2T(n/4)+cn/2)+cn$

  $=2(4T(n/8)+cn/4)+2cn$

  $=8T(n/8)+3cn$

  $=2^kT(1)+kcn$

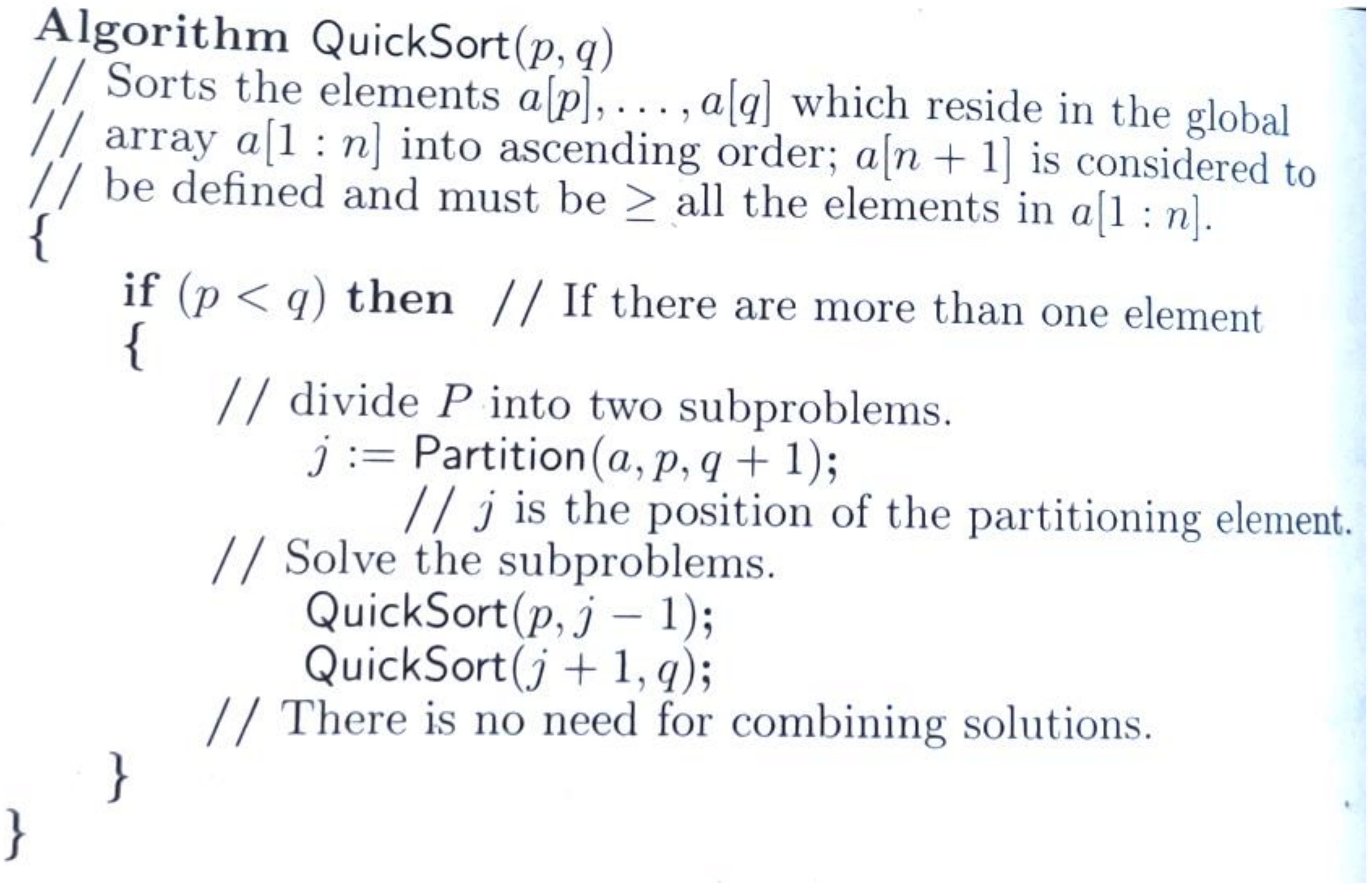
  $=an+\ \ nclogn$ $\qquad[n=2^k]$

  $=O(nlogn)$

Quick Sort

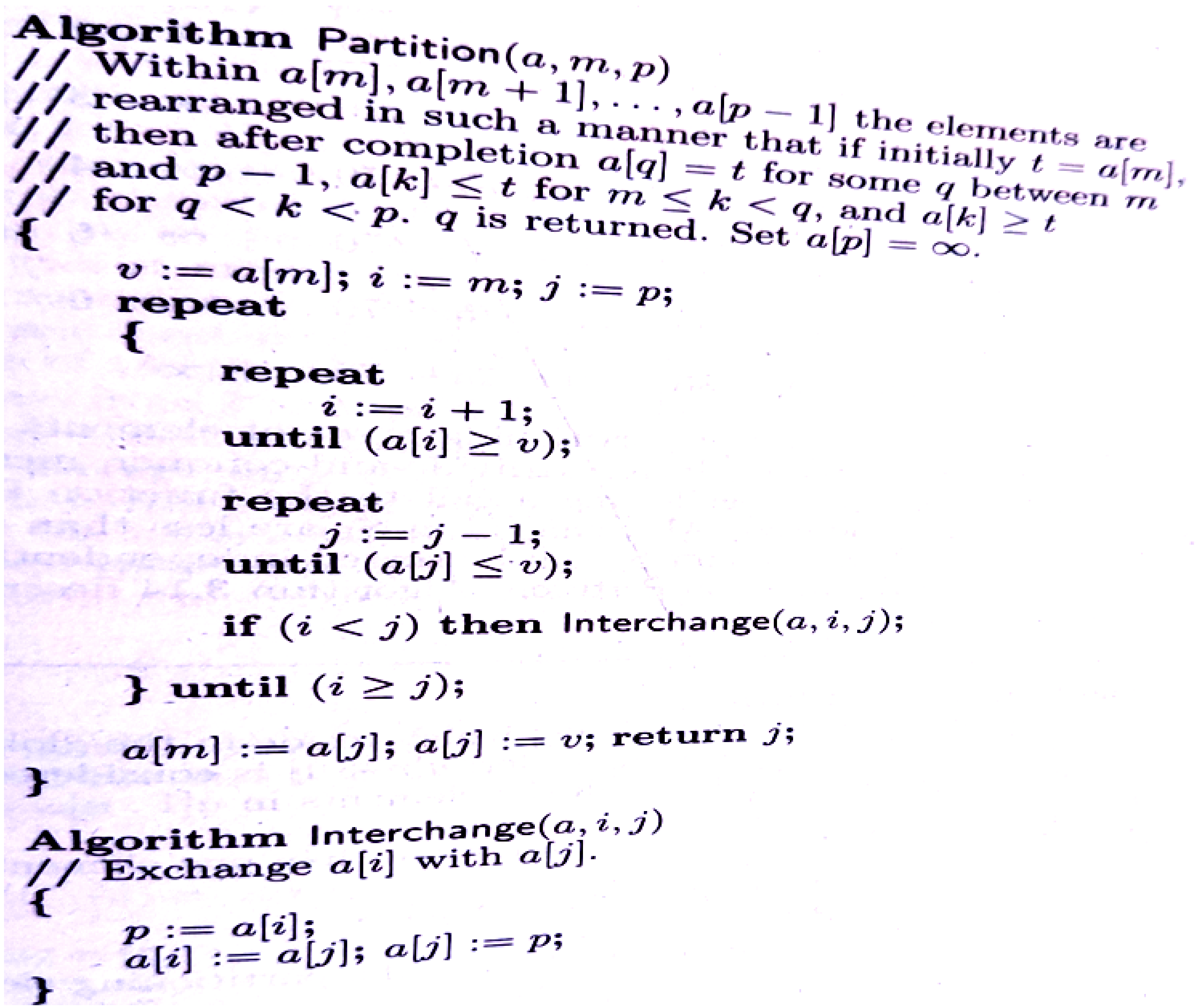
- Uses divide and conquer.
- It finds the element called pivot which divides the array into two halves.
- Elements in the left half are smaller than pivot and elements in the right half are greater than pivot.
- Three steps
  - Find pivot that divides the array into two halves.
  - Quick sort the left half.
  - Quick sort the right half.

Quick Sort



```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```

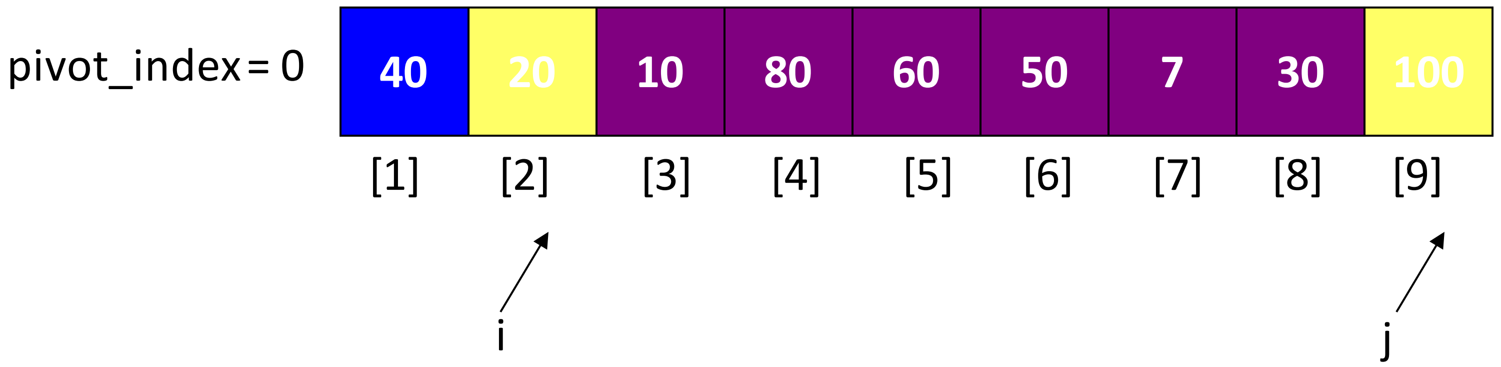
Quick Sort



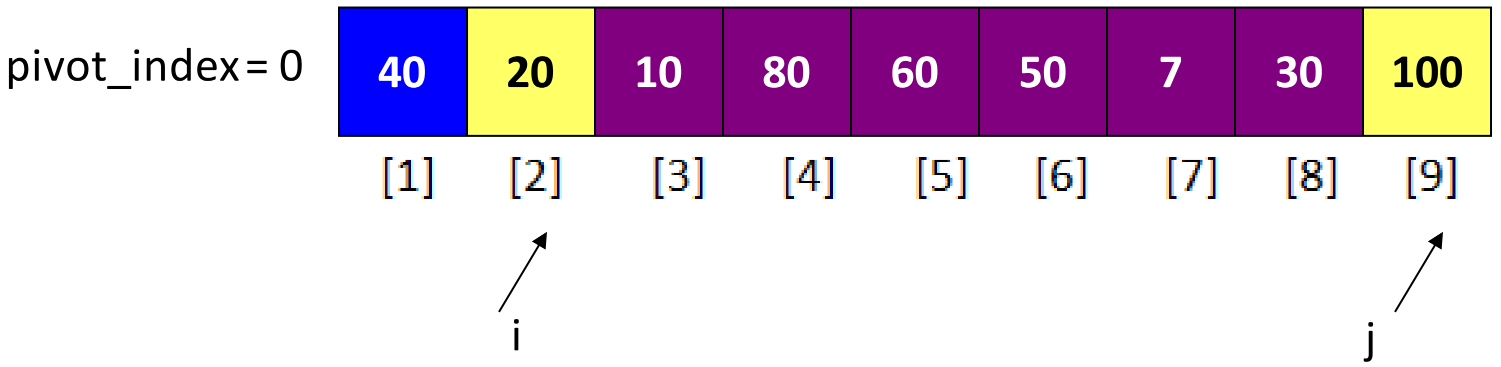
```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Example

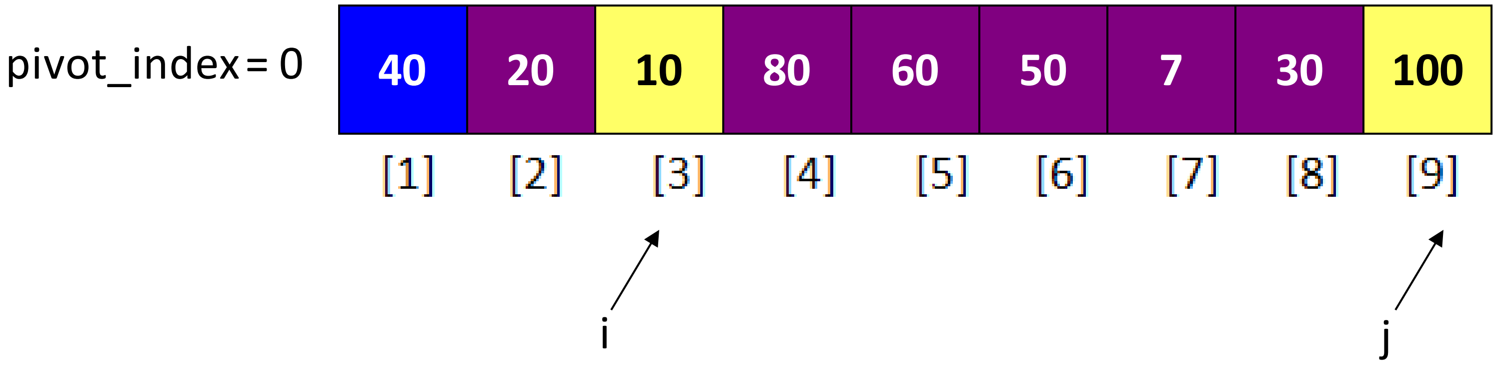
- data={40,20,10,80,60,50,7,30,100}



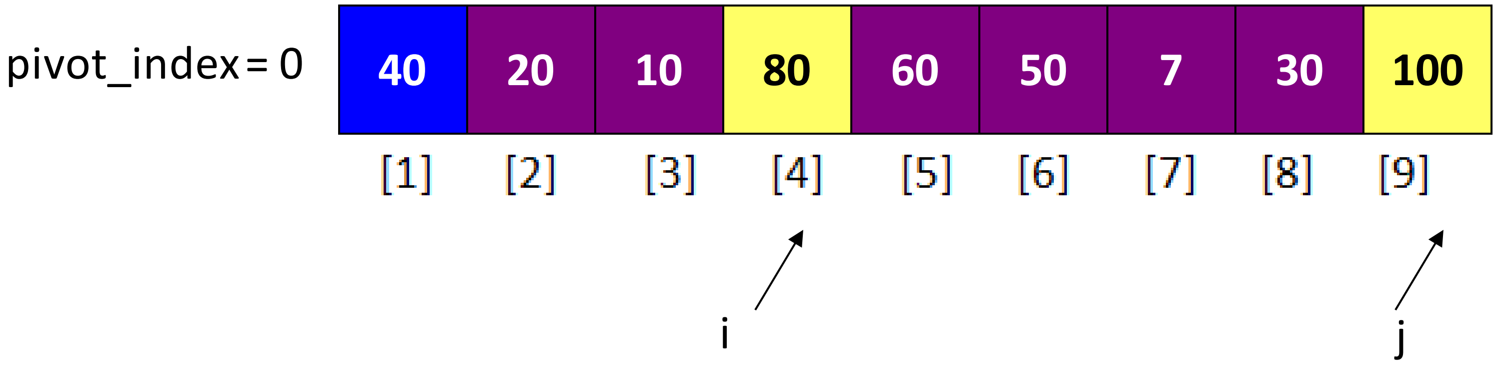
```
1. While data[i] <= data[pivot]
       ++i
```



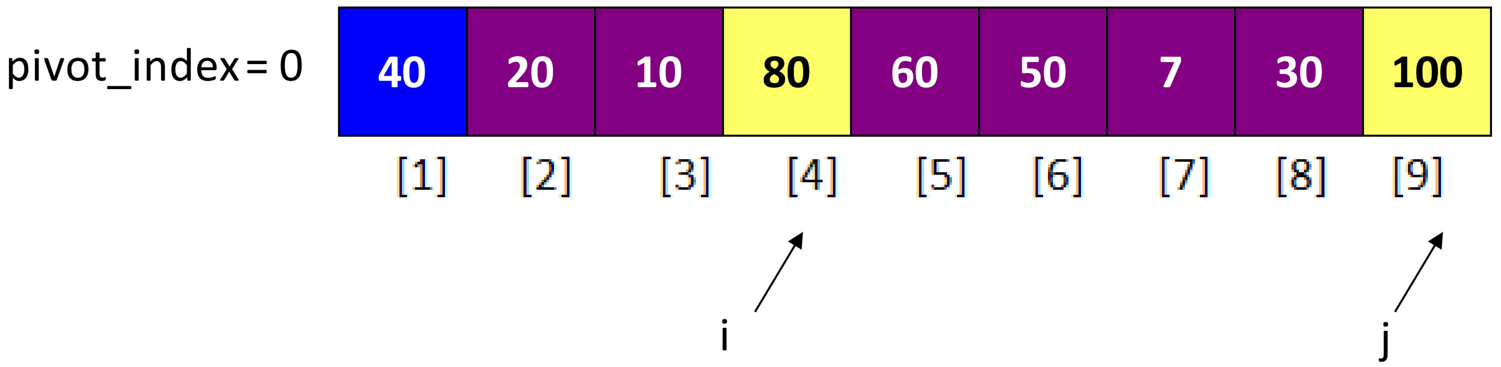
```
1.   While data[i] <= data[pivot]
         ++i
```



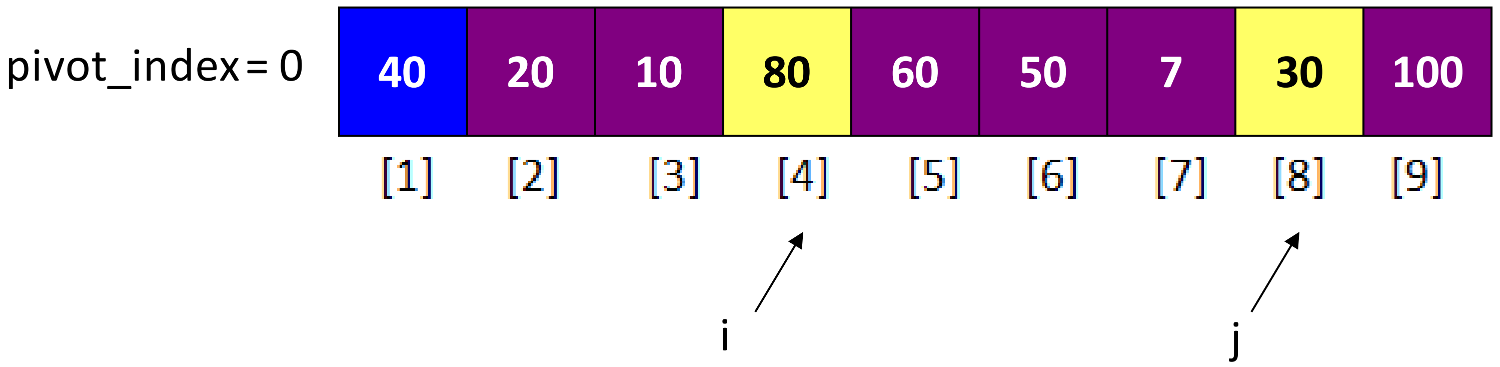
```
1.  While data[i] <= data[pivot]
        ++i
```



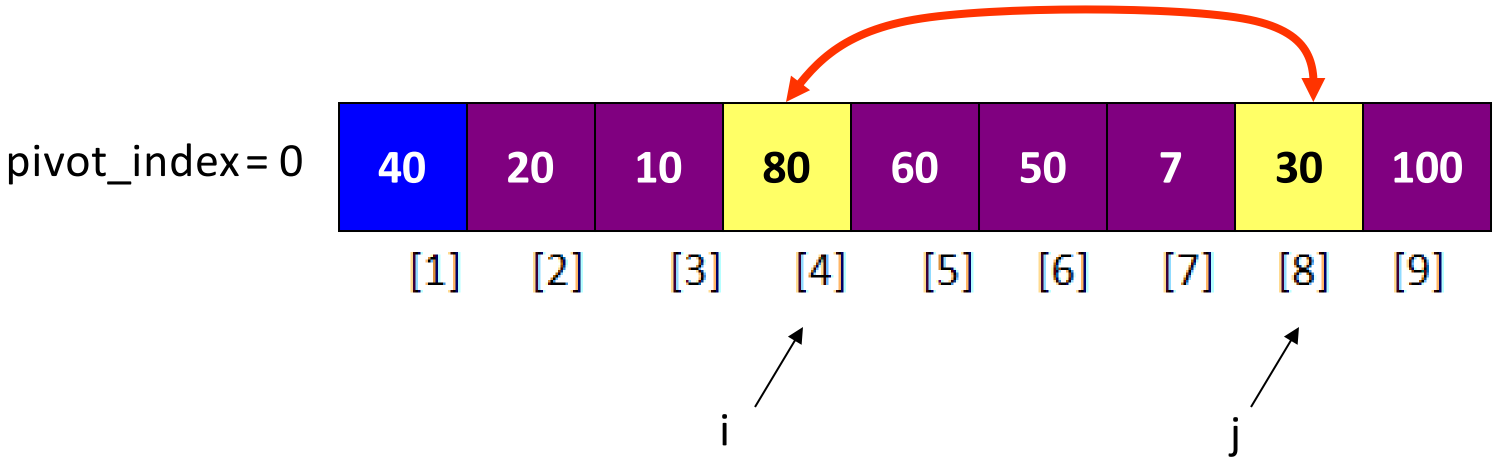
```
1. While data[i] <= data[pivot]
       ++i
2. While data[j] > data[pivot]
       --j
```



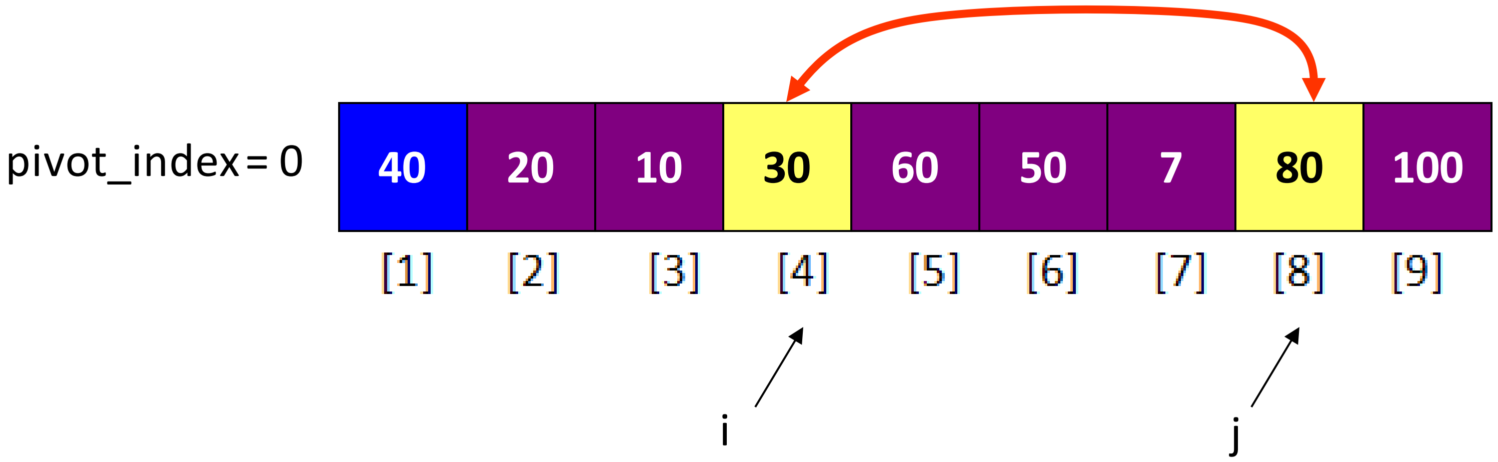
```
1.    While data[i] <= data[pivot]
              ++i
2.    While data[j] >= data[pivot]
              --j
```



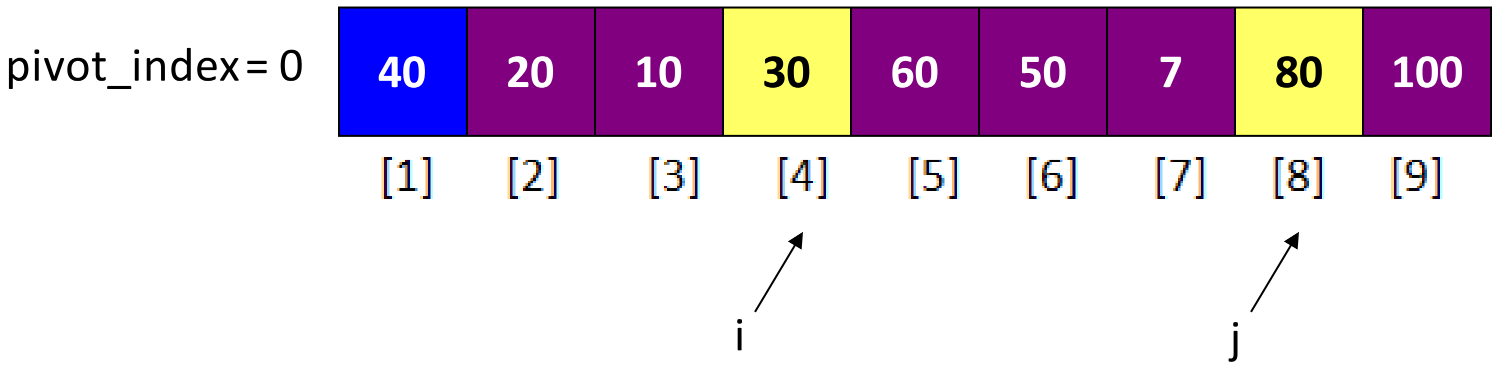
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
```



```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
```

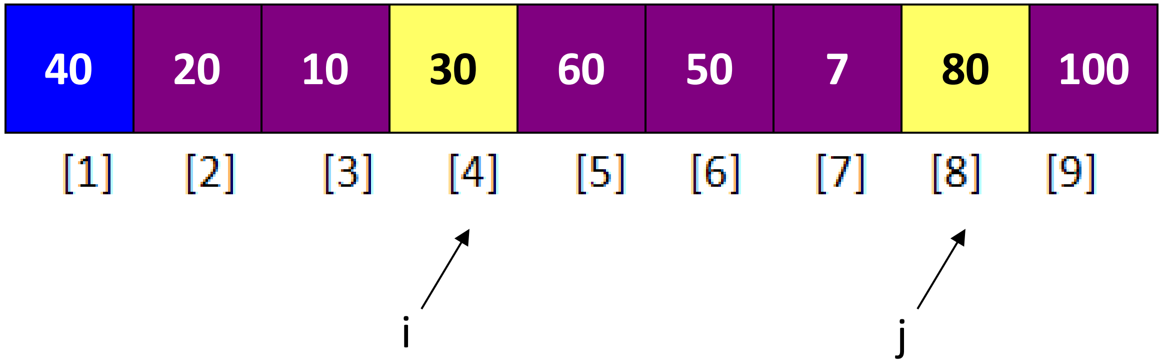


```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

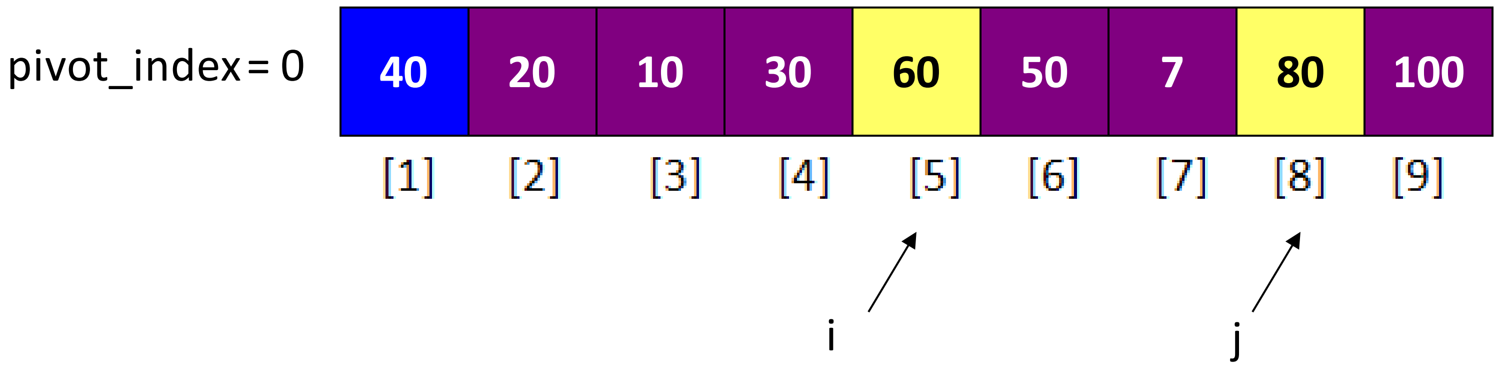


```
→ 1. While data[i] <= data[pivot]
           ++i
  2. While data[j] > data[pivot]
           --j
  3. If i < j
           swap data[i] and data[j]
  4. While j > i, go to 1.
```

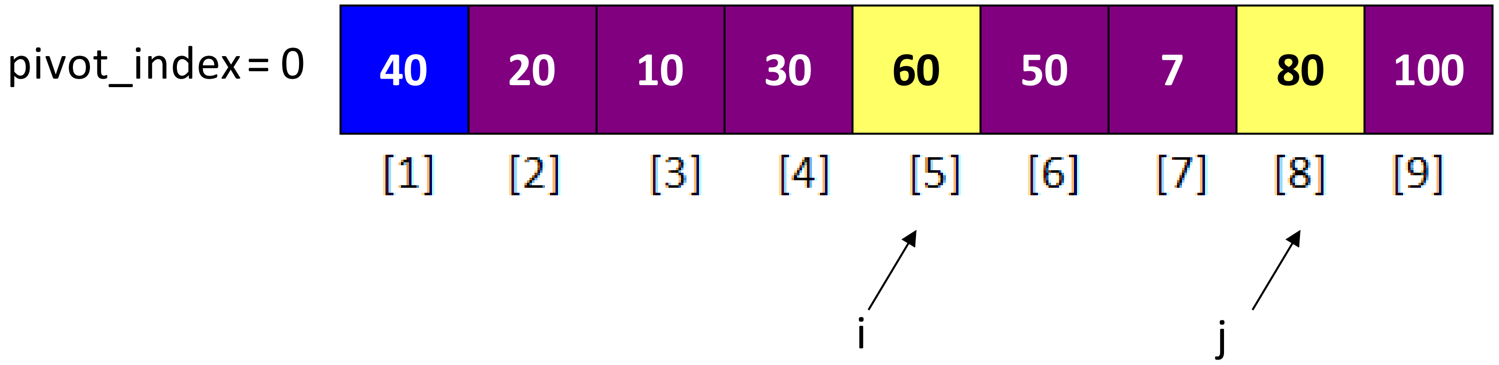
pivot_index = 0



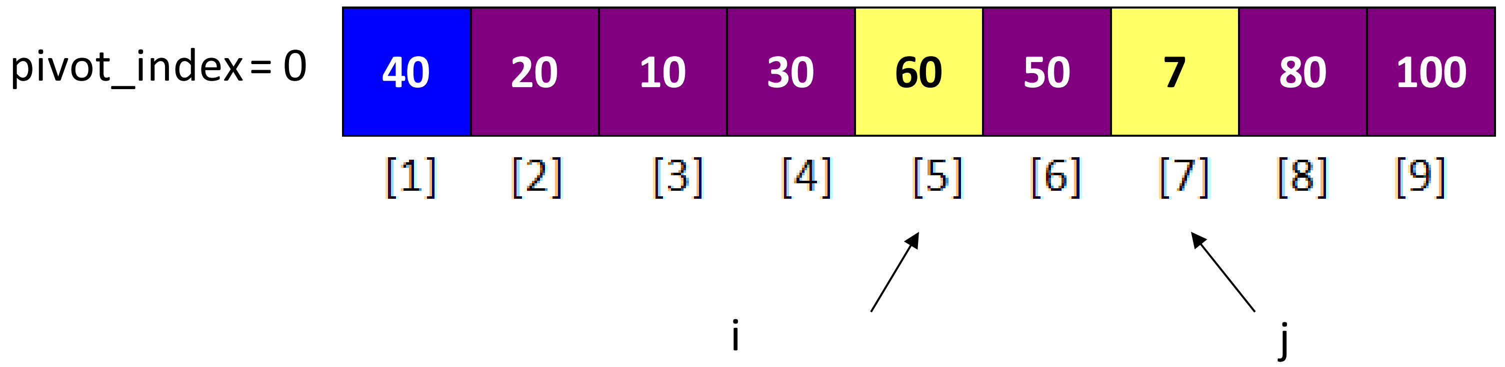
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



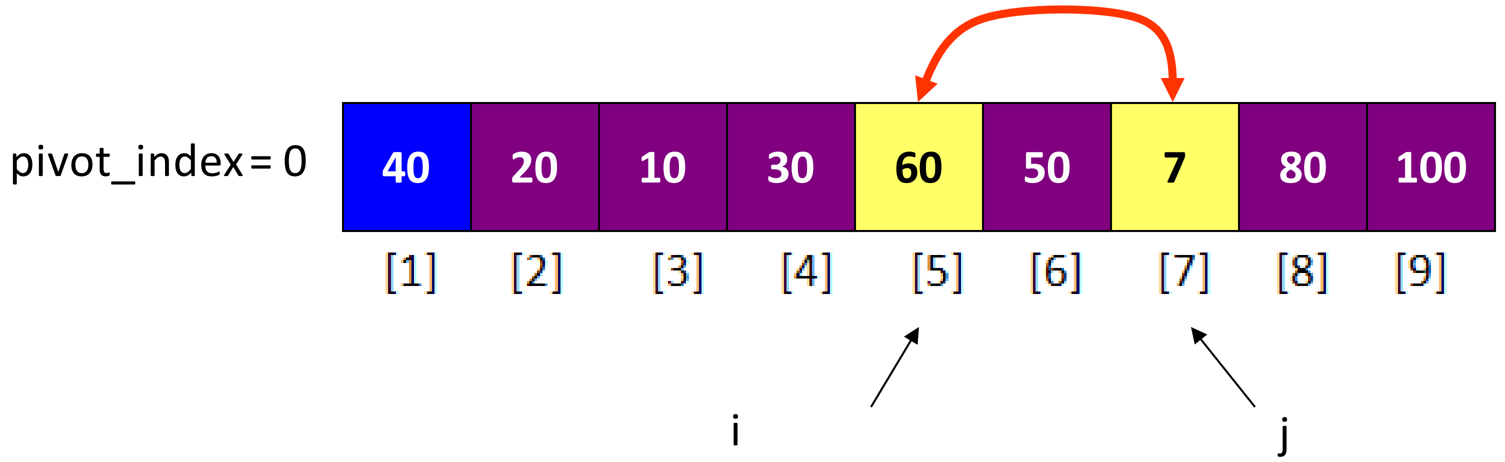
```
1.    While data[i] <= data[pivot]
              ++i
2.    While data[j] > data[pivot]
              --j
3.    If i < j
              swap data[i] and data[j]
4.    While j > i, go to 1.
```



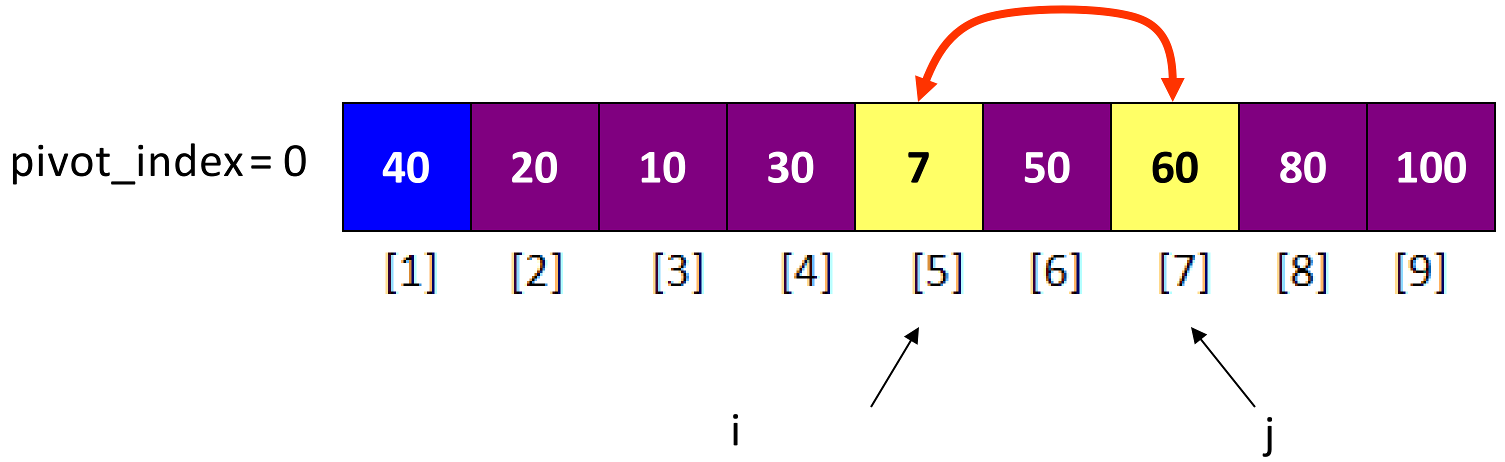
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



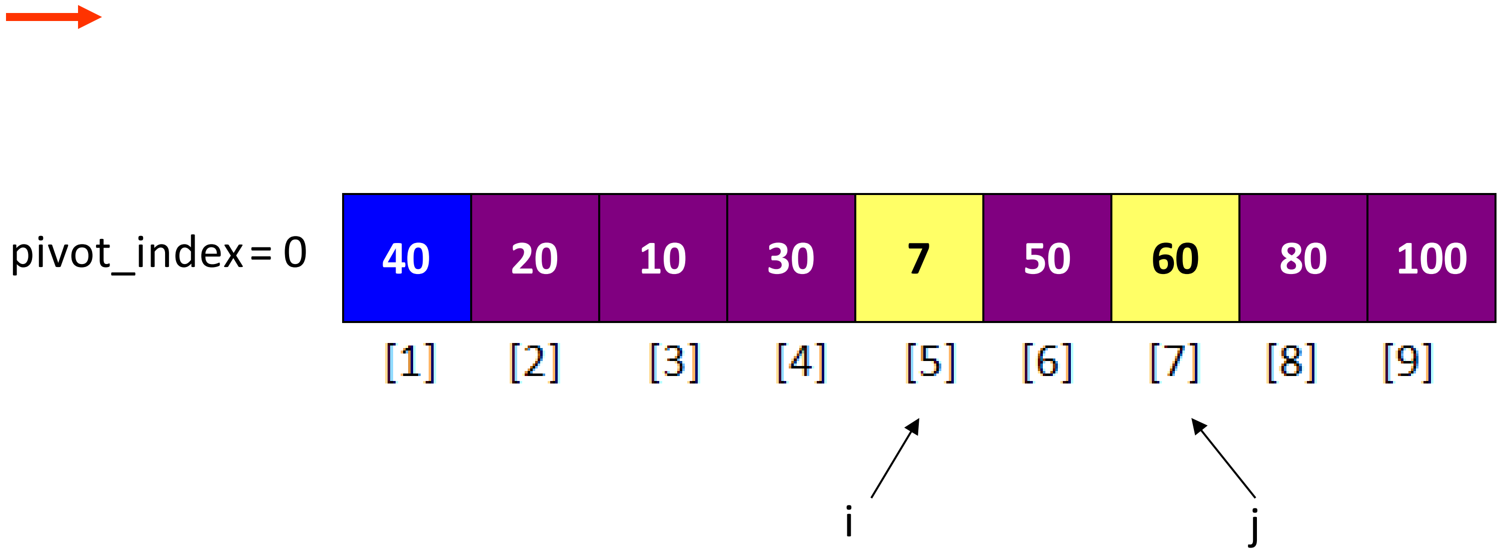
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



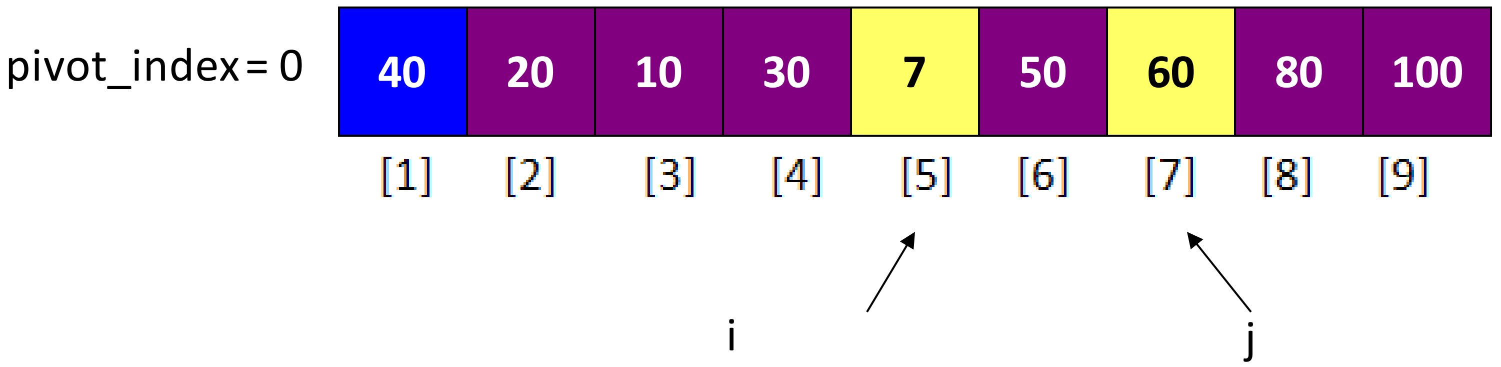
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



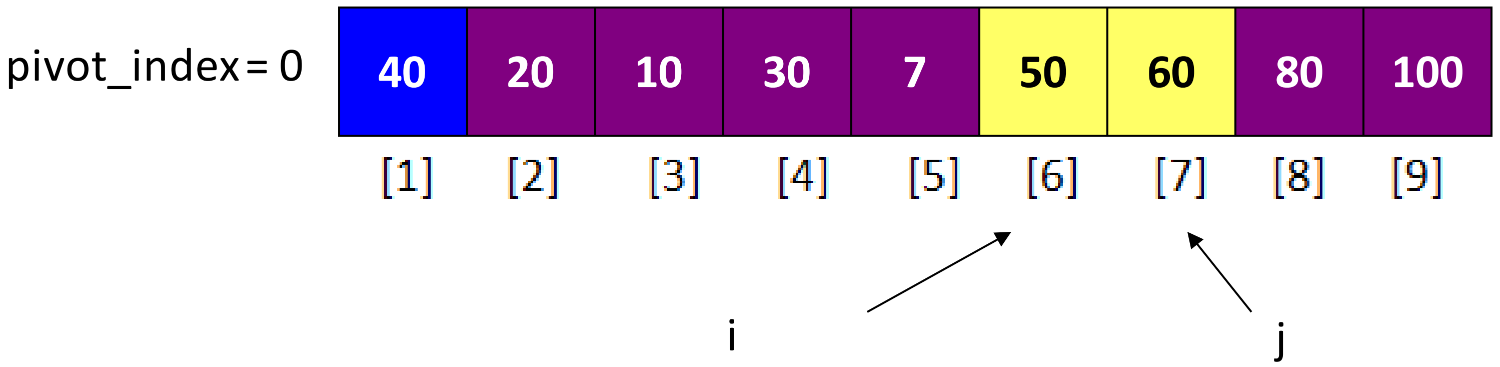
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



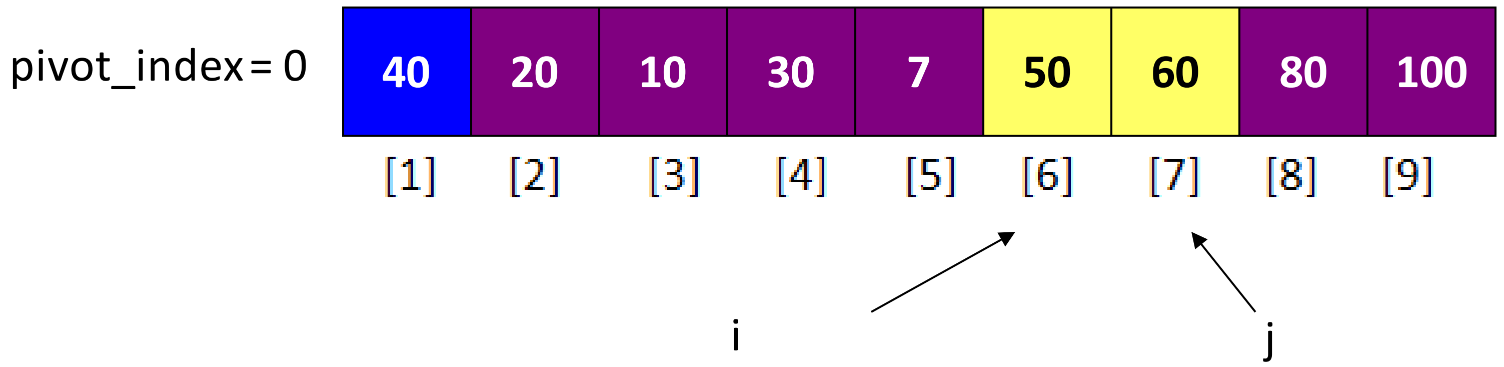
```
→ 1.  While data[i] <= data[pivot]
              ++i
  2.  While data[j] > data[pivot]
              --j
  3.  If i < j
              swap data[i] and data[j]
  4.  While j > i, go to 1.
```



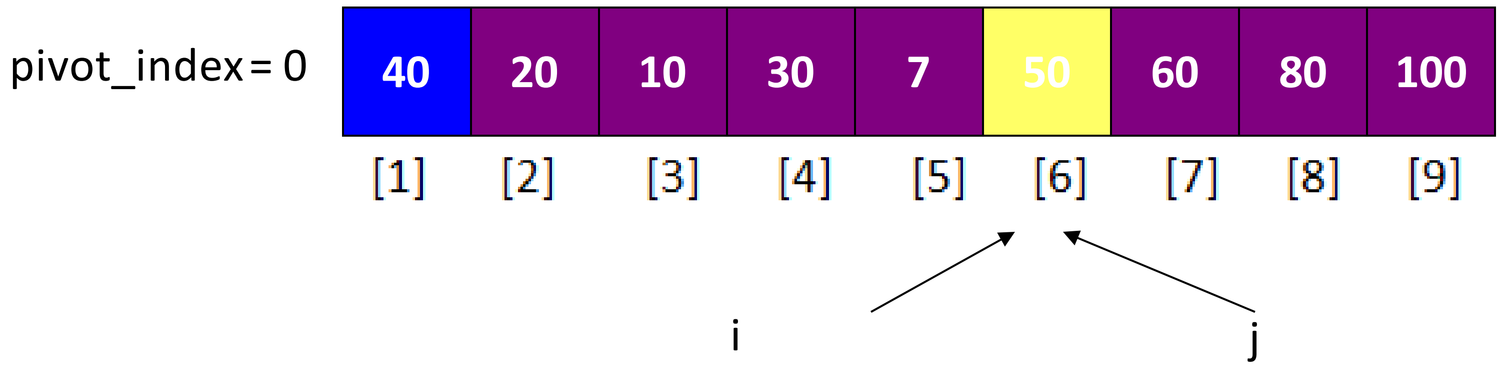
```
→ 1.  While data[i] <= data[pivot]
              ++i
  2.  While data[j] > data[pivot]
              --j
  3.  If i < j
              swap data[i] and data[j]
  4.  While j > i, go to 1.
```



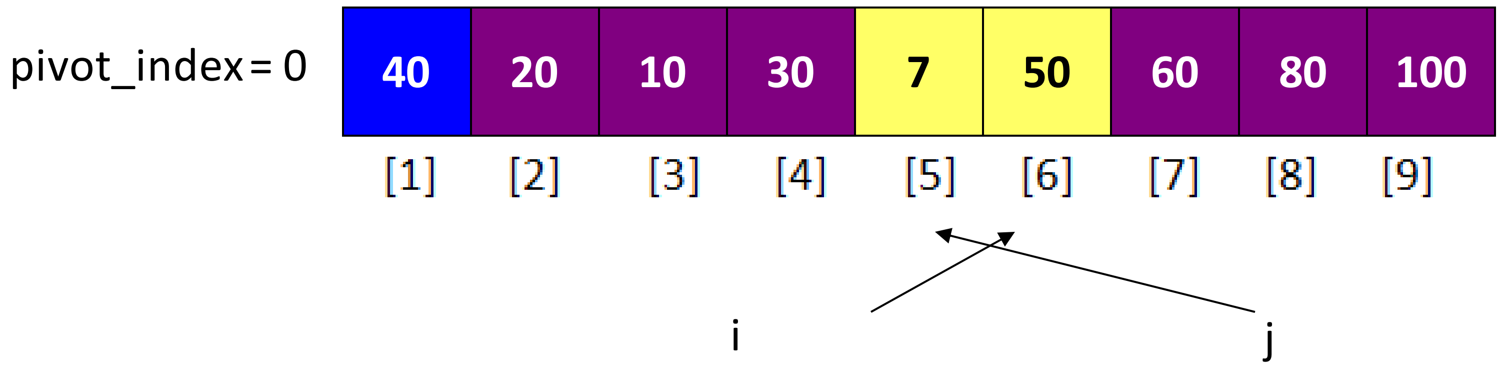
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

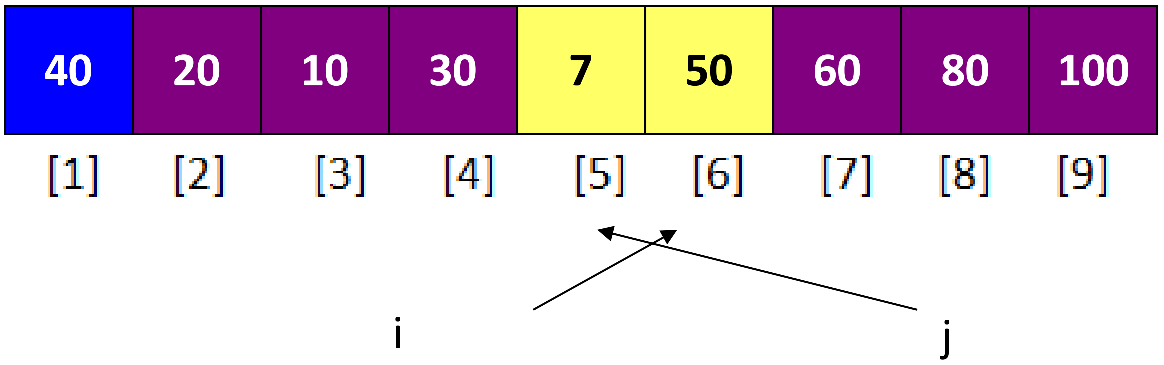


```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

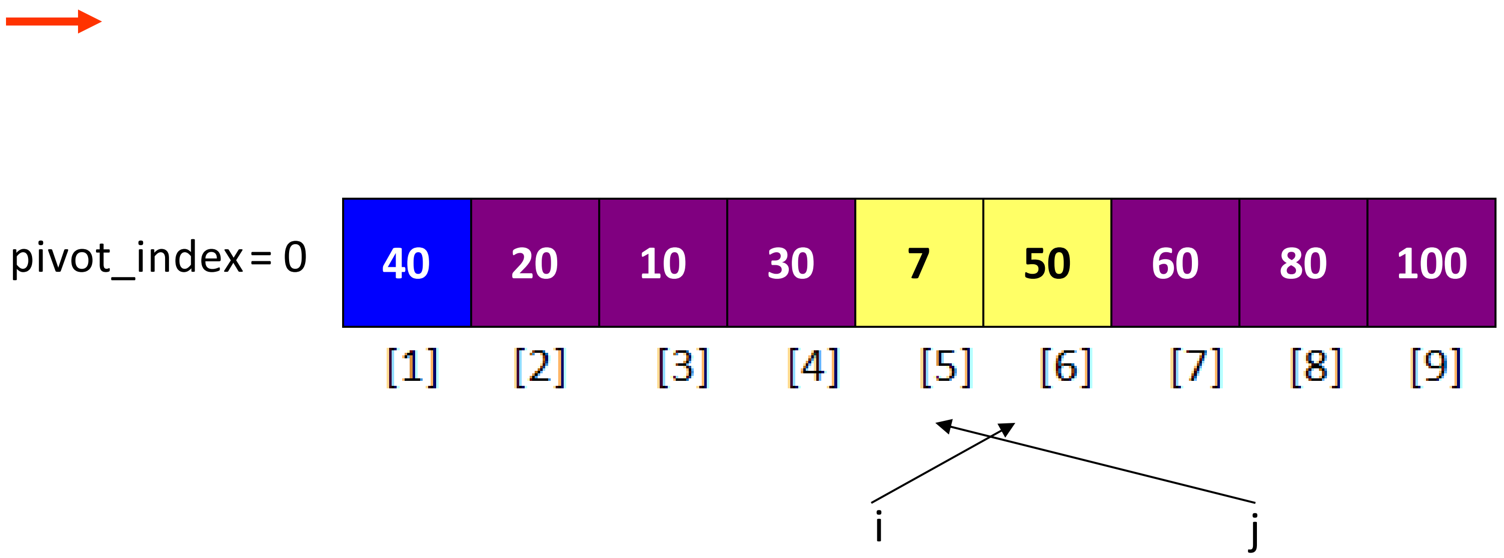


```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```

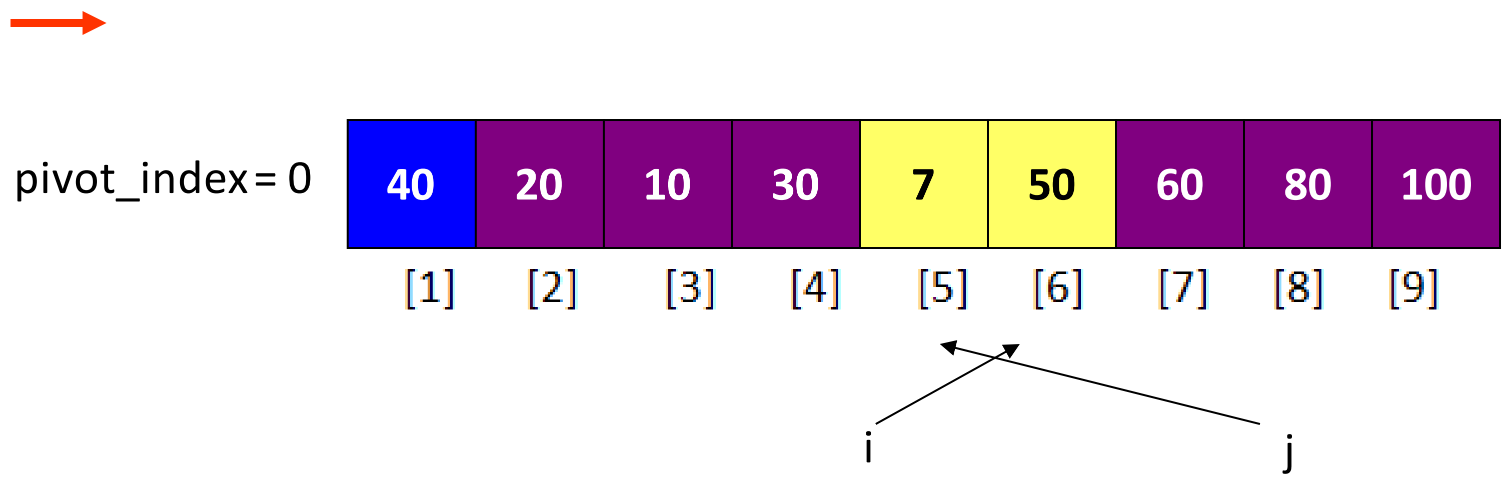
pivot_index = 0



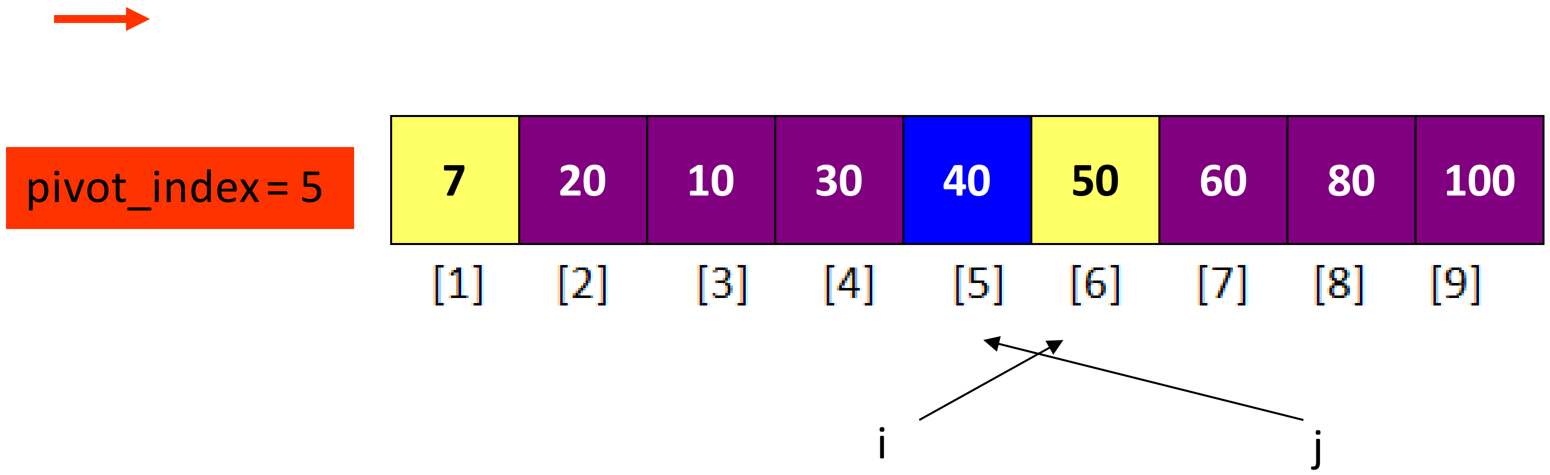
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
```



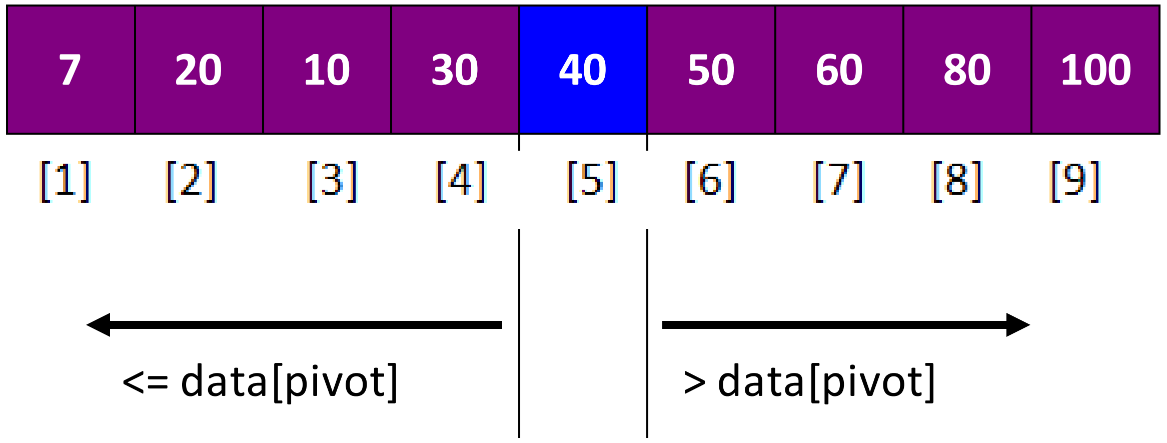
```
1.  While data[i] <= data[pivot]
        ++i
2.  While data[j] > data[pivot]
        --j
3.  If i < j
        swap data[i] and data[j]
4.  While j > i, go to 1.
5.  Swap data[j] and data[pivot_index]
```



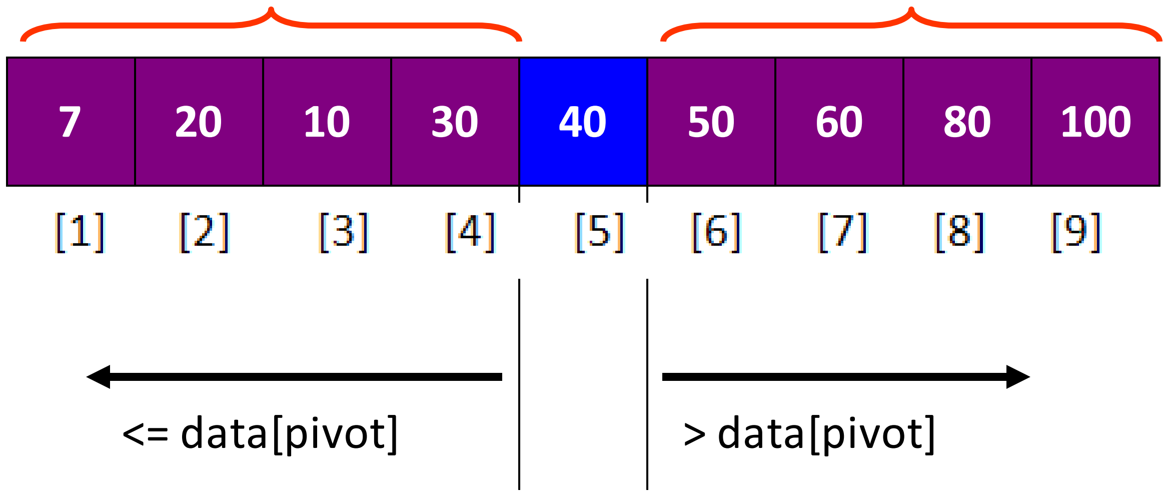
```
1. While data[i] <= data[pivot]
        ++i
2. While data[j] > data[pivot]
        --j
3. If i < j
        swap data[i] and data[j]
4. While j > i, go to 1.
5. Swap data[j] and data[pivot_index]
```



Partition Result



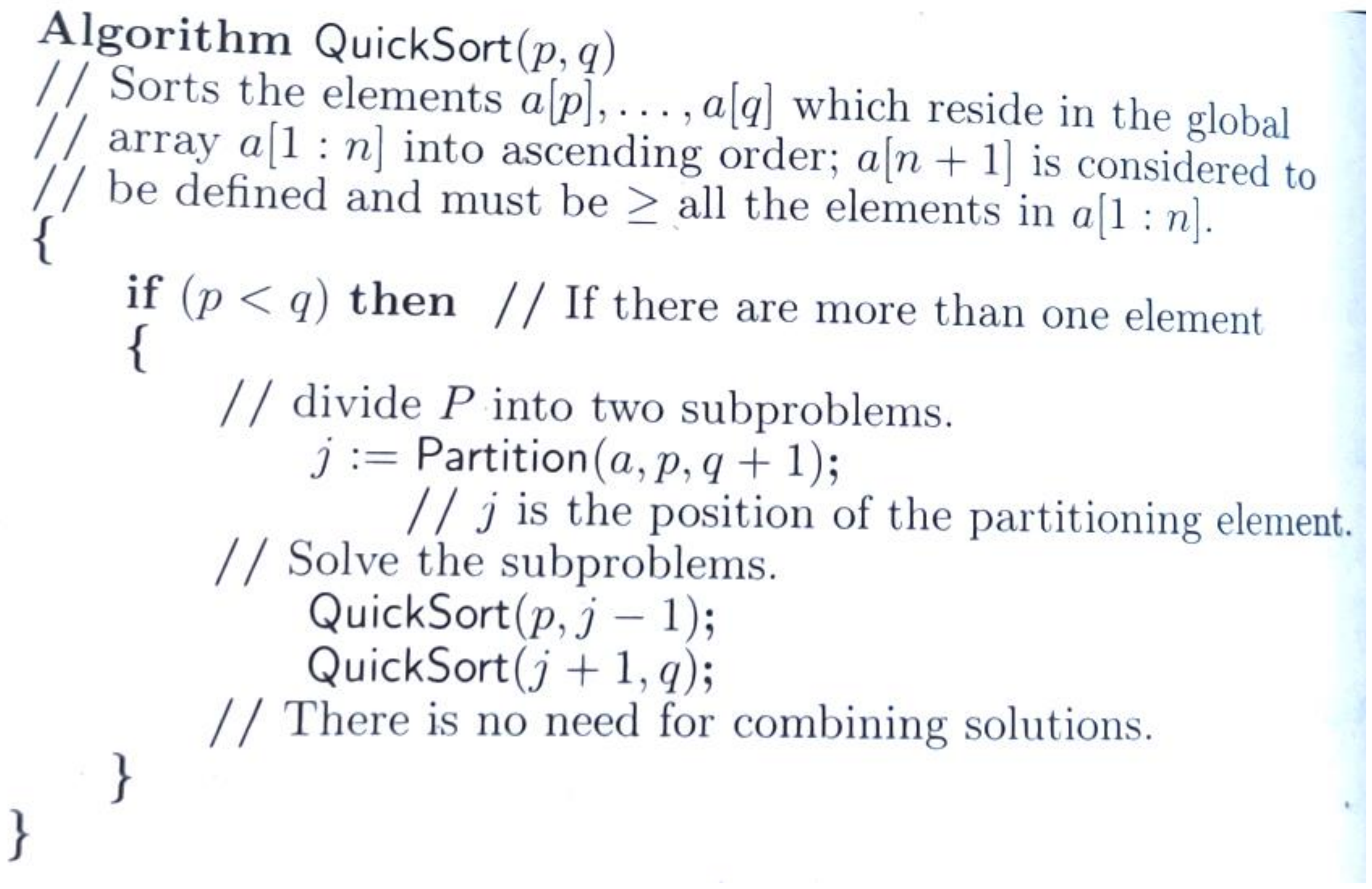
Recursion: Quicksort Sub-arrays



Example

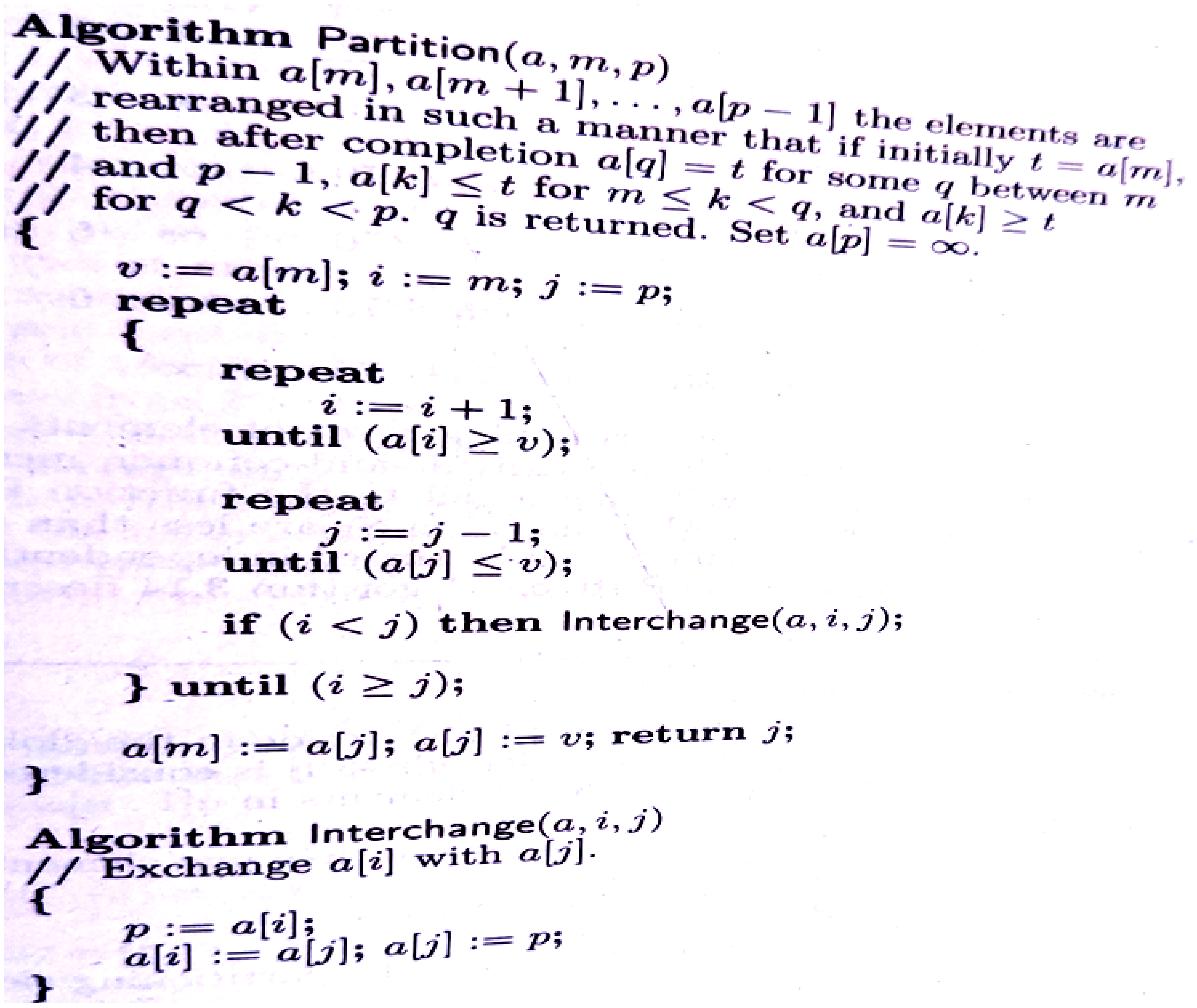
- A={65,70,75,80,85,60,55,50,45}

Quick Sort



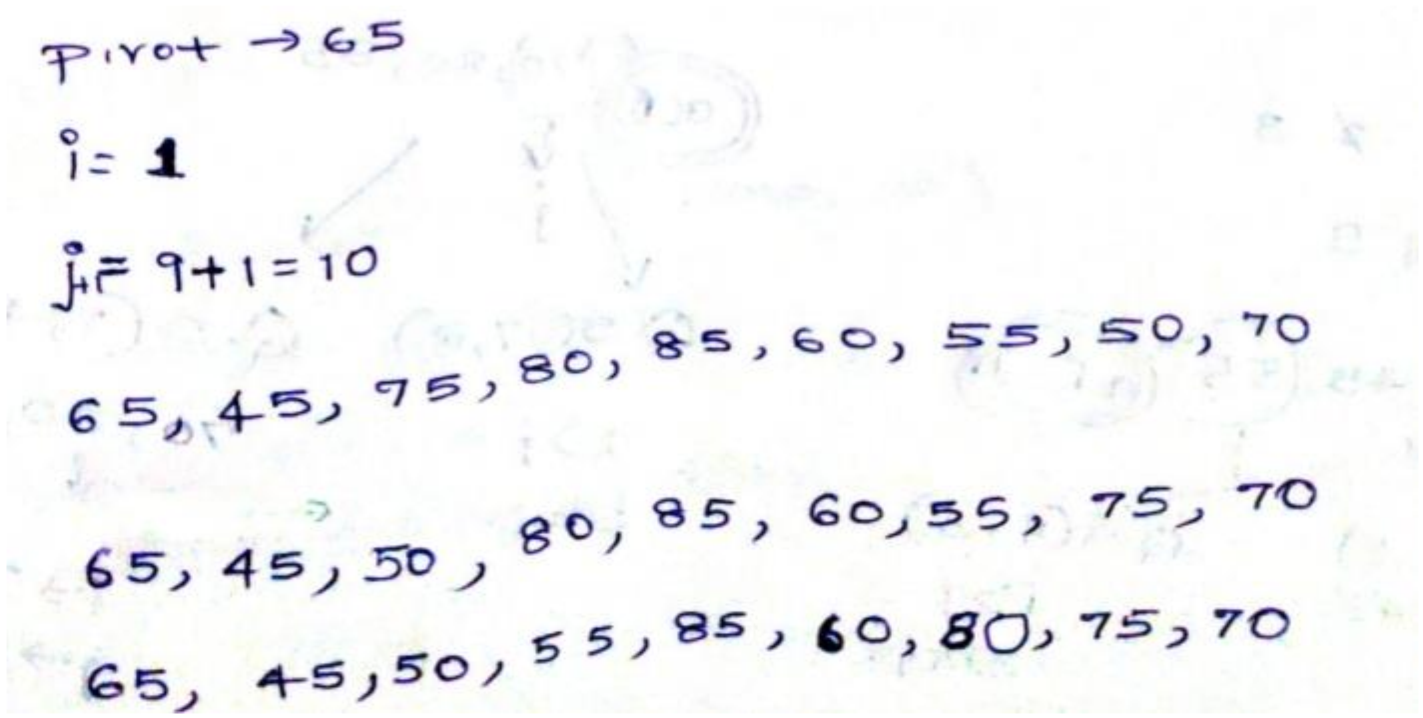
```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1 : n]$ into ascending order; $a[n + 1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1 : n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q + 1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j - 1$);
            QuickSort($j + 1, q$);
        // There is no need for combining solutions.
    }
}
```

Quick Sort



```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p - 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p - 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j - 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Solution



Pivot → 65

i = 1

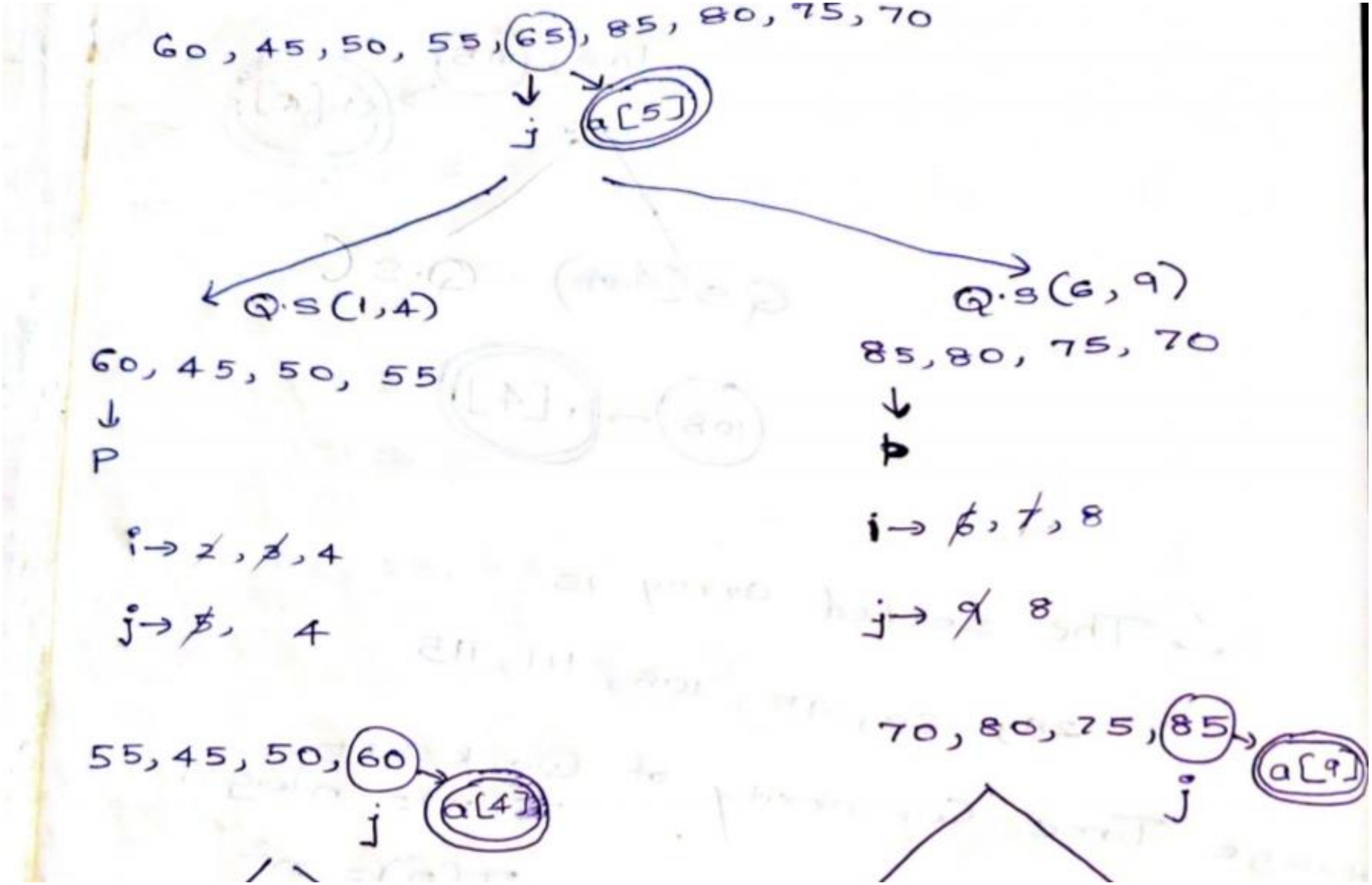
j = 9 + 1 = 10

65, 45, 75, 80, 85, 60, 55, 50, 70

65, 45, 50, 80, 85, 60, 55, 75, 70

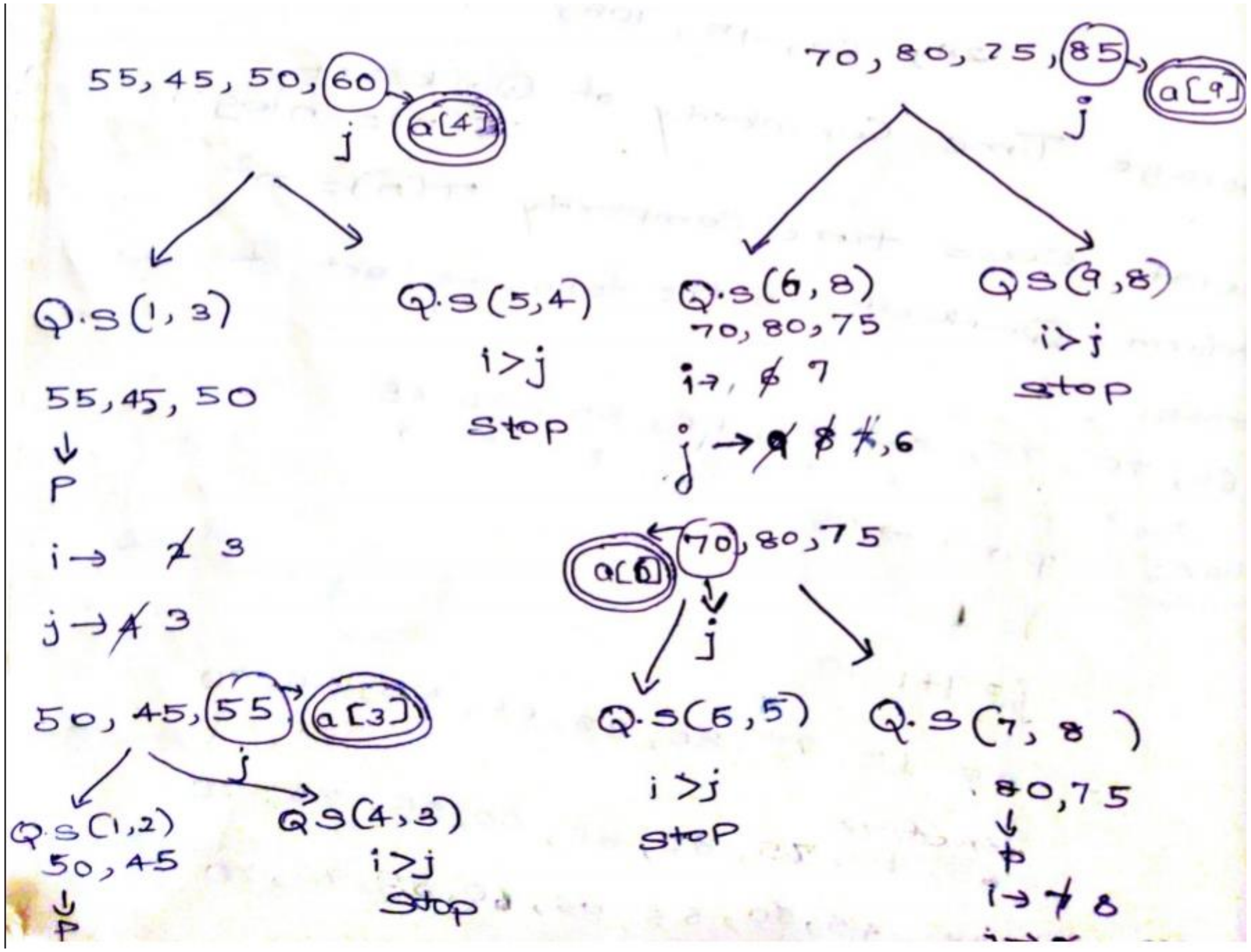
65, 45, 50, 55, 85, 60, 80, 75, 70

Solution



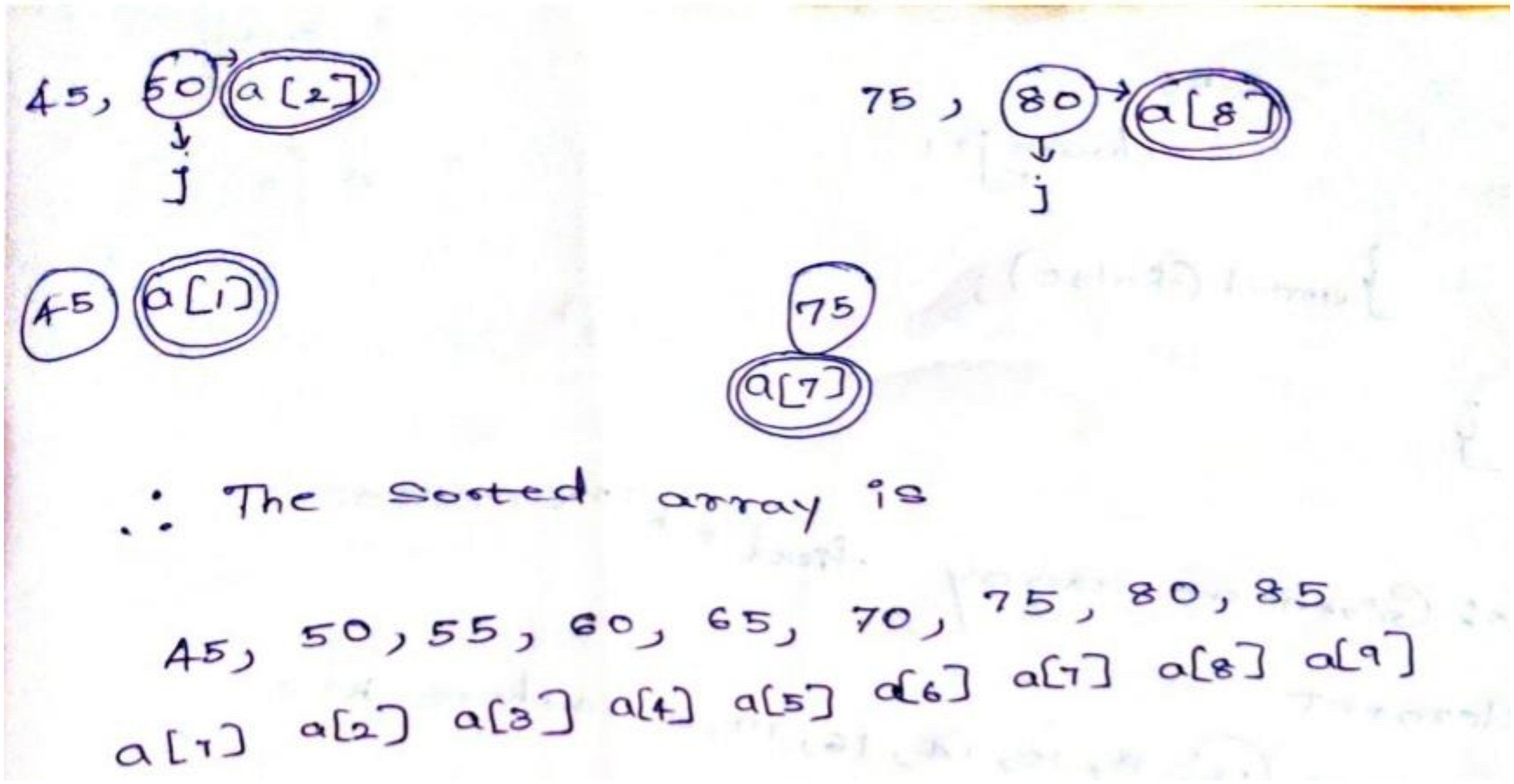
60, 45, 50, 55, 65, 85, 80, 75, 70
j
a[5]
Q.S(1,4)
Q.S(6,9)
60, 45, 50, 55
P
i→ 2, 3, 4
j→ 5, 4
85, 80, 75, 70
P
i→ 6, 7, 8
j→ 9 8
55, 45, 50, 60
j
a[4]
70, 80, 75, 85
j
a[9]

Solution



55, 45, 50, 60
a[4]
j
Q.S(1, 3)
55, 45, 50
P
i → 2 3
j → 4 3
50, 45, 55
a[3]
j
Q.S(1,2)
50, 45
P
Q S(4, 3)
i>j
Stop
Q.S(5,4)
i>j
Stop
70, 80, 75, 85
a[9]
j
Q.S(6, 8)
70, 80, 75
i→, 6 7
j → 9 8 7, 6
70, 80, 75
a[6]
j
Q.S(6, 5)
i>j
stop
Q.S(7, 8)
80, 75
P
i → 7 8
QS(9, 8)
i>j
stop

Solution



$$\begin{tikzcd} 45, & 50 \arrow[r] \arrow[d] & a[2] \\ & j & \end{tikzcd}$$

$$\begin{tikzcd} 75, & 80 \arrow[r] \arrow[d] & a[8] \\ & j & \end{tikzcd}$$

45 a[1]

75
a[7]

∴ The sorted array is

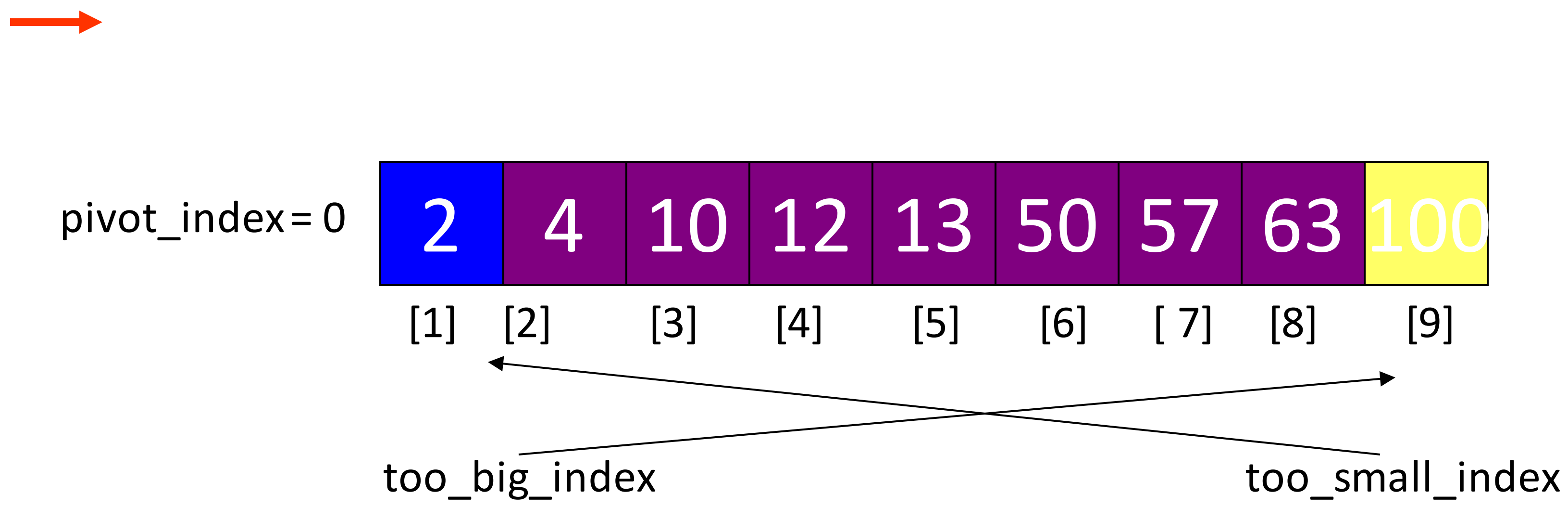
45,	50,	55,	60,	65,	70,	75,	80,	85
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Time Complexity

- Best case running time: $\Omega(n \log_2 n)$
  - $T(n)=T(n/2)+T(n/2)+n$
- Worst case running time: $O(n^2)$

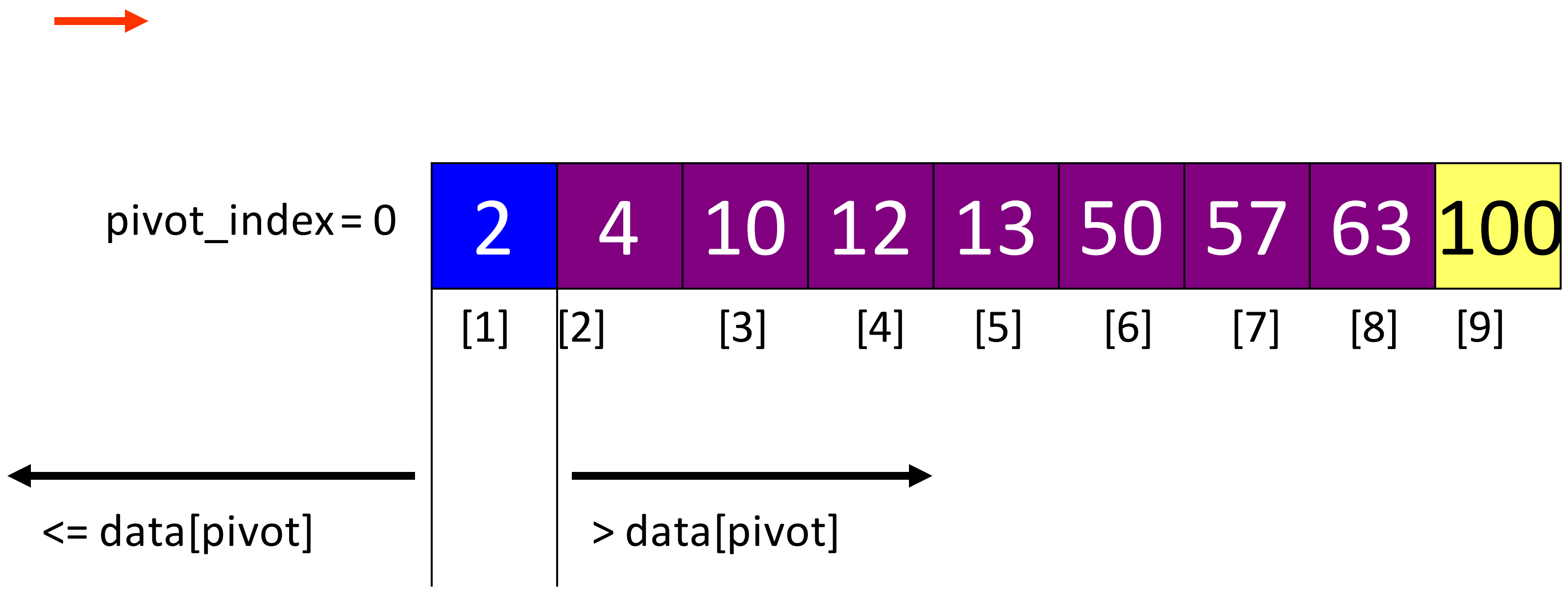
Example of Worst Case Time Complexity

- Quick sort has worst time complexity when the elements in the list are already sorted.



Example of Worst Case Time Complexity

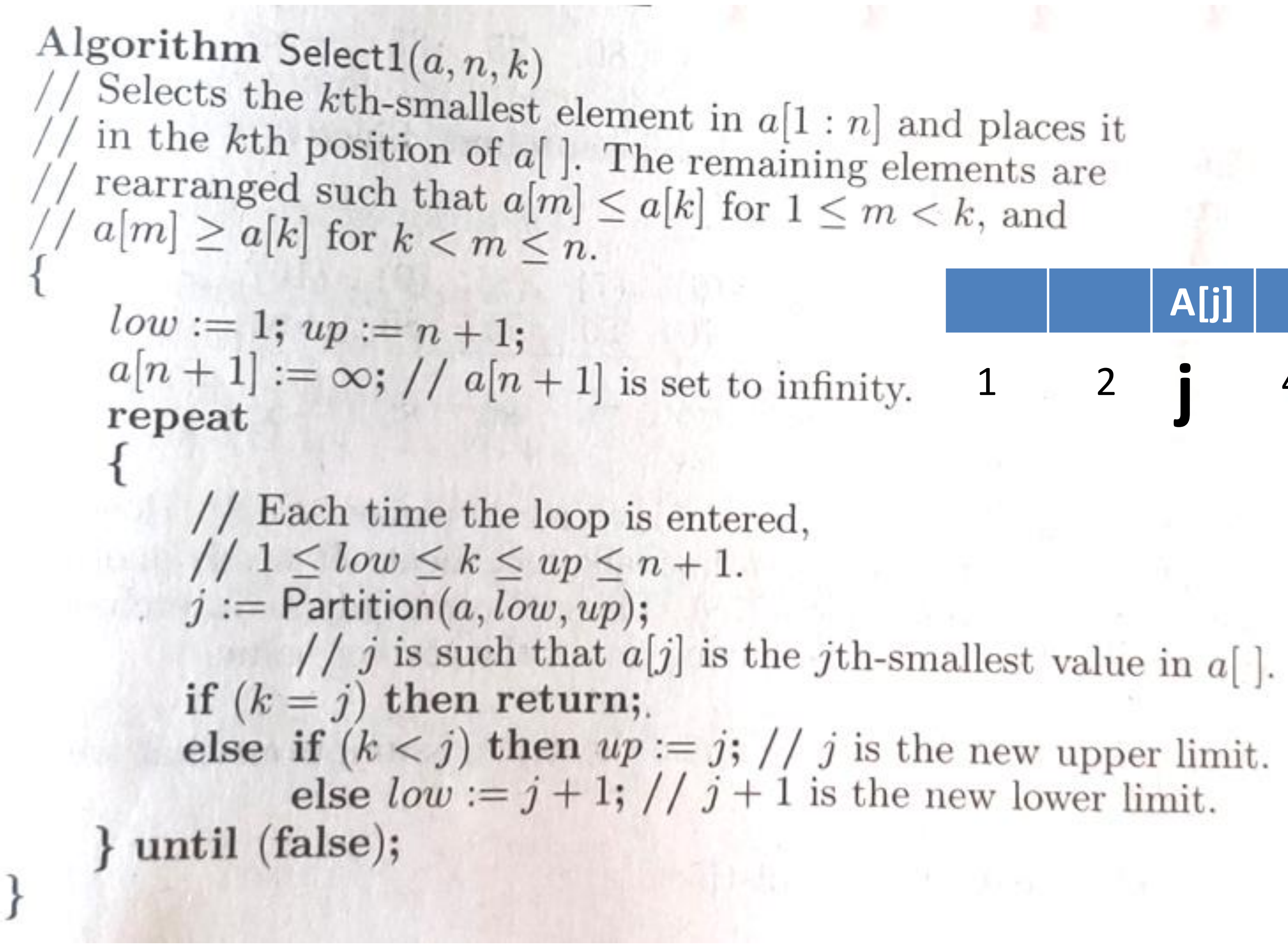
$$T(n)=n+(n-1)+(n-2)+....+2=n(n+1)/2=O(n^2)$$



Selection: Finding kth smallest element

- Select a pivot and partition the array with pivot at correct position j
- If position of pivot, j, is equal to k, return A[j].
- If j is less than k, discard array from start to j, and look for (k-j)th smallest element in right sub array, go to step 1.
- If j is greater than k, discard array from j to end and look for kth element in left subarray, go to step 1

Selection: Finding kth smallest element

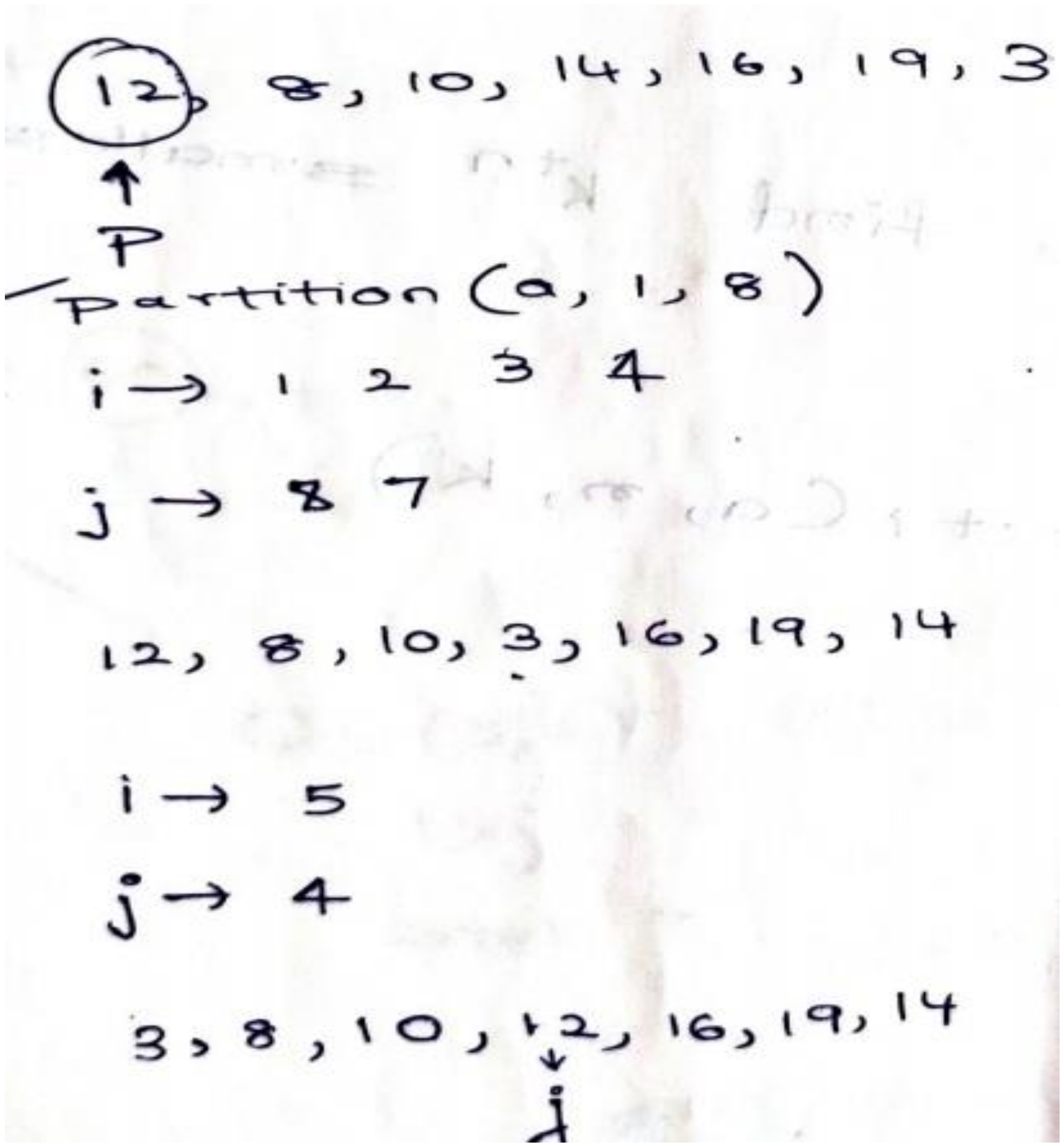


```
Algorithm Select1($a, n, k$)
// Selects the $k$th-smallest element in $a[1:n]$ and places it
// in the $k$th position of $a[\,]$. The remaining elements are
// rearranged such that $a[m] \leq a[k]$ for $1 \leq m < k$, and
// $a[m] \geq a[k]$ for $k < m \leq n$.
{
    $low := 1$; $up := n + 1$;
    $a[n+1] := \infty$; // $a[n+1]$ is set to infinity.
    repeat
    {
        // Each time the loop is entered,
        // $1 \leq low \leq k \leq up \leq n + 1$.
        $j :=$ Partition($a, low, up$);
                // $j$ is such that $a[j]$ is the $j$th-smallest value in $a[\,]$.
        if ($k = j$) then return;
        else if ($k < j$) then $up := j$; // $j$ is the new upper limit.
                else $low := j + 1$; // $j + 1$ is the new lower limit.
    } until (false);
}
```

Selection: Finding kth smallest element

- Ex:{12,8,10,14,16,19,3}
- Find second smallest element in this array.

Selection: Finding kth smallest element



$\boxed{12}$, 8, 10, 14, 16, 19, 3

$\uparrow$

P

partition (a, 1, 8)

$i \rightarrow$ 1 2 3 4

$j \rightarrow$ 8 7

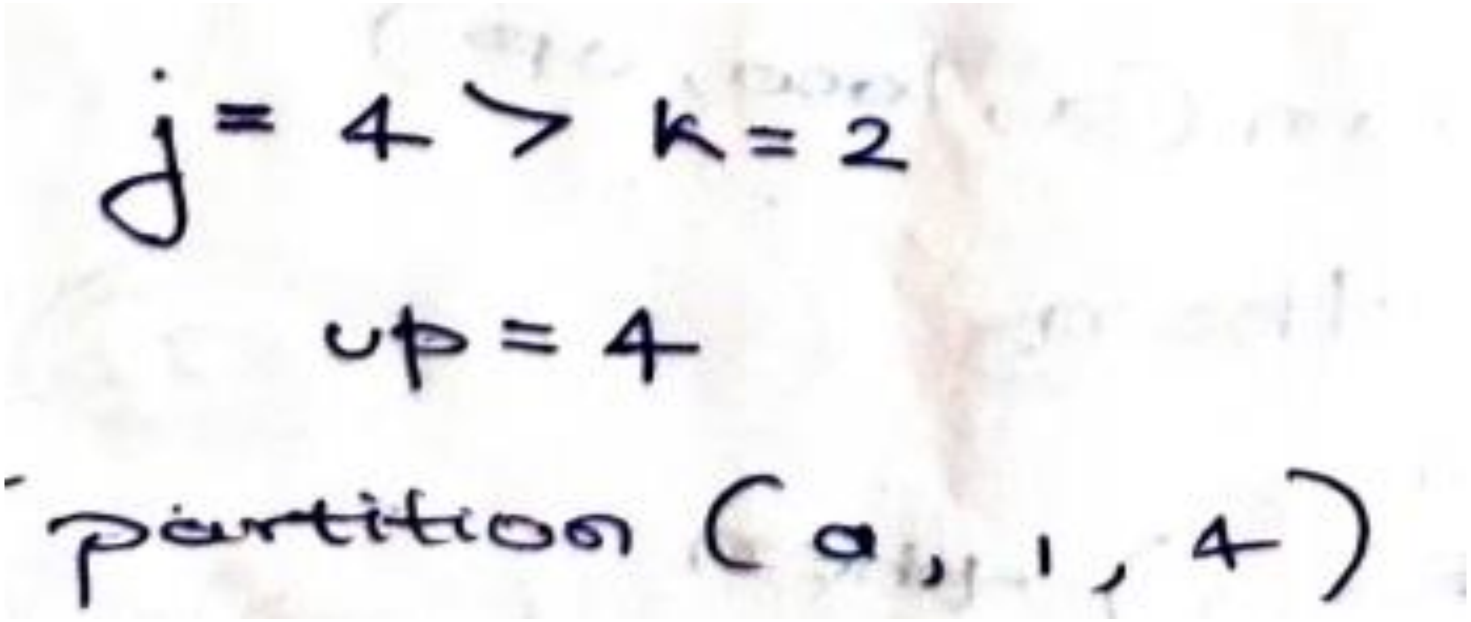
12, 8, 10, 3, 16, 19, 14

$i \rightarrow$ 5

$j \rightarrow$ 4

3, 8, 10, $\underset{\overset{\downarrow}{j}}{12}$, 16, 19, 14

Selection: Finding kth smallest element

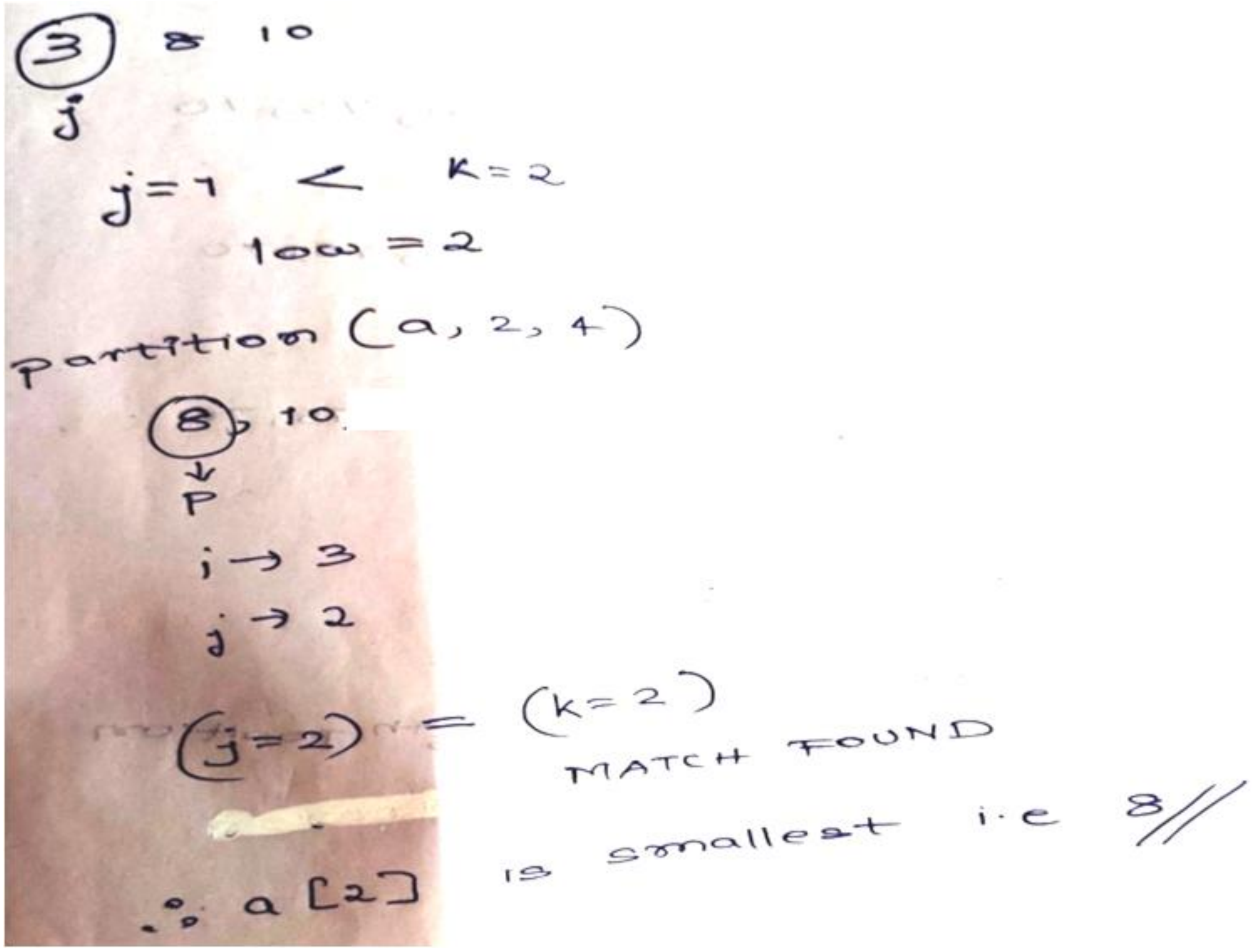


$j = 4 > k = 2$

$up = 4$

partition (a, 1, 4)

Selection: Finding kth smallest element



$\textcircled{3}$ 8 10

j

j = 1 < k = 2

low = 2

partition (a, 2, 4)

$\textcircled{8}$, 10

$\downarrow$

P

i → 3

j → 2

(j = 2) = (k = 2)

MATCH FOUND

∴ a[2] is smallest i.e 8//

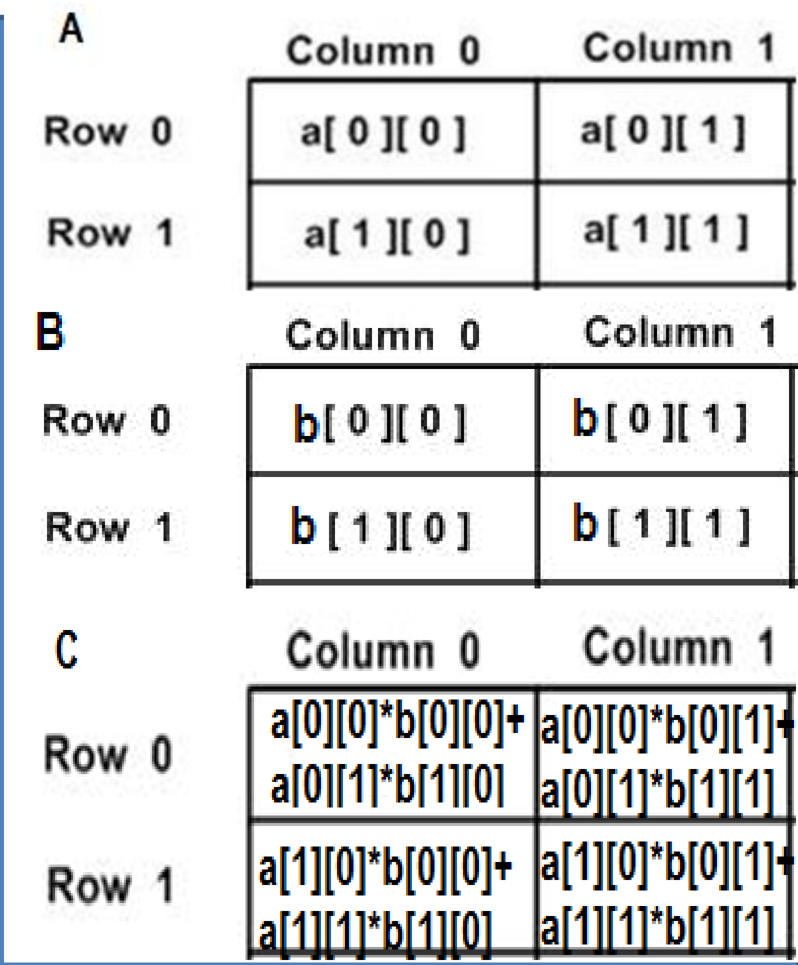
Finding kth Smallest Element Time Complexity

- Time complexity of finding kth smallest element in list is O(nlogn)
- The worst case time complexity of the above solution is still $O(n^2)$.

Strassen’s Matrix Multiplication

- Naïve Matrix Mutiplication

```
for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
    {
      C[i][j] = 0;
      for (int k = 0; k < N; k++)
      {
        C[i][j] += A[i][k]*B[k][j];
      }
    }
  }
```



A	Column 0	Column 1
Row 0	a[0][0]	a[0][1]
Row 1	a[1][0]	a[1][1]

B	Column 0	Column 1
Row 0	b[0][0]	b[0][1]
Row 1	b[1][0]	b[1][1]

C	Column 0	Column 1
Row 0	a[0][0]*b[0][0]+ a[0][1]*b[1][0]	a[0][0]*b[0][1]+ a[0][1]*b[1][1]
Row 1	a[1][0]*b[0][0]+ a[1][1]*b[1][0]	a[1][0]*b[0][1]+ a[1][1]*b[1][1]

Divide and Conquer Matrix Multiplication

- Following is simple Divide and Conquer method to multiply two square matrices:
  1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
  2) Calculate following values recursively.
  ae + bg, af + bh, ce + dg and cf + dh.

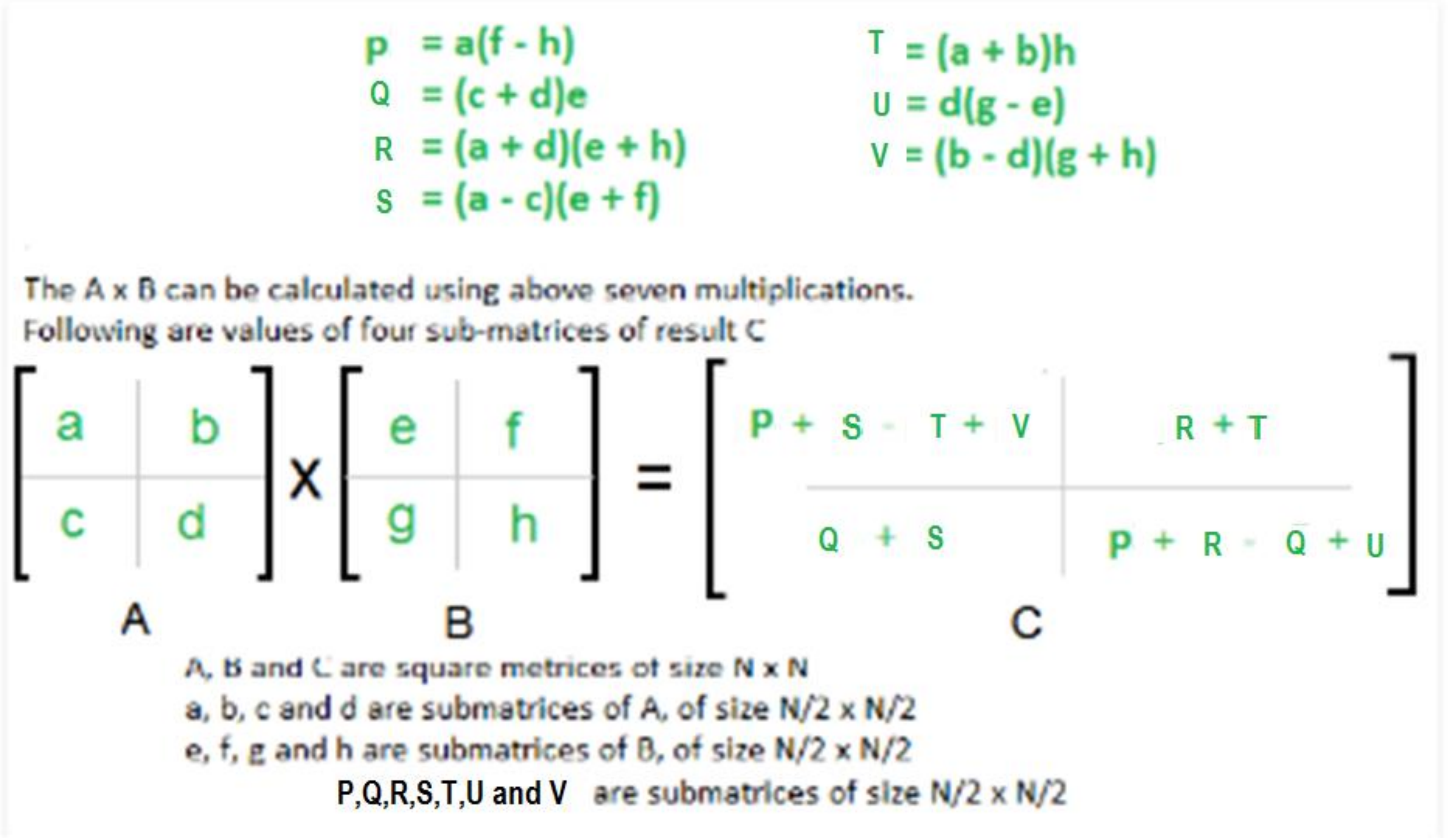
$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

$T(N) = 8T(N/2) + O(N^2)$

8-Multiplication
N^2-Addition

Strassen's Matrix Multiplication



$p = a(f - h)$

$Q = (c + d)e$

$R = (a + d)(e + h)$

$S = (a - c)(e + f)$

$T = (a + b)h$

$U = d(g - e)$

$V = (b - d)(g + h)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} P + S - T + V & R + T \\ Q + S & P + R - Q + U \end{bmatrix}}_{C}$$

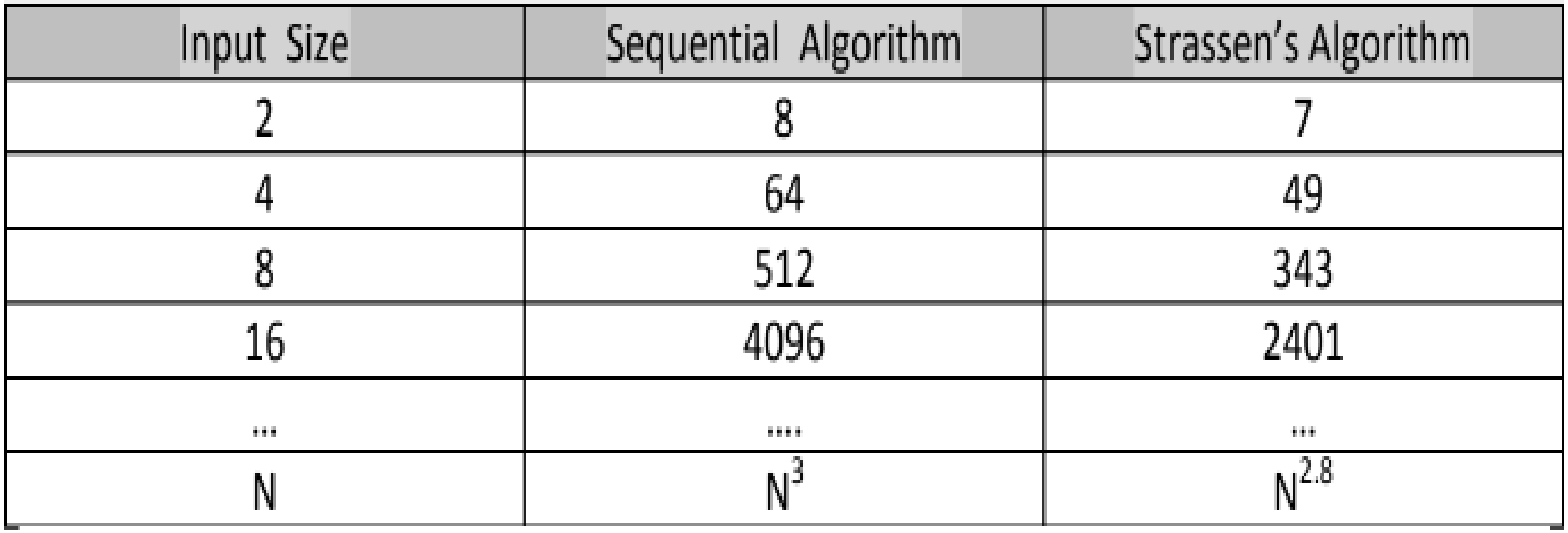
A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
P,Q,R,S,T,U and V are submatrices of size N/2 x N/2

Time Complexity= $T(n)=7T(N/2) + O(N^2)$

Strassen’s Time Complexity

- $T(N) = 7T(N/2) + O(N^2)$
- From Master's Theorem,
- $a=7$, $b=2$, $d=2$
- $a>b^d$, Hence $T(n)=N^{\log_b a}$
- $O(N^{Log 7})$ which is approximately $O(N^{2.8074})$

Performance Enhancement with Strassens



Input Size	Sequential Algorithm	Strassen’s Algorithm
2	8	7
4	64	49
8	512	343
16	4096	2401
...	....	...
N	N^3	$N^{2.8}$

Strassen’s Matrix Multiplication Example

- 1.Perform matrix multiplication using Strassen’s method.
- A= $\begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}$
- B= $\begin{pmatrix} 4 & 3 \\ 3 & 4 \end{pmatrix}$

Strassen’s Matrix Multiplication Example

$A_{11} = 2$, $A_{12} = 3$, $A_{21} = 3$, $A_{22} = 2$

$B_{11} = 4$, $B_{12} = 3$, $B_{21} = 3.$, $B_{22} = 4$

$P = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$P = (4) * (8) = 32$

$Q = (A_{21} + A_{22}) * B_{11}$

$Q = (3 + 2) * 4$

$Q = 20$

$R = A_{11} * (B_{12} - B_{22})$

$R = 2 * (3 - 4)$

$R = -2$

$S = A_{22} * (B_{21} - B_{11})$

$S = -2$

$T = (A_{11} + A_{12}) * B_{22}$

$T = 20$

Strassen's Matrix Multiplication Example

$U = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$U = [1] * 7$

$U = [7]$

$V = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$V = 1 * 7$

$V = 7$

For final matrix C

$C_{11} = P + S - T + V$

$C_{11} = 32 - 2 - 20 + 7$

$C_{11} = 17$

$C_{12} = R + T$

$C_{12} = -2 + 20$

$C_{12} = 18$

$C_{21} = Q + S$

$C_{21} = 20 - 2$

$C_{21} = 18$

$C_{22} = P + R - Q + U$

$C_{22} = 32 - 2 - 20 + 7$

$C_{22} = 17$

$$\therefore \text{Final matrix } C = \begin{bmatrix} 17 & 18 \\ 18 & 17 \end{bmatrix}$$

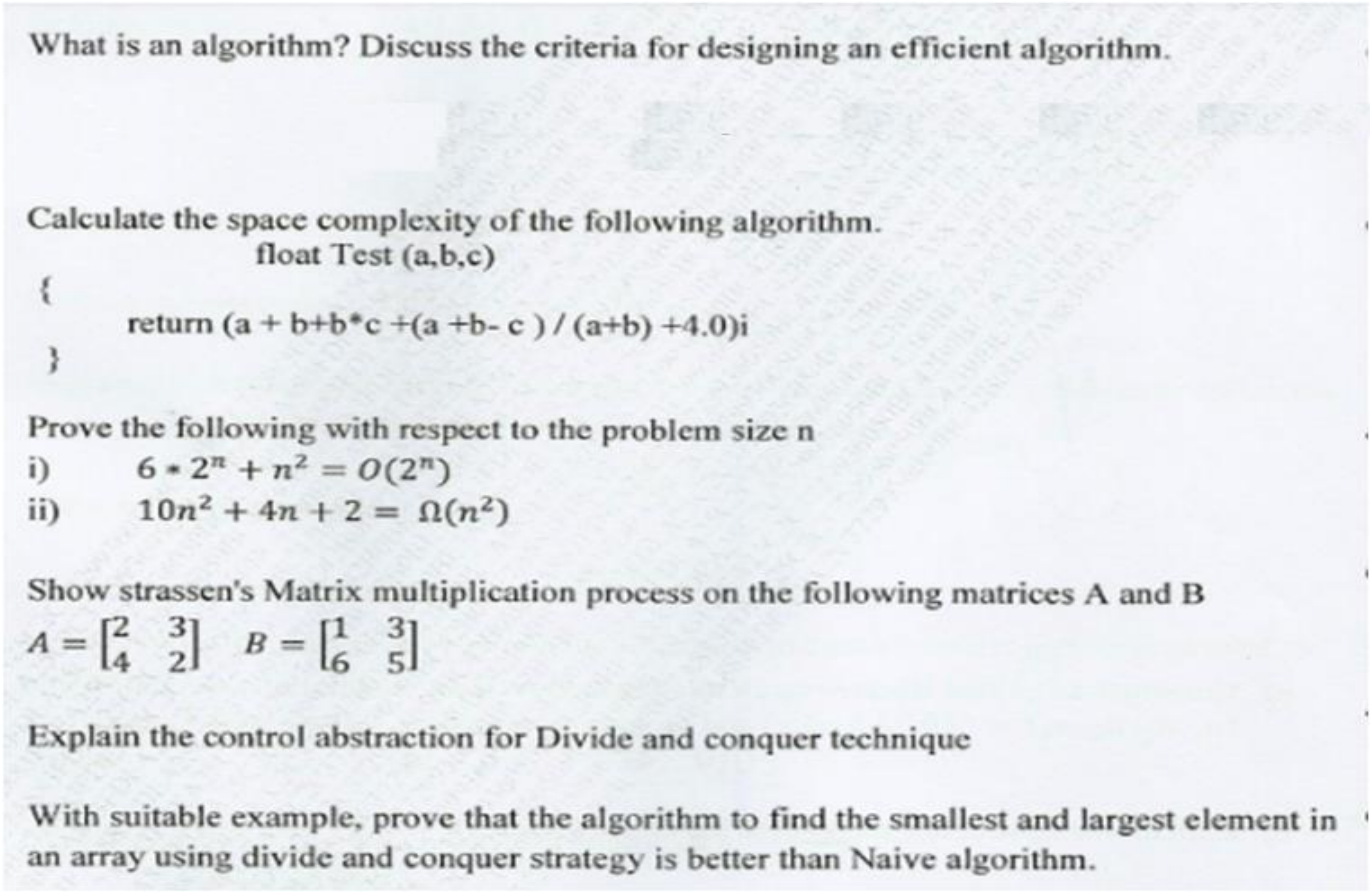
Strassen’s Matrix Multiplication Example

- 1)Solve matrix multiplication using Strassens multiplication

$$A=\begin{pmatrix} 2 & 1 & 1 & 3 \\ 6 & 0 & 4 & 5 \\ 3 & 2 & 1 & 8 \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

$$B=\begin{pmatrix} 1 & 0 & 0 & 3 \\ 2 & 1 & 1 & 6 \\ 7 & 1 & 8 & 3 \\ 0 & 5 & 4 & 0 \end{pmatrix}$$

Questions Asked



What is an algorithm? Discuss the criteria for designing an efficient algorithm.

Calculate the space complexity of the following algorithm.

```
                    float Test (a,b,c)
 {
          return (a + b+b*c +(a +b- c ) / (a+b) +4.0)i
 }
```

Prove the following with respect to the problem size n

i) $6 * 2^n + n^2 = O(2^n)$

ii) $10n^2 + 4n + 2 = \Omega(n^2)$

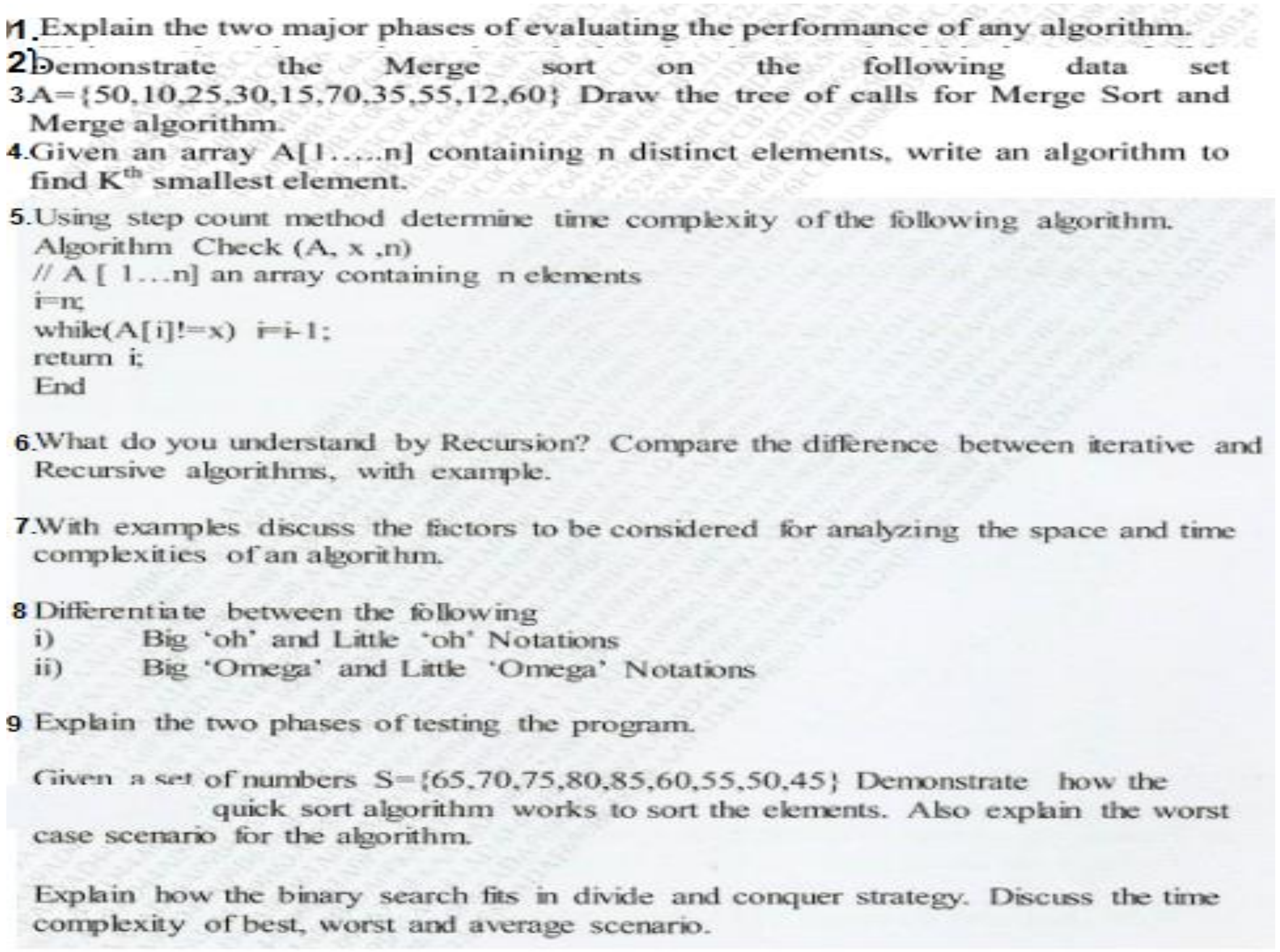
Show strassen's Matrix multiplication process on the following matrices A and B

$A = \begin{bmatrix} 2 & 3 \\ 4 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 3 \\ 6 & 5 \end{bmatrix}$

Explain the control abstraction for Divide and conquer technique

With suitable example, prove that the algorithm to find the smallest and largest element in an array using divide and conquer strategy is better than Naive algorithm.

Questions Asked



1.Explain the two major phases of evaluating the performance of any algorithm.

2.Demonstrate the Merge sort on the following data set

3.A={50,10,25,30,15,70,35,55,12,60} Draw the tree of calls for Merge Sort and Merge algorithm.

4.Given an array A[1.....n] containing n distinct elements, write an algorithm to find Kth smallest element.

5.Using step count method determine time complexity of the following algorithm.

```
Algorithm Check (A, x ,n)
// A [ 1...n] an array containing n elements
i=n;
while(A[i]!=x) i=i-1;
return i;
End
```

6.What do you understand by Recursion? Compare the difference between iterative and Recursive algorithms, with example.

7.With examples discuss the factors to be considered for analyzing the space and time complexities of an algorithm.

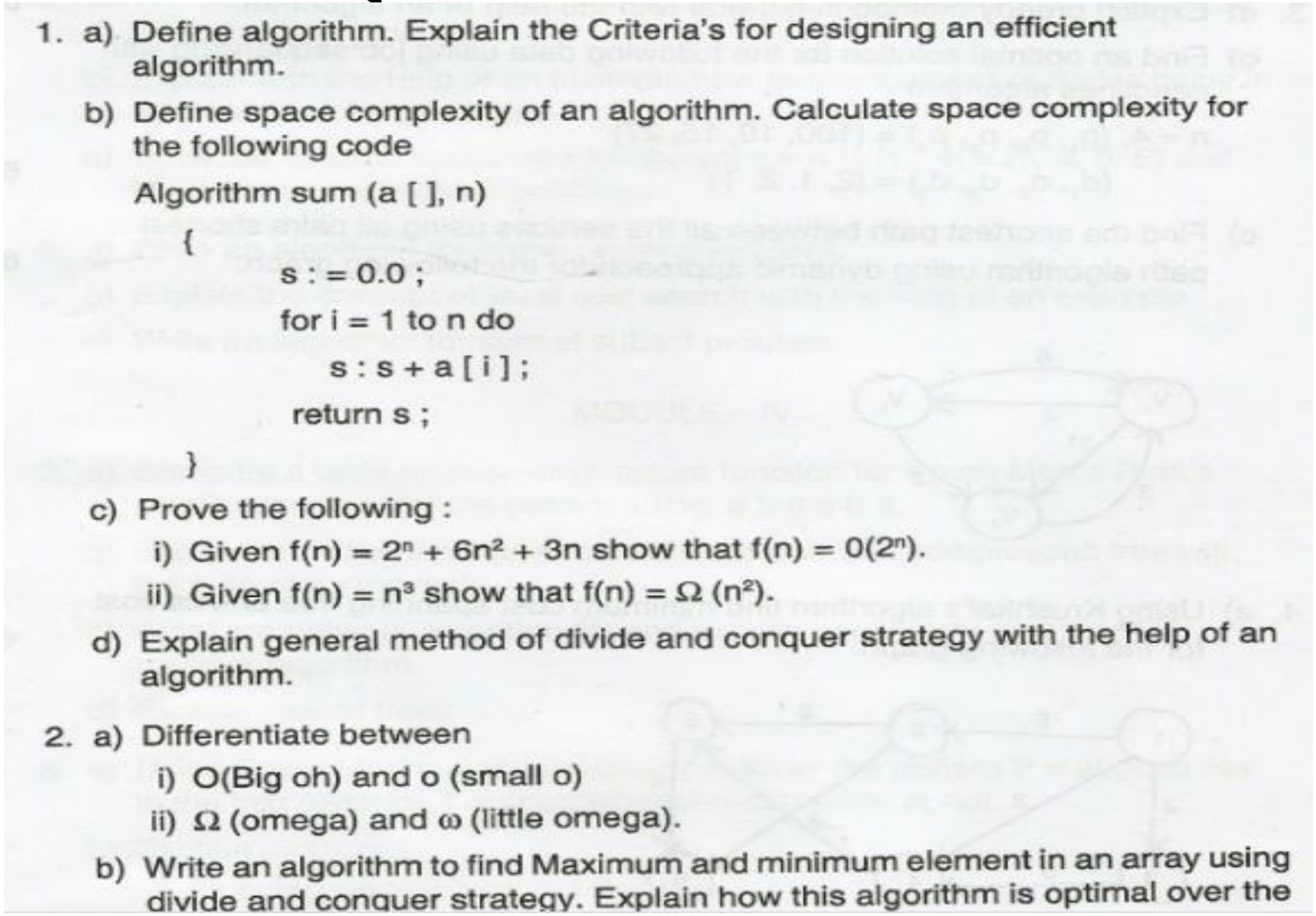
8 Differentiate between the following
i) Big 'oh' and Little 'oh' Notations
ii) Big 'Omega' and Little 'Omega' Notations

9 Explain the two phases of testing the program.

Given a set of numbers S={65,70,75,80,85,60,55,50,45} Demonstrate how the quick sort algorithm works to sort the elements. Also explain the worst case scenario for the algorithm.

Explain how the binary search fits in divide and conquer strategy. Discuss the time complexity of best, worst and average scenario.

Questions Asked



1. a) Define algorithm. Explain the Criteria's for designing an efficient algorithm.

   b) Define space complexity of an algorithm. Calculate space complexity for the following code

```
Algorithm sum (a [ ], n)
{
        s : = 0.0 ;
        for i = 1 to n do
            s : s + a [ i ] ;
         return s ;
}
```

   c) Prove the following :

      i) Given $f(n) = 2^n + 6n^2 + 3n$ show that $f(n) = 0(2^n)$.

      ii) Given $f(n) = n^3$ show that $f(n) = \Omega (n^2)$.

   d) Explain general method of divide and conquer strategy with the help of an algorithm.

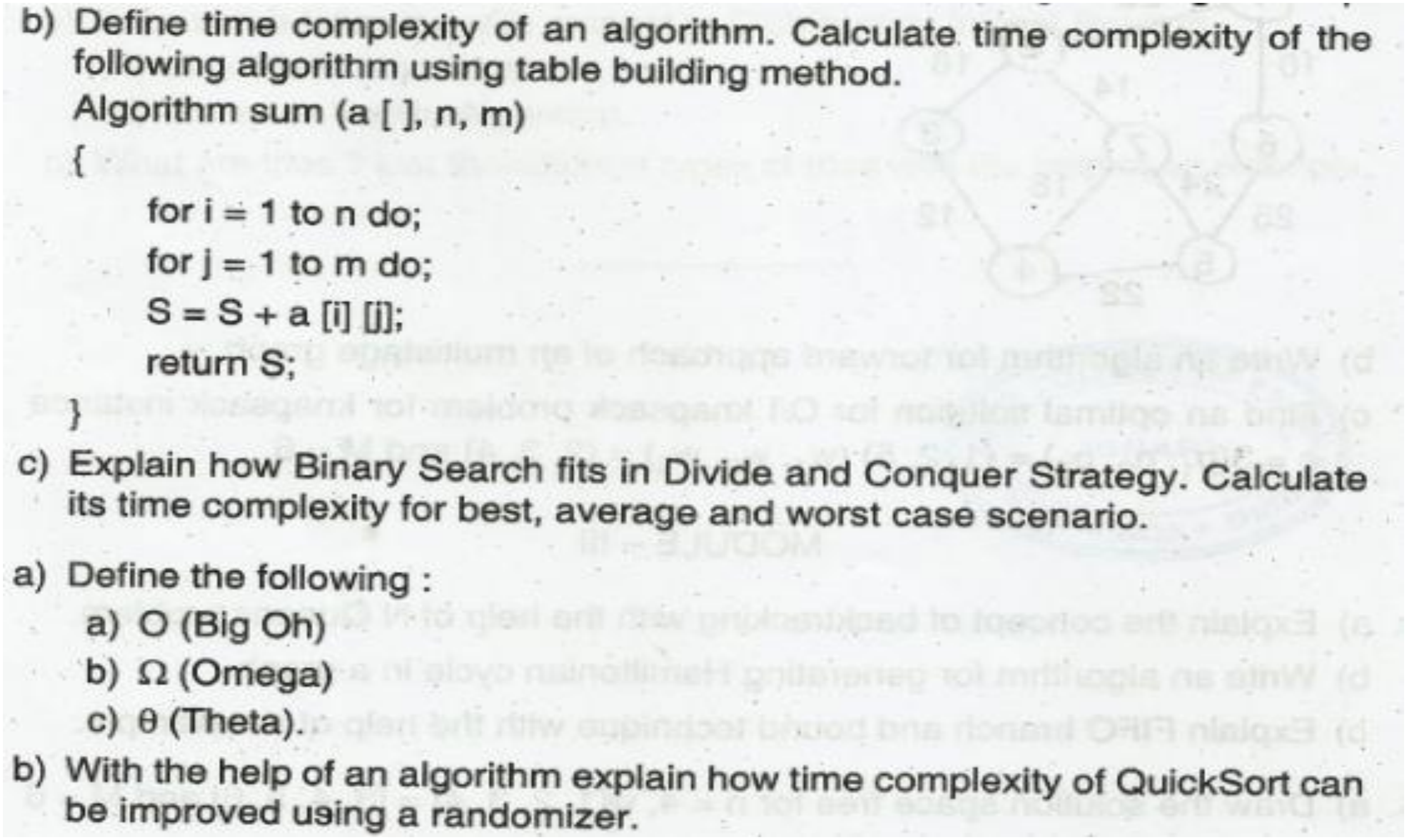
2. a) Differentiate between

      i) O(Big oh) and o (small o)

      ii) Ω (omega) and ω (little omega).

   b) Write an algorithm to find Maximum and minimum element in an array using divide and conquer strategy. Explain how this algorithm is optimal over the

Questions Asked



b) Define time complexity of an algorithm. Calculate time complexity of the following algorithm using table building method.

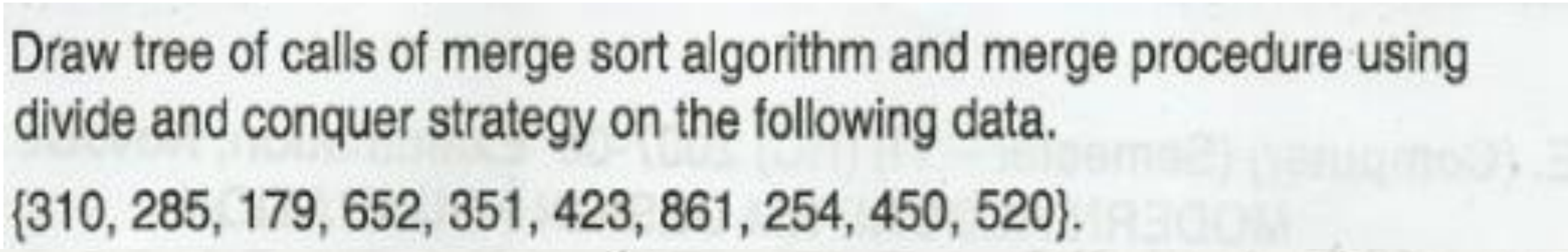
```
Algorithm sum (a [ ], n, m)
{
        for i = 1 to n do;
        for j = 1 to m do;
        S = S + a [i] [j];
        return S;
}
```

c) Explain how Binary Search fits in Divide and Conquer Strategy. Calculate its time complexity for best, average and worst case scenario.

a) Define the following :
  a) O (Big Oh)
  b) Ω (Omega)
  c) θ (Theta).

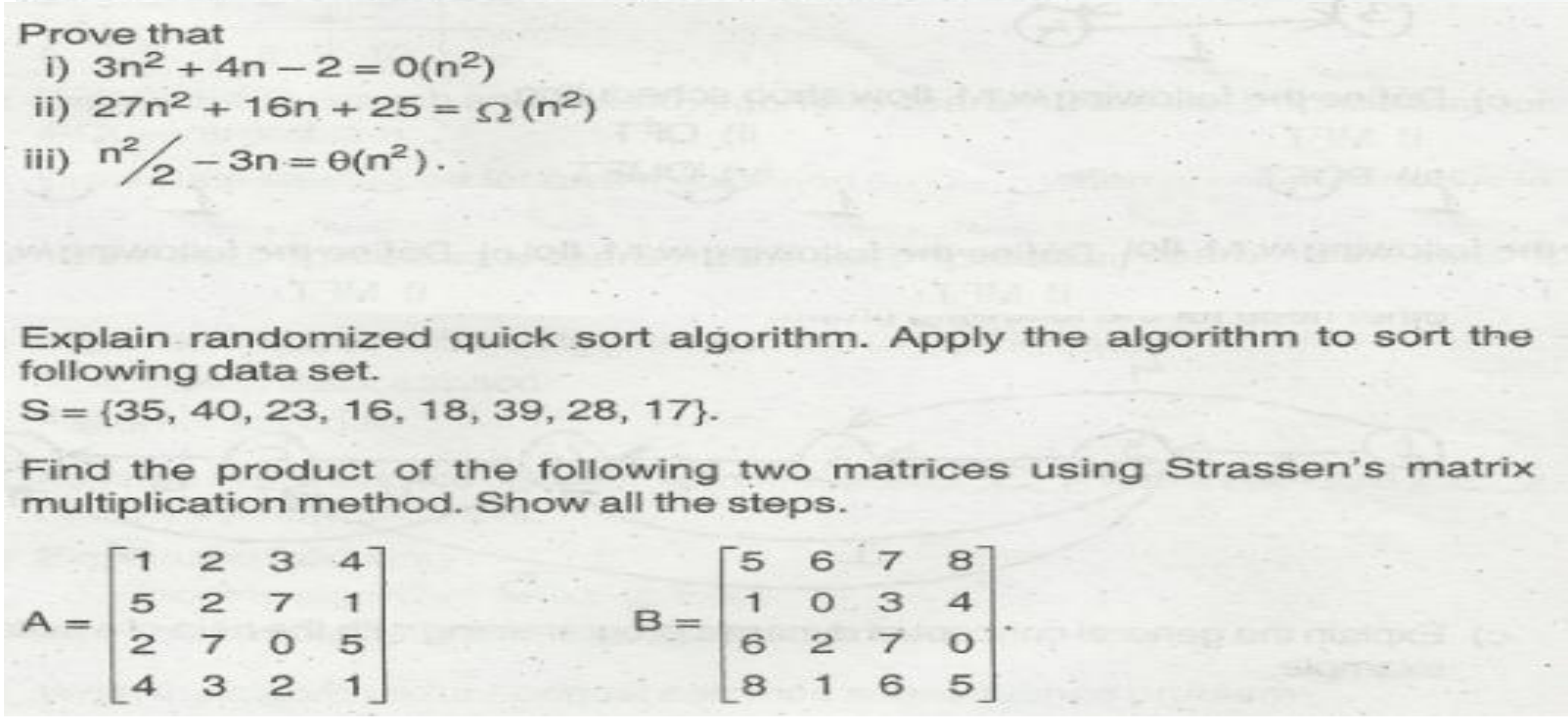
b) With the help of an algorithm explain how time complexity of QuickSort can be improved using a randomizer.

Questions Asked



Draw tree of calls of merge sort algorithm and merge procedure using divide and conquer strategy on the following data.

{310, 285, 179, 652, 351, 423, 861, 254, 450, 520}.



Prove that

i) $3n^2 + 4n - 2 = O(n^2)$

ii) $27n^2 + 16n + 25 = \Omega(n^2)$

iii) $n^2/2 - 3n = \theta(n^2)$.

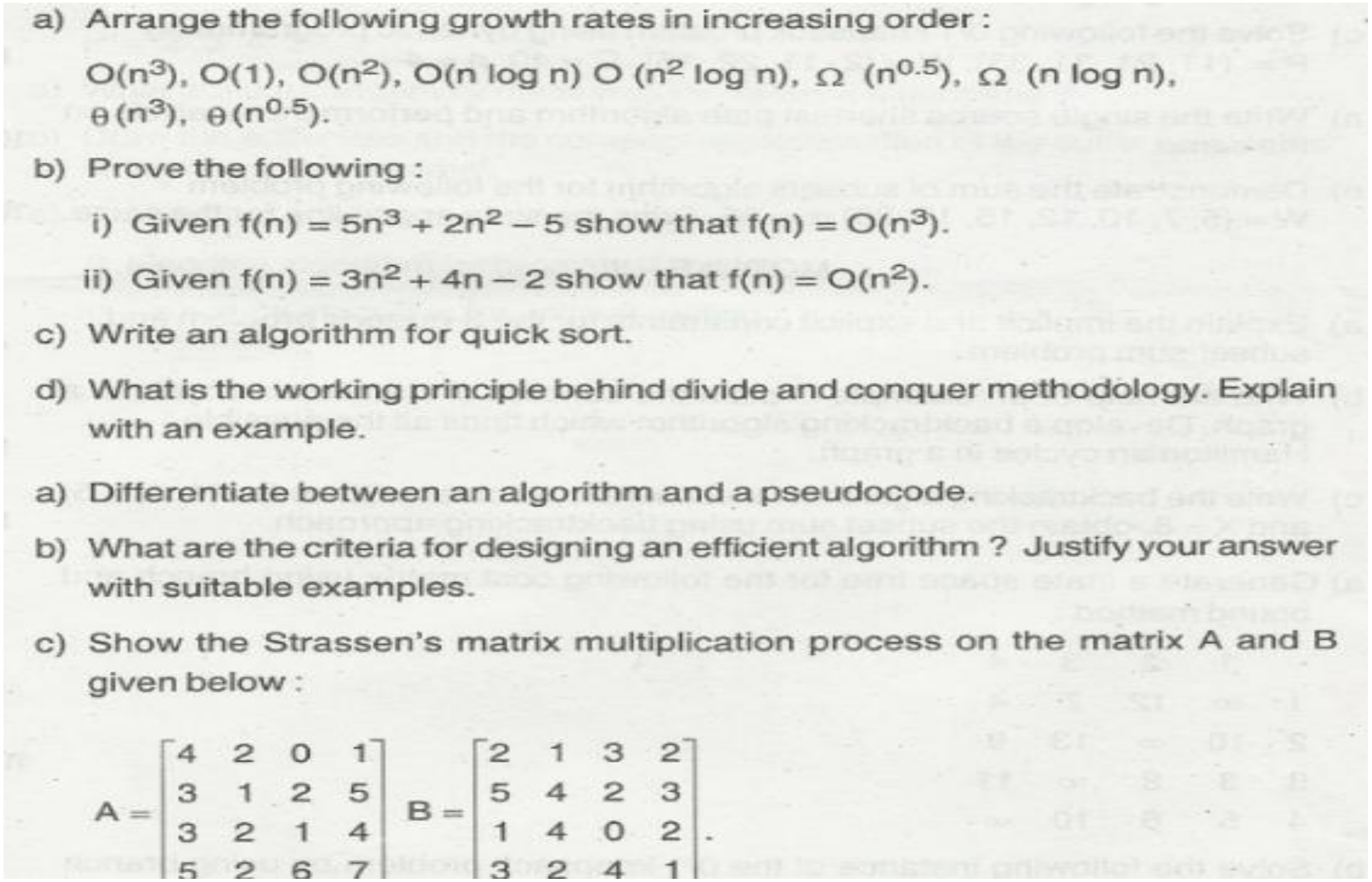
Explain randomized quick sort algorithm. Apply the algorithm to sort the following data set.

S = {35, 40, 23, 16, 18, 39, 28, 17}.

Find the product of the following two matrices using Strassen's matrix multiplication method. Show all the steps.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 2 & 7 & 1 \\ 2 & 7 & 0 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 1 & 0 & 3 & 4 \\ 6 & 2 & 7 & 0 \\ 8 & 1 & 6 & 5 \end{bmatrix}.$$

Questions Asked



a) Arrange the following growth rates in increasing order :

$O(n^3)$, $O(1)$, $O(n^2)$, $O(n \log n)$ $O(n^2 \log n)$, $\Omega(n^{0.5})$, $\Omega(n \log n)$, $\theta(n^3)$, $\theta(n^{0.5})$.

b) Prove the following :

i) Given $f(n) = 5n^3 + 2n^2 - 5$ show that $f(n) = O(n^3)$.

ii) Given $f(n) = 3n^2 + 4n - 2$ show that $f(n) = O(n^2)$.

c) Write an algorithm for quick sort.

d) What is the working principle behind divide and conquer methodology. Explain with an example.

a) Differentiate between an algorithm and a pseudocode.

b) What are the criteria for designing an efficient algorithm ? Justify your answer with suitable examples.

c) Show the Strassen's matrix multiplication process on the matrix A and B given below :

$$A = \begin{bmatrix} 4 & 2 & 0 & 1 \\ 3 & 1 & 2 & 5 \\ 3 & 2 & 1 & 4 \\ 5 & 2 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 1 & 3 & 2 \\ 5 & 4 & 2 & 3 \\ 1 & 4 & 0 & 2 \\ 3 & 2 & 4 & 1 \end{bmatrix}.$$

Text Books

- 1 Fundamentals of Computer Algorithms – E. Horowitz et al, 2nd Edition UP.- Divide and Conquer Strategy
- 2. Introduction to Algorithms, 3th Edition, Thomas H Cormen, Charles E Lieserson, Ronald L Rivest and Clifford Stein, MIT Press/McGraw-Hill.- Recurrence Relation
- 3. Design and Analysis of Algorithm –V. V Muniswamy- Problems on Recurrence Relation(Reference Book)

Thank You

Some Additional Problems

Show that for any real constants a and b, where $b > 0$,
$(n+a)^b = \Theta(n^b)$

Solution:

$$\begin{aligned}
(n+a)^b &\le (n+|a|)^b, \text{ where } n > 0 \\
&\le (n+n)^b \text{ for } n \ge |a| \\
&= (2n)^b \\
&= c_1 \cdot n^b, \text{ where } \; c_1 = 2^b
\end{aligned}$$

Thus

$$(n+a)^b = \Omega(n^b). \tag{1}$$

$$\begin{aligned}
(n+a)^b &\ge (n-|a|)^b, \text{ where } n > 0 \\
&\ge (c_2'n)^b \text{ for } c_2' = 1/2 \text{ where } n \ge 2|a| \\
&\quad \text{as } n/2 \le n-|a|, \text{ for } n \ge 2|a|
\end{aligned}$$

Thus

$$(n+a)^b = O(n^b) \tag{2}$$

The result follows from 1 and 2 with $c_1 = 2^b, c_2 = 2^{-b}$, and $n_0 \ge 2|a|$.

Some Additional Problems

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

Solution:

(a)

Is $2^{n+1} = O(2^n)$? Yes.

$2^{n+1} = 2\ 2^n \leq c2^n$ where $c \geq 2$.

(b)

Is $2^{2n} = O(2^n)$? No.

$2^{2n} = 2^n \cdot 2^n$. Suppose $2^{2n} = O(2^n)$. Then there is a constant $c > 0$ such that $c > 2^n$. Since 2^n is unbounded, no such c can exist.

Hierarchy of functions

$1, \log_2 n, \sqrt[3]{n}, \sqrt{n}, n, n\log_2 n, n\sqrt{n}, n^2, n^3, 2^n, n!, n^n$

Each one is Big-Oh of any function to its right

Basic Rules of Logarithms

If $\log_z (x) = a$, then $x = z^a$

$\log_z (xy)$	$= \log_z (x) + \log_z (y)$
$\log_z (x/y)$	$= \log_z (x) - \log_z (y)$
$\log_z (x^y)$	$= y\log_z (x)$
If $x = y$	then $\log_z (x) = \log_z (y)$
If $x < y$	then $\log_z (x) < \log_z (y)$

$\log_z (-|x|)$ is undefined

Some Equations

- Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Geometric Series:

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} (x \neq 1)$$

•Special Cases of Geometric Series:

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \qquad \text{if } x < 1$$

Some Equations

- Harmonic Series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- Others:

$$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$