RV College of Engineering®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE New Delhi

Divide and Conquer

UNIT 2

Prof. Rajesh R M

Dept. of AIML

RV College of Engineering

Bengaluru

Go, Change the World

Divide and Conquer

Divide-and-conquer is probably the best-known general algorithm design technique.

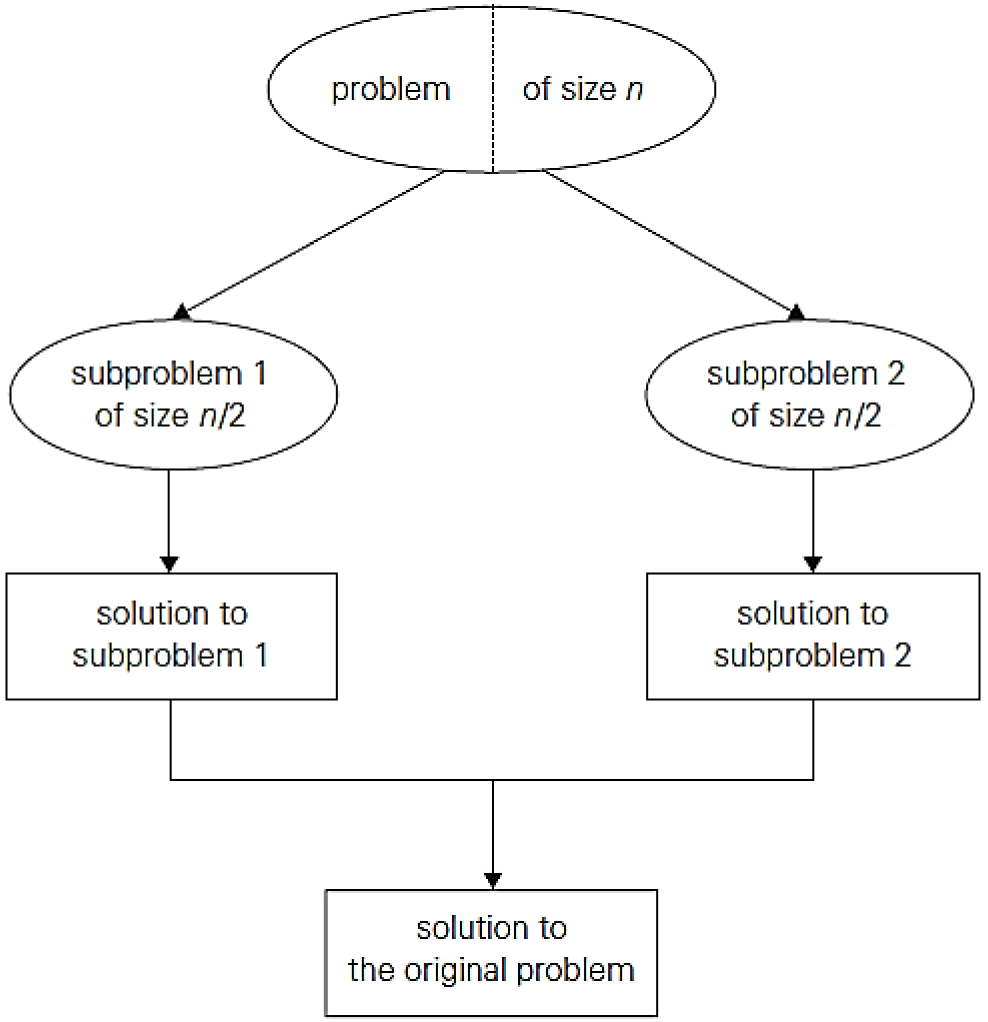
General plan to Divide and Conquer is as follows

1. A problem is divided into several sub problems of the same type, ideally of about equal size.

2. The sub problems are solved (typically recursively, though sometimes a different algorithm is employed, especially when sub problems become small enough).

3. If necessary, the solutions to the sub problems are combined to get a solution to the original problem.

Divide and Conquer


problem
of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

Divide and Conquer

Merge sort

Merge sort is a perfect example of a successful application of the divide-and conquer technique.

- It sorts a given array A[0..n − 1] by dividing it into two halves A[0..n/2 − 1] and A[n/2..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

The ***merging*** of two sorted arrays can be done as follows.

- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
- This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Divide and Conquer

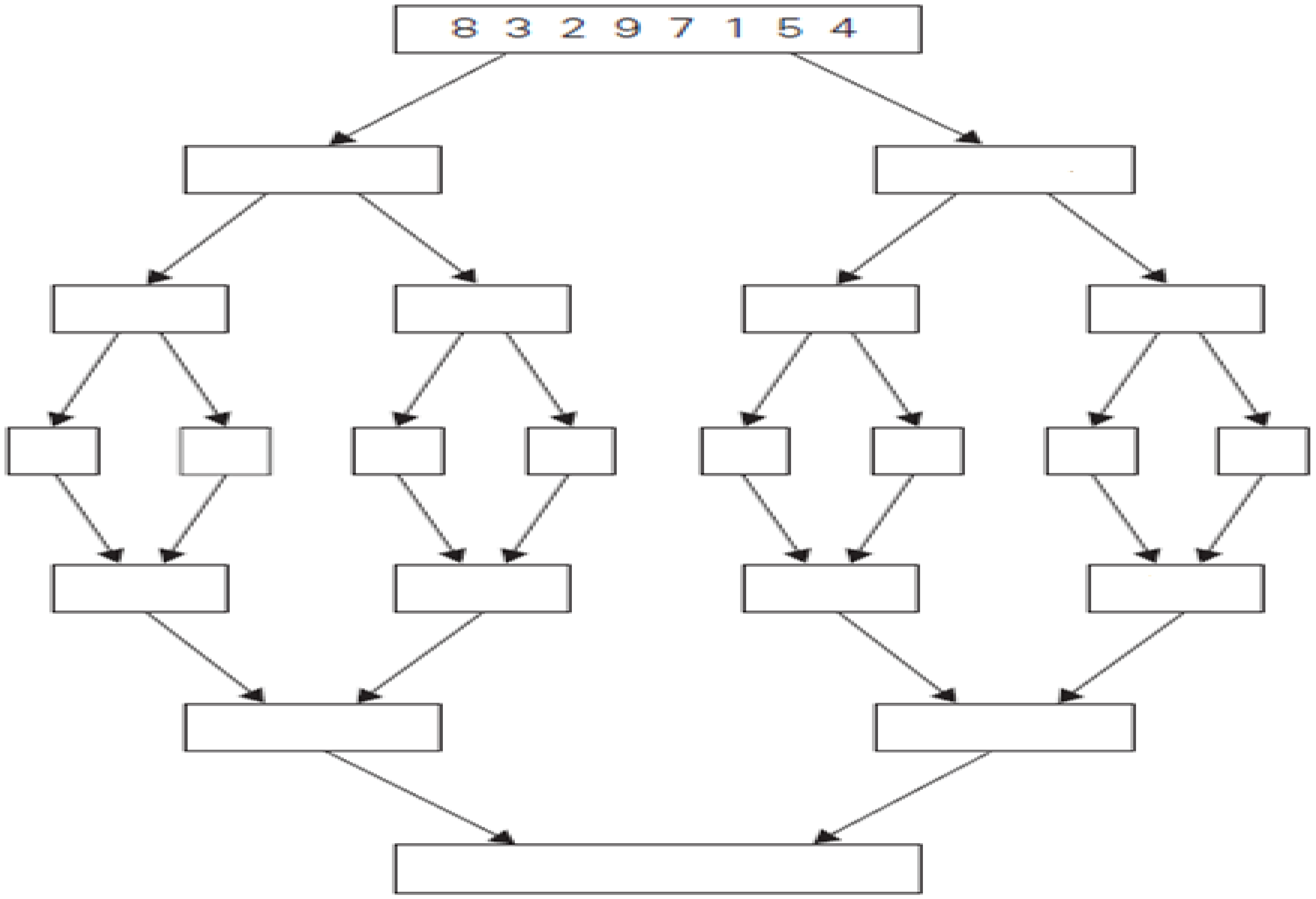
```
ALGORITHM Mergesort(A[0..n − 1])
//Sorts array A[0..n − 1] by recursive merge sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in no decreasing order
if n > 1
        copy A[0..n/2 − 1] to B[0..n/2 − 1]
        copy A[n/2..n − 1] to C[0..n/2 − 1]
        Mergesort(B[0..n/2 − 1])
        Mergesort(C[0..n/2 − 1])
        Merge(B, C, A) //see below
```

Divide and Conquer

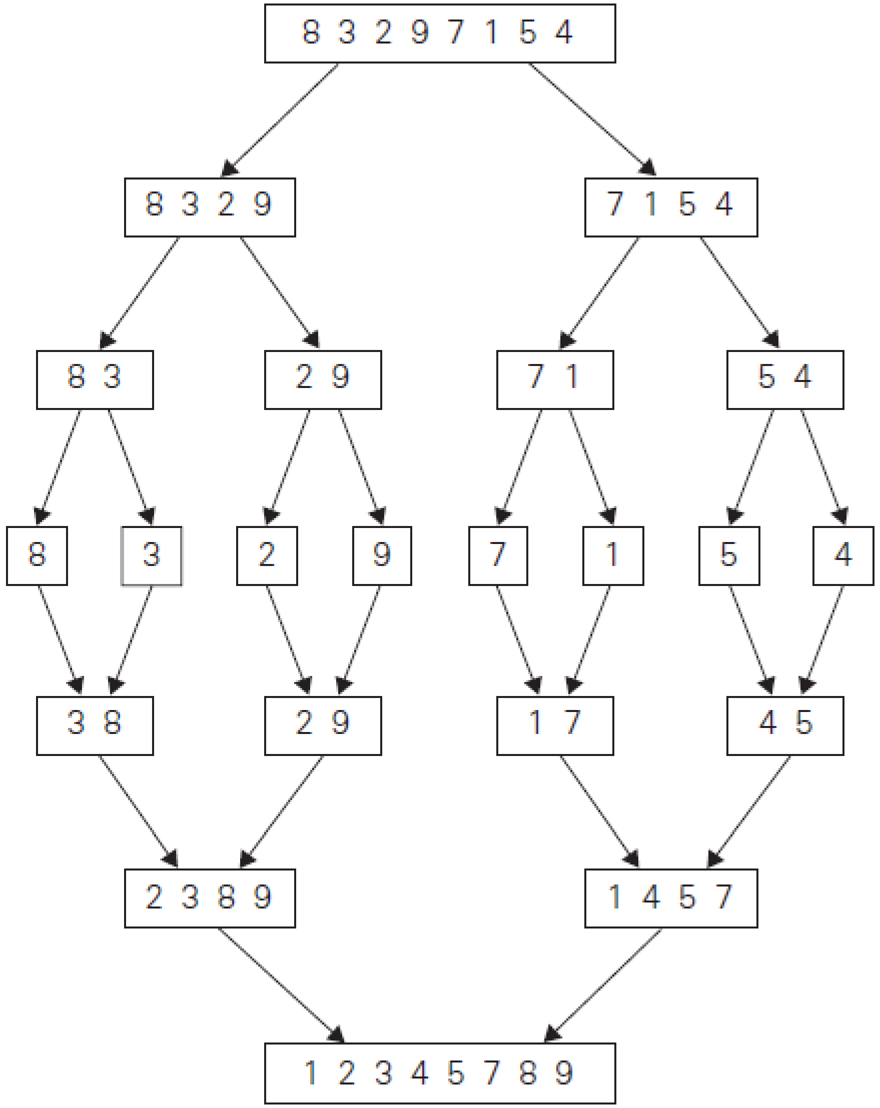
```
ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C

i ←0; j ←0; k←0
while i <p and j <q do
        if B[i]≤ C[j ]
                A[k]←B[i];
                i ←i + 1
        else
                A[k]←C[j ];
                j ←j + 1
        k←k + 1
if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
else
        copy B[i..p − 1] to A[k..p + q − 1]
```

Divide and Conquer



Divide and Conquer



Divide and Conquer

How efficient is mergesort? Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C(n)* is

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n > 1,\ C(1) = 0.$$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage.

At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1.

In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays).

Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ for } n > 1,\ C_{worst}(1) = 0.$$

The time complexity is $C(n)=nlogn$

Divide and Conquer

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence then

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

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Divide and Conquer

T(n)=$9T\left(\frac{n}{3}\right)$ +f(n) where f(n)=n

Here a=9

b=3

d=1

$a>b^d = 9>3^1$

Then the it belongs to $T(n) \in \Theta(n^{\log_b a})$

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

Apply Master Theorem

$$t(n) = n^{log_b a}$$

$$t(n) = n^{log_3 9}$$

$$t(n) = n^2$$

Divide and Conquer

Quick Sort

- Quick sort works based on divide and conquer technique, that means, we divide a problem into sub-problems and then solve them accordingly.
- This sorting algorithm includes selecting a pivot point and finding its appropriate place in the array by putting elements smaller to it on its left side, and the elements greater than it to its right side.
- We then create a partition around this correct position of pivot.
- This process of creating a partition is the backbone of the Quick Sort algorithm.

```
ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
// indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

Divide and Conquer

```
ALGORITHM Partition(A[l..r])
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right indices
l and r (l<r)
//Output: Partition of A[l..r], with the split position returned as
// this function's value
p←A[l]
i ←l; j ←r + 1
repeat
      repeat i ←i + 1 until A[i]≥ p
      repeat j ←j − 1 until A[j ]≤ p
      swap(A[i], A[j ])
until i ≥ j
swap(A[l], A[j ]) //undo last swap when i ≥ j
return j
```

Divide and Conquer

7	6	10	5	9	2	1	15	7

7	6	7	5	9	2	1	15	10

7	6	7	5	1	2	9	15	10

2	6	7	5	1	7	9	15	10

```
if l < r
        s ←Partition(A[l..r])
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

```
repeat
    repeat i ←i + 1 until A[i]≥ p
    repeat j ←j − 1 until A[j ]≤ p
    swap(A[i], A[j ])
until i ≥ j
swap(A[i], A[j ]) //undo last swap when i ≥ j
swap(A[l], A[j ])
return j
```

Divide and Conquer

2	6	7	5	1	7	9	15	10

2	6	7	5	1

2	1	7	5	6

1	2	7	5	6

7	5	6

6	5	7

5	6

9	15	10

9	15	10

9	10	15

1	2	5	6	7	7	9	10	15

Divide and Conquer

The best case scenario of Quicksort is

C(n)= Basic operations + no of comparison done to place pivot in middle position(f(n))

$C(n)=2C(\frac{n}{2}) + n$

Apply master theorem

A=2 $\quad C(n) = n^{log_b a}$

B=2

F(n)=n $\quad C(n) = n^{log_2 2}$

$$c(n) = n^1$$

According to master theorem $\quad a=b^d$

$2=2^1$

$2=2 \rightarrow$ case 2 of master theoren

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

$c(n)=\Theta\ (n^1 logn)$

$c(n)=\Theta\ (n\ logn)$

Divide and Conquer

The worst-case scenario of Quick sort is

$C(n)=C(n-1) + \text{comparison}$

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

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

$=C(n-3)+(n-2)+(n-1)+n+1$

$=3+\ldots\ldots\ldots+(1+n)+(n)+(n-1)(n-2)$

$$= \frac{(n+1)(n+2)}{2} - 3$$

$=n^2$

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

Divide and Conquer

Multiplication of Large Integers

the conventional pen-and-pencil algorithm for multiplying two *n*-digit integers, each of the *n* digits of the first number is multiplied by each of the *n* digits of the second number for the total of *n*2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than *n*2 digit multiplications, this turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^n + c_1 10^{n/2} + c_0$$

where
$c_2 = a_1 * b_1$ is the product of their first digits,
$c_0 = a_0 * b_0$ is the product of their second digits,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the *a*'s digits and the sum of the *b*'s digits minus the sum of c_2 and c_0.

Divide and Conquer

In brute fore method if n denoted the size of integers a and b The maximum number of multiplication done is n^2

By using divide and conquer we decrease the time complexity $n^{1.59}$

Step 1 : $c_0 = a_0 * b_0 \longrightarrow$ 1 Multiplication

Step 2 : $c_2 = a_1 * b_1 \longrightarrow$ 1 Multiplication

Step 3 : $c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0) \longrightarrow$ 1 Multiplication

Step 4 : $\boldsymbol{c = a * b = c_2 10^n + c_1 10^{n/2} + c_0} \longrightarrow$ 0 Multiplication

M(n)=3M$\left(\frac{n}{2}\right)$ + Time(Addition+Subtraction+shift)

M(n)=3M$\left(\frac{n}{2}\right)$ + c$\left(\frac{n}{2}\right)$ → M(n)=3M$\left(\frac{n}{2}\right)$ + cn

Apply Master Theorem M(n)=3M$\left(\frac{n}{2}\right)$ + cn

$$M(n) = n^{log_b a}$$

Here a =3 b = 2

$$M(n) = n^{log_2 3}$$

$$M(n) = n^{1.59}$$

Divide and Conquer

$a=1234$
$b=4512$
$a_0=34 \quad a_1=12$
$b_0=12 \quad b_1=45$

Step 1 : $c0 = a0 * b0 = 34*12 = 408$
Step 2 : $c2 = a1 * b1 = 12*45 = 540$
Step 3 : $c1 = (a1+a0)*(b1+b0)-(c2+c0) = (12+34)*(45+12) - (540+408)$

$= 46*57 - 948$
$= 2622-948$
$= 1674$

Step 4 : $\mathbf{c = a * b}$ $= c_2 10^n + c_1 10^{n/2} + c_0$

$= 540(10^4)+1674(10^2)+408$
$= 5400000+167400+408$
$= 5567808$

Divide and Conquer

Strassen's Matrix Multiplication

- This algorithm was published by V. Strassen in 1969
- Normal brute fore, Time complexity is n^3
- Strassen method is based on divide and conquer in which the time complexity is reduced to $n^{2.81}$

Let a and b are the two matrices of order n x m

A=$\begin{bmatrix} A1 & A2 \\ A3 & A4 \end{bmatrix}$ B=$\begin{bmatrix} B1 & B2 \\ B3 & B4 \end{bmatrix}$ C=$\begin{bmatrix} C1 & C2 \\ C3 & C4 \end{bmatrix}$

```
D=A1(B2-B4)

E=A4(B3-B1)

F=(A3+A4)B1

G=(A1+A2)B4

H=(A3-A1)(B1+B2)

I=(A2-A4)(B3+B4)

J=(A1+A4)(B1+B4)
```

```
To Compute C1,C2,C3,C4
C1=E+I+J-G
C2=D+G
C3=E+F
C4=D+H+J-F
```

Divide and Conquer

Example

A=$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$ B=$\begin{bmatrix}1 & 1\\2 & 2\end{bmatrix}$ C=$\begin{bmatrix}5 & 5\\11 & 11\end{bmatrix}$

```
D=A1(B2-B4)=1(1-2)=-1
E=A4(B3-B1)=4(2-1)=4
F=(A3+A4)B1=(3+4)1=7
G=(A1+A2)B4=(1+2)2=6
H=(A3-A1)(B1+B2)=(3-1)(1+1)=4
I=(A2-A4)(B3+B4)=(2-4)(2+2)=-8
J=(A1+A4)(B1+B4)=(1+4)(1+2)=15
```

```
To Compute C1,C2,C3,C4
C1=E+I+J-G = 4-8+15-6 =5
C2=D+G = -1+6 = 5
C3=E+F = 4+7 = 11
C4=D+H+J-F -1+4+15-7 = 11
```

Divide and Conquer

Time Complexity

D=A1(B2-B4)	→	1
E=A4(B3-B1)	→	1
F=(A3+A4)B1	→	1
G=(A1+A2)B4	→	1
H=(A3-A1)(B1+B2)	→	1
I=(A2-A4)(B3+B4)	→	1
J=(A1+A4)(B1+B4)	→	1

Let M(n) denotes the number of times multiplication operation is done

$M(n)=7M\left(\frac{n}{2}\right)$ + Time(Addition+Subtraction)

$M(n)=7M\left(\frac{n}{2}\right) + cn$

$M(n) = n^{log_b a}$ Here a =7 b = 2

$M(n) = n^{log_2 7}$

$M(n) = n^{2.81}$

$M(n) = \Theta(n^{2.81})$

Decrease and Conquer

- The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Once such a relationship is established, it can be exploited either top down or bottom up.

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

Decrease by a constant

In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

Consider,

Decrease and Conquer

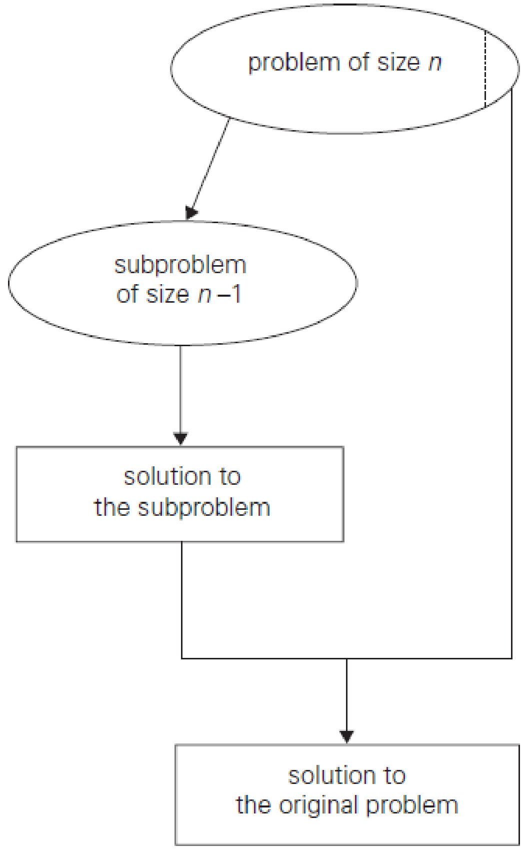
Decrease by a constant

In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

Consider, the exponentiation problem of computing a^n

$a^n = a.a^{n-1}$

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$



Decrease and Conquer

Decrease by a constant

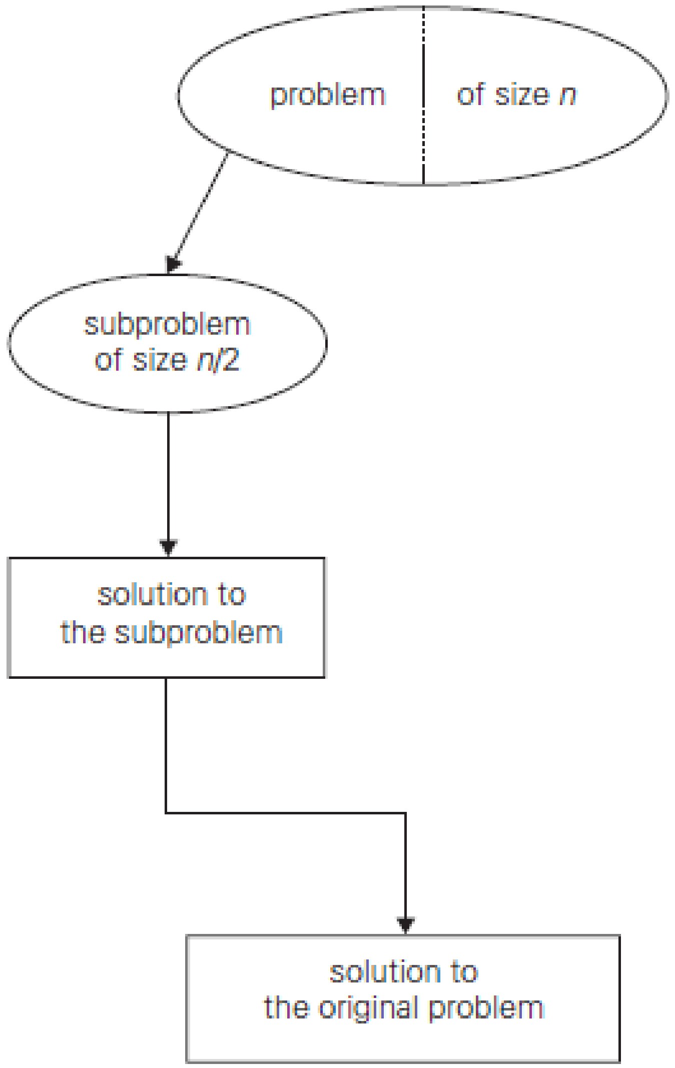
The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two

For an example,
let us revisit the exponentiation problem. If the instance of size n is to compute an, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. B

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Decrease and Conquer

Decrease by a constant



Decrease-(by half)-and-conquer technique.

Decrease and Conquer

variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.

Euclid's algorithm for computing the greatest common divisor provides a good example of such a Situation. Recall that this algorithm is based on the formula

gcd(m, n) = gcd$(n, m \bmod n)$.

Decrease and Conquer

Decrease by a constant

Insertion Sort

we consider an application of the decrease-by-one technique to sorting an array $A[0..n - 1]$.

Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n - 2]$ has already been solved to give us a sorted array of size $n - 1$: $A[0] \leq \ldots \leq A[n - 2]$.

Decrease and Conquer

Insertion Sort

```
ALGORITHM InsertionSort(A[0..n − 1])
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ←1 to n − 1 do
        v ←A[i]
        j ←i − 1
        while j ≥ 0 and A[j ]> v do
                A[j + 1]←A[j ]
                j ←j − 1
        A[j + 1]←v
```

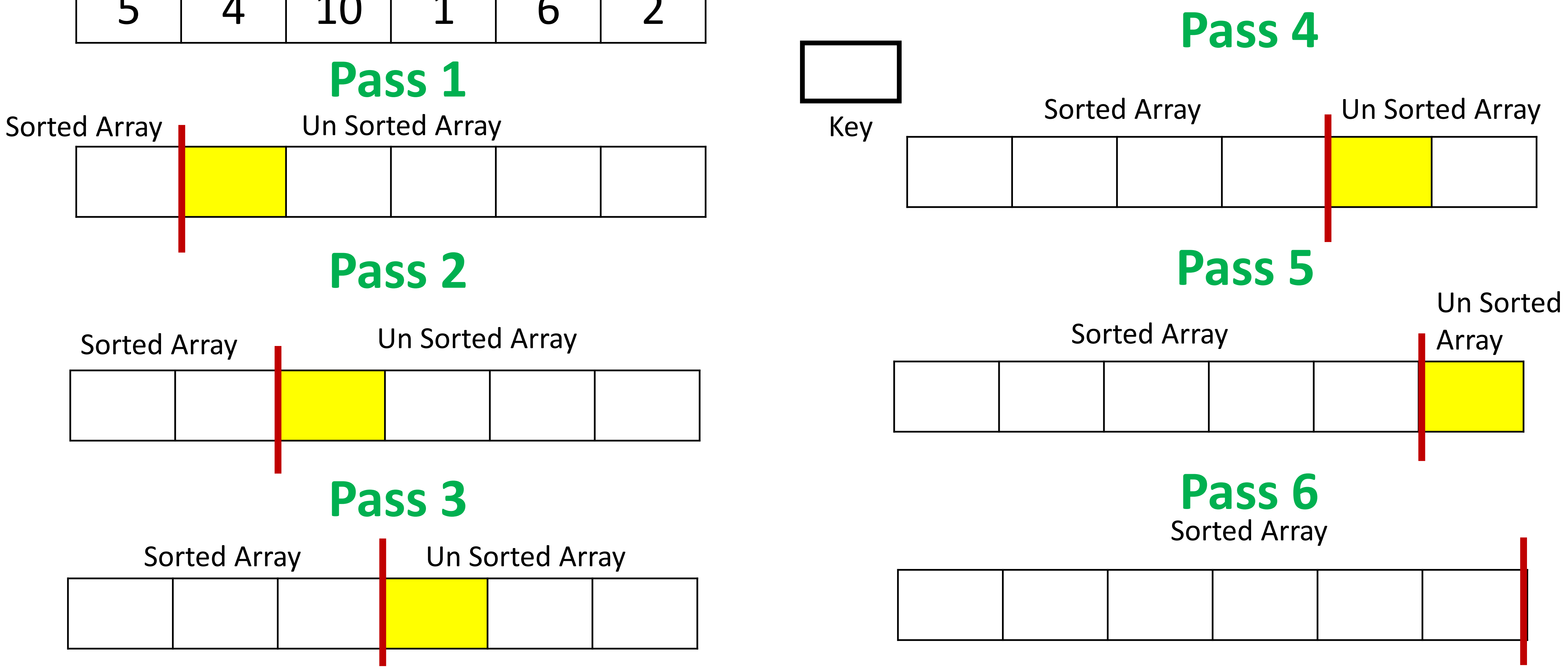
```
void insertionSort()
{
        key=0;
        for(i=1; i<n-1; i++)
        {
                key=a[i];
                j=i-1;
                while(j>=0 && a[j]>key)
                {
                        a[j+1]=a[j];
                        j--;
                }
                a[j+1]=key;
        }
}
```

Exercise:

1) 24, 12, 8, 15, 20
2) 89, 45, 68, 90, 29, 34, 17

Decrease and Conquer

5	4	10	1	6	2



Decrease and Conquer

5	4	10	1	6	2

Key

Pass 1

Sorted Array | Un Sorted Array

5	4	10	1	6	2

Pass 2

Sorted Array | Un Sorted Array

4	5	10	1	6	2

Pass 3

Sorted Array | Un Sorted Array

4	5	10	1	6	2

Pass 4

Sorted Array | Un Sorted Array

1	4	5	10	6	2

Pass 5

Sorted Array | Un Sorted Array

1	4	5	6	10	2

Pass 6

Sorted Array

1	2	4	5	6	10

Decrease and Conquer

Insertion sort

89 | **45** 68 90 29 34 17

45 89 | **68** 90 29 34 17

45 68 89 | **90** 29 34 17

45 68 89 90 | **29** 34 17

29 45 68 89 90 | **34** 17

29 34 45 68 89 90 | **17**

17 29 34 45 68 89 90

Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Decrease and Conquer

Insertion sort

Best-case Efficiency

1, 2, 3, 4, 5

1st iteration 1 Comparison

2nd iteration 1 Comparison

3rd iteration 1 Comparison

4th iteration 1 Comparison

$$C(n)=\sum_{i=1}^{n-1} 1 \;=\; C(n)= n-1-1+1$$

$$C(n)=\Theta n$$

Decrease and Conquer

Insertion sort

Worst-case Efficiency

5, 4, 3, 2, 1

1st iteration 1 Comparison

2nd iteration 2 Comparison

3rd iteration 3 Comparison

4th iteration 4 Comparison

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 \quad =C(n)=\sum_{i=1}^{n-1} i-1-0+1 \quad =C(n)=\sum_{i=1}^{n-1} i$$

$$=C(n)=\sum_{i=1}^{n-1} i \quad = C(n)= 1+2+3+4+\ldots\ldots\ldots+n-1 \quad = C(n)=\frac{n(n-1)}{2}$$

Decrease and Conquer

Graph Traversal

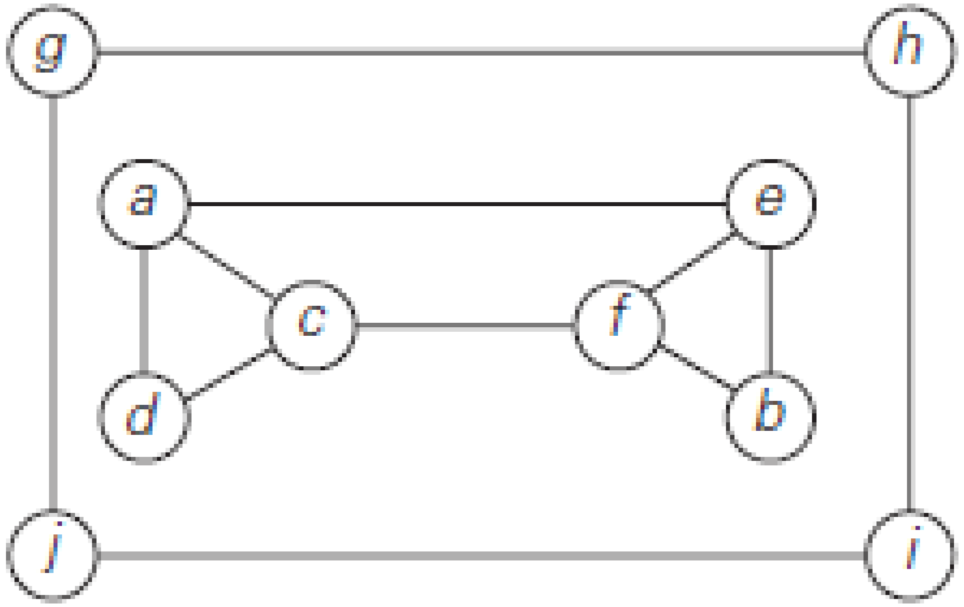
1. **DFS**
2. **BFS**

Depth-first search forest. The starting vertex of the traversal serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest. The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest.

Decrease and Conquer

Graph Traversal

1. **DFS**



Decrease and Conquer

```
ALGORITHM  DFS(G)
    //Implements a depth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //          in the order they are first encountered by the DFS traversal
    mark each vertex in $V$ with 0 as a mark of being "unvisited"
    $count \leftarrow 0$
    for each vertex $v$ in $V$ do
        if $v$ is marked with 0
            $dfs(v)$
```

```
$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex $v$
//by a path and numbers them in the order they are encountered
//via global variable count
$count \leftarrow count + 1$;   mark $v$ with $count$
for each vertex $w$ in $V$ adjacent to $v$ do
    if $w$ is marked with 0
        $dfs(w)$
```

Decrease and Conquer

2. BFS

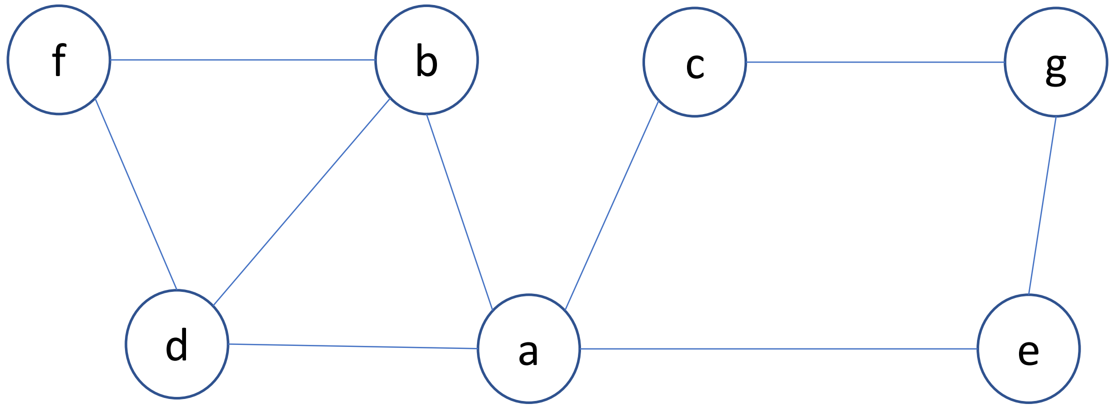
Breadth-first search is a traversal for the cautious. It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

Decrease and Conquer

```
ALGORITHM    BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //            in the order they are visited by the BFS traversal
    mark each vertex in $V$ with 0 as a mark of being "unvisited"
    $count \leftarrow 0$
    for each vertex $v$ in $V$ do
        if $v$ is marked with 0
            $bfs(v)$

    $bfs(v)$
    //visits all the unvisited vertices connected to vertex $v$
    //by a path and numbers them in the order they are visited
    //via global variable $count$
    $count \leftarrow count + 1$;   mark $v$ with $count$ and initialize a queue with $v$
    while the queue is not empty do
        for each vertex $w$ in $V$ adjacent to the front vertex do
            if $w$ is marked with 0
                $count \leftarrow count + 1$;   mark $w$ with $count$
                add $w$ to the queue
        remove the front vertex from the queue
```

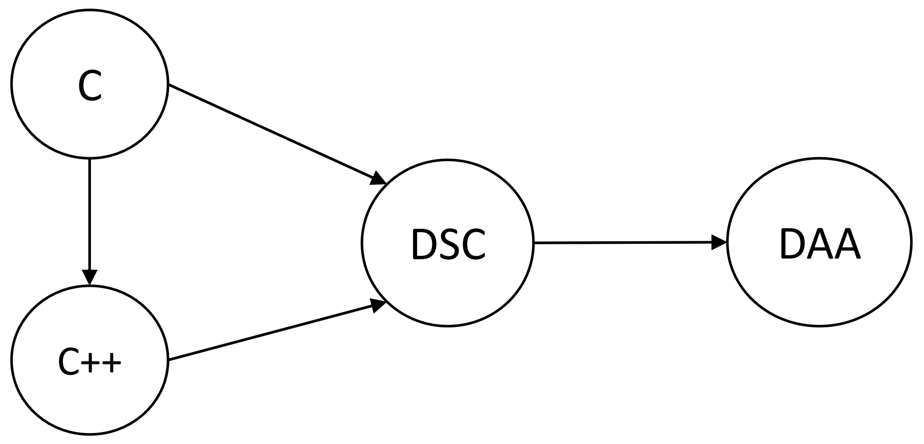
Decrease and Conquer



Decrease and Conquer

Topological Ordering (Topological Sort)

Topological ordering of a directed acyclic graph (DAG) is the linear ordering of all the nodes in the graph, such that if there is an edge from node x to node y, x should be placed before placing y.



Topological Order:- C C++ DSC DAA

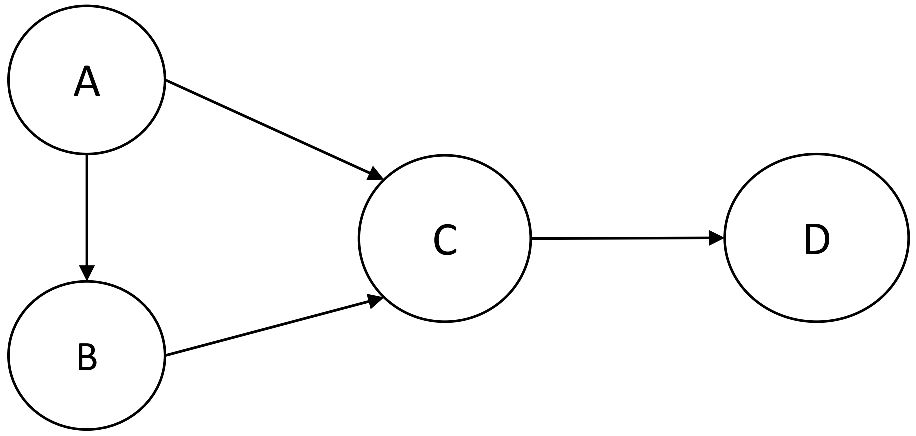
Method to do topological ordering is

1. **DFS**
2. **Vertex Deletion Method**

Decrease and Conquer

Topological ordering using DFS Method

- First traverse the graph using DFS method
- Note down the order in which the nodes become the dead end (note down the popping order of the nodes)
- Reverse the popping order, we will get topological order.



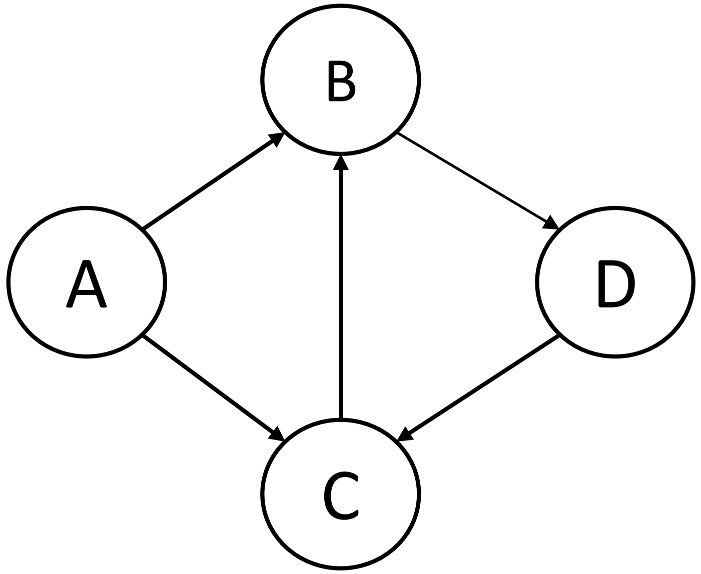
A1, 4
B2, 3
C3, 2
D4, 1

Popping order
D C B A
Reverse the popping order
A B C D

Decrease and Conquer

Topological ordering using DFS Method

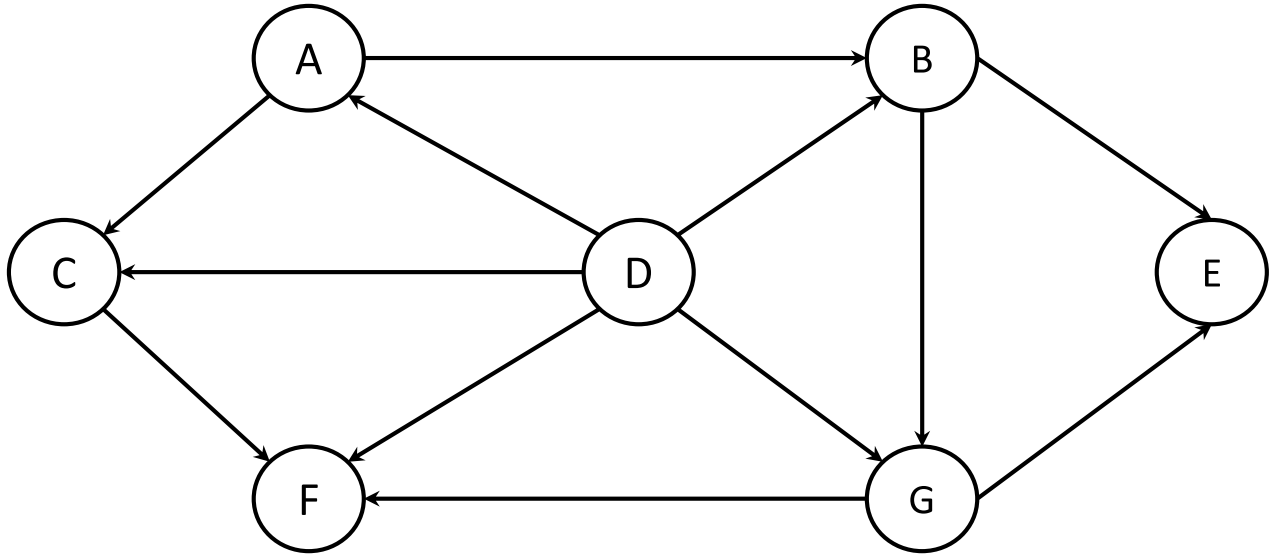
- First traverse the graph using DFS method
- Note down the order in which the nodes become the dead end (note down the popping order of the nodes)
- Reverse the popping order, we will get topological order.



A 1, 4	Popping order
B 2, 3	C D B A
D 3, 2	Topological Order
C 4, 1	A B D C

Decrease and Conquer

Topological ordering using DFS Method



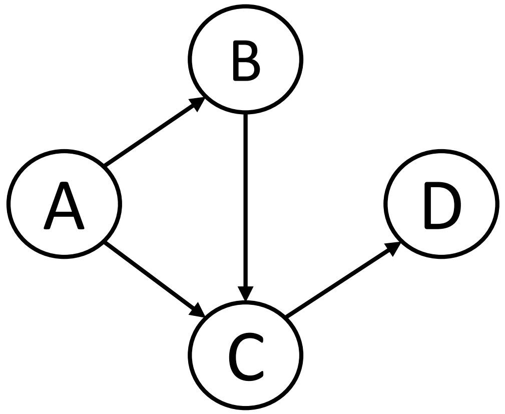
Topological Order : D, A , C, B, G, F, E

Decrease and Conquer

Topological ordering using vertex Deletion Method

Procedure

- Note down the in degree of all the nodes
- Place the node / nodes which has indegree of zero
- Decrement the indegree of the nodes where there was an incoming edge from the node placed in step 2(Deleting the vertex)
- Repeat step 2 and step 3 until all nodes are placed.

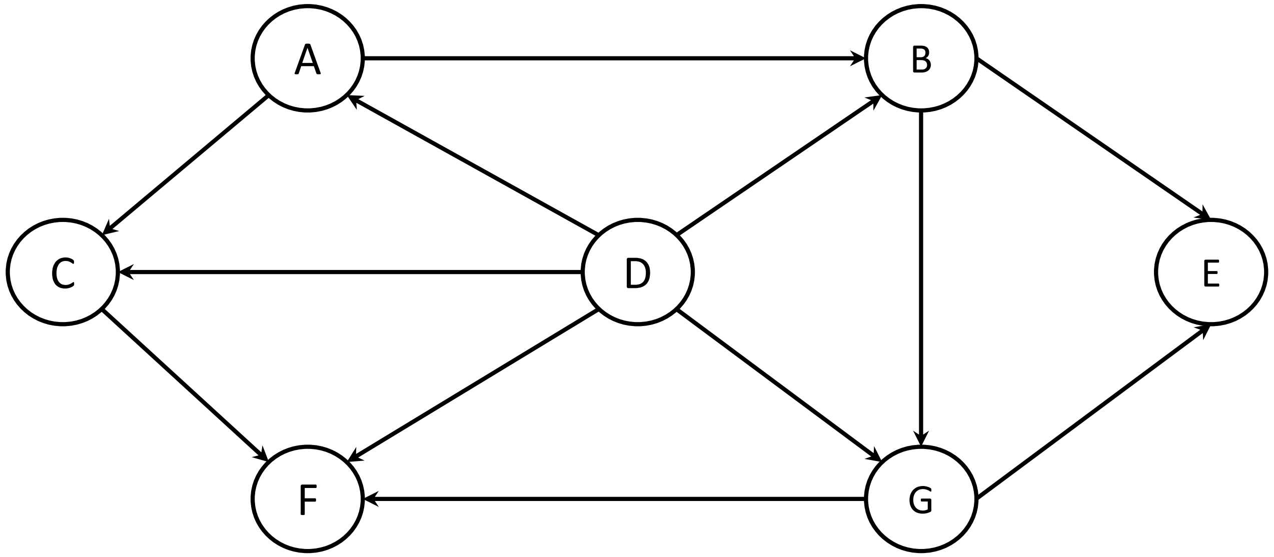

$$\begin{tikzcd} & B \arrow[d] & \\ A \arrow[ur] \arrow[r] & C \arrow[r] & D \end{tikzcd}$$

In(A) = 0
In(B) = 1   0
In(C) = 2   1   0
In(D) = 1   1   1   0

Topological Order is A B D C

Decrease and Conquer

Topological ordering using vertex Deletion Method



In(A) = 1 0

In(B) = 2 1 0

In(C) = 2 1 0 0

In(D) = 0

In(E) = 2 2 2 1 1

In(F) = 3 2 2 2 1

In(G) = 2 1 1 0 0

Topological Order is D A B C G E F

Decrease and Conquer

Applications of Breadth First Search:

1. Shortest Path and Minimum Spanning Tree for unweighted graph
2. Minimum Spanning Tree for weighted graphs
3. Peer-to-Peer Networks
4. Crawlers in Search Engines
5. Social Networking Websites
6. GPS Navigation systems

Decrease and Conquer

Applications of Depth First Search:

1. Detecting cycle in a graph
2. Path Finding
3. Topological Sorting
4. Finding Strongly Connected Components of a graph
5. Solving puzzles with only one solution
6. Web crawlers

Decrease and Conquer

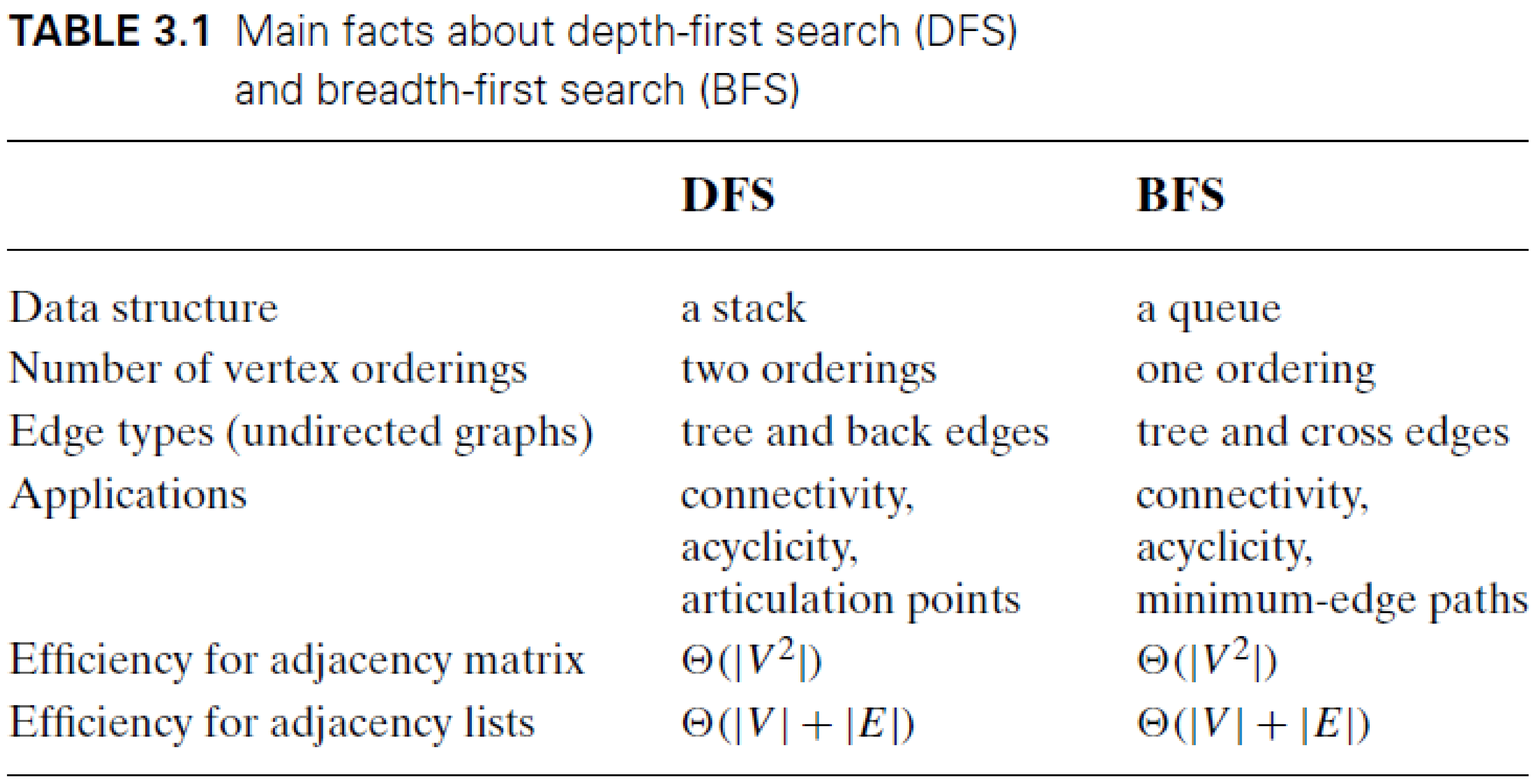


TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	**BFS**
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Divide and Conquer

Assignment Question

An investment company has stock prices for n consecutive days. That is for each day, they have a specific stock price. They want to know on which day they should buy a stock and sell it on some other day so that they make the maximum profit out of it.

Hint: **Think of it as an array $A[0..n]$ of distinct integers, you need to find two indices i and j such that ($j > i$) and $A[j] - A[i]$ is maximum. Design a $O(n \log n)$ algorithm for this problem.**