

Module-2 : Divide and Conquer

Contents

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen's matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

MODULE 2

DIVIDE AND CONQUER

Contents

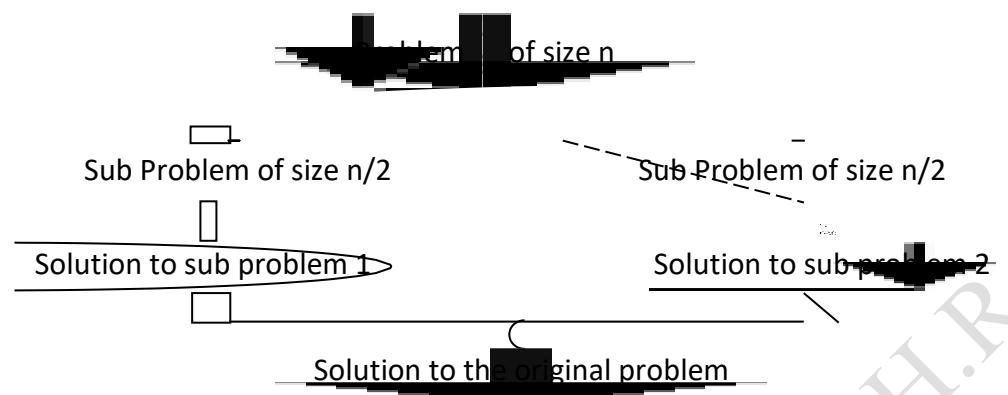
1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen's matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

General method:

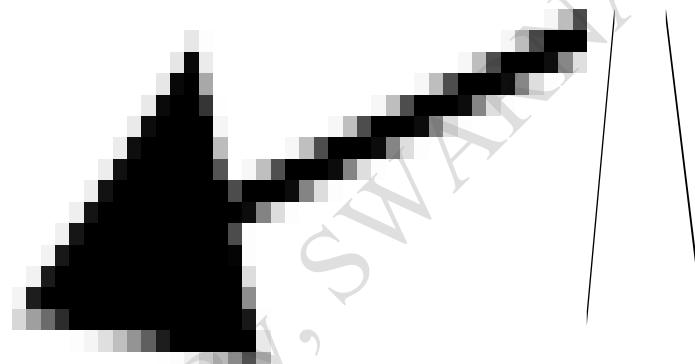
Divide and Conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, $1 < k \leq n$, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

A typical case with $k=2$ is diagrammatically shown below.



Control Abstraction for divide and conquer:



In the above specification,

- Initially $DAndC(P)$ is invoked, where 'P' is the problem to be solved.
- $Small(P)$ is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function ' S ' is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems $P_1, P_2 \dots P_k$ are solved by recursive application of $DAndC$.
- $Combine$ is a function that determines the solution to P using the solutions to the 'k' sub problems.

Recurrence equation for Divide and Conquer:

If the size of problem ‘p’ is n and the sizes of the ‘ k ’ sub problems are $n_1, n_2 \dots n_k$, respectively, then the computing time of divide and conquer is described by the recurrence relation

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

Where,

- $T(n)$ is the time for divide and conquer method on any input of size n and
- $g(n)$ is the time to compute answer directly for small inputs.
- The function $f(n)$ is the time for dividing the problem ‘ p ’ and combining the solutions to sub problems.

For divide and conquer based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe them by using recursion.

More generally, an instance of size n can be divided into b instances of size n/b , with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$). Assuming that size n is a power of b (i.e. $n = b^k$), to simplify our analysis, we get the following recurrence for the running time $T(n)$:

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

where $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Substitution Method - One of the methods for solving the recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.

Master Theorem - The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem.

It states that, in recurrence equation $T(n) = aT(n/b) + f(n)$, If $f(n) \in \Theta(n^d)$ where $d \geq 0$ 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 $a > b^d$,

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

Problems on Substitution method & Master theorem to solve the recurrence relation

~~Exercise 3.1
Homework~~

Solve following recurrence relation.

$$T(n) = 2T(n/2) + n, \quad T(1) = 2 \quad \text{using substitution method}$$

Soln: $T(n) = 2T(n/2) + n.$

$$= 2\left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4T\left(\frac{n}{4}\right) + 2n$$

$$= 4\left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

:

$$= 2^i T\left(\frac{n}{2^i}\right) + in ; \quad 1 \leq i \leq \log_2 n$$

The maximum value of $i = \log_2 n$ [\because then only we get $T(n)$].

$$= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + n \cdot \log_2 n$$

$$= n \cdot T(1) + n \log_2 n$$

$$= 2n + n \log_2 n$$

$$= \underline{\underline{\Theta(n \log_2 n)}}$$

Solution using Master theorem

Here $a=2, b=2, f(n) = n = \Theta(n^1) \Rightarrow d=1$

Also we see that $a=b^d$ [$2=2^1$]

\therefore As per case-2 of master theorem.

$$T(n) = \Theta(n^{d \cdot \log_b n})$$

$$T(n) = \underline{\underline{\Theta(n \log_2 n)}}$$

Exercise 3.2: Solve by substitution method.

$$a=1, b=2, f(n)=c$$

$$\begin{aligned}
 \text{soln: } T(n) &= T\left(\frac{n}{2}\right) + c \\
 &= [T\left(\frac{n}{4}\right) + c] + c = T\left(\frac{n}{4}\right) + 2c \\
 &= [T\left(\frac{n}{8}\right) + c] + c = T\left(\frac{n}{8}\right) + 3c \\
 &\vdots \\
 &= T\left(\frac{n}{2^i}\right) + i c ; \quad 1 \leq i \leq \log_2 n \\
 &= T(1) + c \cdot \log_2 n
 \end{aligned}$$

Assuming $T(1) = k$ some constant

$$T(n) = c \log_2 n + k,$$

$$\underline{T(n) = \Theta(\log_2 n)}.$$

Soln Using master theorem

$$\text{Here } a=1, b=2, f(n)=c = \Theta(1) = \Theta(n^0)$$

$$\Rightarrow d=0,$$

As $a=b^d$ [$1=2^0$], case-2 of master theorem is applied.

$$T(n) = \Theta(n^d \log_2 n).$$

$$\underline{T(n) = \Theta(\log_2 n)}.$$

V

Exercise 3.3 Solve recurrence relation
 $a=2, b=2, f(n)=cn$.

$$\begin{aligned}
 \text{Soln: } T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + cn = 2\left[2 \cdot T\left(\frac{n}{4}\right) + c\frac{n}{2}\right] + cn \\
 &= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot cn = 4\left[2 \cdot T\left(\frac{n}{8}\right) + c\frac{n}{4}\right] + 2cn \\
 &= 8 \cdot T\left(\frac{n}{8}\right) + 3cn \\
 &\vdots \\
 &= 2^i T\left(\frac{n}{2^i}\right) + i cn, \quad 1 \leq i \leq \log_2 n \\
 &\vdots \\
 &= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot cn \\
 &\leq n \cdot T(1) + cn \log_2 n
 \end{aligned}$$

Assuming $T(1) = k$; some constant

$$\begin{aligned}
 T(n) &= c \cdot n \log_2 n + k \cdot n \\
 &= \underline{\underline{\Theta(n \log_2 n)}}
 \end{aligned}$$

Soln Using master theorem

$$\text{Here } a=2, b=2, f(n)=cn = \Theta(n) = \Theta(n^1) \Rightarrow d=1$$

Here $a=b^d$ [$2=2^1$], case-2 of master theorem

$$\begin{aligned}
 T(n) &= \Theta(n^d \log_b n) \\
 &= \underline{\underline{\Theta(n \cdot \log_2 n)}}
 \end{aligned}$$

DAY

Ex. 3.5 Solve $T(n) = 9 \cdot T\left(\frac{n}{3}\right) + 4n^6$; $n \geq 3$
 $n \in \text{powers of } 3$

Soln

$$\begin{aligned}
 T(n) &= 9 \cdot T\left(\frac{n}{3}\right) + 4n^6 \\
 &= 9 \left[9 \cdot T\left(\frac{n}{9}\right) + 4 \left(\frac{n}{3}\right)^6 \right] + 4 \cdot n^6 \\
 &= 81 \cdot T\left(\frac{n}{9}\right) + 9 \cdot 4 \cdot \frac{n^6}{3^6} + 4 \cdot n^6 \\
 &= 81 \cdot T\left(\frac{n}{9}\right) + 4 \cdot \frac{n^6}{3^4} + 4n^6 \\
 &= 3^4 \cdot T\left(\frac{n}{3^4}\right) + \left(\frac{1+3^4}{3^4}\right) 4n^6 \\
 &= 3^4 \cdot T\left(\frac{n}{3^4}\right) + K \cdot n^6. \quad | \text{ } K \text{ is a constant} \\
 &= \dots \\
 &= 3^{2 \cdot \log_3 n} \cdot T\left(\frac{n}{3^{2 \cdot \log_3 n}}\right) + K \cdot n^6 \\
 &= 3^{2 \cdot \log_3 n} \cdot T(1) + K \cdot n^6
 \end{aligned}$$

Assuming $T(1)$ is constant c

$$\begin{aligned}
 &= 9c \cdot n + K \cdot n^6 \\
 &= \underline{\underline{\Theta(n^6)}}
 \end{aligned}$$

Soln using master theorem

$$a=9, b=3, f(n) = 4n^6 = \Theta(n^6) \Rightarrow d=6.$$

Since $a < b^d$ ($9 < 3^6$)

$$T(n) = \Theta(n^d) = \underline{\underline{\Theta(n^6)}}.$$

④ Solve $T(n) = 2 \cdot T\left(\frac{n}{4}\right) + 1$

$$\begin{aligned}
 \underline{\text{Soln}} \quad T(n) &= 2 \cdot [2 \cdot T\left(\frac{n}{4}\right) + 1] + 1 = 4 \cdot T\left(\frac{n}{4}\right) + (2+1) \\
 &= 4 \cdot [2 \cdot T\left(\frac{n}{8}\right) + 1] + (2+1) \\
 &= 8 \cdot T\left(\frac{n}{8}\right) + (4+2+1) \\
 &\vdots \\
 &= 2^i \cdot T\left(\frac{n}{2^i}\right) + (2^{i-1} + 2^{i-2} + \dots + 2+1) \\
 &\quad (\leq i \leq \log_2 n) \\
 &= 2^{\log_2 n} \cdot T(1) + (2^i - 1) \\
 &= n \cdot T(1) + 2^{\log_2 n} - 1
 \end{aligned}$$

Assuming $T(1) = 1$

$$\begin{aligned}
 &= n + n - 1 \\
 &= \underline{\underline{2n-1}} \\
 &= \underline{\underline{\Theta(n)}}
 \end{aligned}$$

Soln Using master theorem

$$a=2, b=4, f(n)=1 \\ = \Theta(1) = \Theta(n^0) \Rightarrow d=0.$$

Since $a > b^d$ ($2 > 4^0$), case-3 is applied

$$\begin{aligned}
 T(n) &= \Theta(n^{\log_b a}) \\
 &= \Theta(n^{\log_2 2}) \\
 &= \underline{\underline{\Theta(n)}}
 \end{aligned}$$

* solve $T(n) = T(\frac{n}{2}) + n$

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + n \\
 &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n \\
 &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n \\
 &\vdots \\
 &= T\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^{i-1}} + \frac{n}{2^{i-2}} + \dots + \frac{n}{4} + \frac{n}{2} + n\right) \\
 &\quad \text{for } 1 \leq i \leq \log_2 n \\
 &= T(1) + \left(\frac{n}{2^{\log_2 n - 1}} + \frac{n}{2^{\log_2 n - 2}} + \dots + \frac{n}{4} + \frac{n}{2} + n\right) \\
 &= T(1) + \frac{n}{(2^{\log_2 n}/2)} + \frac{n}{(2^{\log_2 n}/2^2)} + \dots + \frac{n}{2} + \frac{n}{2} + n \\
 \text{Assume } T(1) = 1, \\
 &= 1 + 2 + 2^2 + \dots + 2^{\log_2 n - 2} + 2^{\log_2 n - 1} + 2^{\log_2 n} \\
 &= 2^{\log_2 n + 1} - 1 \quad (\because 1 + 2 + 2^2 + \dots + 2^k = 2^{k+1} - 1) \\
 &= 2^{\log_2 n} \cdot 2 - 1 \\
 &= n \cdot 2 - 1 = \underline{\underline{2n - 1}} \in \underline{\underline{\Theta(n)}}
 \end{aligned}$$

Solve using Master theorem

Here $a = 1, b = 2, f(n) = n = \Theta(n^d) \Rightarrow d = 1$.

Since $a < b^d$ ($1 < 2^1$)

$$\begin{aligned}
 T(n) &= \Theta(n^{d-1}) \\
 &= \underline{\underline{\Theta(n)}}
 \end{aligned}$$

Binary Search

Problem definition: Let a_i , $1 \leq i \leq n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element's position) such that $a_j=x$. If x is not in the list, then j is set to zero.

Solution: Let $P = (n, a_1 \dots a_l, x)$ denote an arbitrary instance of search problem where n is the number of elements in the list, $a_1 \dots a_l$ is the list of elements and x is the key element to be searched for in the given list. **Binary search** on the list is done as follows:

Step 1: Pick an index q in the middle range $[i, l]$ i.e. $q = \lfloor (n + 1)/2 \rfloor$ and compare x with a_q .

Step 2: if $x = a_q$ i.e key element is equal to mid element, the problem is immediately solved.

Step 3: if $x < a_q$ in this case x has to be searched for only in the sub-list $a_i, a_{i+1}, \dots, a_{q-1}$.

Therefore problem reduces to $(q-i, a_i \dots a_{q-1}, x)$.

Step 4: if $x > a_q$, x has to be searched for only in the sub-list a_{q+1}, \dots, a_l . Therefore problem reduces to $(l-i, a_{q+1} \dots a_l, x)$.

For the above solution procedure, the Algorithm can be implemented as recursive or non-recursive algorithm.

Recursive binary search algorithm

```

int BinSrch(Type a[], int i, int l, Type 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) { // If Small(P)
        if (x==a[i]) return i;
        else return 0;
    }
    else { // Reduce P into a smaller subproblem.
        int mid = (i+l)/2;
        if (x == a[mid]) return mid;
        else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
        else return BinSrch(a,mid+1,l,x);
    }
}

```

Iterative binary search:

```

int BinSearch(Type a[], int n, Type 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.
{
    int low = 1, high = n;
    while (low <= high){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid - 1;
        else if (x > a[mid]) low = mid + 1;
        else return(mid);
    }
    return(0);
}

```

Example Let us select the 14 entries

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

place them in $a[1 : 14]$, and simulate the steps that `BinSearch` goes through as it searches for different values of x . Only the variables low , $high$, and mid need to be traced as we simulate the algorithm. We try the following values for x : 151, -14, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of `BinSearch` on these three inputs. \square

$x = 151$	low	$high$	mid	$x = -14$	low	$high$	mid
	1	14	7		1	14	7
	8	14	11		1	6	3
	12	14	13		1	2	1
	14	14	14		2	2	2
			found		2	1	not found
$x = 9$	low	$high$	mid				
	1	14	7				
	1	6	3				
	4	6	5				
			found				

Analysis

In binary search the basic operation is key comparison. Binary Search can be analyzed with the best, worst, and average case number of comparisons. The numbers of comparisons for the recursive and iterative versions of Binary Search are the same, if comparison counting is relaxed slightly. For Recursive Binary Search, count each pass through the if-then-else block as one comparison. For Iterative Binary Search, count each pass through the while block as one comparison. Let us find out how many such key comparison does the algorithm make on an array of n elements.

Best case – $\Theta(1)$ In the best case, the key is the middle in the array. A constant number of comparisons (actually just 1) are required.

Worst case - $\Theta(\log_2 n)$ In the worst case, the key does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling ($\lceil \log_2 n \rceil$) times. Thus, $\lceil \log_2 n \rceil$ comparisons are required.

Sometimes, in case of the successful search, it may take maximum number of comparisons. $\lceil \log_2 n \rceil$. So worst case complexity of successful binary search is $\Theta(\log_2 n)$.

Average case - $\Theta(\log_2 n)$ To find the average case, take the sum of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in array will be searched for, and that the probabilities of searching for each element are uniform.

successful searches	unsuccessful searches
$\Theta(1), \Theta(\log n), \Theta(\log n)$ best, average, worst	$\Theta(\log n)$ best, average, worst

How to compute Average case complexity?

Space Complexity - The space requirements for the recursive and iterative versions of binary search are different. Iterative Binary Search requires only a constant amount of space, while Recursive Binary Search requires space proportional to the number of comparisons to maintain the recursion stack.

Advantages: Efficient on very big list, Can be implemented iteratively/recursively.

Limitations:

- Interacts poorly with the memory hierarchy
- Requires sorted list as an input
- Due to random access of list element, needs arrays instead of linked list.

Finding the maximum and minimum

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

StraightMaxMin: A simple and straight forward algorithm to achieve this is given below.

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

Explanation:

- **StraightMaxMin** requires $2(n-1)$ comparisons in the best, average & worst cases.
- By realizing the comparison of $a[i] > \text{max}$ is false, improvement in a algorithm can be done. Hence we can replace the contents of the for loop by,
`If(a[i]>Max) then Max = a[i]; Else if (a[i]< min) min=a[i]`
- On the average $a[i]$ is $> \text{max}$ half the time. So, the avg. no. of comparison is $3n/2-1$.

Algorithm based on Divide and Conquer strategy

Let $P = (n, a[i], \dots, a[j])$ denote an arbitrary instance of the problem. Here ‘n’ is the no. of elements in the list ($a[i], \dots, a[j]$) and we are interested in finding the maximum and minimum of the list. If the list has more than 2 elements, P has to be divided into smaller instances.

For example, we might divide ‘P’ into the 2 instances,

$P_1 = ([n/2], a[1], a[n/2])$

$P_2 = (n-[n/2], a[[n/2]+1], \dots, a[n])$

After having divided ‘P’ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:

```
void MaxMin(int i, int j, Type& max, Type& 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) max = min = a[i]; // Small(P)
    else if (i == j-1) { // Another case of Small(P)
        if (a[i] < a[j]) { 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.
    int mid=(i+j)/2; Type max1, min1;
    // Solve the subproblems.
    MaxMin(i, mid, max, min);
    MaxMin(mid+1, j, max1, min1);
    // Combine the solutions.
    if (max < max1) max = max1;
    if (min > min1) min = min1;
}
}

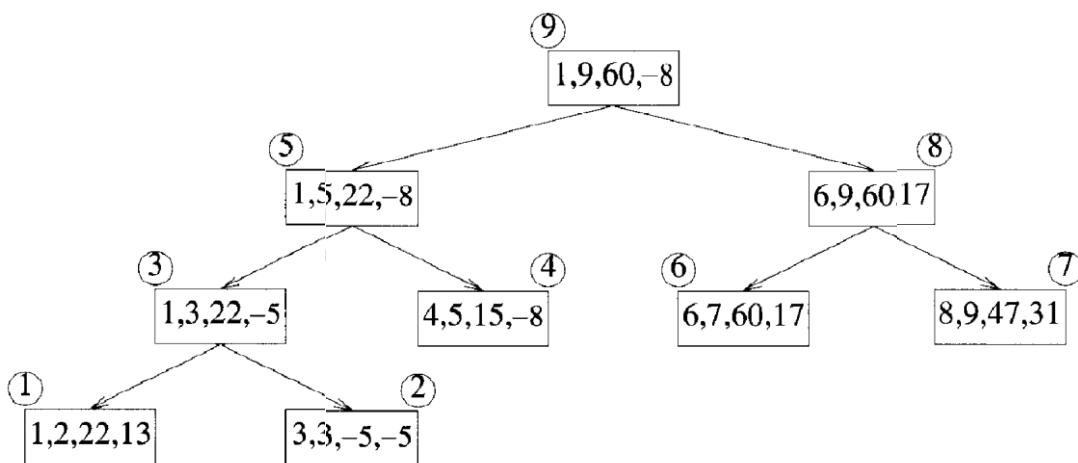
```

Example:

Suppose we simulate MaxMin on the following nine elements:

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

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i , j , max , and min . On the array $a[]$ above, the tree of recursive calls of MaxMin is as follows

**Analysis - Time Complexity**

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

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

When n is a power of two, $n = 2^k$ for some positive integer k , then

$$\begin{aligned}
 T(n) &= 2T(n/2) + 2 \\
 &= 2(2T(n/4) + 2) + 2 \\
 &= 4T(n/4) + 4 + 2 \\
 &\vdots \\
 &= 2^{k-1}T(2) + \sum_{1 \leq i \leq k-1} 2^i \\
 &= 2^{k-1} + 2^k - 2 = 3n/2 - 2
 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the straight forward method ($2n-2$) this method saves 25% in comparisons.

Space Complexity

Compared to the straight forward method, the MaxMin method requires extra stack space for i , j , \max , \min , $\max1$ and $\min1$. Given n elements there will be $\lceil \log_2 n \rceil + 1$ levels of recursion and we need to save seven values for each recursive call. (6 + 1 for return address).

Merge Sort

Merge sort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0 \dots n-1]$ by dividing it into two halves $A[0 \dots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \dots n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort($A[0..n-1]$)*

```

//Sorts array A[0..n-1] by recursive mergesort
//Input: An array A[0..n-1] of orderable elements
//Output: Array A[0..n-1] sorted in nondecreasing 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

```

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.

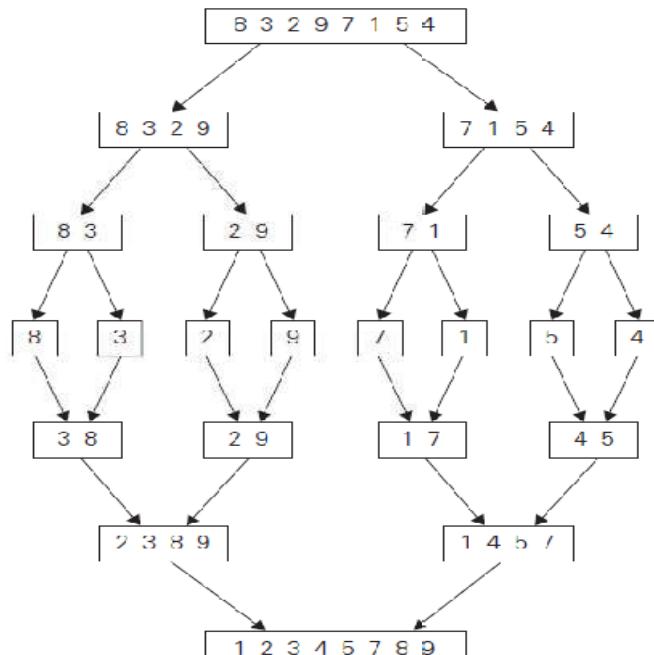
ALGORITHM *Merge(B[0..p - 1], C[0..q - 1], A[0..p + q])*

```

//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
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]
```

Example:

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in the figure



Analysis

Here the basic operation is key comparison. As merge sort execution does not depend on the order of the data, best case and average case runtime are the same as worst case runtime.

Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element leads to the worst case of merge sort. Assuming for simplicity that total number of elements n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

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

where, $C_{\text{merge}}(n)$ is the number of key comparison made during the merging stage.

Let us analyze $C_{\text{merge}}(n)$, 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, $C_{\text{merge}}(n) = n - 1$.

Now,

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

Solving the recurrence equation using **master theorem**:

Here $a = 2$, $b = 2$, $f(n) = n$, $d = 1$. Therefore $2 = 2^1$, case 2 holds in the master theorem

$$C_{\text{worst}}(n) = \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n) \text{ Therefore } C_{\text{worst}}(n) = \Theta(n \log n)$$

Advantages:

- Number of comparisons performed is nearly optimal.
- For large n , the number of comparisons made by this algorithm in the average case turns out to be about $0.25n$ less and hence is also in $\Theta(n \log n)$.
- Mergesort will never degrade to $O(n^2)$
- Another advantage of mergesort over quicksort and heapsort is its **stability**. (A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.)

Limitations:

- The principal shortcoming of mergesort is the linear amount [$O(n)$] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variations of merge sort

1. The algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls.
2. We can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multiway mergesort.

Quick sort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

```
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  $l < r$ 
     $s \leftarrow \text{Partition}(A[l..r])$  // $s$  is a split position
    Quicksort( $A[l..s - 1]$ )
    Quicksort( $A[s + 1..r]$ )
```

Partitioning

We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot. We use the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

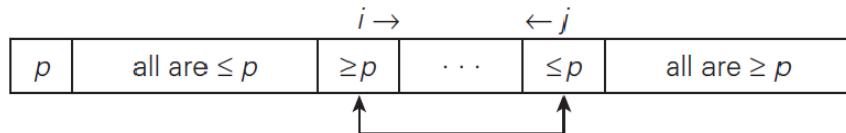
Select the subarray's first element: $p = A[l]$. Now scan the subarray from both ends, comparing the subarray's elements to the pivot.

- The left-to-right scan, denoted below by index pointer i , starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The right-to-left scan, denoted below by index pointer j , starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the

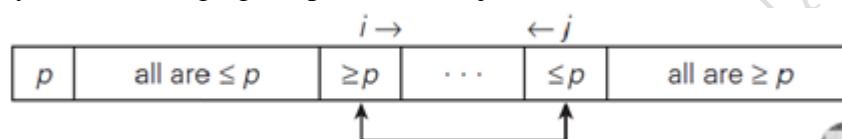
subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

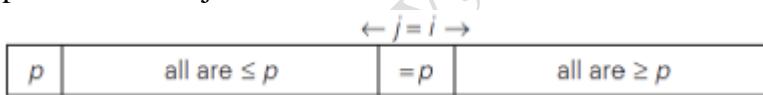
1. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j , respectively:



2. If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



3. If the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p . Thus, we have the subarray partitioned, with the split position $s = i = j$:



We can combine this with the case-2 by exchanging the pivot with $A[j]$ whenever $i \leq j$.

ALGORITHM HoarePartition($A[l..r]$)

```
//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
```

```

 $p \leftarrow A[l]$ 
 $i \leftarrow l; j \leftarrow r + 1$ 
repeat
    repeat  $i \leftarrow i + 1$  until  $A[i] \geq p$ 
    repeat  $j \leftarrow j - 1$  until  $A[j] \leq p$ 
    swap( $A[i]$ ,  $A[j]$ )
until  $i \geq j$ 
swap( $A[i]$ ,  $A[j]$ ) //undo last swap when  $i \geq j$ 
swap( $A[l]$ ,  $A[j]$ )
return  $j$ 
```

Note that index i can go out of the subarray's bounds in this pseudocode.

Example: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	/
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
i	i	1	4				
2	3	1	4				
i	i	1	4				
2	1	3	4				
i	i	3	4				
2	1	3	4				
1	2	3	4				
1							
	3	4					
	3	4					
	4						
		8	9	i	i		
		8	7	i	9		
		8	7	i	9		
		7	8	9	9		
		7					
							9

Analysis

Best Case - Here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide. If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence,

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

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

Worst Case – In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays.

Indeed, if $A[0..n - 1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[n-1]$, indicating the split at position 0: So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n-2 .. n-1]$ has been processed. The total number of key comparisons made will be equal to



Average Case - Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n . A partition can happen in any position s ($0 \leq s \leq n-1$) after $n+1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [n + 1 + C_{avg}(s) + C_{avg}(n - 1 - s)]$$

$$C_{avg}(0) = 0,$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be $\frac{(n+1)n^2}{2}$.

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes. This certainly justifies the name given to the algorithm by its inventor.

Variations

Because of quicksort's importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Better pivot selection methods such as randomized quicksort that uses a random element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array
- Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Limitations

- It is not stable.
- It requires a stack to store parameters of subarrays that are yet to be sorted.
- While Performance on randomly ordered arrays is known to be sensitive not only to implementation details of the algorithm but also to both computer architecture and data type.

Strassen's Matrix multiplication

Direct Method: Suppose we want to multiply two $n \times n$ matrices, A and B. Their product, $C=AB$, will be an n by n matrix and will therefore have n^2 elements. The number of

multiplications involved in producing the product in this way is $\Theta(n^3)$

$$C(i, j) = \sum_{1 \leq k \leq n} A(i, k)B(k, j)$$

DAA-Module 2 .By, S. RAKHAR.

Divide and Conquer method

Multiplication of $n \times n$ matrices – Let A and B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four $n/2 \times n/2$ submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{bmatrix}$$

Multiplication of 2×2 matrices: By using divide-and-conquer approach we can reduce the number of multiplications. Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2×2 matrices A and B with **just seven multiplications** as opposed to the eight required by the brute-force algorithm.

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

DAA-Module 2 .By, SWARAJA R.

Analysis

Here the basic operation is **multiplication**. If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \dots \\ &= 7^iM(2^{k-i}) \dots = 7^kM(2^{k-k}) = 7^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

This implies $M(n) = \Theta(n^{2.807})$ which is smaller than n^3 required by the brute-force algorithm.

Advantages and Disadvantages of Divide & Conquer

Advantages

- **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for execution in multi-processor machines.
- **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.
- It allows solving **difficult** and often impossible looking problems like the Tower of Hanoi. It reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.
- Another advantage to this paradigm is that it often **plays a part in finding other efficient algorithms**, and in fact it was the central role in finding the quick sort and merge sort algorithms.

Disadvantages

- One of the most common issues with this sort of algorithm is the fact that the **recursion is slow**, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more **complicated**

than a basic iterative approach, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.

- Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times. It is solved again. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization.

Decrease and Conquer Approach

Decrease-and-conquer is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (usually recursively) or bottom up.

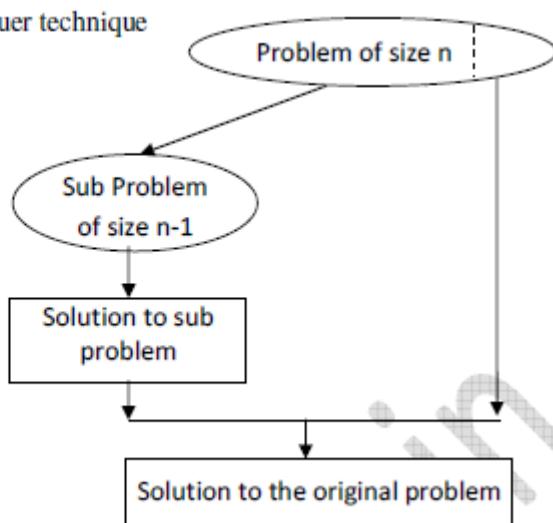
There are three major variations of decrease-and-conquer:

- decrease-by-a-constant, most often by one (e.g., insertion sort)
- decrease-by-a-constant-factor, most often by the factor of two (e.g., binary search)
- variable-size-decrease (e.g., Euclid's algorithm)

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 although other constant size reductions do happen occasionally.

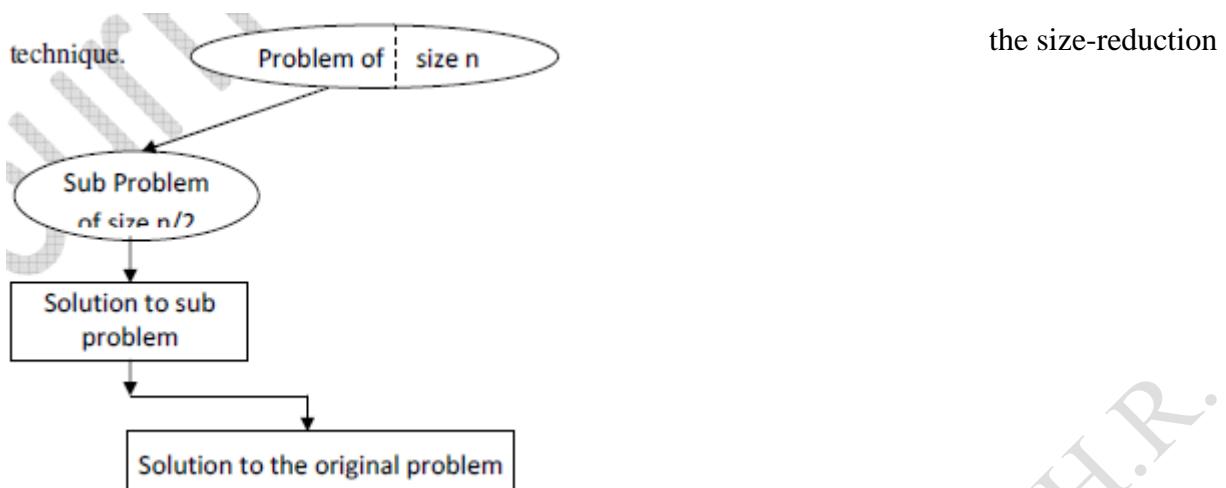
Figure: Decrease-(by one)-and-conquer technique

Example: $a^n = a^{n-1} \times a$



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.

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



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

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

Example: Euclid's algorithm for computing the greatest common divisor. It is based on the formula.
 $gcd(m, n) = gcd(n, m \bmod n)$.

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

DAA-Module 2 -By, SWARNA H.R.

Topological Sort

Background

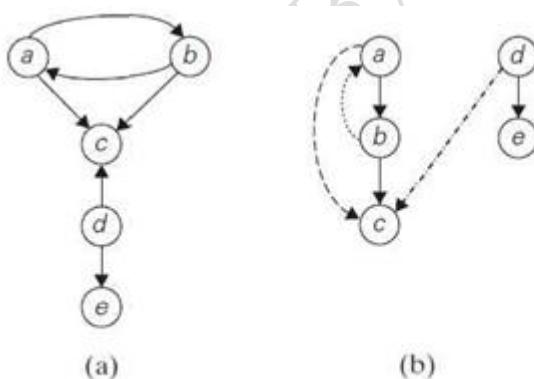
A directed graph, or digraph for short, is a graph with directions specified for all its edges.

The adjacency matrix and adjacency lists are the two principal means of representing a digraph.

There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure, the depth-first search forest (Figure b) exhibits all four types of edges possible in a DFS forest of a directed graph:

- **tree edges** (ab, bc, de),
- **back edges** (ba) from vertices to their ancestors,
- **forward edges** (ac) from vertices to their descendants in the tree other than their children, and
- **cross edges** (dc), which are none of the aforementioned types.



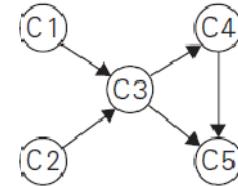
(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a .

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A **directed cycle** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, a, b, a is a directed cycle in the digraph in Figure given above. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a **dag**, an acronym for **directed acyclic graph**.

Motivation for topological sorting

Consider a set of five required courses $\{C_1, C_2, C_3, C_4, C_5\}$ a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C_1 and C_2 have no prerequisites, C_3 requires C_1 and C_2 , C_4 requires C_3 , and C_5 requires C_3 and C_4 . The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.



In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. In other words, can you find such an ordering of this digraph's vertices? This problem is called **topological sorting**.

Topological Sort

For topological sorting to be possible, a digraph in question must be a dag. i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution.

There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem. The first one is based on depth-first search; the second is based on a direct application of the decrease-by-one technique.

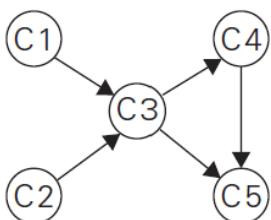
Topological Sorting based on DFS

Method

1. Perform a DFS traversal and note the order in which vertices become dead-ends
2. Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Illustration

- a) Digraph for which the topological sorting problem needs to be solved.
- b) DFS traversal stack with the subscript numbers indicating the popping off order.
- c) Solution to the problem. Here we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.



(a)

C_{5_1}
 C_{4_2}
 C_{3_3}
 $C_{1_4} \quad C_{2_5}$

(b)

The popping-off order:
 C_5, C_4, C_3, C_1, C_2
The topologically sorted list:
 $C_2 \rightarrow C_1 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5$

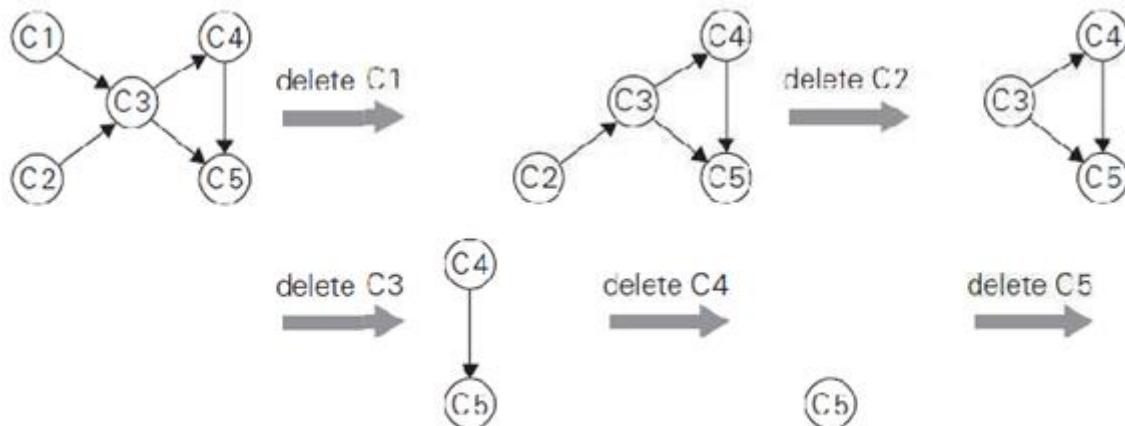
(c)

Topological Sorting using decrease-and-conquer technique:

Method: The algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique:

1. Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved.)
2. The order in which the vertices are deleted yields a solution to the topological sorting problem.

Illustration - Illustration of the source-removal algorithm for the topological sorting problem is given here. On each iteration, a vertex with no incoming edges is deleted from the digraph.



The solution obtained is C_1, C_2, C_3, C_4, C_5

Note: The solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several **alternative solutions**.

Applications of Topological Sorting

- Instruction scheduling in program compilation
- Cell evaluation ordering in spreadsheet formulas,
- Resolving symbol dependencies in linkers.
