Lecture Notes on
Analysis and Design of Algorithms
BCS401

Module-2 :Divide and Conquer

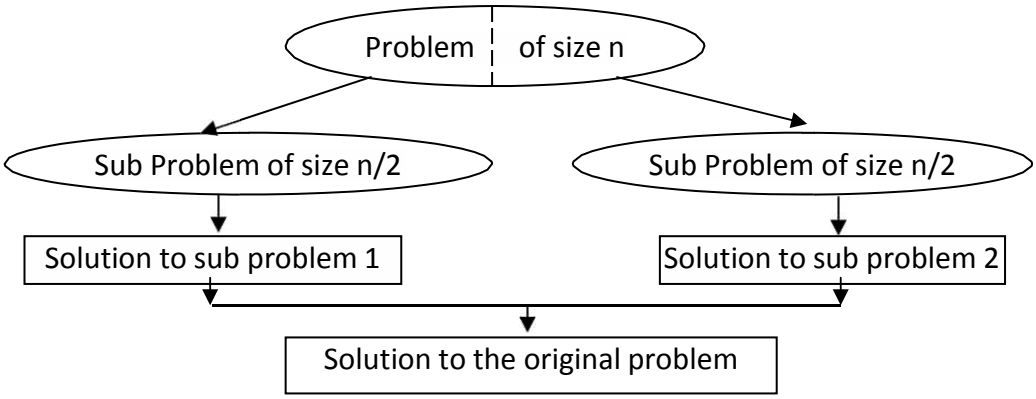
Contents

1. General method

Divide and Conquer is one of the best-known general algorithm design echnique. 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<=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=2is diagrammatically shown below.



Control Abstraction for divide and conquer:

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

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 …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.

2. 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….n_k, respectively, then the computing time of divide and conquer is described by the recurrence relation

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

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>=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):

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

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

The recurrence relation can be solved by i) substitution method or by using ii) master theorem.

1. **Substitution Method** - This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.
2. **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)∈ Θ (n^d) where d ≥ 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 thedivide-and-conquer sum-computation algorithm (see above) on inputs of sizen = 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 Horowitz

Solve following recurrence relation.

$T(n) = 2T(n/2) + n$, $T(1) = 2$. using substitution method

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

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

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

$\vdots$

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

The maximum value of $i = \log_2 n$ [∵ then only we get $T(1)$].

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

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

$= 2n + n \log_2 n$

$= \theta(n \log_2 n)$

Solution using Master theorem

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

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

∴ As per case-2 of master theorem.

$T(n) = \theta(n^d \log_2 n)$

$T(n) = \theta(n \log_2 n)$

Exercise 3.2: Solve by substitution method.

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

Soln: $T(n) = T\left(\frac{n}{2}\right) + c$

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

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

$\vdots$

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

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

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

Assuming $T(1) = K$ some constant

$$T(n) = c \log_2 n + K.$$

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

Soln Using master theorem

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.

$$T(n) = \theta(n^d \cdot \log_2 n)$$

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

Execise 3.3 Solve recurrence relation

$a = 2, b = 2, f(n) = cn$.

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

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

$= 8 \cdot T\left(\frac{n}{8}\right) + 3cn$

$\vdots$

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

$\vdots$

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

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

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

$T(n) = c \cdot n \log_2 n + K \cdot n$.

$= \theta(n(\log_2 n))$

Soln Using master theorem

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

$$T(n)=\theta(n^d \log_b n)$$
$$=\theta(n \cdot \log_2 n)$$

Ex.3.5 Solve $T(n)=9\cdot T(\frac{n}{3})+4n^6$; $n\geq 3$, n is power of 3

Soln
$$T(n)=9\cdot T(\tfrac{n}{3})+4n^6$$
$$=9\left[9\cdot T(\tfrac{n}{9})+4(\tfrac{n}{3})^6\right]+4\cdot n^6$$
$$=81\cdot T(\tfrac{n}{9})+9\cdot 4\cdot\frac{n^6}{3^6}+4\cdot n^6$$
$$=81\cdot T(\tfrac{n}{9})+4\cdot\frac{n^6}{3^4}+4n^6$$
$$=3^4\cdot T(\tfrac{n}{3^2})+\left(\frac{1+3^4}{3^4}\right)4n^6$$
$$=3^4\cdot T(\tfrac{n}{3^2})+K\cdot n^6 \quad \left|\ K \text{ is a constant}\right.$$
$$=\vdots$$
$$=3^{2i}\cdot T(\frac{n}{3^i})+Kn^6;\quad 1\leq i\leq \log_3 n$$
$$=3^{2\cdot\log_3 n}\cdot T(\frac{n}{3^{\log_3 n}})+Kn^6$$
$$=3^2\cdot 3^{\log_3 n}\cdot T(1)+Kn^6$$

Assuming $T(1)$ is constant c

$$=9c\cdot n+Kn^6$$
$$=\theta(n^6)$$

Soln using master theorem

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

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

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

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

Soln $T(n) = 2 \cdot \left[2 \cdot T\left(\frac{n}{4}\right) + 1\right] + 1 = 4\,T\left(\frac{n}{4}\right) + (2+1)$

$= 4\left[2 \cdot T\left(\frac{n}{8}\right) + 1\right] + (2+1)$

$= 8 \cdot T\left(\frac{n}{8}\right) + (4+2+1)$

$\vdots$

$= 2^i T\left(\frac{n}{2^i}\right) + (2^{i-1} + 2^{i-2} + \cdots 2 + 1)$

$(1 \le i \le \log_2 n$

$= 2^{\log_2 n} T(1) + (2^i - 1)$

$= n \cdot T(1) + 2^{\log_2 n} - 1$

Assuming $T(1) = 1$

$= n + n - 1$

$= \underline{2n - 1}$

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

Soln Using master theorem

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

$= \Theta(1) = \Theta(n^0) \Rightarrow d = 0.$

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

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

$= \Theta(n^{\log_2 2})$

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

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

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

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

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

$\vdots$

$= T\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^{i-1}} + \frac{n}{2^{i-2}} + \cdots \frac{n}{4} + \frac{n}{2} + n\right)$

$1 \le i \le \log_2 n$

$$= T(1) + \left(\frac{n}{2^{\log_2 n - 1}} + \frac{n}{2^{\log_2 n - 2}} + \cdots \frac{n}{4} + \frac{n}{2} + n \right)$$

$$= T(1) + \frac{n}{(2^{\log_2 n}/2)} + \frac{n}{\left(\frac{2^{\log_2 n}}{2^2}\right)} + \cdots \frac{n}{4} + \frac{n}{2} + n$$

Assume $T(1) = 1$,

$$= 1 + 2 + 2^2 + \cdots 2^{\log_2 n - 2} + 2^{\log_2 n - 1} + 2^{\log_2 n}$$

$$= 2^{\log_2 n + 1} - 1 \quad \left(\because 1 + 2 + 2^2 \cdots 2^k = 2^{k+1} - 1 \right)$$

$$= 2^{\log_2 n} \cdot 2 - 1$$

$$= n \cdot 2 - 1 = 2n - 1 \in \theta(n)$$

Soln using master theorem

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

Since $a \leq b^d$ $(1 \leq 2^1)$

$$T(n) = \theta(n^d)$$

$$= \theta(n)$$

3. 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 tofind 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, $a_i \ldots a_l$, x)** denote an arbitrary instance of search problem where ***n*** is the number of elements in the list, **$a_i \ldots a_l$**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:

Step1: Pick an index *q* in the middle range [i, *l*] i.e. $q = (n + 1)/2]$ and compare x with a_q.

Step 2: if x = a_qi.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 $a_i, a_{i+1}, \ldots\ldots, a_{q-}$ Therefore, problem reduces to **(q-i, $a_i \ldots a_{q-1}$, x).**

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

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

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

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

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 –Θ(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 - Θ(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(log_2n) times. Thus,] log_2 n] comparisons are required.

Sometimes, in case of the successful search, it may take maximum number of comparisons.] log_2 n]. So worst case complexity of successful binary search is Θ (log_2 n).

Average case - Θ (log_2n) 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**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

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.

4. 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]>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 > 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],……,a [j]) denote an arbitrary instance of the problem. Here 'n' is the no. of elements in the list (a[i],….,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,

P1=([n/2],a[1], a[n/2])

P2= (n-[n/2], a[[n/2]+1], , a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them byrecursivelyinvoking 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]; }
        }
```

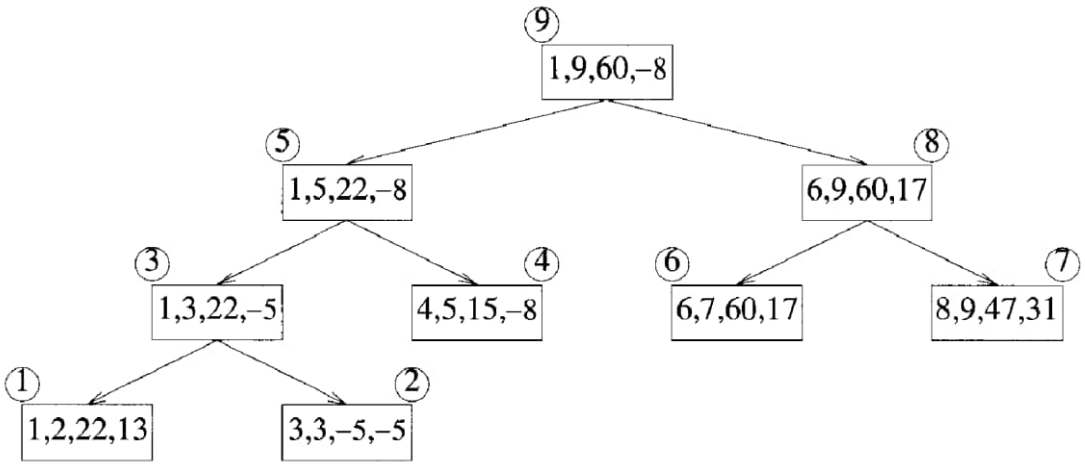
```
    else { // If P is not small
           // divide P into subproblems.
         // Find where to split the set.
           int mid=(i+j)/2; Type max1, min1;
         // Solve the subproblems.
           MaxMin(i, mid, max, min);
           MaxMin(mid+1, j, max1, min1);
         // Combine the solutions.
           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

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

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 $log_2 n] + 1$ levels of recursion and we need to save seven values for each recursive call. (6 + 1 for return address).

5. Merge Sort

Merge sort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A [O ... n - 1] by dividing it into two halves A [0 .. $n/2$]-1] and A [/2] .. 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 $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
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

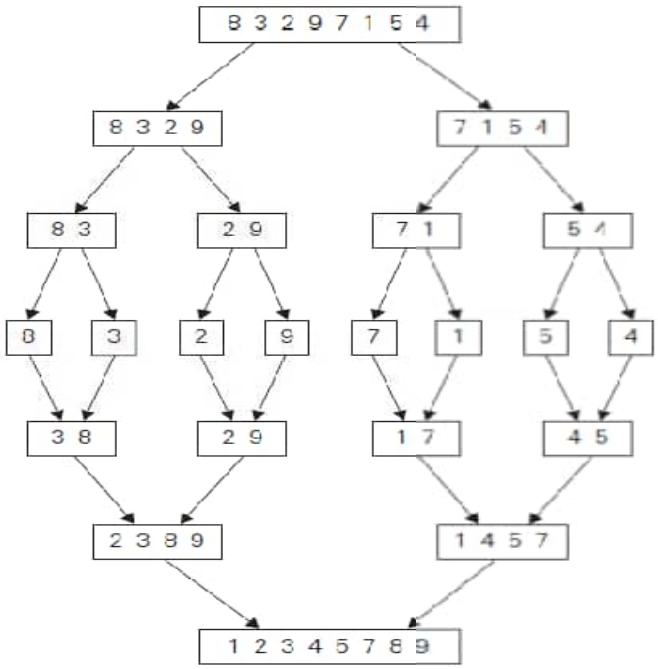
- After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

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

```
//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]
```

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 operationis 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, therecurrence relation for the number of key comparisons $C(n)$ is

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

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

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during themerging stage. At each step, exactly one comparison is made, after which the totalnumber of elements in the two arrays still needing to be processed is reducedby 1. In the worst case, neither of the two arrays becomes empty before theother one contains just one element (e.g., smaller elements may come from thealternating arrays).Therefore, for the worst case,$C_{merge}(n) = n - 1$.
Now,

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

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

Advantages:

- Number of comparisons performed is nearly optimal.
- For large n, the num er 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 $O(n^2)$
- Anotheradvantage 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. Thealgorithm 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 slightbookkeeping complications arise.) This avoids the time and space overhead ofusing a stack to handle recursive calls.
2. We can divide a list to be sortedin more than two parts, sort each recursively, and then merge them tog ther. Thisscheme, which is particularly useful for sorting files residing on secondary memorydevices, is called multiway mergesort.

6. 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:

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are} \ge A[s]}$$

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

In quick sort, the entire workhappens in the division stage, with no work required to combine the solutions tothe subproblems.

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 $l < r$
$s \leftarrow$ *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 valuewe are going to divide the subarray.There are several different strategies forselecting a pivot. We use thesophisticated method suggested byC.A.R. Hoare, the prominent British computer scientist who invented quicksort.

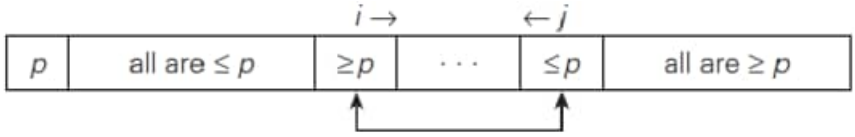
Select the subarray's firstelement: p = A[l].Now scan the subarray from both ends,comparing the subarray's elements to the pivot.

- 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

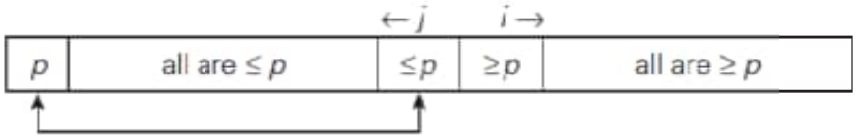
subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or notthe scanning indices have crossed.

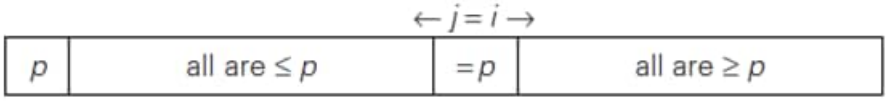
1. If scanning indices i and j have not crossed, i.e., i< j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j, respectively:



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 :



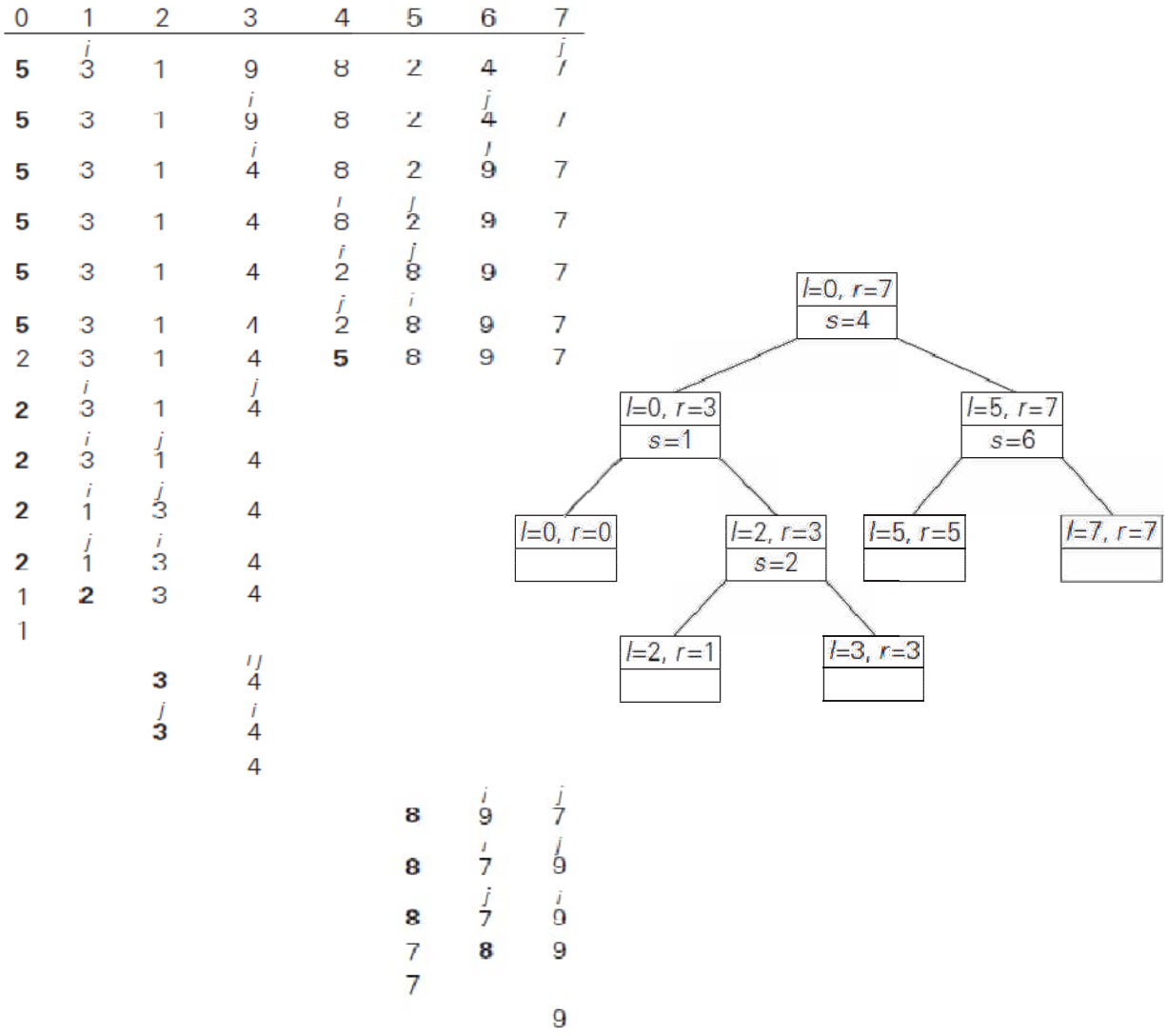
We can combine this with the case-2 byexchanging the pivot with A[j] whenever i≥j

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 ← 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[i], A[j])   //undo last swap when i ≥ 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 pivotsshown in bold. (b) Tree of recursive calls to Quicksort with input values land r of subarray bounds and split position s of a partition obtained.



Analysis

Best Case -Here the basic operation is key comparison. Numberof key comparisons made before a partition is achieved is n + 1 if the scanningindices cross over and n if they coincide. If all the splits happen in themiddle of corresponding subarrays, we will have the best case. The number of keycomparisons 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, $\boldsymbol{C_{best}(n) \in \Theta(n \log_2 n)}$**;** solving it exactly forn = 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 thetwo subarrays will be empty, and the size of the other will be just 1 less than thesize of the subarray being partitioned. This unfortunate situation will happen, inparticular, for increasing arrays. Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as thepivot, the

left-to-right scan will stop on A[1] while the right-to-left scan will go allthe way to reach A[0], indicating the split at position 0:So, after making n + 1 comparisons to get to this partition and exchanging thepivot A[0] with itself, the algorithm will be left with the strictly increasing arrayA[1..n − 1] to sort. This sorting of strictly increasing arrays of diminishing sizes willcontinue until the last one A[n−2.. n−1] has been processed. The total numberof key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case - Let $C_{avg}(n)$ be the average number of key comparisons made byquicksort on a randomly ordered array of size n. A partition can happen in anyposition s ($0 \le s \le n-1$) after n+1comparisons 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 withthe same probability 1/n, we get the following recurrence relation:

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

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turnsout to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

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

Variations: Because of quicksort's importance, there have been persistent efforts over theyears to refine the basic algorithm. Among several improvements discovered byresearchers 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: 1. It is not stable. 2. It requires a stack to store parameters of subarrays that are yet to be sorted. 3. WhilePerformance on randomly ordered arrays is known to be sensitive not only to the implementation details of the algorithm but also to both computer architecture and data type.

7. Advantages and Disadvantages of Divide & Conquer

Advantages

1. **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.
2. **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.
3. 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.
4. 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

5. 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.
6. 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.
7. 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.

8. Decrease and Conquer Approach

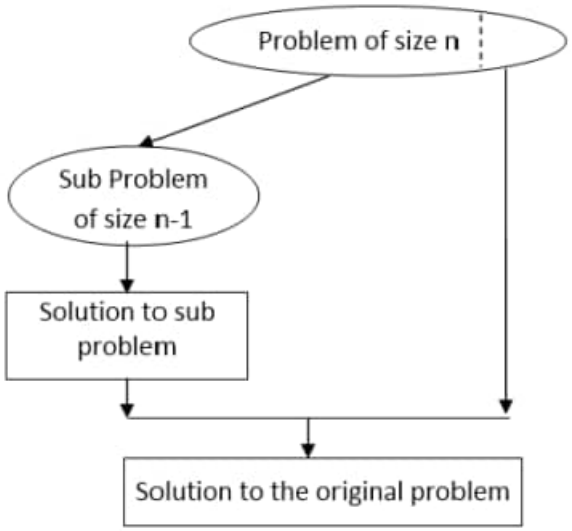
Decrease-and-conquer is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (usually recursively)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

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

In the **decrease-by-a-constant** variation, the size of an instance is reducedby the same constant on each iteration of the algorithm. Typically, this constant is equal to one although other constant size reductions do happenoccasionally.

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

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

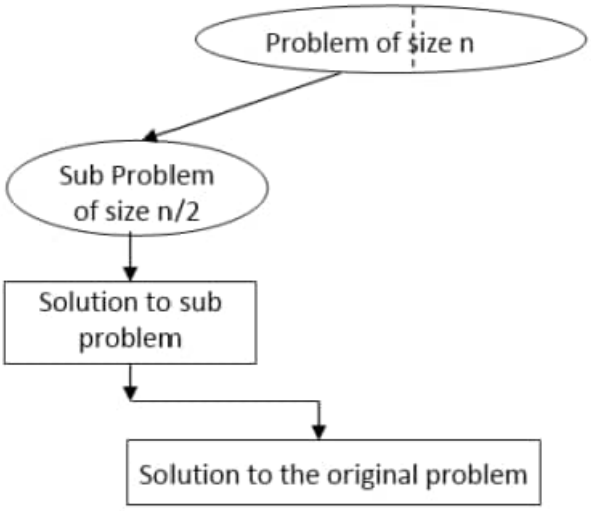


The **decrease-by-a-constant-factor** technique suggests reducing a probleminstance by the same constant factor on each teration of the algorithm. In mostapplications, this constant factor is equal to two.

Figure: Decrease-(byhalf)-and-conquer technique.

Example:

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



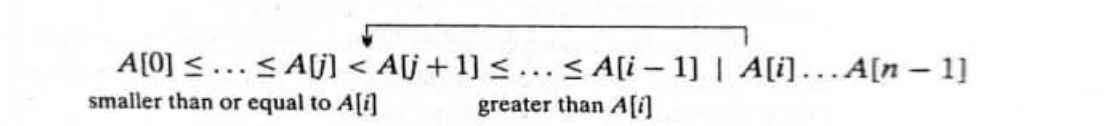
Finally, in the **variable-size-decrease** variety of decrease-and-conquer, thesize-reduction pattern varies from one iteration of an algorithm to another.

Example: Euclid'salgorithm for computing the greatest common divisor. It is based on the formula. *gcd(m, n) = gcd(n, m mod n).*

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

Insertion sort

It is an application for decease-by-one technique to sort an array A[0…n-1]. Everytime the element is inserted, it is stored in its position. So after every insertion the array remains sorted.



Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

There are three reasonable alternatives for doing this. First, we can scan the sorted subarray from left to right until the first element greater than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right before that element. Second, we can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right after that element. These two alternatives are essentially equivalent; usually, it is the second one that is implemented in practice because it is better for sorted and almost-sorted arrays (why?). The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***. The third alternative is to use binary search to find an appropriate position for $A[n-1]$ in the sorted portion of the array. The resulting algorithm is called ***binary 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 $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \ge 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$ is the worst case.

For sorted array, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n).$$

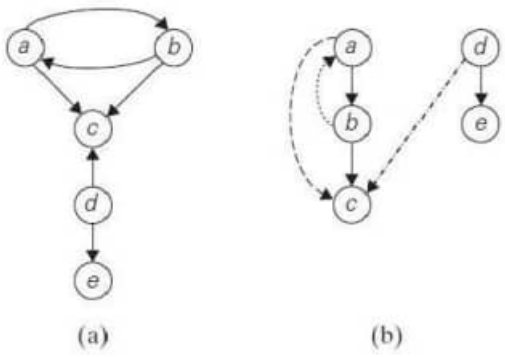
And average case is $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

9. 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.Ther 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 wo) corresponding nodes in the digraph's adjacency lists.

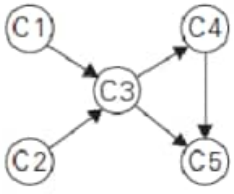
Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure, the depth-first search forest (Figure b) exhibits all 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)*, whichare 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 vertexto its parent. Whether or not it is the case, the presence of a back edge indicatesthat the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequenceof three or more of its vertices that starts and ends with the same vertex and inwhich every vertex is connected to its immediate predecessor by an edge directedfrom 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 backedges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.



Motivation for topological sorting: Consider a set of five required courses {C1, C2, C3, C4,C5} a part-time student has to take in some degree program. The courses can betaken in any order as long as the following course prerequisites are met: C1 andC2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requiresC3 and C4. The student can take only one course per term. In which order shouldthe 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 uestion 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 nodirected 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 sortingproblem.The first one is based on depth-first search; the second is based on a direct application of the decrease-by-one technique.

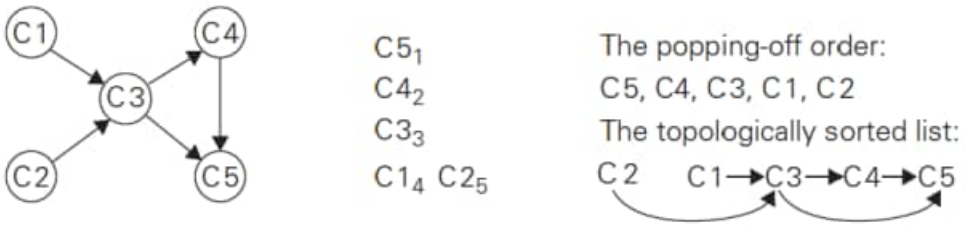
Topological Sorting based on DFS

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 w th 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.

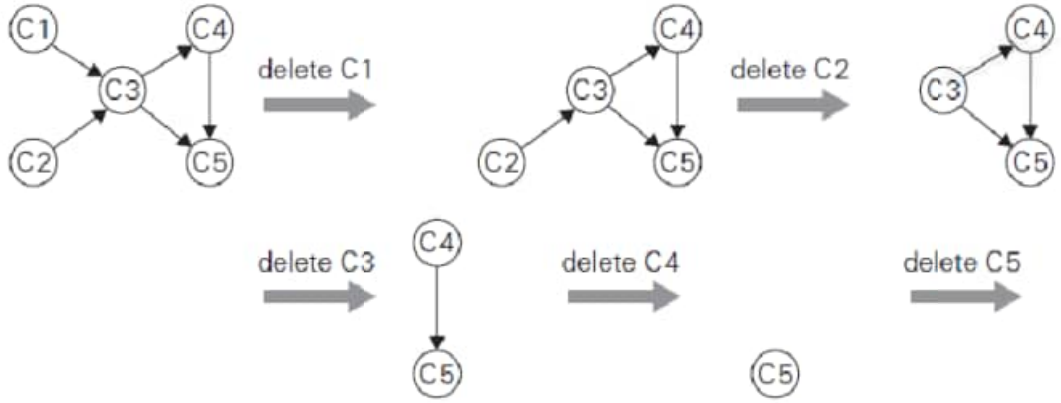


Topological Sorting using decrease-and-conquer technique

Method: The algorithm is based on a direct implementation of the decrease-(byone)-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.



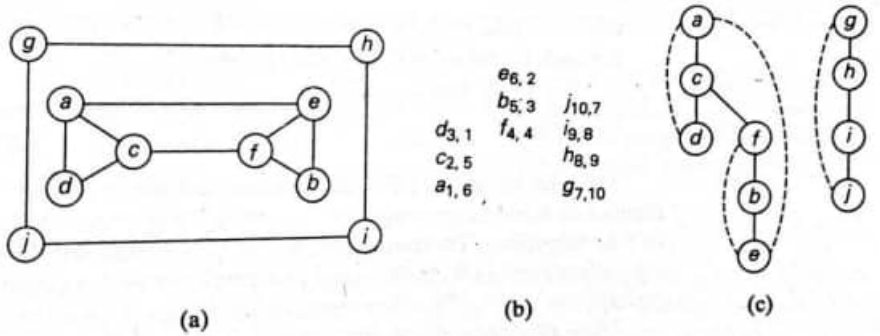
The solution obtained is C1, C2, C3, C4, C5

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**.

Depth first search:

There are three reasonable alternatives for doing this. First, we can scan the sorted subarray from left to right until the first element greater than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right before that element. Second, we can scan the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered and then insert $A[n-1]$ right after that element. These two alternatives are essentially equivalent; usually, it is the second one that is implemented in practice because it is better for sorted and almost-sorted arrays (why?). The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***. The third alternative is to use binary search to find an appropriate position for $A[n-1]$ in the sorted portion of the array. The resulting algorithm is called ***binary insertion sort***. We ask you to implement this

It is also very useful to accompany a depth-first search traversal by constructing the so-called ***depth-first search forest***. The traversal's starting vertex 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. Figure provides an example of a depth-first search traversal, with the traversal's stack and corresponding depth-first search forest shown as well.



Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

ALGORITHM 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've been 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*

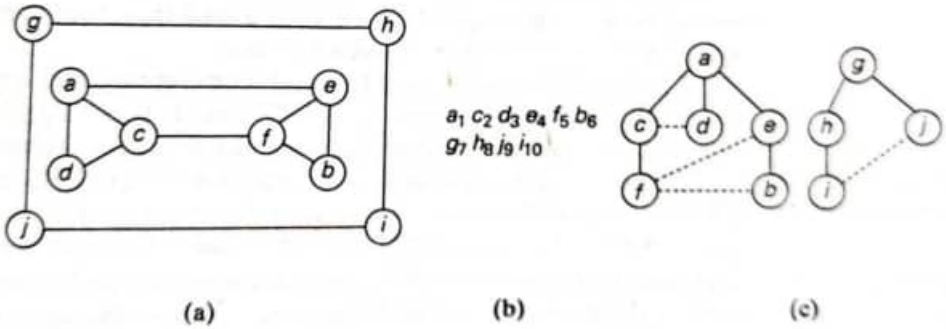
count $\leftarrow$ *count* + 1; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)

Breadth first search:

If depth-first search is a traversal for the brave (the algorithm goes as far from "home" as it can), 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.

It is convenient to use a queue (note the difference from depth-first search!) to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called ***breadth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***. Figure provides an example of a breadth-first search traversal, with the traversal's queue and corresponding breadth-first search forest shown.



(a) (b) (c)

Example of a BFS traversal. (a) Graph. (b) Traversal's queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

```
ALGORITHM BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //in the order they have been visited by the BFS traversal
    mark each vertex in V with 0 as a mark of being "unvisited"
    count ← 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 assigns them the numbers in the order they are visited
    //via global variable count
    count ← 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 ← count + 1;  mark w with count
                add w to the queue
        remove the front vertex from the queue
```

Applications of Topological Sorting

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

***