UNIT II

BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force - String Matching-KMP algorithm - Closest-Pair and Convex-Hull Problems - Exhaustive Search - Travelling Salesman Problem - Knapsack Problem – Assignment problem. Divide and Conquer Methodology - Binary Search - Merge sort - Quick sort – Randomized Quick Sort - Multiplication of Large Integers, Strassen's Matrix Multiplication.

2.1 BRUTE FORCE

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

"Just do it!" would be another way to describe the prescription of the brute-force approach.

2.2 STRING MATCHING

In this problem the two strings are provided are string is called as text string of 'n' characters. Another string is pattern string of 'm' characters. The problem is find the substring in the text string which matches the pattern.

Text t0-------t1 t n-1

Pattern P0----------Pm-1

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//       matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

ANALYSIS

1. The worst case analysis of string matching algorithm is O(mn)
2. The best case analysis of string matching algorithm is O(n+m) or O(n}

3. The average case analysis of string matching algorithm is O (m)

Analysis:

Example:

Text: MYINDIA T=7(n)

Pattern: INDIA p=5(m)

I=0 to 7-5

J=0

While(i<0) and P[0]=T[0]

(M=I)

False

I=1 to 2

J=0

While(0<5)and P[0]=T[1]

(I=Y)

false

I=2 to 2

J=0

While(0<5)and P[0]=T[2]

(I=I)

True (j++)

J=1

While(1<5)and P[1]=T[3]

(N=N)

True

J++

J=2

While(2<5)and P[2]=T[4]

(D=D)

True

J++

J=3

While(3<5)and P[3]=T[5]

(I=I)

True

J++

J=4

While(4<5)and P[4]=T[6]

(A=A)

True

J++

J=5

While(5<5)and P[5]=T[7]

false

if(5==5)

return (5)

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]

6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

P :

a	b	a	b	a	c	a

Solution:

Initially: m = length [p] = 7

Π [1] = 0

K=0

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

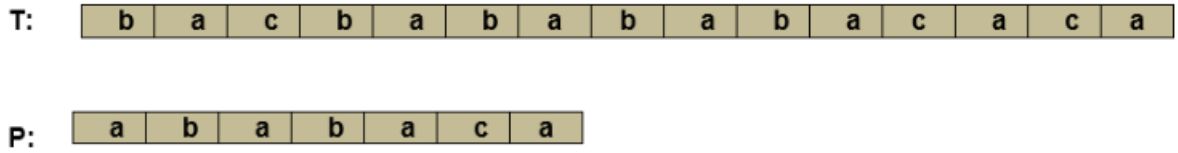
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0 // numbers of characters matched
5. for i ← 1 to n // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q] // next character does not match

8. If P [q + 1] = T [i]
9. then q ← q + 1 // next character matches
10. If q = m // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q] // look for the next match

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

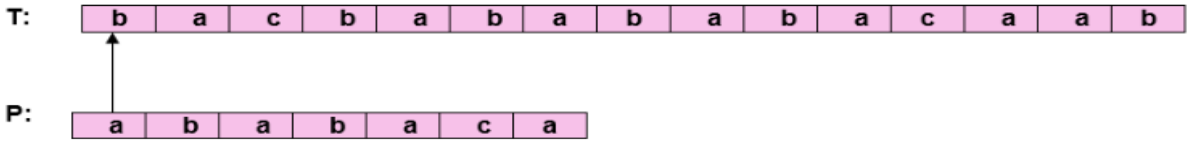
Example: Given a string 'T' and pattern 'P' as follows:



Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

Step1: i=1, q=0

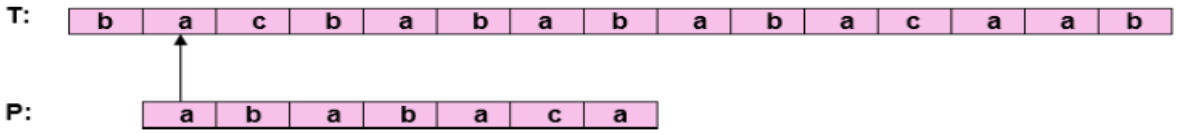
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

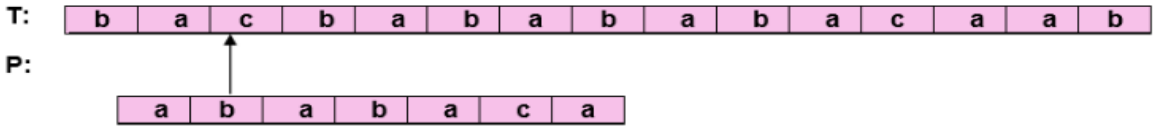
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

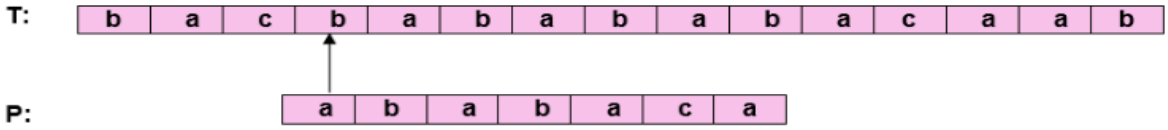
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

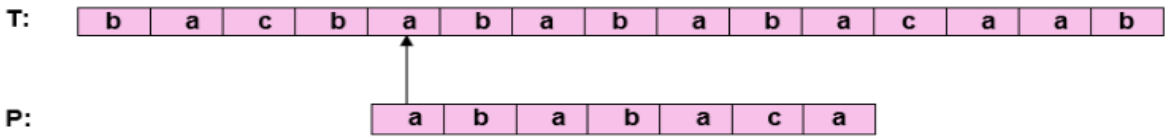
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



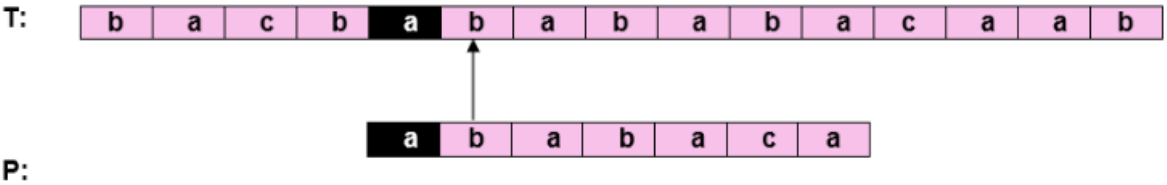
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



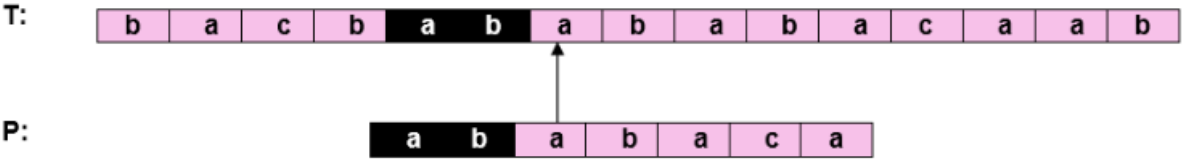
Step 6

Comparing P [2] with T [6] P [2] matches with T [6]



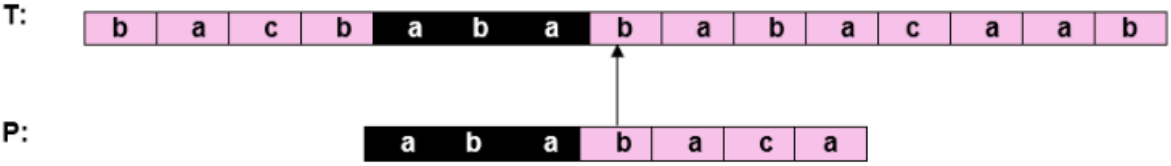
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



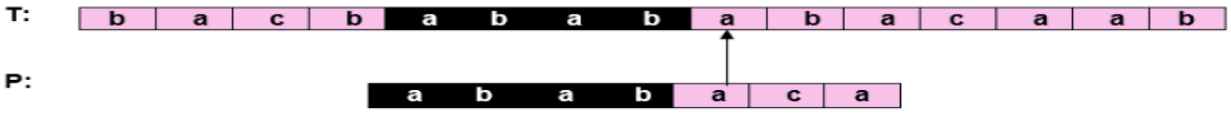
Step8: i = 8, q =3

Comparing P [4] with T [8] P [4] matches with T [8]



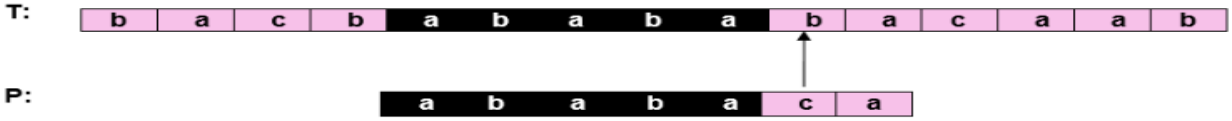
Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]



Step10: i = 10, q = 5

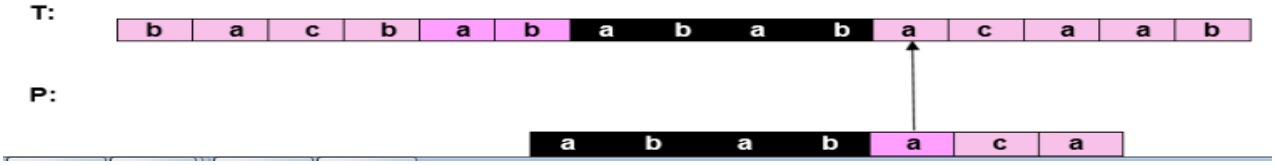
Comparing P [6] with T [10] P [6] doesn't match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

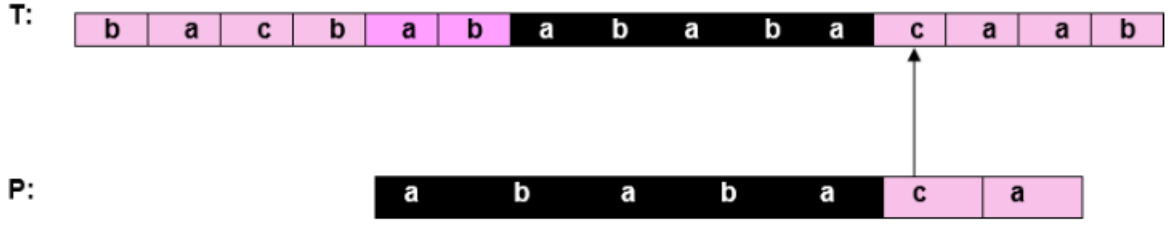
Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]



Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]

Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

2.3 CLOSEST-PAIR AND CONVEX-HULL

2.4.1CLOSEST PAIR PROBLEM

A point (2D case) is an ordered pair of values (x, y).

The (Euclidean) distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is:

$$d(p_i, p_j) = \text{root}(q(x_i - x_j)^2 + (y_i - y_j)^2)$$

The closest-pair problem is finding the two closest points in a set of n points.

The brute force algorithm checks every pair of points, which will make it $O(n^2)$.

We can avoid computing square roots by using squared distance.

Closest-Pair Brute-Force Algorithm

ALGORITHM *BruteForceClosestPair*(P)

```
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points p1(x1, y1), . . . , pn(xn, yn)
//Output: The distance between the closest pair of points
d ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← min(d, sqrt((xi − xj)^2 + (yi − yj)^2)) //sqrt is square root
return d
```

Analysis:

The basic operation of the algorithm is computing the square root. In the age of electronic calculators with a square-root button, one might be led to believe that computing the square root is as simple an operation as, say, addition or multiplication. Of course, it is not. For starters, even for most integers, square roots are irrational numbers that therefore can be found only approximately. Moreover, computing such approximations is not a trivial matter. But, in fact, computing square roots in the loop can be avoided!

The trick is to realize that we can simply ignore the square-root function and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves. We can do this because the smaller a number of which we take the square root, the smaller its square root, or, as mathematicians say, the square-root function is strictly increasing.

Then the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i)$$

$$= 2[(n - 1) + (n - 2) + \cdots + 1] = (n - 1)n \in \Theta(n^2).$$

2.4.2 THE CONVEX HULL PROBLEM

A region (set of points) in the plane is convex if every line segment between two points in the region is also in the region.

The convex hull of a finite set of points P is the smallest convex region containing P.

Theorem: The convex hull of a finite set of points P is a convex polygon whose vertices is a subset of P.

The convex hull problem is finding the convex hull given P.

Examples of Convex Sets

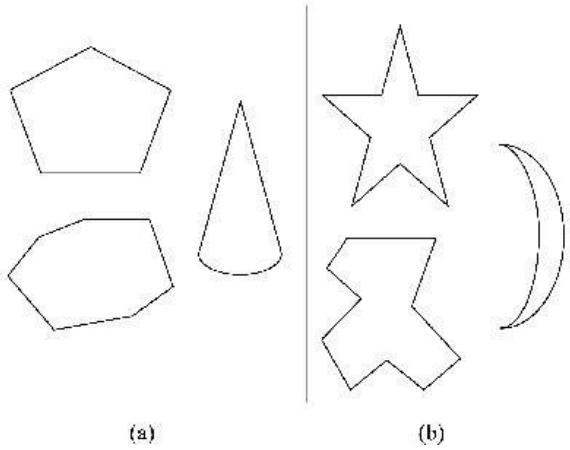
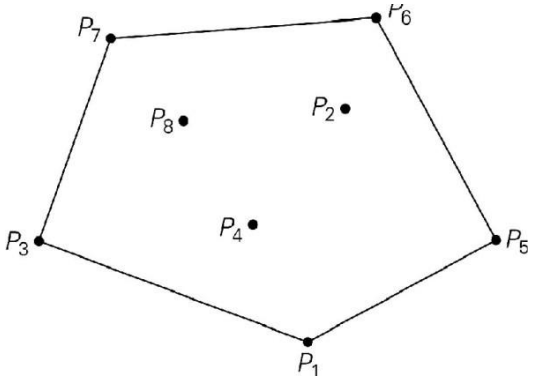


Fig (a) Convex sets (b) Sets that are not Convex

Example of Convex Hull



Idea for Solving Convex Hull

Consider the straight line that goes through two points P_i and P_j .

Suppose there are points in P on both sides of this line.

- This implies that the line segment between P_i and P_j is not on the boundary of theconvex hull.

Suppose all the points in P are on one side of the line (or on the line).

- This implies that the line segment between P_i and P_j is on the boundary of theconvex hull.

Development of Idea for Convex Hull

The straight line through $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ can be defined by a nonzero solution for:

$a\ x_i + b\ y_i = c$

$a\ x_j + b\ y_j = c$

One solution is $a = y_j - y_i$, $b = x_i - x_j$, and $c = x_i y_j - y_i x_j$.

The line segment from P_i to P_j is on the convex hull if either $a\ x + b\ y \geq c$ or $a\ x + b\ y \leq c$ is true for all the points.

Brute force algorithm is $O(n^3)$.

2.4 EXHAUSTIVE SEARCH

Exhaustive search generates all combinatorial objects (e.g., permutations, combinations, subsets) for a problem.

2.5.1 THE TRAVELING SALESMAN PROBLEM

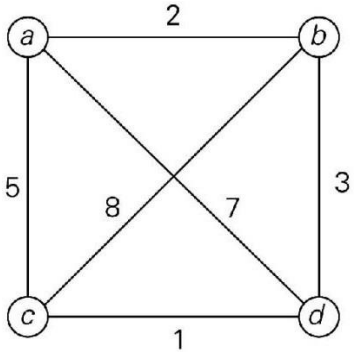
Find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once)

It is easy to see that a Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct.

- Brute Force Approach: Calculate the distance of all cycles of n vertices in a weighted graph.

TSP Example



TSP Solution

Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE Solution to a small instance of the traveling salesman problem by exhaustive search

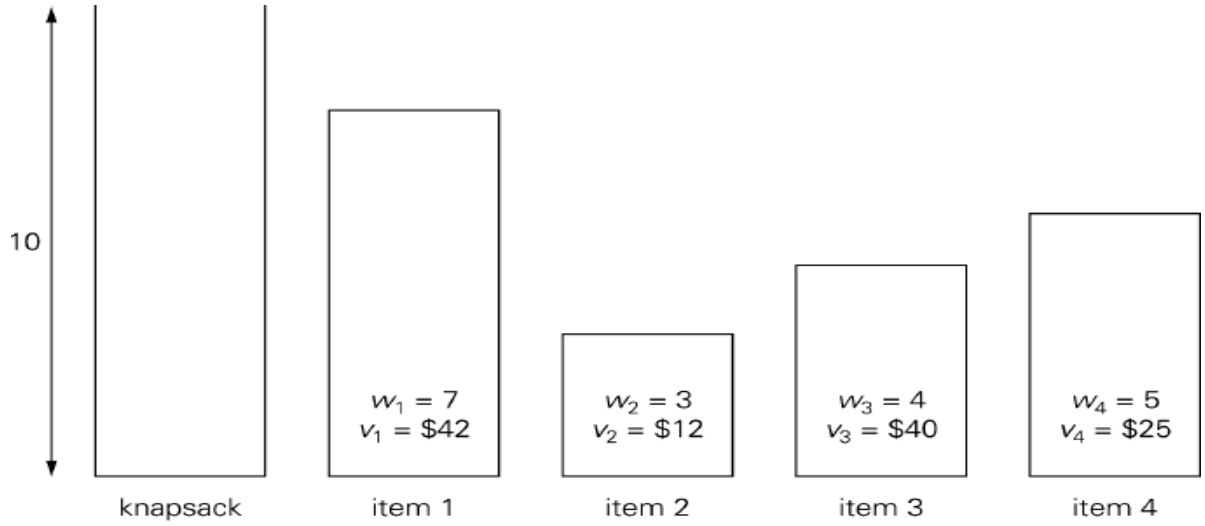
We can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them.

The total number of permutations needed is $1/2\ (n - 1)!$, which makes the exhaustive-search approach impractical for all but very small values of n.

2.5.2 KNAPSACK PROBLEM EXAMPLE

Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

- Brute Force Approach: generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.



(a)

KNAPSACK PROBLEM SOLUTION

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

(a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)

Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

2.5.3 ASSIGNMENT PROBLEM

There are *n* people who need to be assigned to execute *n* jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the *i*th person is assigned to the *j*th job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

The exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, . . . , *n*, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

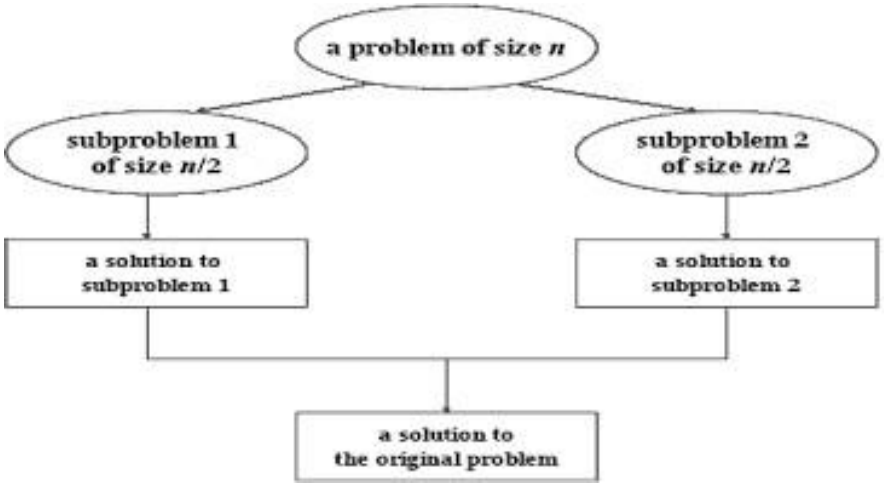
Since the number of permutations to be considered for the general case of the assignment problem is *n*!, exhaustive search is impractical for all but very small instances of the problem.

2.5 DIVIDE AND CONQUER METHODOLOGY

2.6.1 GENERAL METHOD

Divide and Conquer is the most-well known algorithm design technique. It works according to the following general plan:

1. Divide instance of the problem into two or more smaller instances
2. Solve the smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions



Control Abstraction

A control abstraction we mean a procedure whose flow of control is clear but whose primary operations are by other procedures whose precise meanings are left undefined.

```
Algorithm DC( P)
{
        if small(P) then return S(P)
        else
        {
                divide P into smaller instance p1,p2, …pk, k ≥ 1
                Apply DC to each of these subproblems
                Return combine (DC(p1) DC(p2), ---, DC(pk));
        }
}
```

Divide-and-Conquer Recurrence Relation

A problem's instance of size n is divided into two instances of size n/2. Generally, an instance of size n can be divided into several instances of size n/b,with a of them needed to be solved. Here 'a' and 'b' are constants; $a \geq 1$and $b > 1$.

Assuming that the size n is a power of b, we get the following recurrence for the running time T(n):

$T(n) = aT(n/b) + f(n)$

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

The order of growth of T(n) depends on the values of the constants a and b and the order of growth of f(n).

Master Theorem

The recurrence relation for the divide and conquer technique is given by

$T(n) = aT(n/b) + f(n)$

where

a : the number of subproblems

T(n/b) : the time spent on solving a subproblem of size n/b

f(n) : a function that represents the time spent on dividing the problem into smaller ones and on combining their solutions.

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the recurrence equation, then

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

The same results hold with O and Ω notations, too.

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

$$A(n) = 2\,A(n/2) + 1$$

Thus for this example, a = 2, b=2, and d = 0.

We have $2 > 2^0$ - $a > b^d$

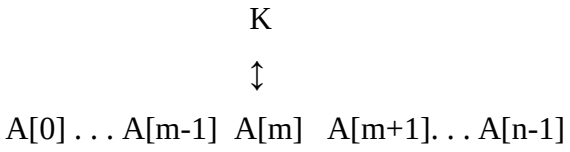
Hence, $A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

Divide-and-Conquer Examples

- Binary search
- Mergesort
- Quicksort
- Multiplication of large integers

2.6.2 BINARY SEARCH

- Binary Search is a very efficient algorithm for searching in sorted array.
- It works by comparing a search key K with the array's middle element A[m].
- If they match, the algorithm stops;otherwise, the same operation is repeated recursively for the first half of the array if K<A[m] and for the second half if K>A[m].



Search here if K < A[m]

Search here if K > A[m]

- The method reduces the number of elements needed to be checked by a factor of twoeach time, and finds the target value, if it exists in logarithmic time.

- The Binary Search is a much more effective search method than a sequential or linearsearch, for example, especially for large sets of data.

- An ideal application of the Binary Search would be searching for a record in an ordered list of employees which are kept in order by the employees' unique ID.

ALGORTIHM BinarySearch(A[0..n-1],K)
// Implements non-recursive binary search
// Input : An array A[0..n-1] sorted in ascending order and a search key K
//Output : An index of the array's element that is equal to K or -1 if there is no such element
l ← 0; r ←n-1
while (l ≤ r) do
m ← (l+r) / 2
if K = A[m] return m
else if K < A[m] r ← m-1
else l ← m+1
return -1

Example

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	27	31	39	42	55	70	74	81	85	93	98

Itn 1 : l m r

Itn 2: l m r

Itn 3 : l,m r

Analysis:

The standard way to analyze the efficiency of this algorithm is to count the number of times the search key is compared with an element of the array.
The worst-case recurrence relation is given by

$C_w(n) = C_w(n/2) + 1$ for $n > 1$, $C_w(1) = 1$

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

The average –case efficiency is given by

$C_{avg}(n) \approx \log_2 n$

2.6.3 MERGE SORT

- Merge sort is a sorting algorithm based on the divide and conquer technique.
- The general concept is that we first break the list into two smaller lists of roughly the same size.
- Then use merge sort recursively on the subproblems, until they cannot subdivide anymore (i.e. when they contain zero or one elements).
- Finally, we can merge by stepping through the lists in linear time.
- Merge sort is a stable algorithm meaning that it preserves the input order of equal elements in the sorted output. Mergesort divides the input elements according to their position in the array.
- It works by sorting the a given array A[0..n-1] by dividing it into two halves A[0..(n/2) – 1] and A[(n/2)..n-1], sorting each of them recursively and then merging the two smaller sorted arrays into a single sorted one.

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)

The merging of two sorted arrays can be done as follows:

- Two pointers are initialized to point to the first elements of the arrays being merged;
- Then the elements pointed to are compared and the smaller of them is added to a newarray being constructed;
- After that, the index of that smaller element is incremented to point to its immediatesuccessor in the array;
- This operation is continued until one of the two given arrays is exhausted and the

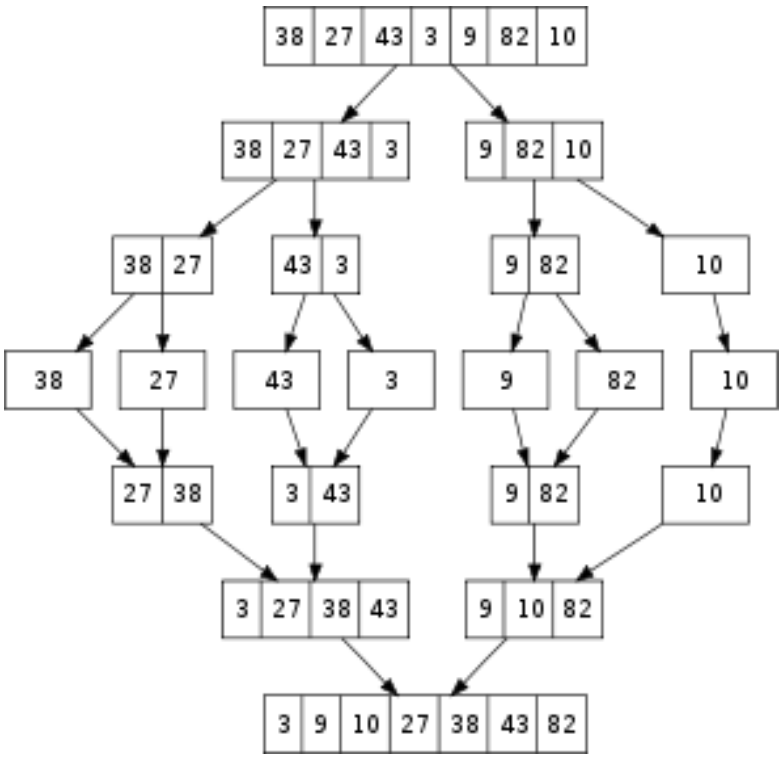
remaining elements of the other array are copied to the end of the new array.

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 38,27 ,43, 3 , 9, 82, 10 is illustrated below.



Analysis

Assuming that n is a power of 2, the recurrence relation for the number of key comparisons is

$C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$

and the Initial Condition is

$C(1) = 0$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage.At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one.

In the worst case, neither of the two arrays becomes empty before the other one contains just one element (ie., the smaller elements may come from the alternating arrays).Therefore, for the worst case, $C_{merge}(n) = n-1$, and we have the recurrence

$C_{worst}(n) = 2C_{worst}(n/2) + n-1$ for $n>1$ and

$C_{worst}(1) = 0$

According to the Master Theorem,

$C_{worst}(n) \in \Theta(n \log n)$

2.6.3 QUICK SORT

- Quick sort is a well-known sorting algorithm developed by C. A. R. Hoare.
- Quick sort is also known as "partition-exchange sort".
- It has two phases:

  Partition phase- Most of the work is done in the partition phase

  Sort phase - simply sorts the two smaller problems that are generated in the partition phase
- It is significantly faster in practice than other algorithms. It is not a stable sort algorithm but an in-place partition algorithm.
- Quick sort divides the input elements according to their value.

The algorithm works as follows:

- Pick an element, called a pivot, from the list. It is usually the first element.
- Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

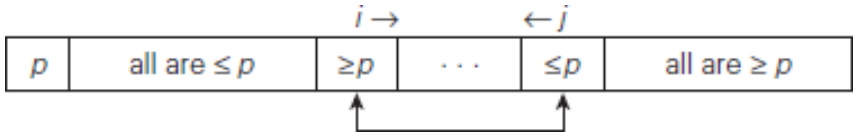
```
ALGORITHM Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //       indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r])  //s is a split position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

```
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
```

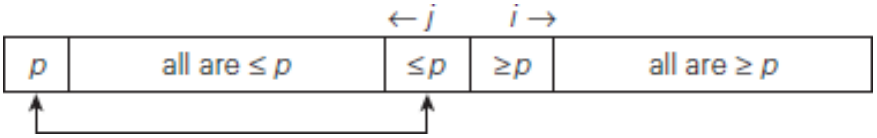
- A partition of A[0..n-1] and its subarray A[l..r] can be achieved by the above algorithm.
- First we select an element with respect to whose value we are going to divide the subarray.
- This element is called the **pivot**. We use the first element as the pivot element: p = A[l].
- The elements are then rearranged to achieve a partition.
- The method is based on two scans of the subarray:one is from left-to-right and the otherfrom right-to-left,each comparing the subarray's elements with the
- pivot.

- The left-to-right scan starts with the second element.Since we want elements smaller than the pivot to be in the first part of the subarray,this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot.
- The right-to-left scan starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second art 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.
- Depending on whether the scanning indices have crossed or not ,three situations may arise

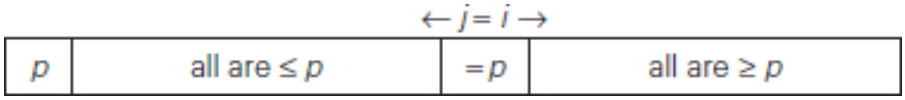
(i) If scanning indices i and j have not crossed, i.e i < j, we simply exchange A[i] and A[j] and continue the scans by incrementing i and decrementing j, respectively.



(ii) If the scanning indices have crossed over i.e. i > j, we exchange the pivot with A[j] and partition the array .At this point, the pivot will be placed in its correct position.



(iii)Finally, 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.



Example

Consider the array 5 ,3 ,1 ,9 ,8 ,2 ,4 ,7

Here n = 8, pivot = A[0] = 5

i = 1 and j = 7

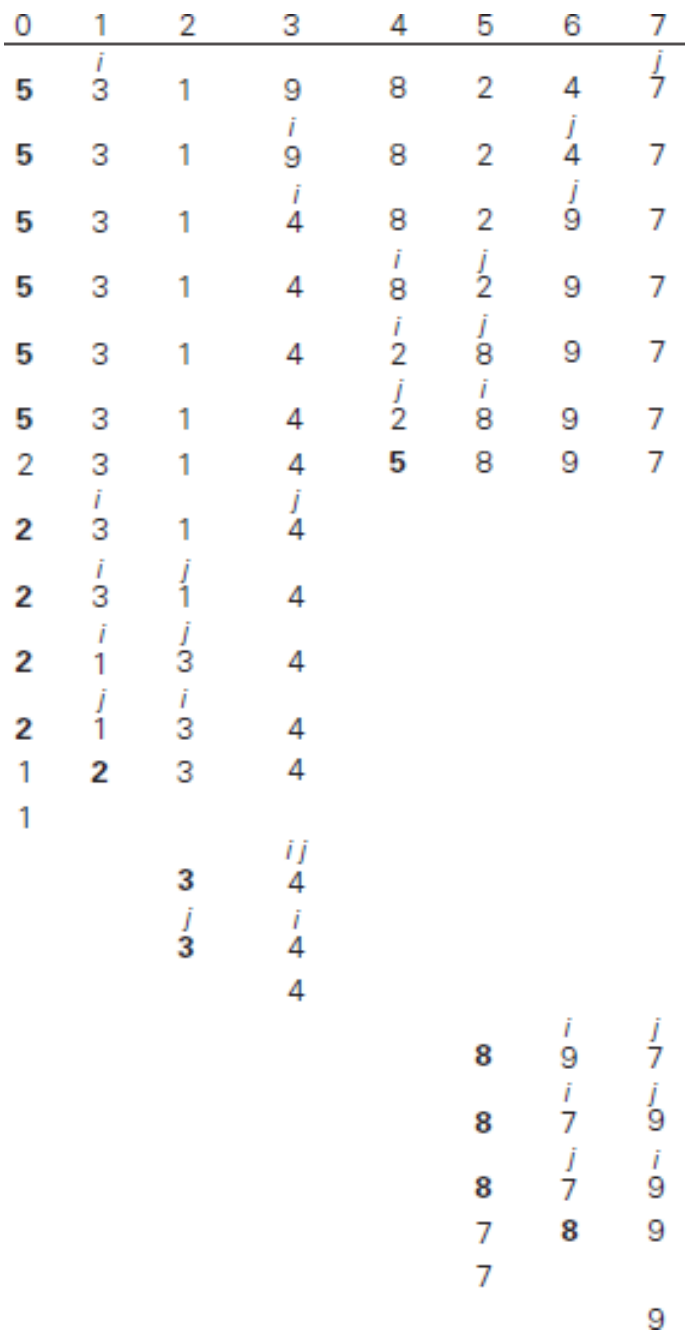


Figure. Array's transformations with pivots shown in bold.

Analysis

The number of key comparisons made before a partition is achieved in 'n+1', if the scanning indices cross over and 'n' if they coincide.

If all the splits happen in the middle of the corresponding subarrays, we will have the best case.

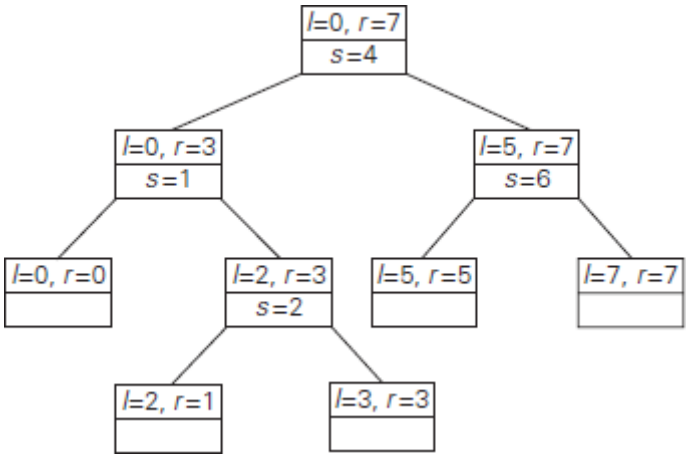


Figure. Tree of recursive calls to *Quicksort* with input values *l* and *r* of subarray bounds and split position *s* of a partition obtained.

The number of key comparisons in the best case is given by

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n>1$

$C_{best}(1) = 0$

According to Master Theorem,

$C_{best}(n) \in \Theta(n \log_2 n)$

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 in the worst case is given by

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

$C_{worst}(n) \in \Theta(n^2)$

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 $(0 \le s \le n-1)$ 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:

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

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

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

Randomized Quick Sort

Randomized quicksort is a variant of the quicksort algorithm that randomly selects a pivot element from the array to be sorted, instead of always choosing the first or last element as the pivot. The pivot is then used to partition the array into two sub-arrays, one with elements less than the pivot and one with elements greater than the pivot. The sub-arrays are then recursively sorted.

Algorithm for random pivoting using Lomuto Partitioning

```
partition(arr[], lo, hi)
pivot = arr[hi]
        i = lo     // place for swapping
for j := lo to hi – 1 do
        if arr[j] <= pivot then
swap arr[i] with arr[j]
i = i + 1
swap arr[i] with arr[hi]
return i


partition_r(arr[], lo, hi)
   r = Random Number from lo to hi
   Swap arr[r] and arr[hi]
   return partition(arr, lo, hi)


quicksort(arr[], lo, hi)
   if lo < hi
      p = partition_r(arr, lo, hi)
```

quicksort(arr, lo , p-1)

quicksort(arr, p+1, hi)

2.6.4 MULTIPLICATION OF LARGE INTEGERS

- Let us consider two n-digit integers a and b where n is a positive integer.
- The multiplication is achieved as follows :divide both the numbers in the middle .
- We denote the first half of a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0.
- The notations $a = a_1a_0$ implies $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$.
- The multiplication is carried out using the formula

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

where

$c_2 = a_1 * b_1 \rightarrow$ the product of their first halves

$c_0 = a_0 * b_0 \rightarrow$ the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0) \rightarrow$ the product of sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Example

2135 * 4014

Given a = 2135 and b = 4014 and n = 4

Solution:

Let a = a_1a_0 = 2135 = a_1:21 and a_0:35

$= a_1 10^{n/2} + a_0 = (21*10^2 + 35)$

||| ly , let $b = b_1b_0$ = 4014 = b_1:40 and b_0:14

$= b_1 10^{n/2} + b_0 = (40*10^2 + 14)$

Now we will compute the product using the formula

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

$c_2 = (21*40) = 840$

$c_0 = (35*14) = 490$

$c_1 = (21+35) * (40+14) - (840 + 490) = 56 * 54 - 1330 = 1694$

Substituting in the formula ,we get

$c = (840 * 10^4) + (1694 * 10^2) + 490$

$c = 8400000 + 169400 + 490$

$c = 8569890$

Analysis

The analysis depends upon the number of multiplications. Since multiplication of n-digit numbers requires three multiplications of n/2 digit numbers, the recurrence relation for the number of multiplications is given by

$M(n) = 3\ M(n/2)$ for $n > 1$, $M(1) = 1$.

Let $n = 2^k$.Hence we have

$M(2^k) = 3\ M(2^{k-1})$,

$M(2^0) = 1$

Solving by backward substitutions we get

$M(2^k) = 3\ M(2^{k-1})$

$= 3[3\ M(2^{k-2})] = 3^2\ M(2^{k-2})$

$= 3^2[3\ M(2^{k-3})] = 3^3\ M(2^{k-3})$

…

$= 3^i\ M(2^{k-i})$

Substituting i = k we get

$= 3^k\ M(2^{k-k}) = 3^k\ M(2^0) = 3^k(1) = 3^k$.

Since $n = 2^k$, $k = \log_2 n$,

$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$.

Strassen's Matrix Multiplication Algorithm

Using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit. Strassen's Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n.

Divide X, Y and Z into four (n/2)×(n/2) matrices as represented below −

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following −

M1:=(A+C)×(E+F)
M2:=(B+D)×(G+H)
M3:=(A−D)×(E+H)
M4:=A×(F−H)
M5:=(C+D)×(E)
M6:=(A+B)×(H)
M7:=D×(G−E)
Then,
I:=M2+M3−M6−M7
J:=M4+M6
K:=M5+M7
L:=M1−M3−M4−M5

Analysis

$$T(n) = \begin{cases} c & if\ n = 1 \\ 7\ x\ T(\frac{n}{2}) + d\ x\ n^2 & otherwise \end{cases}$$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$.