# Design and analysis of Divide and Conquer Algorithms

## DIVIDE AND CONQUER ALGORITHM:

Divide and conquer is a powerful algorithmic technique used to solve complex problems by breaking them down into smaller, more manageable subproblems, solving the subproblems recursively, and then combining their solutions to form the solution to the original problem. This technique is widely used in computer science and has applications in various fields such as computer graphics, numerical algorithms, and data structures.

The basic steps of the divide and conquer approach are as follows:

**Divid**e: Break the problem into smaller, more manageable subproblems. This typically involves dividing the input data into two or more parts.
**Conquer:** Solve each subproblem recursively. This step involves solving the subproblems independently, often using the same divide and conquer approach.
**Combine**: Merge the solutions of the subproblems to form the solution to the original problem. This step typically involves combining the solutions of the subproblems in a way that produces the overall solution.

The divide and conquer approach is particularly useful for solving problems that exhibit the following characteristics:

**Optimal Substructure**: The problem can be broken down into smaller subproblems, and the optimal solution to the original problem can be constructed from the optimal solutions of its subproblems.
**Overlapping Subproblems**: The same subproblems occur multiple times in the solution process, and it is more efficient to solve them only once and store their solutions for future use.
Some classic examples of algorithms that use the divide and conquer technique include:

**Merge Sort**: A sorting algorithm that divides the input array into two halves, sorts each half recursively, and then merges the sorted halves.
**Quick Sort**: Another sorting algorithm that partitions the input array into two subarrays around a pivot element, recursively sorts each subarray, and then combines them.
**Binary Search**: A search algorithm that divides the search space in half at each step, discarding the half that does not contain the target element.
**Strassen's Algorithm**: An algorithm for multiplying two matrices that divides the matrices into smaller submatrices, performs recursive matrix multiplications, and combines the results.

These examples demonstrate the versatility and effectiveness of the divide and conquer technique in solving a wide range of problems efficiently.

- In this approach ,we solve a problem recursively by applying 3 steps

1. **DIVIDE**-break the problem into several sub problems of smaller size.
2. **CONQUER**-solve the problem recursively.
3. **COMBINE**-combine these solutions to create a solution to the original problem.

CONTROL ABSTRACTION FOR DIVIDE AND CONQUER ALGORITHM

```
Algorithm D and C (P)
{
    if small(P)
        then return S(P)
    els
    e
        { divide P into smaller instances $P_1$ ,$P_2$   $P_k$
        Apply D and C to each sub problem
          Return combine (D and C($P_1$)+ D and C($P_2$)+   +D and C($P_k$))
    }
```

}

Let a recurrence relation is expressed as T(n)=

θ(1), if n<=C

aT(n/b) + D(n)+ C(n) ,otherwise

then n=input size a=no. Of sub-problemsn/b= input size of the sub-problems

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

## Binary Search:

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found.If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past ofthe file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\mathbf{\log_2 n}$

*low* and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i:l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor(i + l)/2\rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                  return BinSrch($a, i, mid - 1, x$);
              else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Recursive binary search

```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1:n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor(low + high)/2\rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else if ($x > a[mid]$) then $low := mid + 1$;
             else return $mid$;
    }
    return 0;
}
```

Iterative binary search

**Example for Binary Search**

Let us illustrate binary search on the following 9 elements:

| *Index* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|
| *Elements* | -15 | -6 | 0 | 7 | 9 | 23 | 54 | 82 | 101 |

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

| low | high | mid |
|---|---|---|
| 1 | 9 | 5 |
| 6 | 9 | 7 |
| 8 | 9 | 8 |
| 9 | 9 | 9 |
| | | found |

Number of comparisons = 4

2. Searching for x = 82

| low | high | mid |
|---|---|---|
| 1 | 9 | 5 |
| 6 | 9 | 7 |
| 8 | 9 | 8 |
| | | found |

Number of comparisons = 3

3. Searching for x = 42

| low | high | mid |
|---|---|---|
| 1 | 9 | 5 |
| 6 | 9 | 7 |
| 6 | 6 | 6 |
| 7 | 6 not found | |

Number of comparisons = 4

4. Searching for x = -14

| low | high | mid |
|---|---|---|
| 1 | 9 | 5 |
| 1 | 4 | 2 |
| 1 | 1 | 1 |
| 2 | 1 | not found |

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

| *Index* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|
| *Elements* | -15 | -6 | 0 | 7 | 9 | 23 | 54 | 82 | 101 |
| *Comparisons* | *3* | *2* | *3* | *4* | *1* | *3* | *2* | *3* | *4* |

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 orapproximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$

The time complexity for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$.

| **Successful searches** | | | **un-successful searches** |
|---|---|---|---|
| $\Theta(1)$, | $\Theta(\log\ n)$, | $\Theta(\log\ n)$ | $\Theta(\log n)$ |
| Best | average | worst | best, average and worst |

**Analysis for worst case**

Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

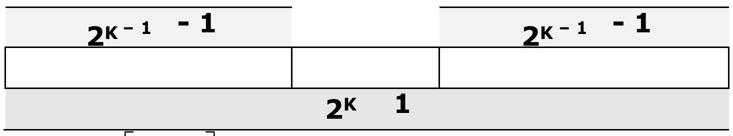
$$T(0) = 0$$

$$T(n) = 1 \qquad \text{if } x = a\,[mid]$$

$$= 1 + T([(n + 1) / 2] - 1) \qquad \text{if } x < a\,[mid]$$

$$= 1 + T(n - [(n + 1)/2]) \qquad \text{if } x > a\,[mid]$$

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2^K - 1 + 1}{2} \right\rfloor = 2^{K-1}$ for K > 1

Giving,

$$T(0) = 0$$

$$T(2^k - 1) = 1 \qquad \text{if } x = a\,[mid]$$

$$= 1 + T(2^{K-1} - 1) \qquad \text{if } x < a\,[mid]$$

$$= 1 + T(2^{k-1} - 1) \qquad \text{if } x > a\,[mid]$$

In the worst case the test x = a[mid] always fails, so

$$w(0) = 0$$

$$w(2^k - 1) = 1 + w(2^{k-1} - 1)$$

This is now solved by repeated substitution:

$$w(2^k - 1) = 1 + w(2^{k-1} - 1)$$

$$
\begin{aligned}
&= 1 + [1 + w(2^{k-2} - 1)] \\
&= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\
&= \ldots\ldots\ldots \\
&= \ldots\ldots\ldots \\
&= i + w(2^{k-i} - 1)
\end{aligned}
$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

## Maximum and Minimum:

- Let us consider another simple problem that can be solved by the divide-and-conquer technique.

- The problem is to find the maximum and minimum items in a set of ‘n’ elements.

- In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.

- More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of character, the cost of an element comparison is much higher than the cost of the other operations.

- Hence, the time is determined mainly by the total cost of the element comparison.

```
1. Algorithm straight MaxMin(a,n,max,min)
2. // set max to the maximum & min to the minimum of a[1:n]
3. {
4. max:=min:=a[1];
5. for I:=2 to n do
6. {
7. if(a[I]>max) then max:=a[I];
8. if(a[I]<min) then min:=a[I];
9. }
10. }
```

**Algorithm:** Straight forward Maximum & Minimum

- Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.
- An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false.

- Hence we can replace the contents of the for loop by,

```
If(a[I]>max) then max:=a[I];
Else if (a[I]<min) then min:=a[I];
```

- Now the best case occurs when the elements are in increasing order.

→ The no. of element comparison is (n-1).

- The worst case occurs when the elements are in decreasing order.

  → The no. of elements comparison is 2(n-1)

- The average no. of element comparison is < than 2(n-1)

- On the average a[I] is > than max half the time, and so, the avg. no. of comparison is

3n/2-1.

- A divide- and conquer algorithm for this problem would proceed as follows:

  → 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 by recursively invoking the same divide-and-conquer algorithm.

**Algorithm:** Recursively Finding the Maximum & Minimum

```
Algorithm MaxMin (I,j,max,min)
//a[1:n] is a global array, parameters I & j
//are integers, 1<=I<=j<=n.The effect is to
//set max & min to the largest & smallest value
//in a[I:j], respectively.
{
if(I=j) then max:= min:= a[I];
else if (I=j-1) then // Another case of small(p)
{
if (a[I]<a[j]) then
{
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 mid:=[(I+j)/2];
//solve the subproblems
MaxMin(I,mid,max.min);
MaxMin(mid+1,j,max1,min1);
//combine the solution
if (max<max1) then max=max1;
if(min>min1) then min = min1;
}
}
```

- The procedure is initially invoked by the statement,
  MaxMin(1,n,x,y)
- Suppose we simulate MaxMin on the following 9 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 4 items of information: I, j, max & imin. • Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin.

• This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size. • From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)

• The include no.s in the upper left corner of each node represent the order in which max & min are assigned values.

No. of element Comparison:

• If T(n) represents this no., then the resulting recurrence relations is

T(n)={ T([n/2]+T[n/2]+2 n>2
1 n=2
0 n=1

→ When 'n' is a power of 2, n=2^k for some +ve integer 'k', then
T(n) = 2T(n/2) +2
= 2(2T(n/4)+2)+2
= 4T(n/4)+4+2
*
*
= 2^k-1T(2)+
= 2^k-1+2^k-2
= 2^k/2+2^k-2
= n/2+n-2
= (n+2n)/2)-2
**T(n)=(3n/2)-2**

*Note that (3n/3)-3 is the best-average, and worst-case no. of comparisons when 'n' is a power of 2.

## Merge sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numbers which we will assume is stored in an array A [1...n]. The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

Divide: Split A down the middle into two sub-sequences, each of size

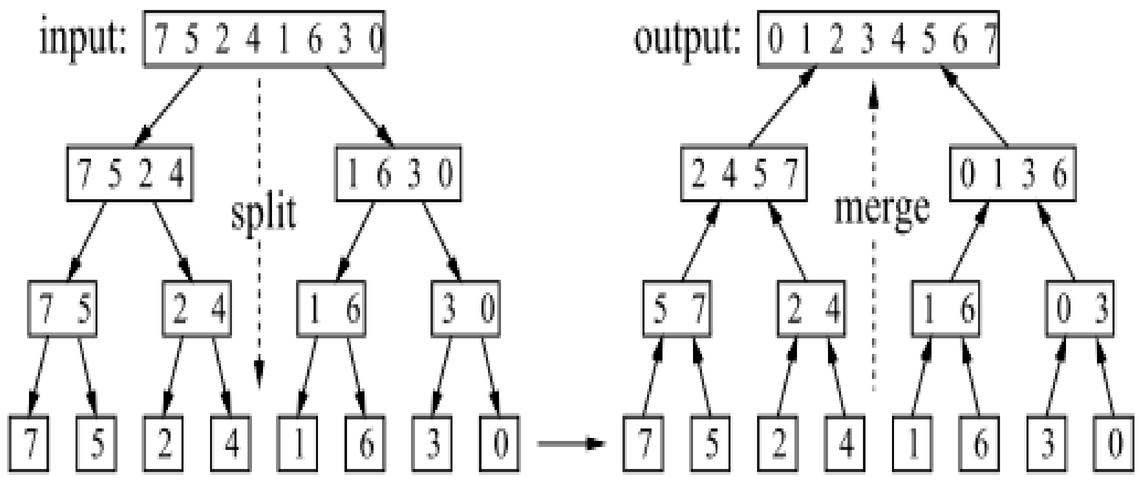
roughly n/2 . Conquer: Sort each subsequence (by calling MergeSort

recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases areshown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure that mergestwo sorted list is available to us. We’ll implement it later. Because the algorithm is called recursively on sub lists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the sub-array A [ p..r ] and return the sorted result in the same subarray.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may return immediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = ( p + r ) / 2 (rounded down to then nearest integer). Then we split the array into subarrays A [ p..q ] and A [ q
+ 1 ..r ] . Call Merge Sort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) which merges these two subarrays into a single sorted array.

```
MergeSort(array A, int p, int r) {
        if      (p < r) {                                   // we have at least 2
        itemsq = (p + r)/2
        MergeSort(A, p, q)                          // sort A[p..q]
        MergeSort(A, q+1, r)                        // sort A[q+1..r]

        Merge(A, p, q, r)                           // merge everything together
        }
        }
```

Merging: All that is left is to describe the procedure that merges two sorted lists. Merge(A, p, q, r)assumes that the left subarray, A [ p..q ] , and the right subarray, A [ q + 1 ..r ] , have already been sorted. We merge these two subarrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, B [ p..r ] . We have to indices i and j, that point to the current elements of each subarray. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A.

```
Merge(array A, int p, int q, int r) {               // merges A[p..q] with
        A[q+1..r]array B[p..r]
        i = k = p                                   //initialize pointers
        j = q+1
        while (i <= q and j <= r) {                 // while both subarrays are
        nonempty
        if (A[i] <= A[j]) B[k++] = A[i++]           // copy from left subarray
        else B[k++] = A[j++]                        // copy from right subarray
        }
        while (i <= q) B[k++] = A[i++]              // copy any leftover
        to Bwhile (j <= r) B[k++] = A[j++]
        for i = p to r do A[i] = B[i]         // copy B back to A
        }
```

Analysis: What remains is to analyze the running time of MergeSort. First let us consider the running time of the procedure Merge(A, p, q, r). Let $n = r - p + 1$ denote the total length of both the left and right subarrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for elements.) Thus the running time to Merge n items is $\Theta(n)$ . Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$ ), which are defined explicitly.

Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort onan array of length n. For concreteness we could count whatever we like: numberof lines of pseudocode,number of comparisons, number of array accesses, since these will only differ by a constant factor.Since all of the real work is done in the Merge procedure, we will count the total time spent in theMerge procedure.

First observe that if we call MergeSort with a list containing a single element, then the running time is aconstant. Since we are ignoring constant factors, we can just write $T(n) = 1$ .

When we call MergeSort with a list of length $n > 1$ ,

e.g. Merge(A, p, r), where $r - p + 1 = n$,

the algorithm first computes $q = (p + r) / 2$ .

The subarray $A[p..q]$ , which contains $q - p + 1$ elements.

You can verify that is of size $n/2$ .

Thus the remaining subarray $A[q+1..r]$ has $n/2$ elements in it.

How long does it taketo sort the left subarray? We do not know this, but because $n/2 < n$ for $n > 1$, we can express this as $T(n/2)$. Similarly, we can express the time that it takes to sort the right subarray as $T(n/2)$.

Finally, to merge both sorted lists takes n time, by the comments made above. In conclusion we have

$T(n) = 1$ if $n = 1$,

$2T(n/2) + n$ otherwise.

Solving the above recurrence we can see that merge sort has a time complexity of $\Theta(n \log n)$.

## QUICKSORT

- Worst-case running time: *O (n2)*.
- Expected running time: *O (n* lg*n)*.
- Sorts in place.

**Description of quicksort**

Quicksort is based on the three-step process of divide-and-conquer.

• To sort the subarray$A[p \, . \, . \, r]$:

**Divide:** Partition $A[p \, . \, . \, r]$, into two (possibly empty) subarrays$A[p \, . \, . \, q - 1]$ and $A[q + 1 \, . \, . \, r]$, such that each element in the Þrstsubarray$A[p \, . \, . \, q - 1]$ is $\leq A[q]$ and

$A[q]$ is $\leq$ each element in the second subarray$A[q + 1 \, . \, . \, r]$.

**Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.

**Combine:** No work is needed to combine the subarrays, because they are sorted in place.

• Perform the divide step by a procedure PARTITION, which returns the index $q$ that marks theposition separating the subarrays.

```
QUICKSORT (A, p, r)
if p < r
then q ← PARTITION(A, p, r )
        QUICKSORT (A, p, q − 1)
        QUICKSORT (A, q + 1, r)
```

Initial call is QUICKSORT *(A,* 1*, n)*

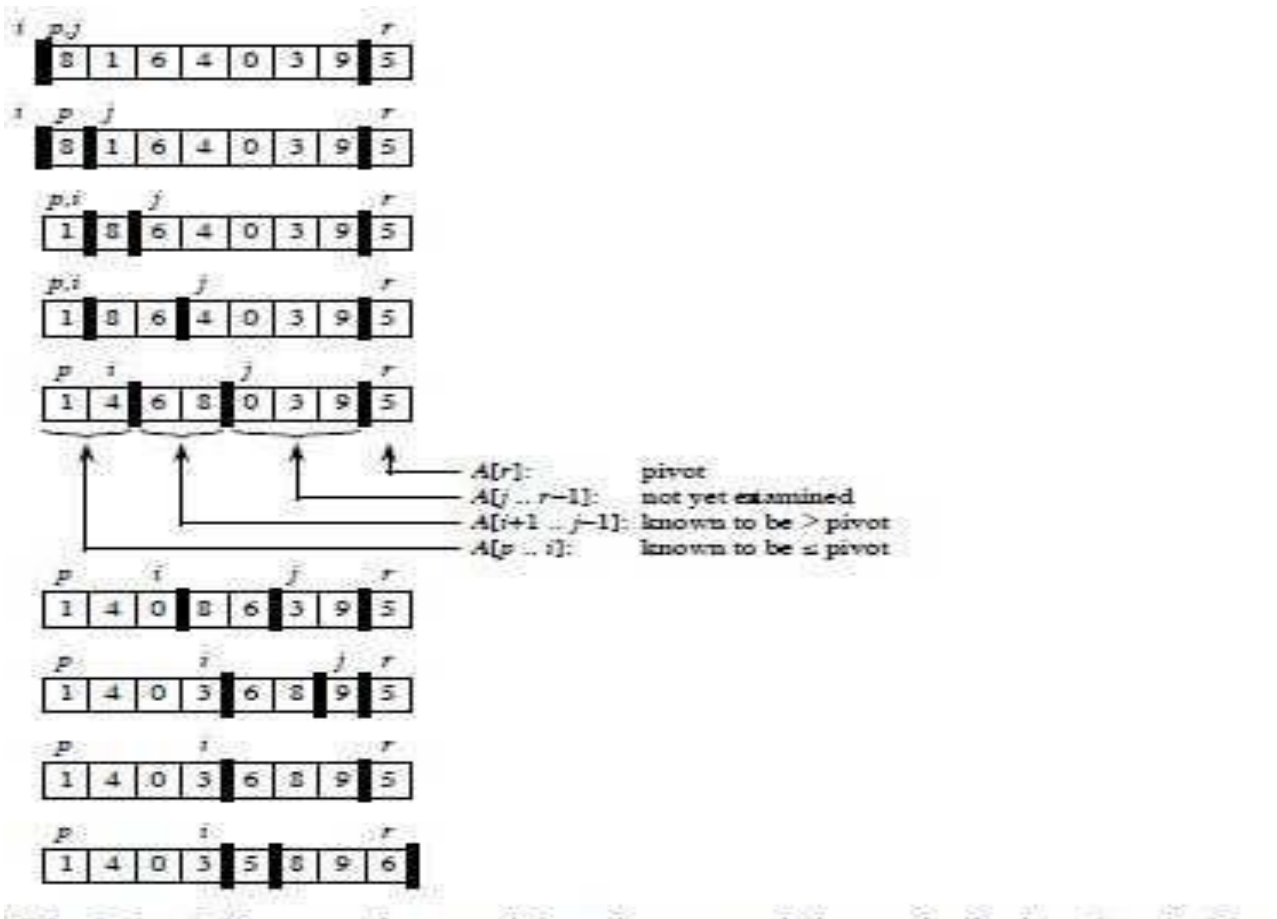
**Partitioning**

Partition subarray$A\ [p \, . \, . \, . \, r]$ by the following procedure:

```
PARTITION (A, p, r)
  x ← A[r ]
  i ← p −1
  for j ← p to r −1
do if A[ j ] ≤ x
then i ← i + 1
```

```
exchange$A[i] \leftrightarrow A[j]$
        exchange$A[i+1] \leftrightarrow A[r]$
**return** $i + 1$
```

- PARTITION always selects the last element $A[r]$ in the subarray $A[p\,.\,.\,r]$ as the ***pivot*** the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:



A[r]: pivot
A[j .. r−1]: not yet examined
A[i+1 .. j−1]: known to be > pivot
A[p .. i]: known to be ≤ pivot

[The index j disappears because it is no longer needed once the for loop is exited.]

## Performance of Quicksort

The running time of Quicksort depends on the partitioning of the subarrays:

• If the subarrays are balanced, then Quicksort can run as fast as mergesort.

• If they are unbalanced, then Quicksort can run as slowly as insertion sort.

**Worst case**

• Occurs when the subarrays are completely unbalanced.

• Have 0 elements in one subarray and $n - 1$ elements in the other subarray.

• Get the recurrence

$T(n) = T(n - 1) + T(0) + \Theta(n)$

$= T(n - 1) + \Theta(n)$

$= O(n2)$ .

• Same running time as insertion sort.

• In fact, the worst-case running time occurs when Quicksort takes a sorted array as input, butinsertion sort runs in $O(n)$ time in this case.

**Best case**

• Occurs when the subarrays are completely balanced every time.

• Each subarray has $\leq n/2$ elements.

• Get the recurrence

$T(n) = 2T(n/2) + \Theta(n) = O(n \lg n)$.

**Balanced partitioning**

• QuickPort's average running time is much closer to the best case than to the worst case.

• Imagine that PARTITION always produces a 9-to-1 split.

• Get the recurrence

$T(n) \leq T(9n/10) + T(n/10) + \_(n) = O(n \lg n)$.

• Intuition: look at the recursion tree.

• It's like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$.

• Except that here the constants are different; we get log10 $n$ full levels and log10/9 $n$ levels thatare nonempty.

• As long as it's a constant, the base of the log doesn't matter in asymptotic notation.

• Any split of constant proportionality will yield a recursion tree of depth $O(\log n)$.

## Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \dots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out ofplace to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.
- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.
- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], . . . . . . . x[j-1] and x[j+1], x[j+2], x[high].
- It calls itself recursively to sort the left sub-array x[low], x[low+1], . . . . . . . x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . . . . . . . . x[high] between positions j+1 and high.

**Example**

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

| *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *Remarks* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 38 | 08 | 16 | 06 | 79 | 57 | 24 | 56 | 02 | 58 | 04 | 70 | 45 | |
| pivot | | | | i | | | | | | j | | | swap i & j |
| | | | | 04 | | | | | | 79 | | | |
| | | | | | i | | | j | | | | | swap i & j |
| | | | | | 02 | | | 57 | | | | | |
| | | | | | | j | i | | | | | | |
| (24 | 08 | 16 | 06 | 04 | 02) | **38** | (56 | 57 | 58 | 79 | 70 | 45) | swap pivot & j |
| pivot | | | | | j, i | | | | | | | | swap pivot & j |
| (02 | 08 | 16 | 06 | 04) | **24** | | | | | | | | |
| pivot, j | i | | | | | | | | | | | | swap pivot & j |
| **02** | (08 | 16 | 06 | 04) | | | | | | | | | |
| | pivot | i | | j | | | | | | | | | swap i & j |
| | | 04 | | 16 | | | | | | | | | |
| | | | j | i | | | | | | | | | |
| | (06 | 04) | **08** | (16) | | | | | | | | | swap pivot & j |
| | pivot, j | i | | | | | | | | | | | |
| | (04) | **06** | | | | | | | | | | | swap pivot & j |
| | **04** pivot, j, i | | | | | | | | | | | | |
| | | | | **16** pivot, j, i | | | | | | | | | |
| **(02** | **04** | **06** | **08** | **16** | **24)** | 38 | | | | | | | |
| | | | | | | | (56 | 57 | 58 | 79 | 70 | 45) | |
| | | | | | | | pivot | i | | | | j | swap i & j |
| | | | | | | | | 45 | | | | 57 | |
| | | | | | | | | j | i | | | | |
| | | | | | | | (45) | **56** | (58 | 79 | 70 | 57) | swap pivot & j |
| | | | | | | | **45** pivot, j, i | | | | | | swap pivot & j |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | (58 pivot | 79 i | 70 | 57) j | swap i & j |
| | | | | | | | | | | 57 | | 79 | |
| | | | | | | | | | | j | i | | |
| | | | | | | | | | (57) | **58** | (70 | 79) | swap pivot & j |
| | | | | | | | | | **57** pivot, j, i | | | | |
| | | | | | | | | | | | (70 | 79) | |
| | | | | | | | | | | | pivot, j | i | swap pivot & j |
| | | | | | | | | | | | **70** | | |
| | | | | | | | | | | | | **79** pivot, j, i | |
| | | | | | | | **(45** | **56** | **57** | **58** | **70** | **79)** | |
| **02** | **04** | **06** | **08** | **16** | **24** | **38** | **45** | **56** | **57** | **58** | **70** | **79** | |

**Analysis of Quick Sort:**

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in $S_1$.

**Worst Case Analysis**

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + Cn \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

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

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

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

$$= \mathbf{O(n^2)} \qquad - \qquad (3)$$

**Best and Average Case Analysis**

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) + T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) +------ + T(n-2)] \

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [ n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)
nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

| | | |
|---|---|---|
| T(n)/(n+1) | = | 2/(n+1) + T(n-1)/n |
| T(n-1)/n | = | 2/n + T(n-2)/(n-1) |
| T(n-2)/(n-1) | = | 2/(n-1) + T(n-3)/(n-2) |
| T(n-3)/(n-2) | = | 2/(n-2) + T(n-4)/(n-3) |
| . | | . |
| . | | . |
| T(3)/4 | = | 2/4 + T(2)/3 |
| T(2)/3 | = | 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0) |

Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + --------------+ T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + --------------+ T(2)/3 + T(1)/2] + T(0) +
[2/(n+1) + 2/n + 2/(n-1) + -----------+2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+ --------------- +1/n+1/(n+1)]

T(n) = (n+1)2[ $\sum_{2 \le k \le n+1} 1/k$

=2(n+1) [ – ]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
**T(n)= O(n log n)**

# BRUTE FORCE AND DIVIDE-AND-CONQUER

## 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.

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.

Examples:

1. Computing $a^n$ : a * a * a * … * a ( n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

## 2.2 KNAPSACK PROBLEM

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.

Real time examples:

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane’s capacity.

The exhaustive-search approach to this problem leads to 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.

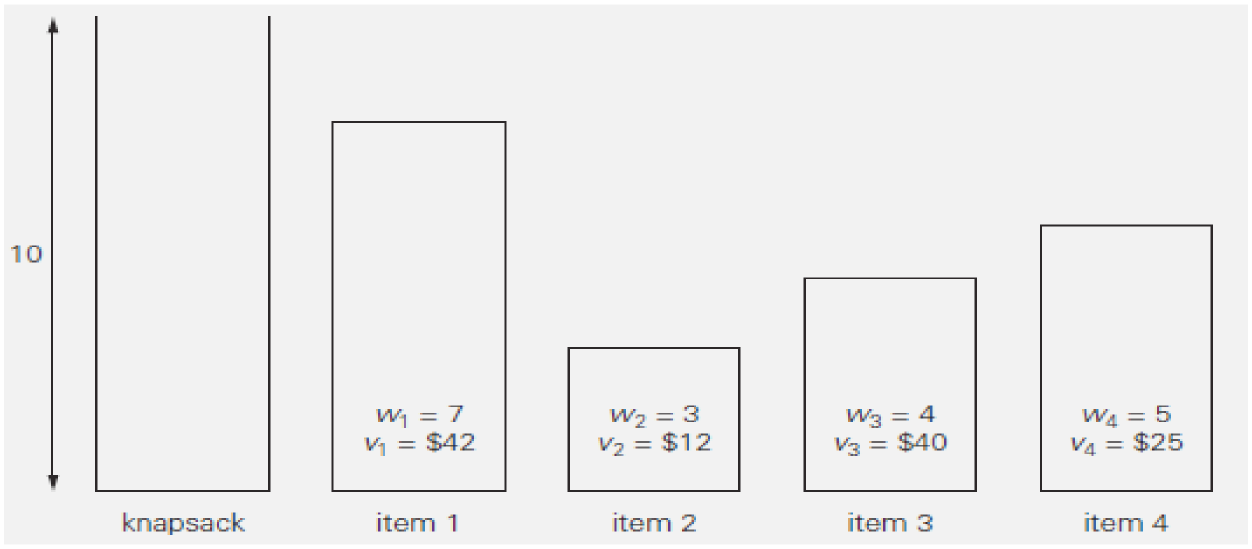


**FIGURE 2.5** Instance of the knapsack problem.

| **Subset** | **Total weight** | **Total value** |
|---|---|---|
| Φ | 0 | $0 |
| {1} | 7 | $42 |
| {2} | 3 | $12 |
| {3} | 4 | $40 |
| {4} | 5 | $25 |
| {1, 2} | 10 | $54 |
| {1, 3} | 11 | not feasible |
| {1, 4} | 12 | not feasible |
| {2, 3} | 7 | $52 |
| {2, 4} | 8 | $37 |
| **{3, 4}** | **9** | **$65 (Maximum-Optimum)** |
| {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 |

**FIGURE 2.6** knapsack problem's solution by exhaustive search. The information about the optimal selection is in bold.

**Time efficiency:** As given in the example, the solution to the instance of Figure 2.5 is given in Figure 2.6. 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.

**Note:** Exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***NP-hard problems***. **No polynomial-time** algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

## The Knapsack Problem (KP)

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a best solution from among many other solutions. It is concerned with a knapsack that has positive integer volume (or capacity) $V$. There are $n$ distinct items that may potentially be placed in the knapsack. Item $i$ has a positive integer volume $V_i$ and positive integer benefit $B_i$. In addition, there are $Q_i$ copies of item $i$ available, where quantity $Q_i$ is a positive integer satisfying $1 \le Q_i \le \infty$.

Let $X_i$ determines how many copies of item $i$ are to be placed into the knapsack. The goal is to:

Maximize

$$\sum_{i=1}^{N} B_i X_i$$

Subject to the constraints

$$\sum_{i=1}^{N} V_i X_i \le V$$

And

$$0 \le X_i \le Q_i.$$

If one or more of the $Q_i$ is infinite, the KP is *unbounded;* otherwise, the KP is *bounded* [1]. The bounded KP can be either *0-1 KP* or *Multiconstraint KP*. If $Q_i = 1$ for $i = 1, 2, \ldots, N$, the problem is a *0-1 knapsack problem* In the current paper, we have worked on the bounded *0-1 KP*, where we cannot have more than one copy of an item in the knapsack.

## Different Approaches

### Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. If there are $n$ items to choose from, then there will be $2^n$ possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0's and 1's is generated which is of length *n*. If the ith symbol of a bit string is 0, then the ith item is not chosen and if it is 1, the ith item is chosen.

```
ALGORITHM BruteForce (Weights [1 … N], Values [1 … N], A[1…N])
//Finds the best possible combination of items for the KP
//Input: Array Weights contains the weights of all items
        Array Values contains the values of all items
        Array A initialized with 0s is used to generate the bit strings
//Output: Best possible combination of items in the knapsack bestChoice [1 .. N]

for i = 1 to $2^n$ do
    j ← n
    tempWeight ← 0
    tempValue ← 0
    while ( A[j] != 0 and j > 0)
        A[j] ← 0
        j ← j – 1
    A[j] ← 1
    for k ← 1 to n do
        if (A[k] = 1) then
            tempWeight ← tempWeight + Weights[k]
            tempValue ← tempValue + Values[k]
    if  ((tempValue > bestValue) AND (tempWeight ≤ Capacity)) then
        bestValue ← tempValue
        bestWeight ← tempWeight

    bestChoice ← A
return bestChoice
```

**Complexity**

$$\sum_{i=1}^{2^n} \Big[ \sum_{j=n}^{1} + \sum_{k=1}^{n} \Big] = \sum_{i=1}^{2^n} [\{1+..+1\}(\text{n times}) + \{1+..+1\}(\text{n times})]$$

$$= (2n)^* [1+1+1\ldots..+1]\ (2^n \text{ times})$$

$$= O(2n^*2^n)$$

$$= O(n^*2^n)$$

Therefore, the complexity of the Brute Force algorithm is *$O(n2^n)$*. Since the complexity of this algorithm grows exponentially, it can only be used for small instances of the KP. Otherwise, it does not require much programming effort in order to be implemented. Besides the memory used to store the values and weights of all items, this algorithm requires a two one dimensional arrays (A[] and bestChoice[]).

# UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER

## 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.

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

## 2.2 EXHAUSTIVE SEARCH

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete searchor* direct *search.***

*Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).*

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only **general approach** for which it is more difficult to point out problems it *cannot* tackle.
2. Second, for some important problems, e.g., sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustive Search is applied to the important problems like

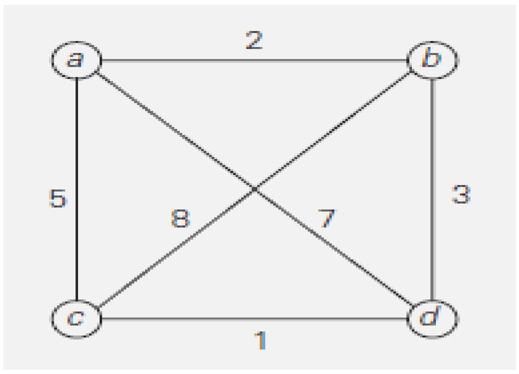
- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

## 2.3 TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. The problem asks to 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).

A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $vi_0$, $vi_1$, . . . , $vi_{n-1}$, $vi_0$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



| Tour | Length |
|---|---|
| a ---> b ---> c ---> d ---> a | I = 2 + 8 + 1 + 7 = 18 |
| a ---> b ---> d ---> c ---> a | **I = 2 + 3 + 1 + 5 = 11 optimal** |
| a ---> c ---> b ---> d ---> a | I = 5 + 8 + 3 + 7 = 23 |
| a ---> c ---> d ---> b ---> a | **I = 5 + 1 + 3 + 2 = 11 optimal** |
| a ---> d ---> b ---> c ---> a | I = 7 + 3 + 8 + 5 = 23 |
| a ---> d ---> c ---> b ---> a | I = 7 + 1 + 8 + 2 = 18 |

**FIGURE 2.4** Solution to a small instance of the traveling salesman problem by exhaustive search.

**Time efficiency**

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. **(n - 1)!**
- Consider two intermediate vertices, say, *b* and *c*, and then only permutations in which *b* precedes *c*. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

# UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER

## 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.

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.

Examples:

1. Computing $a^n$ : a * a * a * … * a ( n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

## 2.2 CLOSEST-PAIR AND CONVEX-HULL PROBLEMS

We consider a straight forward approach (Brute Force) to two well-known problems dealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

### Closest-Pair Problem

The closest-pair problem finds the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

Consider the two-dimensional case of the closest-pair problem. The points are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance.

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

The following algorithm computes the distance between each pair of distinct points and finds a pair with the smallest distance.

```
ALGORITHM BruteForceClosestPair(P)
    //Finds distance between two closest points in the plane by brute force
    //Input: A list P of n (n ≥ 2) points $p_1(x_1, y_1), \dots, p_n(x_n, y_n)$
    //Output: The distance between the closest pair of points
    $d \leftarrow \infty$
    for $i \leftarrow 1$ to $n - 1$ do
        for $j \leftarrow i + 1$ to $n$ do
            $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //sqrt is square root
    return $d$
```

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

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

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

$$= 2[(n-1) + (n-2) + \dots + 1]$$

$$= (n-1)n \in \Theta(n^2).$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor, but it cannot improve its asymptotic efficiency class.

**Convex-Hull Problem**

**Convex Set**

A set of points (finite or infinite) in the plane is called ***convex*** if for any two points *p* and *q* in the set, the entire line segment with the endpoints at *p* and *q* belongs to the set.

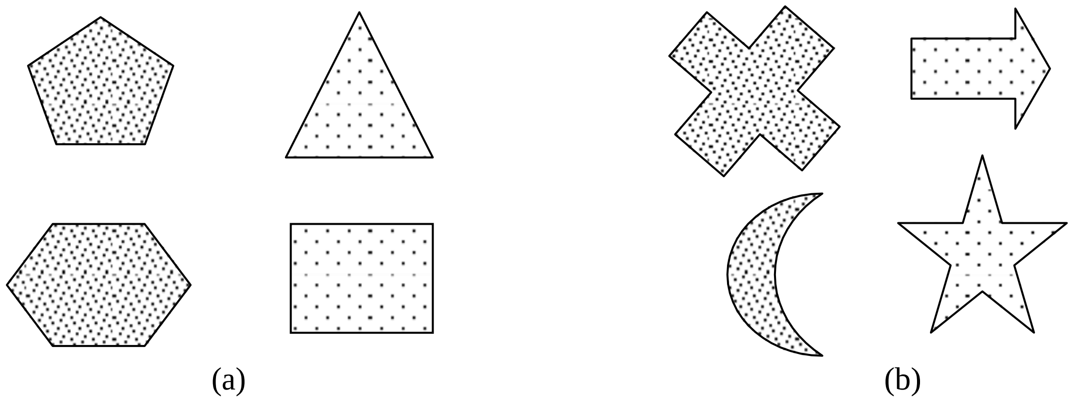


**FIGURE 2.1** (a) Convex sets. (b) Sets that are not convex.

All the sets depicted in Figure 2.1 (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.

On the other hand, the sets depicted in Figure 2.1 (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in Figure 2.2

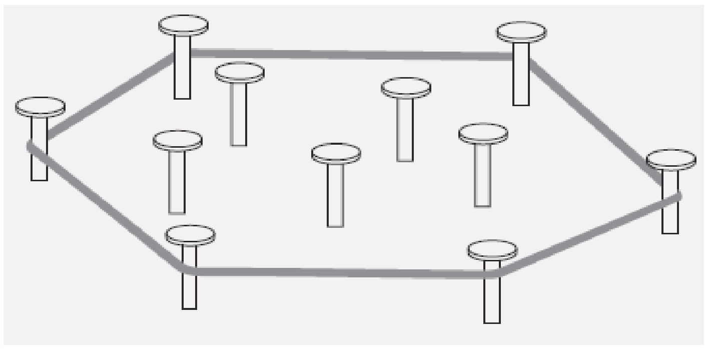


**FIGURE 2.2** Rubber-band interpretation of the convex hull.

**Convex hull**

The ***convex hull*** of a set *S* of points is the smallest convex set containing *S*. (The smallest convex hull of *S* must be a subset of any convex set containing *S*.)

If *S* is convex, its convex hull is obviously *S* itself. If *S* is a set of two points, its convex hull is the line segment connecting these points. If *S* is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on thesame line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 2.3.

**THEOREM**

The convex hull of any set *S* of $n>2$ points not all on the same line is a convex polygon with the vertices at some of the points of *S*. (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of *S*.)

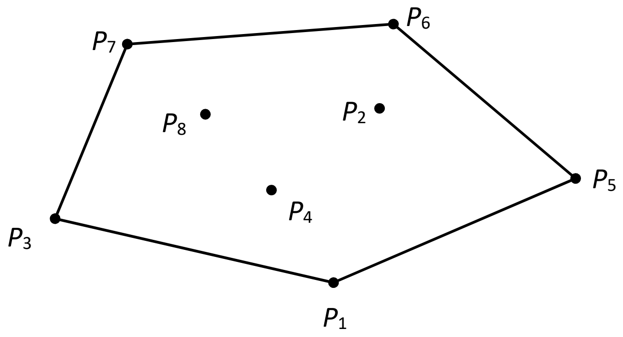


**FIGURE 2.3** The convex hull for this set of eight points is the convex polygon with vertices at $p_1$, $p_5$, $p_6$, $p_7$, and $p_3$.

The ***convex-hull problem*** is the problem of constructing the convex hull for a given set *S* of *n* points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon "extreme points." By definition, an ***extreme point*** of a convex set is a point of this set that is *not a middle point of any line segment with endpoints in the set*. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 2.3 are $p_1$, $p_5$, $p_6$, $p_7$, and $p_3$.

**Application**

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the ***simplex method***, This algorithm solves ***linear programming*** **Problems.**

We are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of *n* points of a given set are extreme points of the set's convex hull. we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counterclockwise order.

We can solve the convex-hull problem by brute-force manner. The convex hull problem is one with no obvious algorithmic solution. there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points *pi* and *pj* of a set of *n* points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

**Facts**

A few elementary facts from analytical geometry are needed to implement the above algorithm.

- First, the straight line through two points $(x_1, y_1)$, $(x_2, y_2)$ in the coordinate plane can be defined by the equation $ax + by = c$,where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.
- Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.

**Time efficiency of this algorithm.**

Time efficiency of this algorithm is in $O(n^3)$: for each of $n(n-1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.

## 1.8 Graphs (BFS ,DFS)

In Java, you can implement various graph traversal techniques such as Depth-First Search (DFS) and Breadth-First Search (BFS) using different data structures like adjacency lists or adjacency matrices. Here, I'll provide examples for both DFS and BFS using adjacency lists in Java.

1. **Depth-First Search (DFS):**

DFS explores as far as possible along each branch before backtracking.

It's often used for topological sorting, cycle detection, and solving maze problems.

Example graph:

```
      A
     /  \
    B    C
   / \    \
  D   E --- F
```

DFS traversal starting from node 'A':

A, B, D, E, F, C

DFS explores as far as possible along each branch before backtracking. Here's how you can implement DFS in Java using recursion:

```
import java.util.*;


public class DFSGraphTraversal {
    private Map<Integer, List<Integer>> graph;


    public DFSGraphTraversal(Map<Integer, List<Integer>> graph) {
        this.graph = graph;
    }


    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        dfsHelper(start, visited);
    }


    private void dfsHelper(int node, Set<Integer> visited) {
        visited.add(node);
        System.out.print(node + " ");

        List<Integer> neighbors = graph.getOrDefault(node, new ArrayList<>());
        for (int neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                dfsHelper(neighbor, visited);
            }
        }
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(2));
        graph.put(2, Arrays.asList(0, 3));
        graph.put(3, Arrays.asList(3));

        DFSGraphTraversal dfsTraversal = new DFSGraphTraversal(graph);
        System.out.println("DFS Traversal:");
        dfsTraversal.dfs(2);
    }
}
```

**2. Breadth-First Search (BFS):**

Graph traversal techniques are algorithms used to visit and explore all the nodes in a graph. Two common graph traversal techniques are Breadth-First Search (BFS) and Depth-First Search (DFS). Let's discuss these techniques with examples of graphs:

Breadth-First Search (BFS):

```
    A
   / \
  B   C
  / \    \
 D   E --- F
```

BFS traversal starting from node 'A':

Level 0: A

Level 1: B, C

Level 2: D, E, F

BFS explores nodes level by level, visiting all the neighbors of a node before moving on to the next level.

It's often used to find the shortest path in unweighted graphs.

Example graph:

BFS explores nodes level by level, visiting all the neighbors of a node before moving on to the next level. Here's how you can implement BFS in Java using a queue:

```
import java.util.*;


public class BFSGraphTraversal {
    private Map<Integer, List<Integer>> graph;


    public BFSGraphTraversal(Map<Integer, List<Integer>> graph) {
        this.graph = graph;
    }


    public void bfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        visited.add(start);


        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");


            List<Integer> neighbors = graph.getOrDefault(node, new ArrayList<>());
            for (int neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    queue.offer(neighbor);
                    visited.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(0, Arrays.asList(1, 2));
        graph.put(1, Arrays.asList(2));
        graph.put(2, Arrays.asList(0, 3));
        graph.put(3, Arrays.asList(3));

        BFSGraphTraversal bfsTraversal = new BFSGraphTraversal(graph);
        System.out.println("BFS Traversal:");
        bfsTraversal.bfs(2);
    }
}
```

# UNIT II  BRUTE FORCE AND DIVIDE-AND-CONQUER

## 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.

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very  useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

## 2.2 EXHAUSTIVE SEARCH

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete searchor*** **direct** ***search.***

*Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).*

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only **general approach** for which it is more difficult to point out problems it *cannot* tackle.
2. Second, for some important problems, e.g., sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustive Search is applied to the important problems like

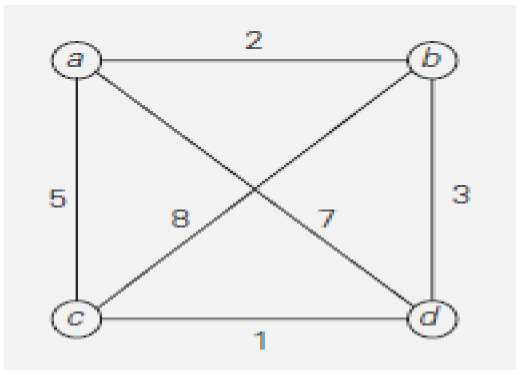
- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

## 2.3 TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. The problem asks to 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).

A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $vi_0$, $vi_1, \ldots, vi_{n-1}, vi_0$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



| Tour | Length |
|---|---|
| a ---> b ---> c ---> d ---> a | I = 2 + 8 + 1 + 7 = 18 |
| a ---> b ---> d ---> c ---> a | **I = 2 + 3 + 1 + 5 = 11 optimal** |
| a ---> c ---> b ---> d ---> a | I = 5 + 8 + 3 + 7 = 23 |
| a ---> c ---> d ---> b ---> a | **I = 5 + 1 + 3 + 2 = 11 optimal** |
| a ---> d ---> b ---> c ---> a | I = 7 + 3 + 8 + 5 = 23 |
| a ---> d ---> c ---> b ---> a | I = 7 + 1 + 8 + 2 = 18 |

**FIGURE 2.4** Solution to a small instance of the traveling salesman problem by exhaustive search.

**Time efficiency**

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. **(n - 1)!**
- Consider two intermediate vertices, say, *b* and *c*, and then only permutations in which *b* precedes *c*. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.