UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER

Brute Force - Closest-Pair and Convex-Hull Problems-Exhaustive Search – Traveling Salesman Problem - Knapsack Problem - Assignment problem. Divide and conquer methodology – Merge sort – Quick sort – heap sort– Binary search – Multiplication of Large Integers – Strassen's Matrix Multiplication-Closest-Pair and Convex-Hull Problems.

2.1 Brute Force

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

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. And often, the brute-force strategy is indeed the one that is easiest to apply. As an example, consider the exponentiation problem: compute a^n for a nonzero number a and a nonnegative integer n. Although this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design strategies, including the brute force .

Selection Sort and Bubble Sort

In this section, we consider the application of the brute-force approach to the problem of sorting: given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order. As we mentioned earlier, dozens of algorithms have been developed for solving this very important problem. The two algorithms discussed here—selection sort and bubble sort—seem to be the two prime candidates.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n

$-$ 1 elements and exchange it with the second element, putting the second smallest element in its final position.

```
ALGORITHM SelectionSort(A[0..n − 1])
    //Sorts a given array by selection sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ←0 to n − 2 do
        min←i
        for j ←i + 1 to n − 1 do
```

```
            if $A[j] < A[min]$ $min \leftarrow j$
        swap $A[i]$ and $A[min]$
```

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i $(0 \le i \le n - 2)$ of bubble sort can be represented by the following diagram.

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

```
    //Sorts a given array by bubble sort
    //Input: An array $A[0..n - 1]$ of orderable elements
    //Output: Array $A[0..n - 1]$ sorted in nondecreasing order
    for $i \leftarrow 0$ to $n - 2$ do
        for $j \leftarrow 0$ to $n - 2 - i$ do
            if $A[j + 1] < A[j]$ swap $A[j]$ and $A[j + 1]$
```

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size *n*; it is obtained by a sum that is almost identical to the sum for selection

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$
$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Sequential Search and Brute-Force String Matching

We saw in the previous section two applications of the brute-force approach to the sorting problem. Here we discuss two applications of this strategy to the problem of searching. The first deals with the canonical problem of searching for an item of a given value in a given list. The second is different in that it deals with the string-matching problem.

Sequential Search

We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search. To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudo code of this enhanced version.

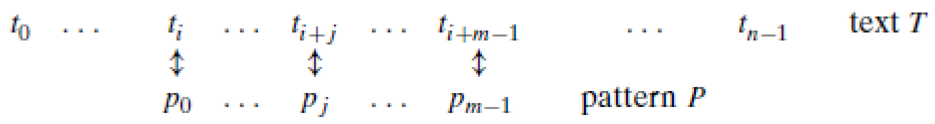
```
ALGORITHM SequentialSearch2(A[0..n], K)
    //Implements sequential search with a search key as a sentinel
    //Input: An array A of n elements and a search key K
    //Output: The index of the first element in A[0..n − 1] whose value is
    // equal to K or −1 if no such element is found
    A[n]←K
    i ←0
    while A[i] _= K do
        i ←i + 1
    if i < n return i
    else return −1
```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered. Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency). The efficiency results obtained for the standard version of sequential search change for the enhanced version only very slightly, so that the algorithm remains linear in both the worst and average cases.

Brute-Force String Matching

Recall the string-matching problem introduced in earlier section: given a string of n characters called the ***text*** and a string of m characters $(m \leq n)$ called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that

$t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$:


t_0 . . . t_i . . . t_{i+j} . . . t_{i+m-1} . . . t_{n-1} text T
p_0 . . . p_j . . . p_{m-1} pattern P

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string- matching problem is quite obvious: align the pattern against the first *m* characters of the text and start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

Note that the last position in the text that can still be a beginning of a matching substring is $n - m$(provided the text positions are indexed from 0 to $n - 1$). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
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 <mand P[j ]= T [i + j ] do
            ←j + 1
        if j = m return i
    return −1
```

Note that for the example, the algorithm shifts the pattern almost always after a single character comparison. The worst case is much worse: the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries

2.2 Closest-Pair and Convex-Hull Problems by Brute Force

In this section, we consider a straightforward approach to two well-known problems dealing with a finite set of points in the plane. These problems, aside from their theoretical interest, arise in two important applied areas: computational geometry and operations research.

2.2.1 Closest-Pair Problem

The closest-pair problem calls for finding 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. An air-traffic controller might be interested in two closest planes as the most probable collision candidates. A regional postal service manager might need a solution to the closestpair problem to find candidate post-office locations to be closed.

One of the important applications of the closest-pair problem is cluster analysis in statistics. Based on *n* data points, hierarchical cluster analysis seeks to organize them in a hierarchy of clusters based on some similarity metric.

For numerical data, this metric is usually the Euclidean distance; for text and other nonnumerical data, metrics such as the Hamming distance A bottom-up algorithm begins with each element as a separate cluster and merges them into successively larger clusters by combining the closest pair of clusters.

For simplicity, we consider the two-dimensional case of the closest-pair problem. We assume that the points in question are specified in a standard fashion by their *(x, y)* Cartesian coordinates and that the distance between two points *pi(xi, yi)* and *pj(xj, yj)* is the standard Euclidean distance

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

The brute-force approach to solving this problem leads to the following obvious algorithm:

- compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice.
- To avoid doing so, we consider only the pairs of points *(pi, pj)* for which *i < j*. Pseudocode below computes the distance between the two closest points; getting the closest points themselves requires just a trivial modification.

```
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
```

The basic operation of the algorithm is computing the square root. square roots are irrational numbers that therefore can be found only approximately. But, in fact, computing square roots in the loop can be avoided! 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:

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

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

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.

2.1.1 Convex-Hull Problem

On to the other problem—that of computing the convex hull. Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important. Several such applications are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given. They are also used for detecting outliers by some statistical techniques. An efficient algorithm for computing a diameter of a set of points, which is the largest distance between two of the points, needs the set's convex hull to find the largest distance between two of its extreme points (see below). Finally, convex hulls are important for solving many optimization problems, because their extreme points provide a limited set of solution candidates.

DEFINITION 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. A formal definition of the convex hull that is applicable to arbitrary sets, including sets of points that happen to lie on the same line, follows.

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

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*.)

The ***convex-hull problem*** is the problem of constructing the convex hull for a given set *S* of *n* points. 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 3.6 are *p*1, *p*5, *p*6, *p*7, and *p*3.

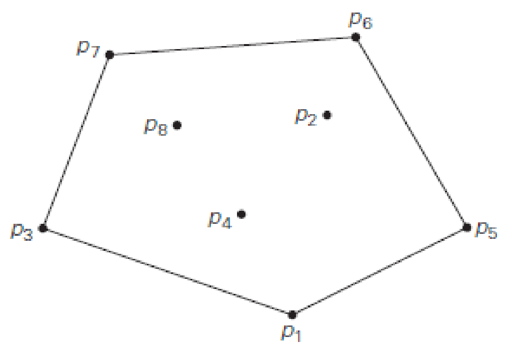


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, and p_3.

Here, however, we are interested in extreme points because their identification solves the convex-hull problem

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.

First, the straight line through two points $(x1, y1)$, $(x2, y2)$ in the coordinate plane can be defined by the equation $ax + by = c$, where $a = y2 - y1$, $b = x1 - x2$, $c = x1y2 - y1x2$.

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, $ax + by < c$.

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.

What is the time efficiency of this algorithm? It 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.

2.3 Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects

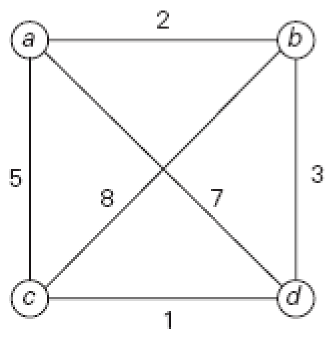
2.3.1 Traveling Salesman Problem

The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, 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.)

It is easy to see that a Hamiltonian circuit can also be defined as a sequence of *n* + 1 adjacent vertices *vi*0*, vi*1*, . . . , vin*−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. Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex Thus, 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.

Figure a small instance of the problem and its solution by this method. An inspection of Figure reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, *b* and *c,* and then consider only permutations in which *b* precedes *c*. This trick implicitly defines a tour's direction.

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

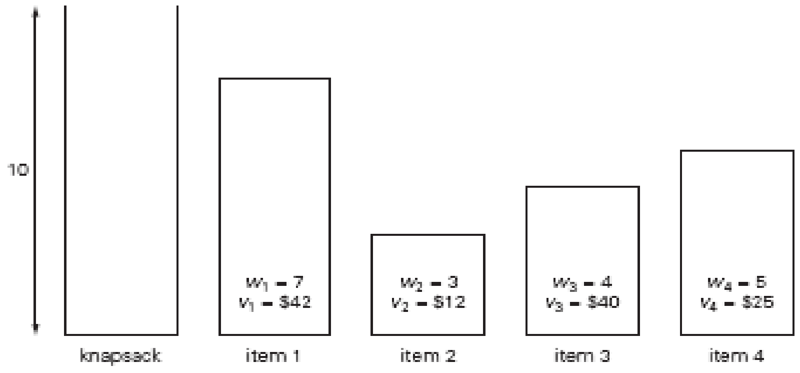


Tour	Length	
a ---> *b* ---> *c* ---> *d* ---> *a*	*l* = 2 + 8 + 1 + 7 = 18	
a ---> *b* ---> *d* ---> *c* ---> *a*	*l* = 2 + 3 + 1 + 5 = 11	optimal
a ---> *c* ---> *b* ---> *d* ---> *a*	*l* = 5 + 8 + 3 + 7 = 23	
a ---> *c* ---> *d* ---> *b* ---> *a*	*l* = 5 + 1 + 3 + 2 = 11	optimal
a ---> *d* ---> *b*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

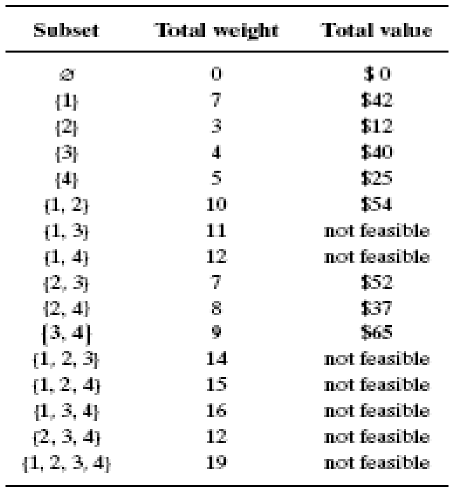
2.1.1 Knapsack Problem

Here is another well-known problem in algorithmics. Given *n* items of known weights *w*1, *w*2, . . . , *wn* and values *v*1, *v*2, . . . , *vn* and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack. 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. Since the number of subsets of an *n*-element set is 2*n*, the exhaustive search leads to a Ω*(*2*n)* algorithm, no matter how efficiently individual subsets are generated.

Thus, for both the traveling salesman and knapsack problems considered above, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so called ***NP-hard problems***.



(a)



Subset	Total weight	Total value
∅	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
{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)

2.3.3 Assignment Problem

In our third example of a problem that can be solved by exhaustive search, 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

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix *C*. In terms of this matrix, the problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible. In fact, the smallest element in the entire matrix need not be a component of an optimal solution.

We can describe feasible solutions to the assignment problem as *n*-tuples {*j*1, . . . , *jn*} in which the *i*th component, *i* = 1, . . . , *n*, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th person). For example, for the cost matrix above, {2, 3, 4, 1} indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. A few first iterations of applying this algorithm to the

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

<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.

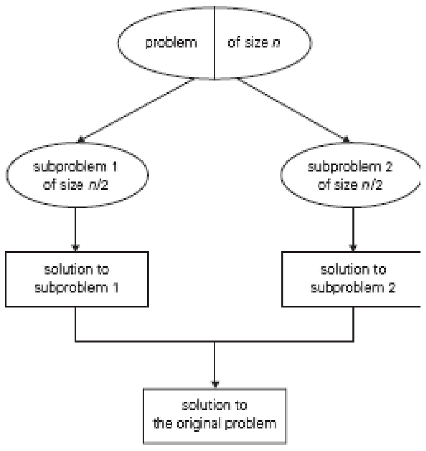
2.4 Divide-and-Conquer

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and- conquer algorithms work according to the following general plan:

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

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

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



Assuming that size *n* is a power of *b* to simplify our analysis, 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 accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and combining their solutions. (For the sum example above, *a* = *b* = 2 and *f (n)* = 1.) Recurrence (5.1) is called the ***general divide-and-conquer recurrence***. Obviously, the order of growth of its solution *T (n)* depends on the values of the constants *a* and *b* and the order of growth of the function *f (n)*.

2.4.1 Mergesort

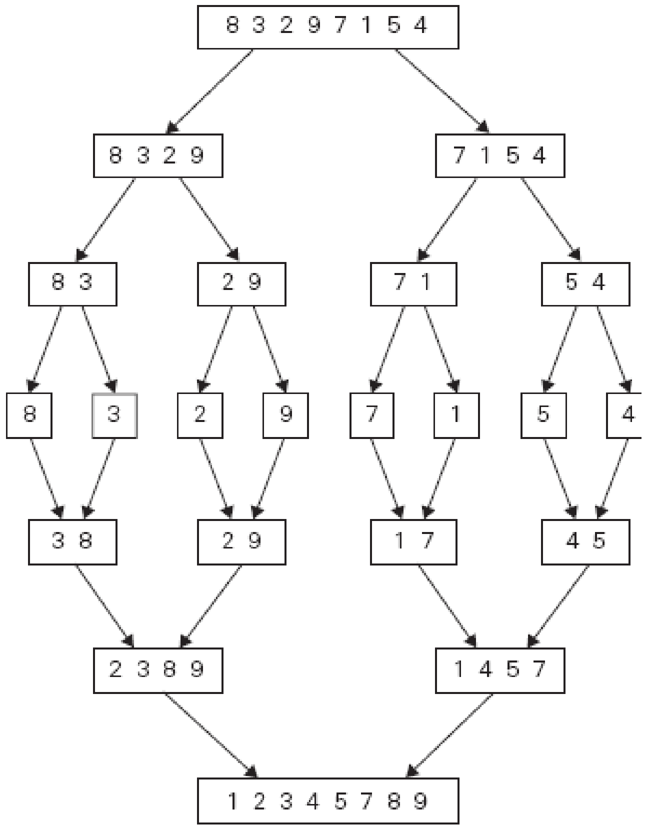
Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array *A*[0..*n* − 1] by dividing it into two halves *A*[0.._*n/*2_ − 1] and *A*[_*n/*2_..*n* − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a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

```
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
    A[0.._n/2_ − 1] to B[0.._n/2_ − 1]
    A[_n/2_..n − 1] to C[0.._n/2_ − 1]
    Mergesort(B[0.._n/2_ − 1])
    Mergesort(C[0.._n/2_ − 1])
    Merge(B, C, A) //see below
```

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to

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

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



```
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]
```

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

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

2.4.2 Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element *A*[*s*] are less than or equal to *A*[*s*], and all the elements to the right of *A*[*s*] are greater than or equal to it:

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

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

```
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]
```

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

If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

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

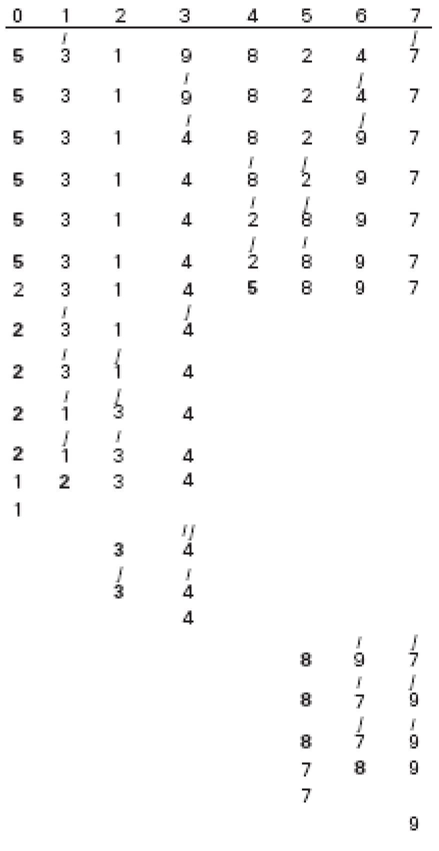
This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n - 2..n - 1]$ has been processed. The total number of key comparisons made will be equal to

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

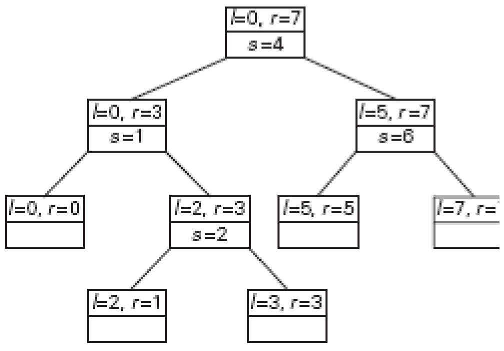
Thus, the question about the utility of quicksort comes down to its average case behavior. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

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

Thus, on the average, quicksort makes only 39% more comparisons than in the best case.



(a)



(b)

2.4.3 Binary Search

It is possible to take greater advantage of the ordered list if we are clever with our comparisons. In the sequential search, when we compare against the first item, there are at most $n-1$ more items to look through if the first item is not what we are looking for. Instead of searching the list in sequence, a **binary search** will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.

```
Algorithm binarySearch(A[ ],
item):
```

```
first = 0
last = len(A)-1
found = False

while first<=last and not found:
midpoint = (first + last)//2
if alist[midpoint] == item:
found = True
else:
if item < alist[midpoint]:
last = midpoint-1
else:
first = midpoint+1

return found
```

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, N, is odd then both sub-intervals will contain $(N-1)/2$ elements, while if N is even then the two sub-intervals contain $N/2-1$ and $N/2$ elements. The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is $O(\log n)$.

2.5 Multiplication of Large Integers

Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \,.\, 10^1 + 3 \,.\, 10^0 \text{ and } 14 = 1 \,.\, 10^1 + 4 \,.\, 10^0.$$

Now let us multiply them:

$$23 * 14 = (2 \,.\, 10^1 + 3 \,.\, 10^0) * (1 \,.\, 10^1 + 4 \,.\, 10^0)$$
$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$$

Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products 2 * 1 and 3 * 4 that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

For any pair of two-digit numbers $a = a1a0$ and $b = b1b0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

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

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique.We denote the first half of the a's digits by $a1$ and the second half by $a0$; for b, the notations are $b1$ and $b0$, respectively.

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

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

Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

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

Let $A(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides 3 $A(n/2)$ of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$$A(n) = 3A(n/2) + cn \text{ for } n > 1, A(1) = 1.$$

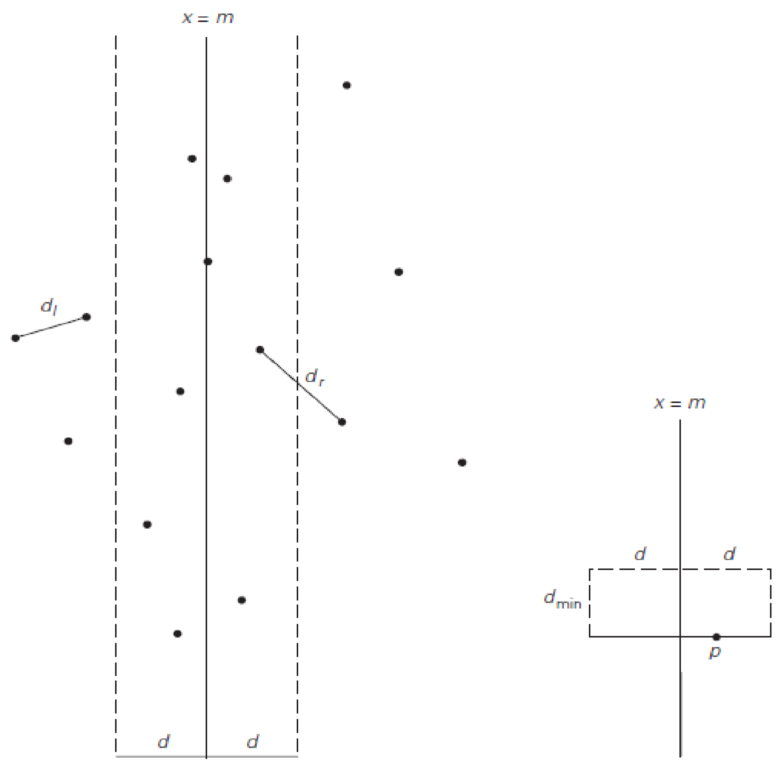
2.6 The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer

Closest-Pair

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we can divide the points into two subsets Pl and Pr of $(n/2)⌉$ and $(n/2)⌋$ points, respectively, by

- drawing a vertical line through the median m of their x coordinates so that $(n/2)⌉$ points lie to the

- left of or on the line itself, and (*n/*2) ⌋ points lie to the right of or on the line. Then we can solve
- the closest-pair problem recursively for subsets *Pl* and *Pr* . Let *dl* and *dr* be the smallest distances between pairs of points in *Pl* and *Pr* , respectively, and let *d* = min{*dl, dr* }.
- Note that *d* is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line.
- Obviously, we can limit our attention to the points inside the symmetric vertical strip of width 2*d* around the separating line, since the distance between any other pair of points is at least *d*
- Let *S* be the list of points inside the strip of width 2*d* around the separating line, obtained from *Q* and hence ordered in nondecreasing order of their *y* coordinate. We will scan this list, updating the information about *d*min*,* the minimum distance seen so far, if we encounter a closer pair of points. Initially, *d*min = *d,* and subsequently *d*min ≤ *d.*
- Let *p(x, y)* be a point on this list. For a point *p(x, y)* to have a chance to be closer to *p* than *d*min, the point must follow *p* on list *S* and the difference between their *y* coordinates must be less than *d*min Geometrically, this means that *p* must belong to the rectangle shown in Figure
- The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance *d* apart.



(a) (b)

```
ALGORITHM EfficientClosestPair(P, Q)
 if $n \le 3$
         return the minimal distance found by the brute-force algorithm
 else
         copy the first _n/2_ points of P to array Pl
         copy the same _n/2_ points from Q to array Ql
         copy the remaining _n/2_ points of P to array Pr
         copy the same _n/2_ points from Q to array Qr
         dl ←EfficientClosestPair(Pl, Ql)
         dr ←EfficientClosestPair(Pr, Qr)
         d ←min{dl, dr}
         m←P[_n/2_ − 1].x
         copy all the points of Q for which |x − m| < d into array S[0..num − 1]
         dminsq ←$d^2$
         for i ←0 to num − 2 do
```

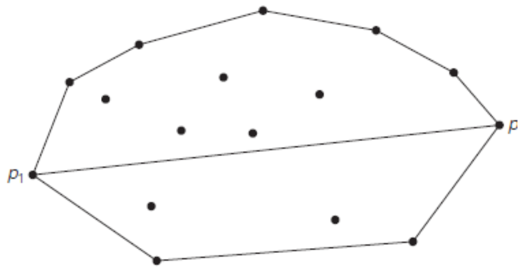
```
                k ← i + 1
                while k ≤ num − 1 and $(S[k].y − S[i].y)^2 < dminsq$
                        dminsq ← min($(S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2$, dminsq)
                        k ← k + 1
        return sqrt(dminsq)
```

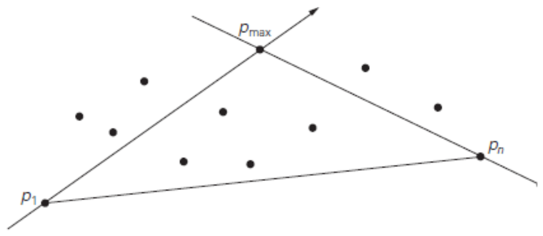
The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that *n* is a power of 2, we have the following recurrence for the running time of the algorithm:

$$T(n) = 2T(n/2) + f(n), \text{ where } f(n) \in \Theta(n).$$

Convex-Hull Problem

Find the smallest convex polygon that contains *n* given points in the plane. We consider here a divide-and-conquer algorithm called ***quickhull*** because of its resemblance to quicksort.
Let *S* be a set of*n*>1 points $p1(x1, y1), \ldots, pn(xn, yn)$ in the Cartesian plane. We assume that the points are sorted in nondecreasing order of their *x* coordinates, with ties resolved by increasing order of the *y* coordinates of the points involved. It is not difficult to prove the geometrically obvious fact that the leftmost point *p*1 and the rightmost point *pn* are two distinct extreme points of the set's
.





Let $\overrightarrow{p1pn}$ be the straight line through points *p*1 and *pn* directed from *p*1 to *pn*. This line separates the points of *S* into two sets: *S*1 is the set of points to the left of this line, and *S*2 is the set of points to the right of this line. The points of *S* on the line $\overrightarrow{p1pn}$, other than *p*1 and *pn*, cannot be extreme points of the convex hull and hence are excluded from further consideration.

The boundary of the convex hull of *S* is made up of two polygonal chains: an "upper" boundary and a "lower" boundary. The "upper" boundary, called the ***upper hull***, is a sequence of line segments with vertices at *p*1, some of the points in *S*1 (if *S*1 is not empty) and *pn.* The "lower" boundary, called the ***lower hull***, is a sequence of line segments with vertices at *p*1, some of the points in *S*2 (if *S*2 is not empty) and *pn.* The fact that the convex hull of the entire set *S* is composed of the upper and lower hulls, which can be constructed independently and in a similar fashion, is a very useful observation exploited by several algorithms for this problem.

For concreteness, let us discuss how quickhull proceeds to construct the upper hull; the lower hull can be constructed in the same manner. If *S*1 is empty, the upper hull is simply the line segment with the endpoints at *p*1 and *pn.* If *S*1 is not empty, the algorithm identifies point *p*max in *S*1, which is the farthest from the line $\overrightarrow{p1pn}$. If there is a tie, the point that maximizes the angle _ *p*max *p pn* can be selected. It is not difficult to prove the following: *p*max is a vertex of the upper hull. The points inside _*p*1*p*max*pn* cannot be vertices of the upper hull (and hence can be eliminated from further consideration).

There are no points to the left of both lines $\overrightarrow{p1p\text{max}}$ and $\overrightarrow{p\text{max}pn}$. Therefore, the algorithm can continue constructing the upper hulls of *p*1 *S*1,1 *p*max and *p*max *S*1,2 *pn* recursively and then simply concatenate them to get the upper hull of the entire set *p*1 *S*1 *pn.*

Heap Sort

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

Consider an array Arr which is to be sorted using Heap Sort.

- Initially build a max heap of elements in Arr.
- The root element, that is Arr[1], will contain maximum element of Arr. After that, swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Implementation:

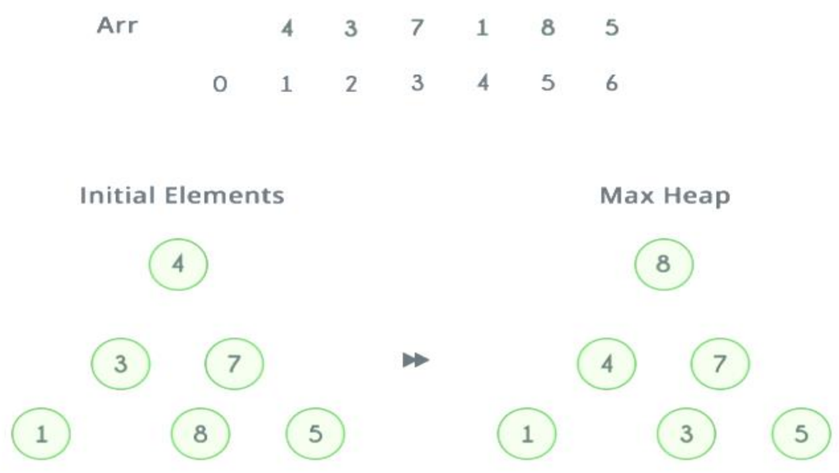
```
void heap_sort(int Arr[ ])

{
   int heap_size = N;

   build_maxheap(Arr);
   for(int i = N; i >= 2 ; i-- )
   {
      swap|(Arr[ 1 ], Arr[ i ]);
      heap_size = heap_size - 1;
      max_heapify(Arr, 1, heap_size);
   }
}
```

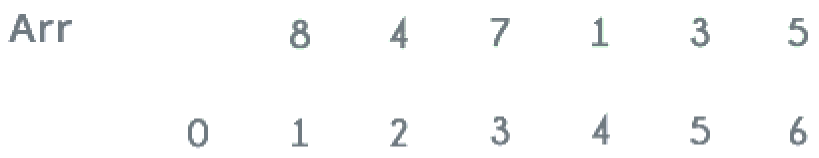
Complexity:
max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify N−1times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).

Example:
In the diagram below,initially there is an unsorted array Arr having 6 elements and then max-heap will be built.



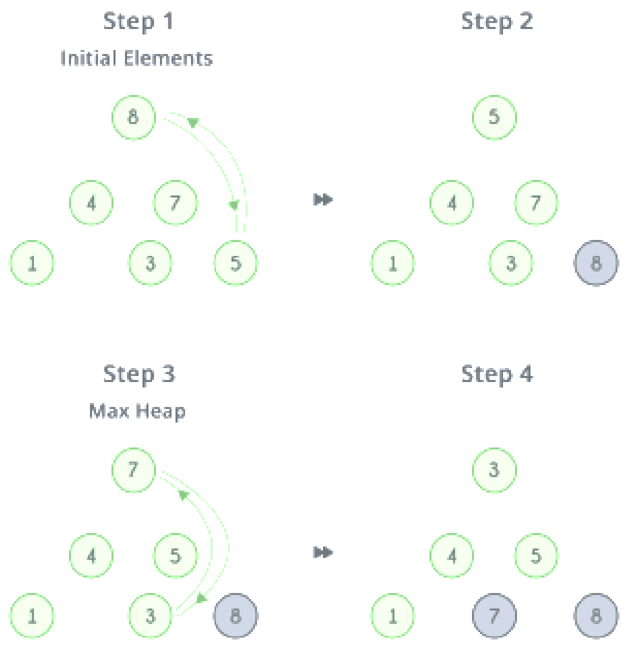
After building max-heap, the elements in the array Arr will be:

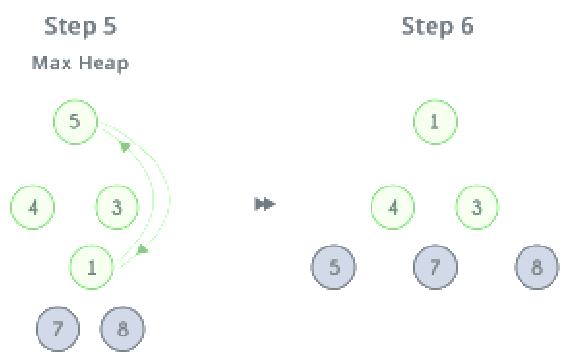
After building max-heap, the elements in the array Arr will be:

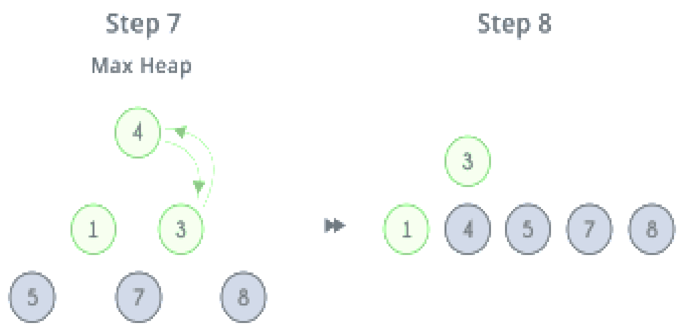


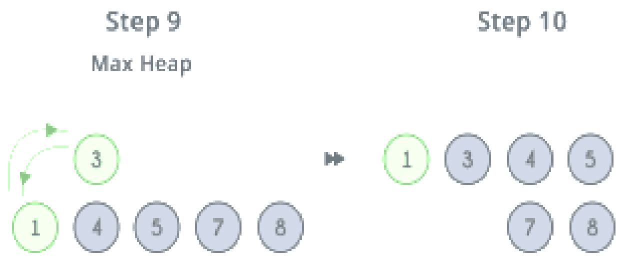
Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now and.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.









After all the steps, we will get a sorted array.

Arr		1	3	4	5	7	8
	0	1	2	3	4	5	6