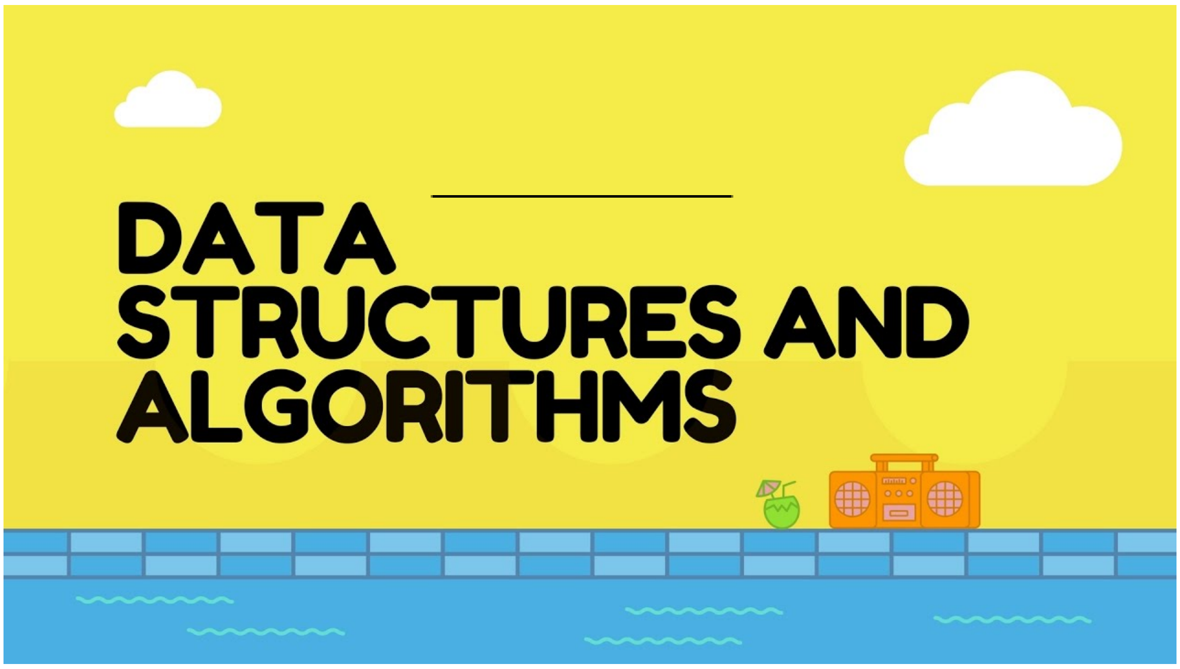


Summer of Science

Mid-Term Report

Data Structure & Algorithm

Bhavesh Khichi

Mentor: Vibhor Jain

Maths & Physics Club

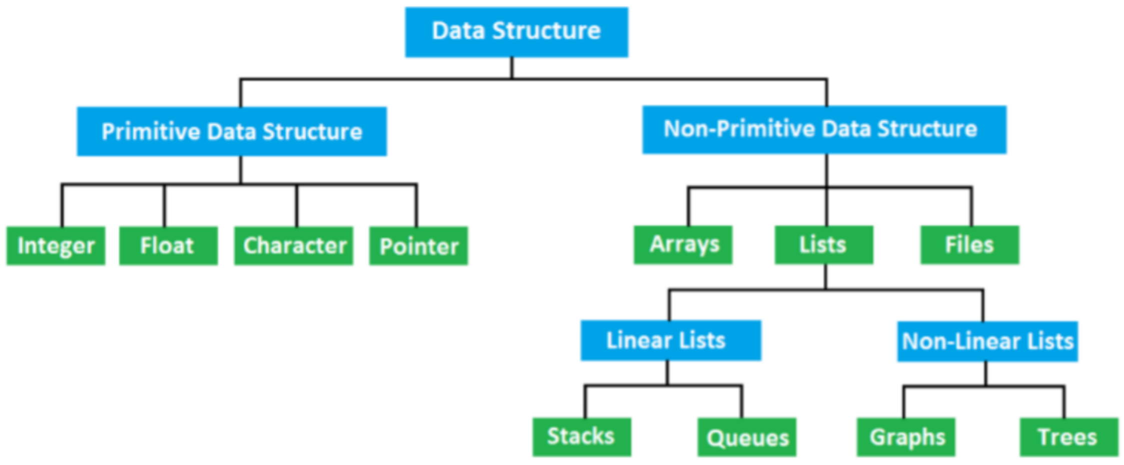
IIT Bombay

Project Specification

To study how the operations on data structure and algorithms are performed. And how the values are compared in a sorting algorithm and swapped. Total Number of comparison and exchanges performed in a sorting algorithm. And the corresponding code performed while sorting. .To get a clear idea about various data structures and operations on it. And how can we implement a data structure?

Chapter 1: **Foundations of DSA**

In computer science, a **Data Structure** is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.



An **Algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

1.1 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify the most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

- ❖ Before we can analyze an algorithm, we must have a model of the implementation technology that we will use,
- ❖ In the RAM model, instructions are executed one after another, with no concurrent operations. we should precisely define the instructions of the RAM model and their costs. Yet we must be careful not to abuse the RAM model.
- ❖ The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

1.2 Designing algorithms

We can choose from a wide range of algorithm design techniques. A given problem can be solved in various different approaches and some approaches deliver much more efficient results than others. Algorithm analysis is a technique used to measure the effectiveness and performance of the algorithms. It helps to determine the quality of an algorithm based on several parameters such as user-friendliness, maintainability, security, space usage and usage of other resources.

1. **Brute-force or exhaustive search** :- In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

2. **Divide and Conquer** :- In computer science, divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

3. **Greedy Algorithms :-** A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

4. **Dynamic Programming** :- DP is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

5. **Branch and Bound Algorithm** :- Branch and bound is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.

6. **Randomized Algorithm** :- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot.

7. **Backtracking** :- Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time,

removing those solutions that fail to satisfy the constraints of the problem at any point of time.

1.3 Growth of Functions

We need to analyse the performance of algorithms in terms of its running time or the memory required for its implementation.

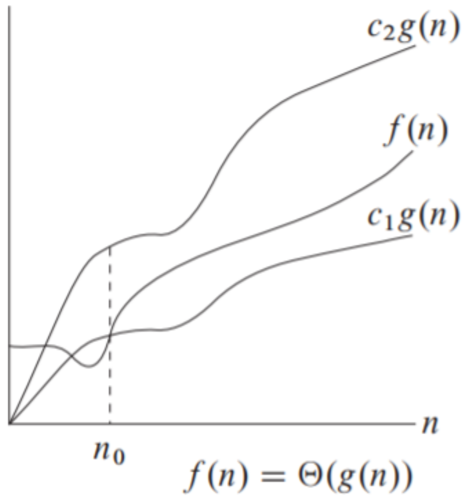
1.3.1 Asymptotic notation, functions, and running times

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N = {0, 1, 2,.....} Such notations are convenient for describing the worst-case running-time function T(n),

Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make a blanket statement that covers all inputs, not just the worst case. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

1.3.1.1 Θ-notation

A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between c_1g(n) and c_2g(n) for sufficiently large n. Because Θ(g(n)) is a set, we could write "f(n) ∈ Θ(g(n))" to indicate that f(n) is a member of Θ(g(n)). Instead, we will usually write "f(n) = Θ(g(n))".



Given an intuitive picture of functions f(n) and g(n), where f(n) = Θ(g(n)). For all values of n at and to the right of n_0, the value of f(n) lies at or above c_1g(n) and at or below c_2g(n). In other words, for all n ≥ n_0, the function f(n) is equal to g(n) to within a constant factor. We say that g(n) is an asymptotically tight bound for f(n).

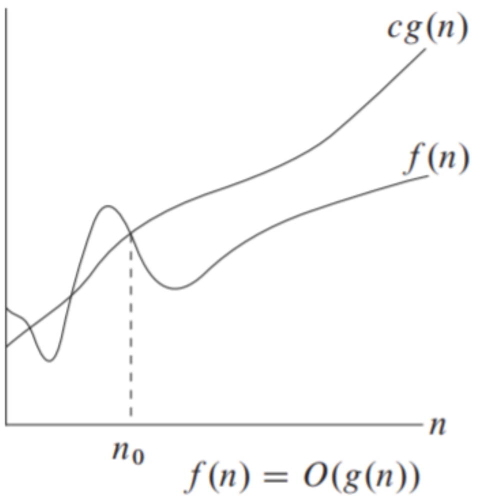
The definition of Θ(g(n)) requires that every member f(n) ∈ Θ(g(n)) be asymptotically nonnegative, that is, that f(n) be nonnegative whenever n is sufficiently large. Consequently, the function g(n) itself must be asymptotically nonnegative, or else the set Θ(g(n)) is empty.

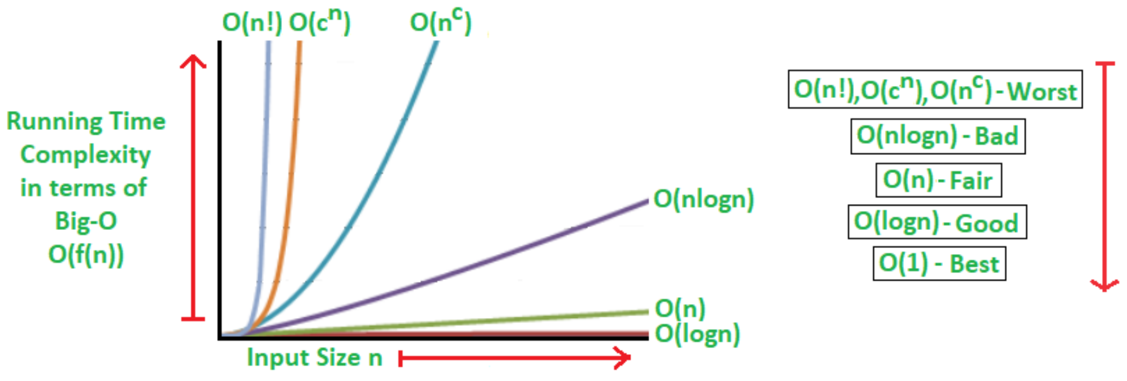
1.3.1.2 O-notation

When we have only an asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n))

O(g(n)) = {f(n) : there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$}

Since O-notation describes an upper bound, when we use it to bound the worst case running time of an algorithm. Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure.



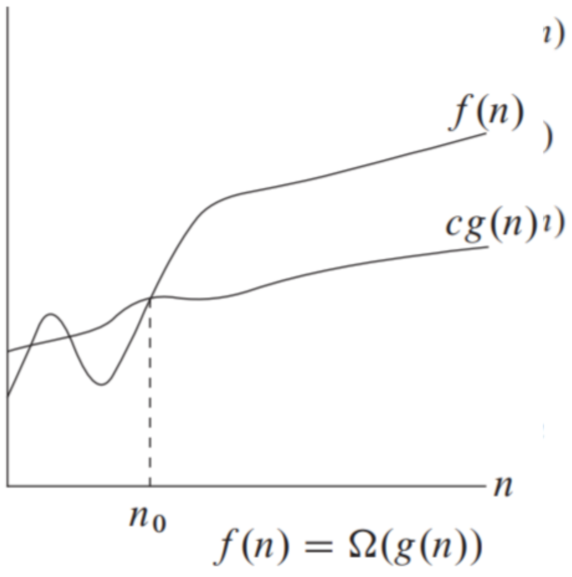


1.3.1.3 Ω-notation

Just as O-notation provides an asymptotic upper bound on a function, -notation provides an asymptotic lower bound. For a given function g(n), we denote by Ω(g(n)) the set of functions

Ω(g(n)) = {f(n) : there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}

Theorem :-
For any two functions f(n) and g(n), we have f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).



1.3.2 Asymptotic notation in equations and inequalities

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation. For example, the worst-case running time of merge sort as the recurrence

$$\mathbf{T(n) = 2T(n/2) + \Theta(n)}$$

If we are interested only in the asymptotic behavior of T(n), there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term Θ(n).

1.3.2.1 o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define o(g(n)) as the set

o(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n_0 \le n$}

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. The definitions of O-notation and o-notation are similar. The main difference is that in f(n) = O(g(n)), the bound $0 \le f(n) \le cg(n)$ holds for some constant c > 0, but in f(n) = o(g(n)), the bound $0 \le f(n) < cg(n)$ holds for all constants c > 0. Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

1.3.2.2 ω-notation

By analogy, ω-notation is similar to Ω-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

ω(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n_0 \le n$}

For example $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation f(n) = ω(g(n)) implies that

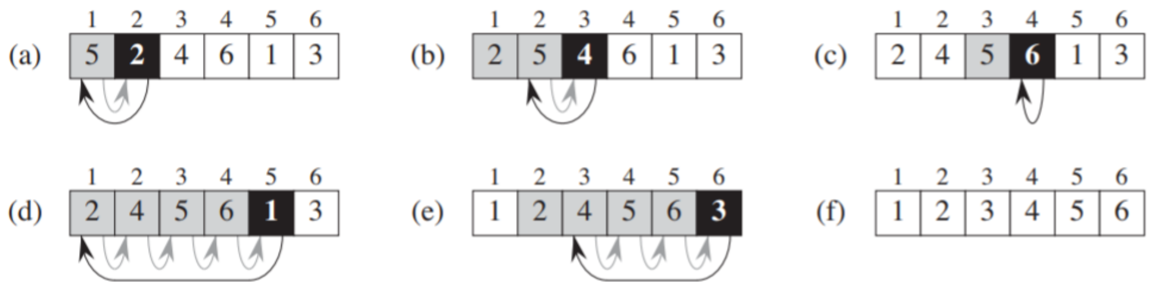
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Chapter 2: **Sorting and Order Statistics**

2.1 Sorting algorithms

2.1.1 Insertion Sort

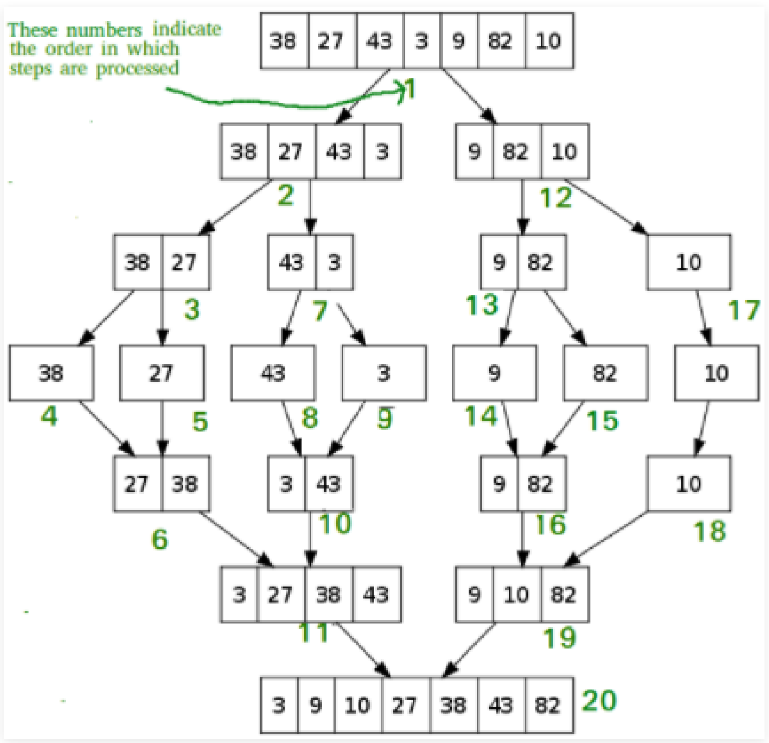
pseudocode for insertion sort as a procedure called INSERTIONSORT, which takes as a parameter an array A[1….n] containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by A.length.) The algorithm sorts the input numbers in place: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.



```
INSERTION-SORT(A)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

2.1.2 Merge Sort

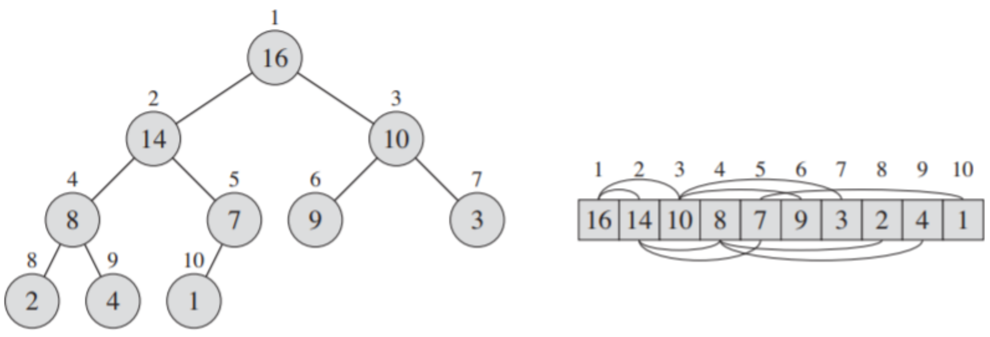
Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one. See the following C implementation for details.



```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

2.1.3 Heapsort

Heaps:-The (binary) heap data structure is an array object that we can view as a nearly complete binary tree, as shown in Figure Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: A.length, which gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A. That is, although A[1...A.length] may contain numbers, only the elements in A[1...A.heap-size], where 0 ≤ A.heap-size ≤ A.length, are valid elements of the heap. The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:



Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and the right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while the size of the heap is greater than 1.

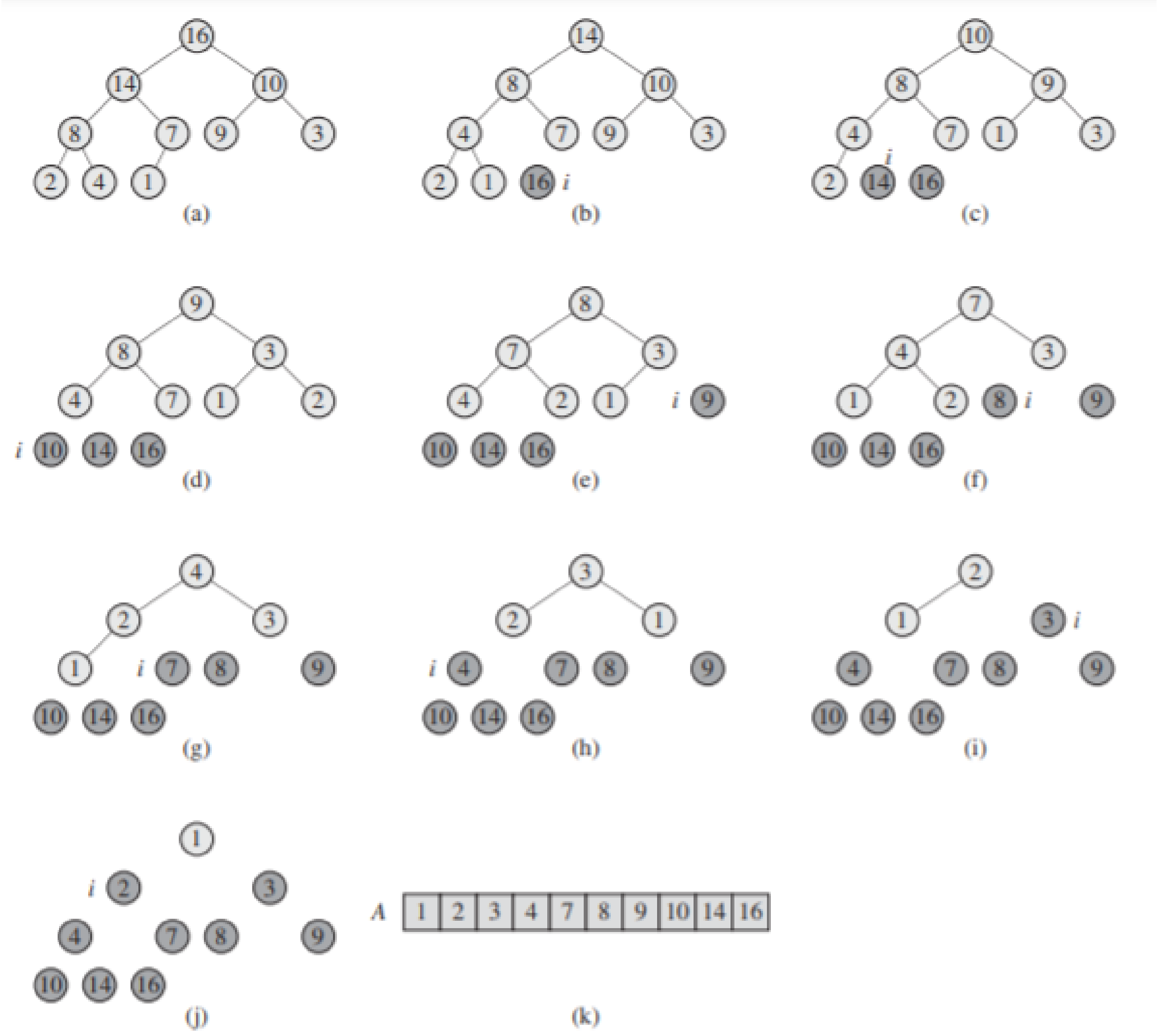
Building the heap:

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

The heapsort algorithm:

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

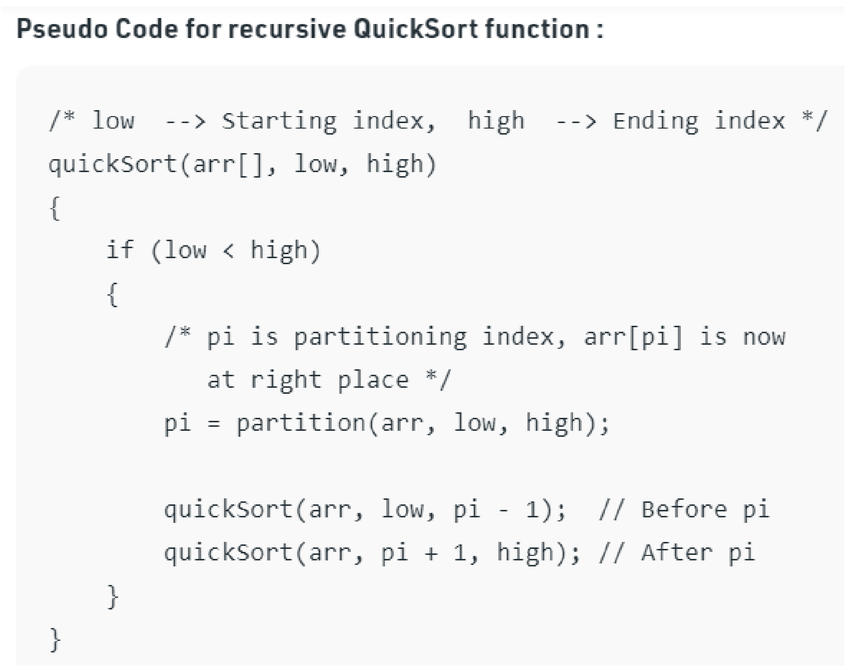


2.1.4 Quicksort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as pivot.
2. Always pick the last element as pivot.
3. Pick a random element as pivot
4. pick median as pivot

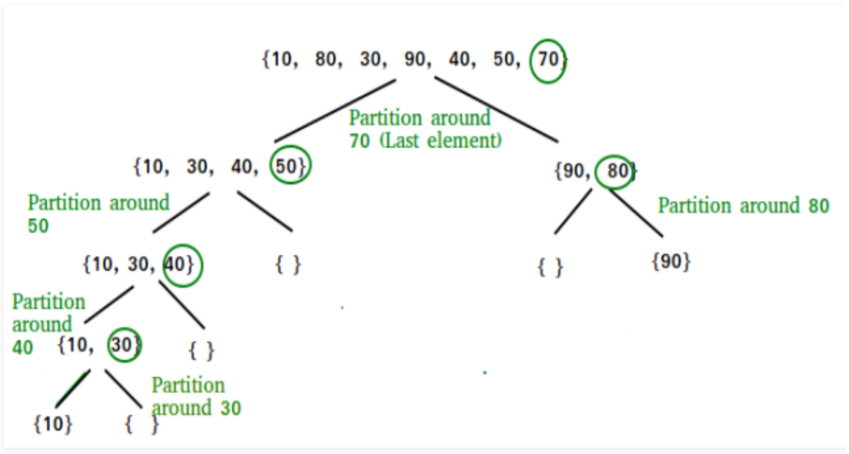
The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

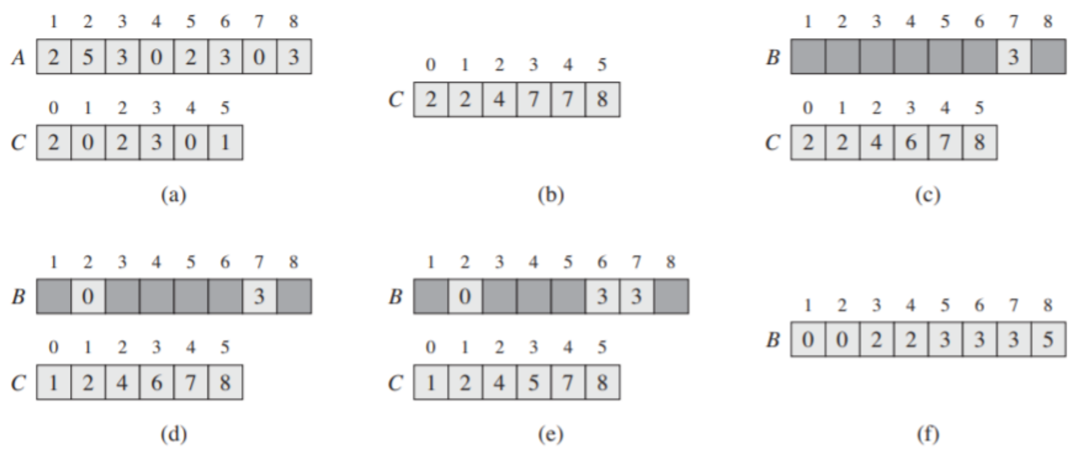


2.1.5 Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n), the sort runs in Θ(n) time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For example,

In the code for counting sort, we assume that the input is an array A[1….n], and thus A.length = n. We require two other arrays: the array B[1…..n] holds the sorted output, and the array C[0….k] provides temporary working storage.



```
COUNTING-SORT(A, B, k)
1   let $C[0..k]$ be a new array
2   for $i = 0$ to $k$
3       $C[i] = 0$
4   for $j = 1$ to $A.length$
5       $C[A[j]] = C[A[j]] + 1$
6   // $C[i]$ now contains the number of elements equal to $i$.
7   for $i = 1$ to $k$
8       $C[i] = C[i] + C[i-1]$
9   // $C[i]$ now contains the number of elements less than or equal to $i$.
10  for $j = A.length$ downto 1
11      $B[C[A[j]]] = A[j]$
12      $C[A[j]] = C[A[j]] - 1$
```

2.1.6 Radix Sort

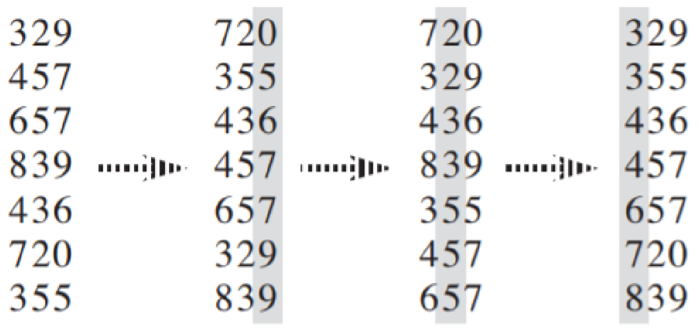
The Radix Sort Algorithm

Do the following for each digit i where i varies from least significant digit to the most significant digit.

➔ Sort input array using counting sort (or any stable sort) according to the i'th digit.

```
RADIX-SORT(A, d)
1   for $i = 1$ to $d$
2       use a stable sort to sort array $A$ on digit $i$
```

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.



The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

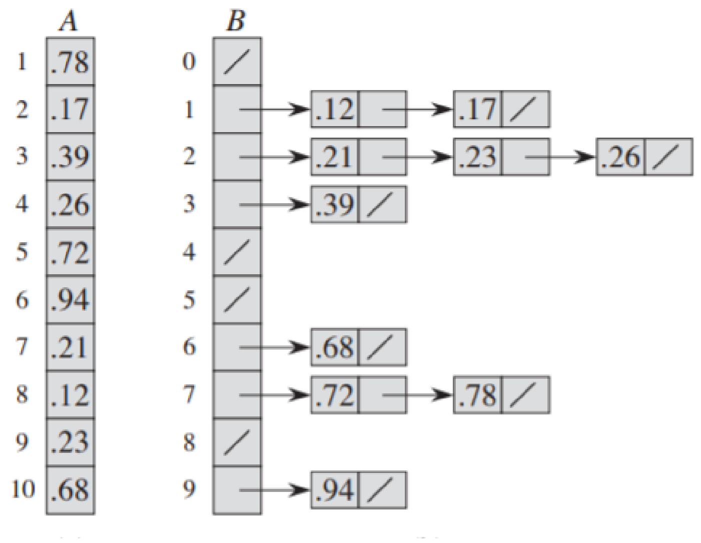
2.1.7 Bucket Sort

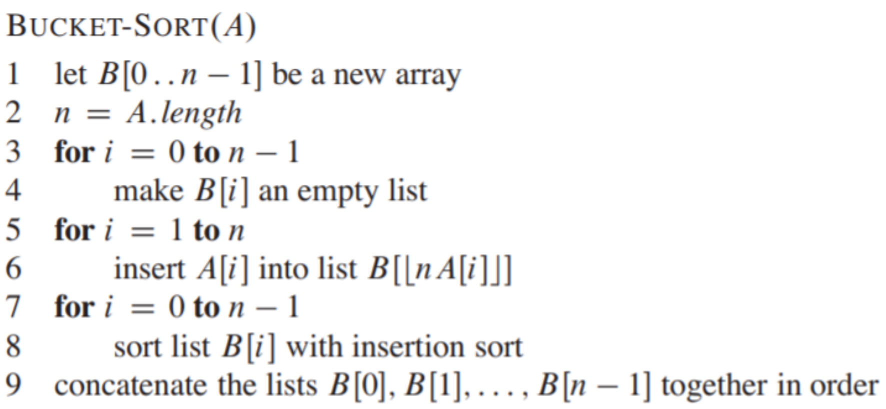
Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithms (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn. Counting sort can not be applied here as we use keys as indexes in counting sort. Here keys are floating point numbers.

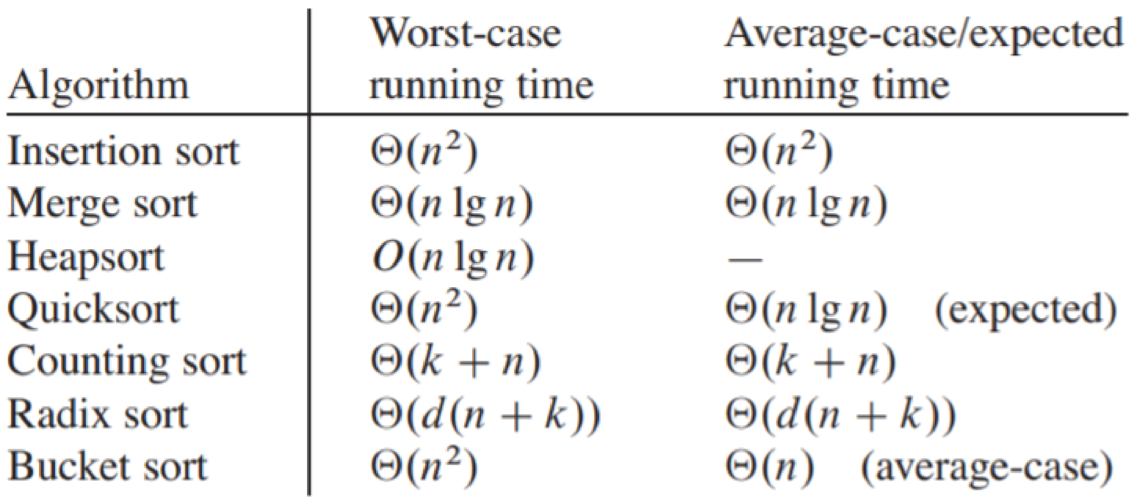
The idea is to use bucket sort. Following is the bucket algorithm.

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```





```
BUCKET-SORT(A)
1  let $B[0..n-1]$ be a new array
2  $n = A.length$
3  for $i = 0$ to $n-1$
4      make $B[i]$ an empty list
5  for $i = 1$ to $n$
6      insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7  for $i = 0$ to $n-1$
8      sort list $B[i]$ with insertion sort
9  concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
```



Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Chapter 3: **Data Structure**

3.1 Elementary Data Structures

3.1.1 Stacks and Queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

Stacks:-

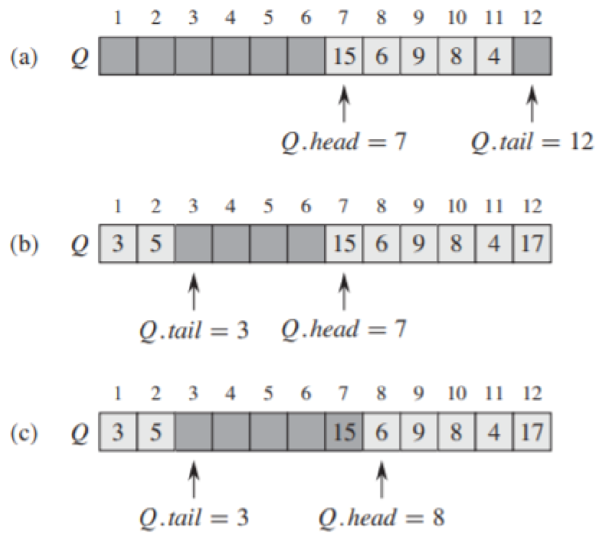
The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, we can implement a stack of at most n elements with an array S[1….n]. The array has an attribute S.top that indexes the most recent.

When S.top = 0, the stack is empty. We can test to see whether the stack is empty by query operation STACK-EMPTY. If we attempt to pop an empty stack, we say the stack underflows, which is normally an error. If S.top exceeds n, the stack overflows.

```
STACK-EMPTY(S)
1  if $S.top == 0$
2      return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  $S.top = S.top + 1$
2  $S[S.top] = x$
```

```
POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else $S.top = S.top - 1$
4      return $S[S.top + 1]$
```



Queues:-

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE, like the stack operation POP, DEQUEUE takes no element argument. The elements in the queue reside in locations Q.head, Q.head + 1…...Q.tail - 1, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order.

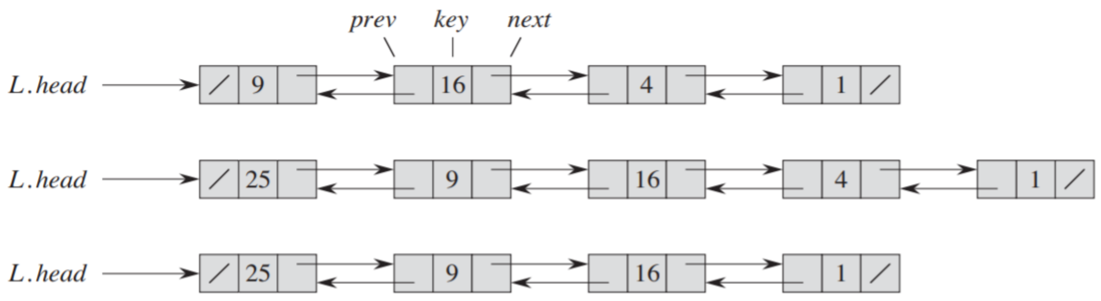
```
ENQUEUE(Q, x)
1  $Q[Q.tail] = x$
2  if $Q.tail == Q.length$
3      $Q.tail = 1$
4  else $Q.tail = Q.tail + 1$
```

```
DEQUEUE(Q)
1  $x = Q[Q.head]$
2  if $Q.head == Q.length$
3      $Q.head = 1$
4  else $Q.head = Q.head + 1$
5  return $x$
```

3.1.2 Linked lists

A **linked list** is a data structure in which the objects are arranged in a linear order, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

A **doubly linked list** L is an object with an attribute key and two other pointer attributes: next and prev. The object may also contain other satellite data. Given an element x in the list, x.next points to its successor in the linked list, and x.prev points to its predecessor. If x.prev = NIL, the element x has no predecessor and is therefore the first element, or head, of the list. If x.next = NIL, the element x has no successor and is therefore the last element, or tail, of the list. An attribute L.head points to the first element of the list. If L.head = NIL, the list is empty.



➔ **Searching a linked list**

```
LIST-SEARCH(L, k)
1  $x = L.head$
2  while $x \neq$ NIL and $x.key \neq k$
3      $x = x.next$
4  return $x$
```

➔ **Inserting into a linked list**

```
LIST-INSERT(L, x)
1  $x.next = L.head$
2  if $L.head \neq$ NIL
3      $L.head.prev = x$
4  $L.head = x$
5  $x.prev =$ NIL
```

➔ **Deleting from a linked list**

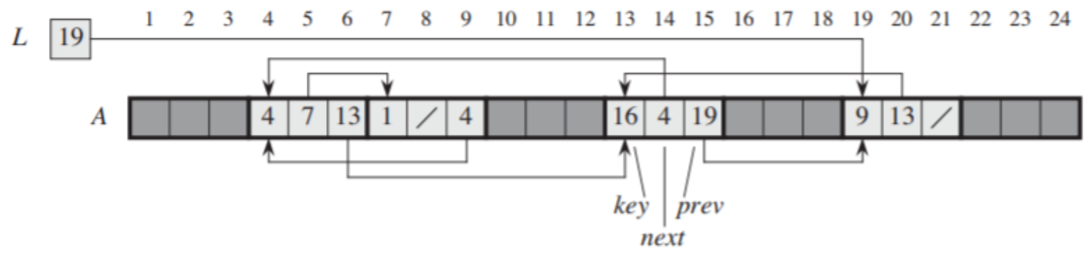
```
LIST-DELETE($L, x$)
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

3.1.3 Implementing pointers and objects

➔ *A multiple-array representation of objects :-*

The words in a computer memory are typically addressed by integers from 0 to M 1, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and we can address other memory locations within the object by adding an offset to the pointer.

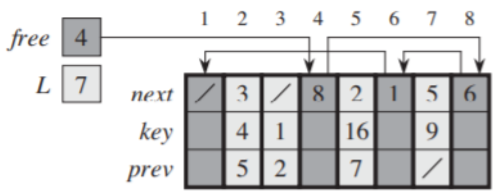
The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array. The problem of managing such a heterogeneous collection of objects is more difficult than the problem of managing a homogeneous collection, where all objects have the same attributes. Since most of the data structures we shall consider are composed of homogeneous elements, it will be sufficient for our purposes to use the multiple-array representation of objects.



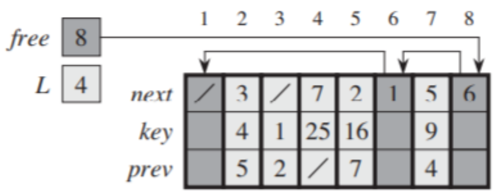
➔ *Allocating and freeing objects :-*

To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a garbage collector is responsible for determining which objects are unused. Many applications, however, are simple enough that they can bear responsibility for returning an unused object to a storage manager. We shall now explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.

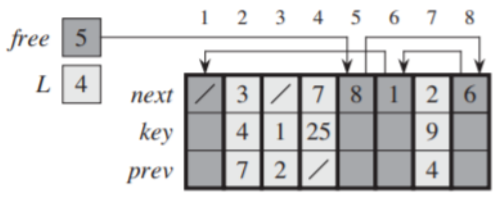
The free list acts like a stack: the next object allocated is the last one freed. We can use a list implementation of the stack operations PUSH and POP to implement the procedures for allocating and freeing objects, respectively. We assume that the global variable free used in the following procedures points to the first element of the free list.



(a)



(b)

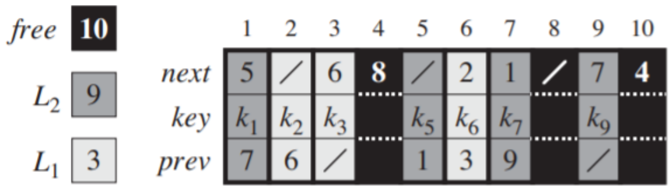


(c)

```
ALLOCATE-OBJECT()
1  if free == NIL
2      error "out of space"
3  else x = free
4      free = x.next
5      return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

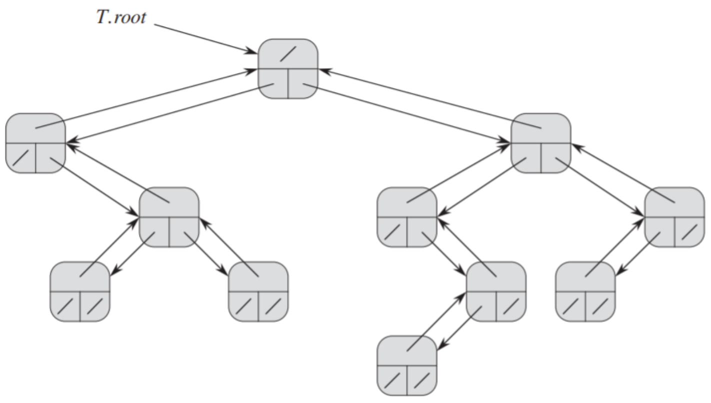
The free list initially contains all n unallocated objects. Once the free list has been exhausted, running the ALLOCATE-OBJECT procedure signals an error. We can even service several linked lists with just a single free list.



3.1.4 Representing rooted trees

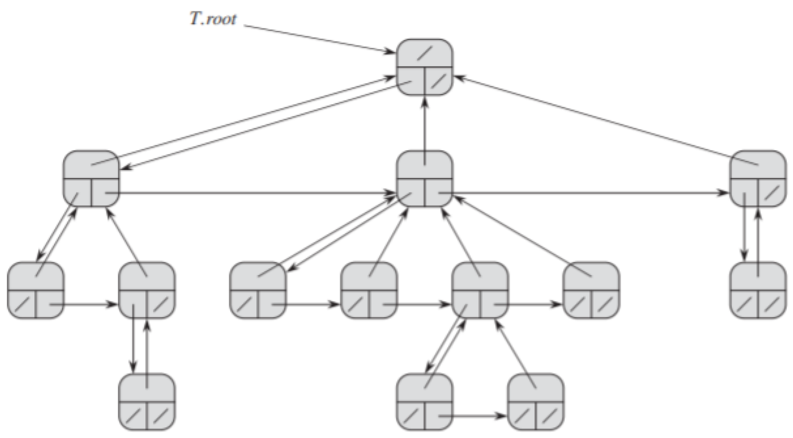
➔ Binary trees

Figure shows how we use the attributes p, left, and right to store pointers to the parent, left child, and right child of each node in a binary tree T . If x.p = NIL, then x is the root. If node x has no left child, then x.left = NIL, and similarly for the right child. The root of the entire tree T is pointed to by the attribute T.root. If T.root = NIL, then the tree is empty.



➔ Rooted trees with unbounded branching

The scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: we replace the left and right attributes by $child_1$, $child_2$, ..., $child_k$. Fortunately, there is a clever scheme to represent trees with arbitrary numbers of children. It has the advantage of using only O(n) space for any n-node rooted tree. The left-child, right-sibling representation appears in Figure



3.2 Hash Tables

In computing, a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

3.2.1 Direct-address tables

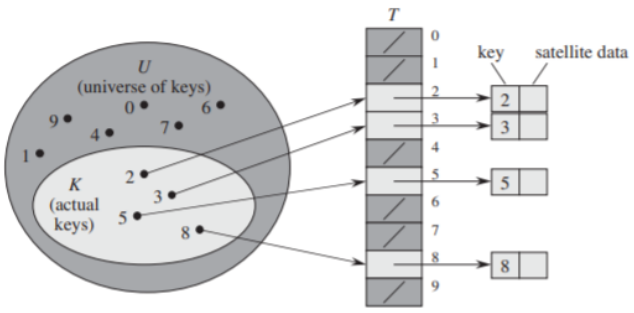
Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = [0, 1…m-1], where m is not too large. To represent the dynamic set, we use an array, or direct-address table, denoted by T = [0...m-1], in which each position, or slot, corresponds to a key in the universe U.

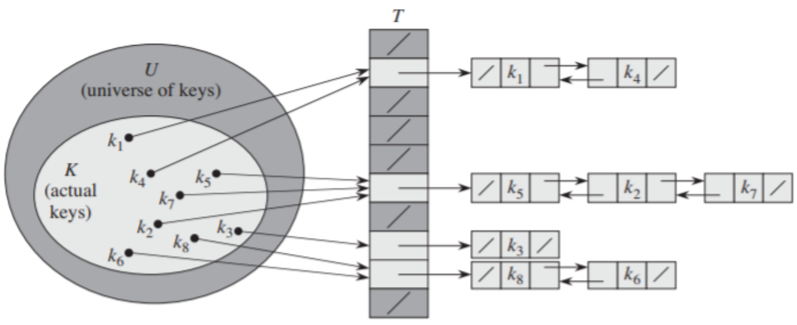
```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Each of these operations takes only $O(1)$ time.



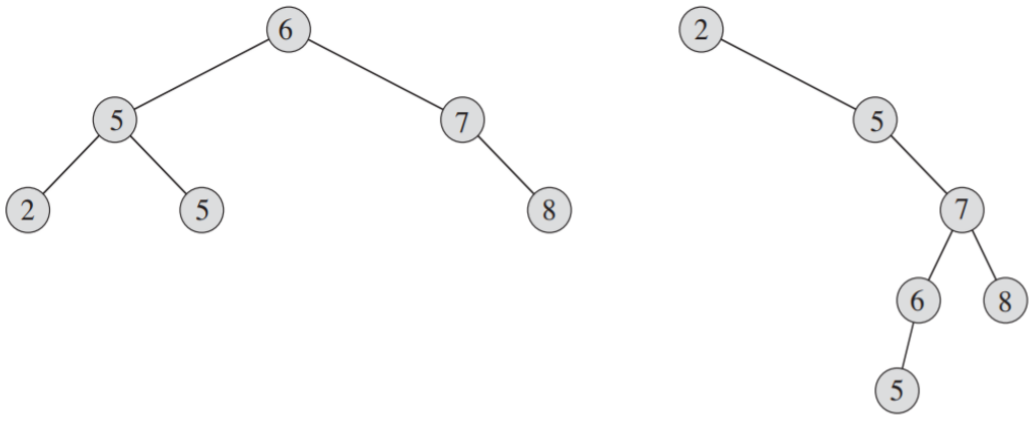


3.3 Binary Search Trees

A binary search tree is organized in a binary tree, as shown in Figure. We can represent such a tree by a linked data structure in which each node is an object. In addition to a key and satellite data, each node contains attributes left, right, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively.

If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL. The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key.



The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.

To use the following procedure to print all the elements in a binary search tree T , we call INORDER-TREE-WALK(T.root).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

3.3.1 Querying a binary search tree

We often need to search for a key stored in a binary search tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show how to support each one in time O(h) on any binary search tree of height h.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH$(x, k)$
1  while $x \neq$ NIL and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

```
TREE-MINIMUM$(x)$
1  while $x.left \neq$ NIL
2      $x = x.left$
3  return $x$
```

```
TREE-MAXIMUM$(x)$
1  while $x.right \neq$ NIL
2      $x = x.right$
3  return $x$
```

```
TREE-SUCCESSOR$(x)$
1  if $x.right \neq$ NIL
2      return TREE-MINIMUM$(x.right)$
3  $y = x.p$
4  while $y \neq$ NIL and $x == y.right$
5      $x = y$
6      $y = y.p$
7  return $y$
```

3.3.2 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

```
TRANSPLANT$(T, u, v)$
1  if $u.p ==$ NIL
2      $T.root = v$
3  elseif $u == u.p.left$
4      $u.p.left = v$
5  else $u.p.right = v$
6  if $v \neq$ NIL
7      $v.p = u.p$
```

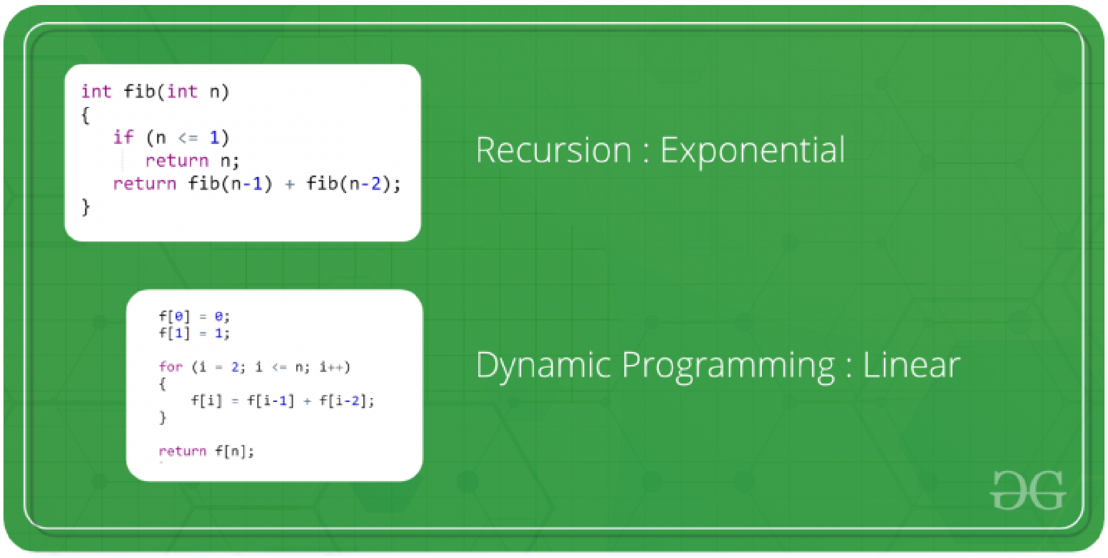
```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z        // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Chapter 4: **Design and Analysis Techniques**

4.1 Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solutions for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

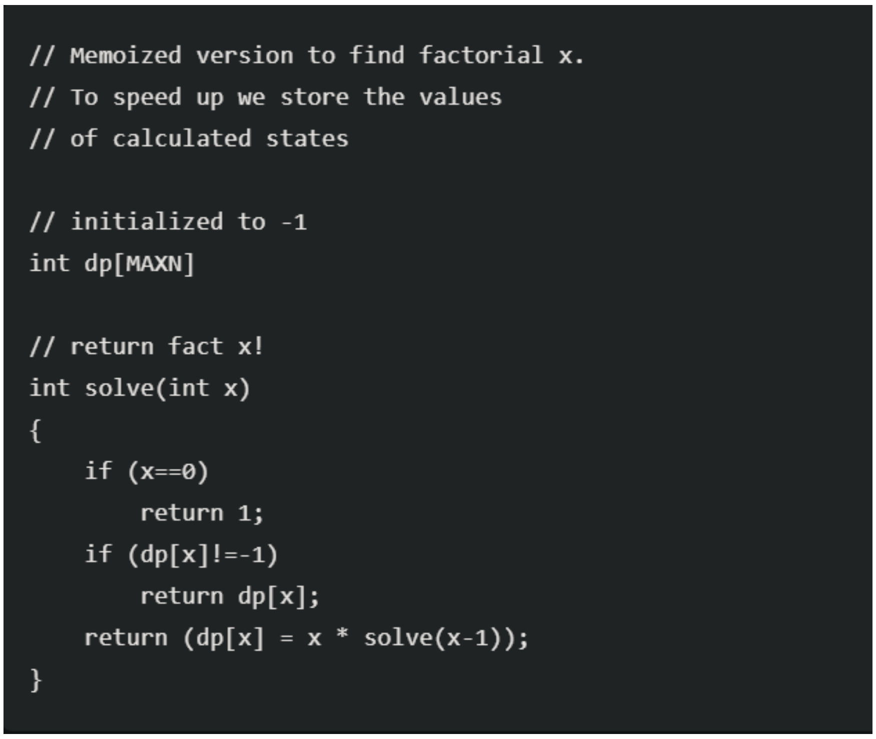
4.1.1 Tabulation vs Memoization

1. **Tabulation Method:-** Bottom Up Dynamic Programming, starting from the bottom and culminating answers to the top

```
// Tabulated version to find factorial x.
int dp[MAXN];

// base case
int dp[0] = 1;
for (int i = 1; i< =n; i++)
{
    dp[i] = dp[i-1] * i;
}
```

2. **Tabulation Method:-** Top Down Dynamic Programming, Once again let's describe it in terms of state transition. If we need to find the value for some state say dp[n] and instead of starting from the base state that i.e dp[0] we ask our answer from the states that can reach the destination state dp[n] following the state transition relation, then it is the top-down fashion of DP.



```
// Memoized version to find factorial x.
// To speed up we store the values
// of calculated states

// initialized to -1
int dp[MAXN]

// return fact x!
int solve(int x)
{
    if (x==0)
        return 1;
    if (dp[x]!=-1)
        return dp[x];
    return (dp[x] = x * solve(x-1));
}
```

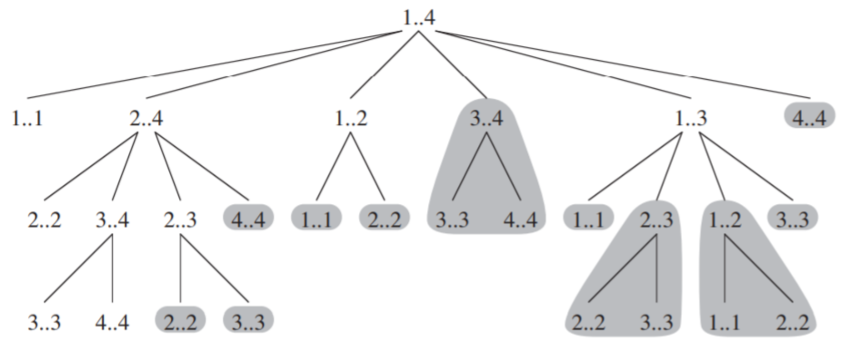
4.1.2 Optimal Substructure Property

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply. In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

$$\begin{tikzcd}
q \arrow[r, shift left] \arrow[d, shift right] & r \arrow[l, shift left] \arrow[d, shift right] \\
s \arrow[u, shift right] \arrow[r, shift left] & t \arrow[l, shift left] \arrow[u, shift right]
\end{tikzcd}$$

4.1.3 Optimal Subproblems Property

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.



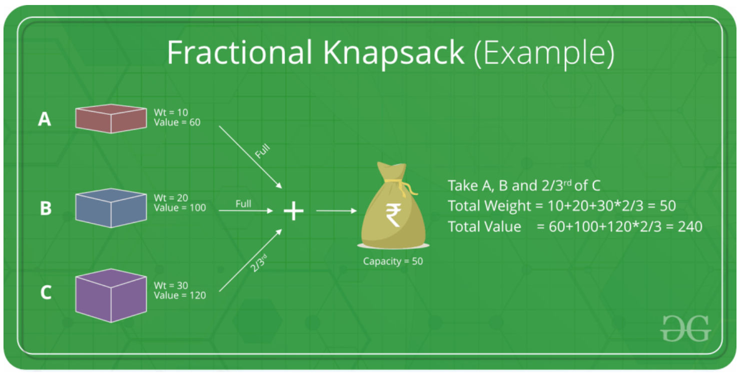
```
RECURSIVE-MATRIX-CHAIN$(p, i, j)$
if $i == j$
    return 0
$m[i, j] = \infty$
for $k = i$ to $j - 1$
    $q =$ RECURSIVE-MATRIX-CHAIN$(p, i, k)$
        $+$ RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
        $+ p_{i-1} p_k p_j$
    if $q < m[i, j]$
        $m[i, j] = q$
return $m[i, j]$
```
,

4.2 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solutions are best fit for Greedy.

For example consider the Fractional Knapsack Problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to a global optimal solution because we are allowed to take fractions of an item.



4.2.1 An activity-selection problem

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied.The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems.

Following are some standard algorithms that are Greedy algorithms.

1. **Kruskal's Minimum Spanning Tree (MST):** In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.
2. **Prim's Minimum Spanning Tree:** In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.
3. **Dijkstra's Shortest Path:** Dijkstra's algorithm is very similar to Prim's algorithm. The shortest-path tree is built up, edge by edge. We maintain two sets: a set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.
4. **Huffman Coding:** Huffman Coding is a loss-less compression technique. It assigns variable-length bit codes to different characters. The Greedy Choice is to assign the least bit length code to the most frequent character.

4.2.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes the choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. To develop a greedy algorithm was a bit more involved than is typical. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

➔ **The greedy-choice property:** we can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

➔ **A problem exhibits optimal substructure:** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

➔ **Cases of failure:** For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the traveling salesman problem mentioned above: for each number of cities, there is an assignment of distances between the cities for which the nearest-neighbor heuristic produces the unique worst possible tour.

➔ **Submodular functions:** A function *f* defined on subsets of a set Ω is called submodular if for every S,T ⊆ Ω we have that

$$f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$$

References

1. Data Structures and Algorithms
2. Introduction_to_algorithms-3rd Edition
3. GeeksforGeeks - DSA
4. wikipedia