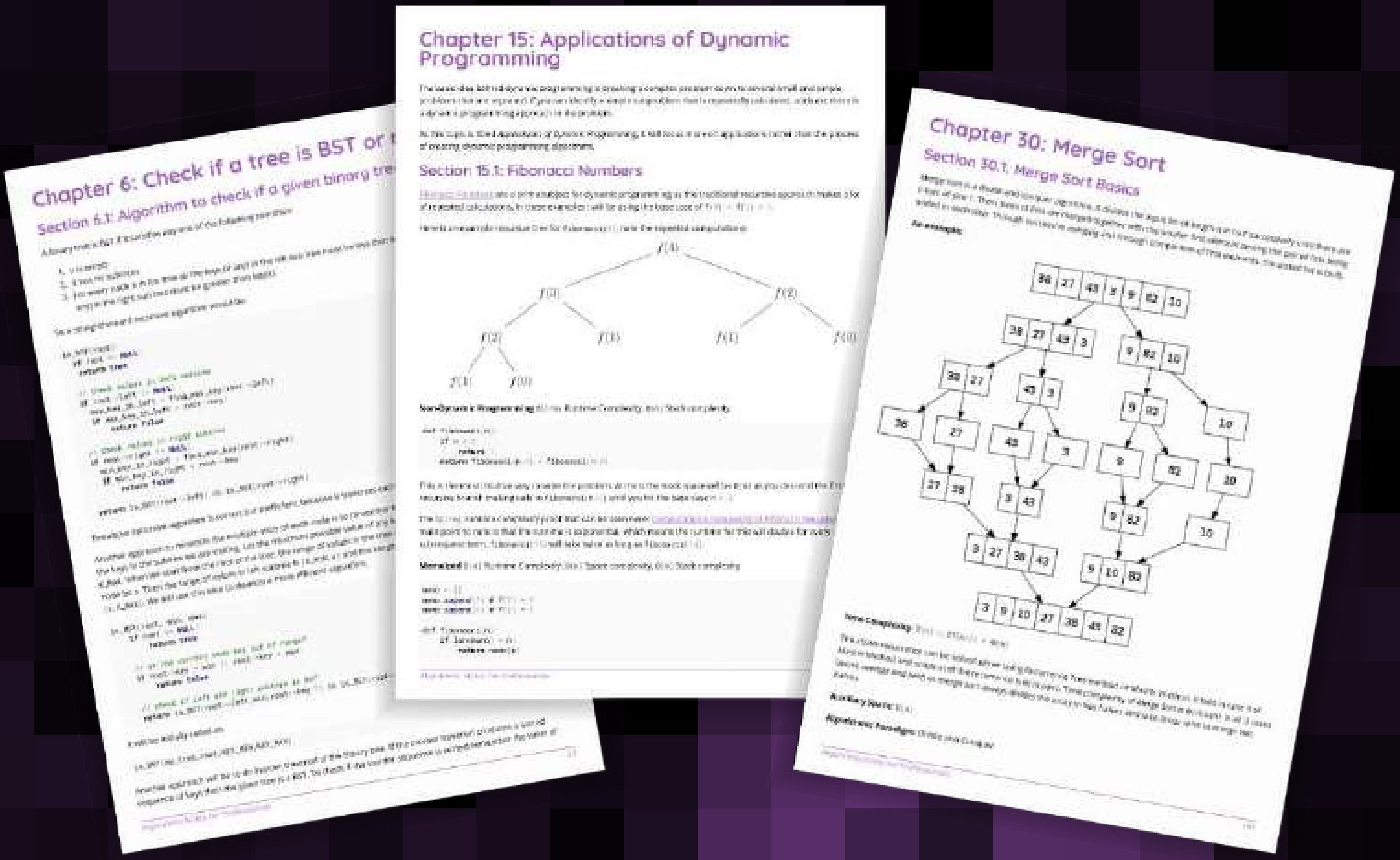
Algorithms

Notes for Professionals



Chapter 6: Check if a tree is BST or

Section 6.1: Algorithm to check if a given binary tre

Chapter 15: Applications of Dynamic Programming

Section 15.1: Fibonacci Numbers

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

200+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Algorithms group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

Chapter 28: Sorting

Parameter	Description
Stability	A sorting algorithm is **stable** if it preserves the relative order of equal elements after sorting.
In place	A sorting algorithm is **in-place** if it sorts using only `O(1)` auxiliary memory (not counting the array that needs to be sorted).
Best case complexity	A sorting algorithm has a best case time complexity of `O(T(n))` if its running time is **at least** `T(n)` for all possible inputs.
Average case complexity	A sorting algorithm has an average case time complexity of `O(T(n))` if its running time, **averaged over all possible inputs**, is `T(n)`.
Worst case complexity	A sorting algorithm has a worst case time complexity of `O(T(n))` if its running time is **at most** `T(n)`.

Section 28.1: Stability in Sorting

Stability in sorting means whether a sort algorithm maintains the relative order of the equals keys of the original input in the result output.

So a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Consider a list of pairs:

```
(1, 2) (9, 7) (3, 4) (8, 6) (9, 3)
```

Now we will sort the list using the first element of each pair.

A **stable sorting** of this list will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)
```

Because `(9, 3)` appears after `(9, 7)` in the original list as well.

An **unstable sorting** will output the below list:

```
(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)
```

Unstable sort may generate the same output as the stable sort but not always.

Well-known stable sorts:

- Merge sort
- Insertion sort
- Radix sort
- Tim sort
- Bubble Sort

Well-known unstable sorts:

- Heap sort
- Quick sort

Chapter 29: Bubble Sort

Parameter	Description
Stable	Yes
In place	Yes
Best case complexity	O(n)
Average case complexity	O(n^2)
Worst case complexity	O(n^2)
Space complexity	O(1)

Section 29.1: Bubble Sort

The `BubbleSort` compares each successive pair of elements in an unordered list and inverts the elements if they are not in order.

The following example illustrates the bubble sort on the list `{6,5,3,1,8,7,2,4}` (pairs that were compared in each step are encapsulated in '**'):

```
{6,5,3,1,8,7,2,4}
{**5,6**,3,1,8,7,2,4} -- 5 < 6 -> swap
{5,**3,6**,1,8,7,2,4} -- 3 < 6 -> swap
{5,3,**1,6**,8,7,2,4} -- 1 < 6 -> swap
{5,3,1,**6,8**,7,2,4} -- 8 > 6 -> no swap
{5,3,1,6,**7,8**,2,4} -- 7 < 8 -> swap
{5,3,1,6,7,**2,8**,4} -- 2 < 8 -> swap
{5,3,1,6,7,2,**4,8**} -- 4 < 8 -> swap
```

After one iteration through the list, we have `{5,3,1,6,7,2,4,8}`. Note that the greatest unsorted value in the array (8 in this case) will always reach its final position. Thus, to be sure the list is sorted we must iterate n-1 times for lists of length n.

Graphic:

Section 29.2: Implementation in C & C++

An example implementation of `BubbleSort` in `C++`:

```
   void bubbleSort(vector<int>numbers)
   {
       for(int i = numbers.size() - 1; i >= 0; i--) {
           for(int j = 1; j <= i; j++) {
               if(numbers[j-1] > numbers[j]) {
                   swap(numbers[j-1],numbers(j));
               }
           }
       }
   }
```

C Implementation

```
void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */

        t          = list[d];
        list[d]    = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

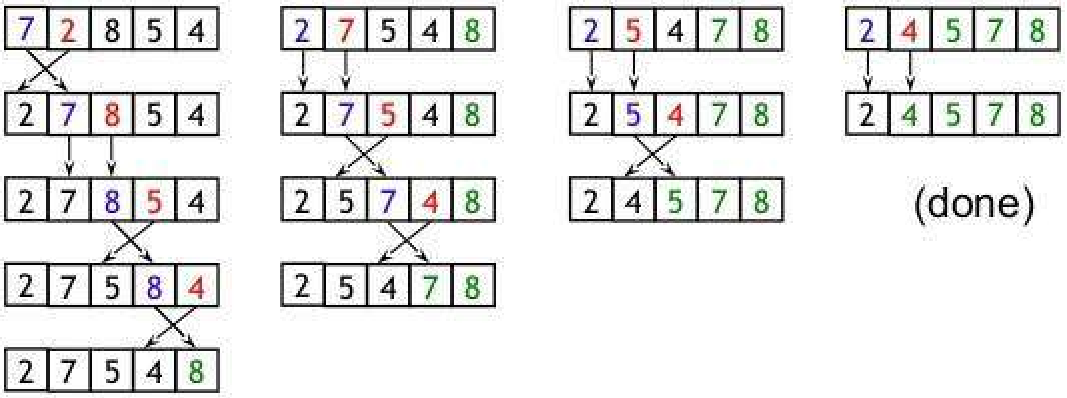
Bubble Sort with pointer

```
void pointer_bubble_sort(long * list, long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if ( * (list + d ) > *(list+d+1))
      {
        /* Swapping */

        t          = * (list + d );
        * (list + d )   = * (list + d + 1 );
        * (list + d + 1) = t;
      }
    }
  }
}
```

Section 29.3: Implementation in C#

Bubble sort is also known as **Sinking Sort**. It is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Bubble sort example



Implementation of Bubble Sort

I used C# language to implement bubble sort algorithm

```
public class BubbleSort
{
    public static void SortBubble(int[] input)
    {
        for (var i = input.Length - 1; i >= 0; i--)
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--)
            {
                if (input[j] <= input[j + 1]) continue;
                var temp = input[j + 1];
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBubble(input);
        return input;
    }
}
```

Section 29.4: Python Implementation

```
#!/usr/bin/python

input_list = [10,1,2,11]

for i in range(len(input_list)):
  for j in range(i):
    if int(input_list[j]) > int(input_list[j+1]):
      input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list
```

Section 29.5: Implementation in Java

```java
public class MyBubbleSort {

    public static void bubble_srt(int array[]) {//main logic
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
                if (array[i] > array[k]) {
                    swapNumbers(i, k, array);
                }
            }
            printNumbers(array);
        }
    }

    private static void swapNumbers(int i, int j, int[] array) {

        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void printNumbers(int[] input) {

        for (int i = 0; i < input.length; i++) {
            System.out.print(input[i] + ", ");
        }
        System.out.println("\n");
    }

    public static void main(String[] args) {
        int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
        bubble_srt(input);

    }
}
```

Section 29.6: Implementation in Javascript

```javascript
        function bubbleSort(a)
          {
                var swapped;
                do {
                    swapped = false;
                    for (var i=0; i < a.length-1; i++) {
                        if (a[i] > a[i+1]) {
                            var temp = a[i];
                            a[i] = a[i+1];
                            a[i+1] = temp;
                            swapped = true;
                        }
                    }
                } while (swapped);
            }

   var a = [3, 203, 34, 746, 200, 984, 198, 764, 9];
```

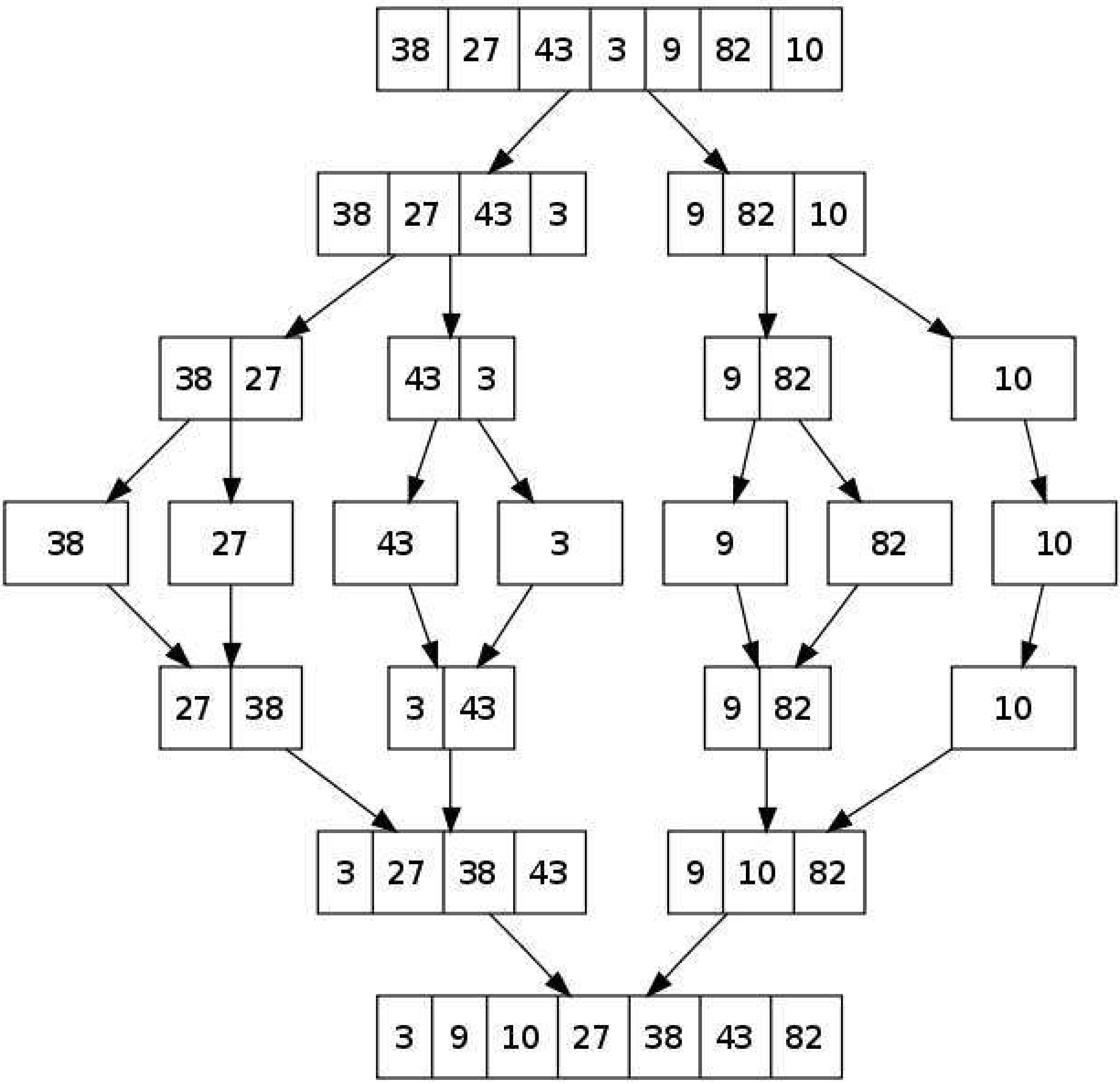
```
    bubbleSort(a);
    console.log(a); //logs [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]
```

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

Merge Sort is a divide-and-conquer algorithm. It divides the input list of length n in half successively until there are n lists of size 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built.

An example:



Time Complexity: `T(n) = 2T(n/2) + Θ(n)`

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is `Θ(nLogn)`. Time complexity of *Merge Sort* is `Θ(nLogn)` in all 3 cases (*worst, average and best*) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: `O(n)`

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: Not in a typical implementation

Stable: Yes

Section 30.2: Merge Sort Implementation in Go

```go
package main

import "fmt"

func mergeSort(a []int) []int {
    if len(a) < 2 {
        return a
    }
    m := (len(a)) / 2

    f := mergeSort(a[:m])
    s := mergeSort(a[m:])

    return merge(f, s)
}

func merge(f []int, s []int) []int {
    var i, j int
    size := len(f) + len(s)

    a := make([]int, size, size)

    for z := 0; z < size; z++ {
        lenF := len(f)
        lenS := len(s)

        if i > lenF-1 && j <= lenS-1 {
            a[z] = s[j]
            j++
        } else if j > lenS-1 && i <= lenF-1 {
            a[z] = f[i]
            i++
        } else if f[i] < s[j] {
            a[z] = f[i]
            i++
        } else {
            a[z] = s[j]
            j++
        }
    }

    return a
}

func main() {
    a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
    fmt.Println(a)
    fmt.Println(mergeSort(a))
}
```

Section 30.3: Merge Sort Implementation in C & C#

C Merge Sort

```
int merge(int arr[],int l,int m,int h)
{
  int arr1[10],arr2[10];  // Two temporary arrays to
  hold the two arrays to be merged
  int n1,n2,i,j,k;
  n1=m-l+1;
  n2=h-m;

  for(i=0; i<n1; i++)
    arr1[i]=arr[l+i];
  for(j=0; j<n2; j++)
    arr2[j]=arr[m+j+1];

  arr1[i]=9999;  // To mark the end of each temporary array
  arr2[j]=9999;

  i=0;
  j=0;
  for(k=l; k<=h; k++) { //process of combining two sorted arrays
    if(arr1[i]<=arr2[j])
      arr[k]=arr1[i++];
    else
      arr[k]=arr2[j++];
  }

  return 0;
}

int merge_sort(int arr[],int low,int high)
{
  int mid;
  if(low<high) {
    mid=(low+high)/2;
    // Divide and Conquer
    merge_sort(arr,low,mid);
    merge_sort(arr,mid+1,high);
    // Combine
    merge(arr,low,mid,high);
  }

  return 0;
}
```

C# Merge Sort

```
public class MergeSort
    {
        static void Merge(int[] input, int l, int m, int r)
        {
            int i, j;
            var n1 = m - l + 1;
            var n2 = r - m;

            var left = new int[n1];
            var right = new int[n2];

            for (i = 0; i < n1; i++)
            {
                left[i] = input[l + i];
            }
            for (j = 0; j < n2; j++)
            {
                right[j] = input[m + j + 1];
            }

            i = 0;
            j = 0;
            var k = l;

            while (i < n1 && j < n2)
            {
                if (left[i] <= right[j])
                {
                    input[k] = left[i];
                    i++;
                }
                else
                {
                    input[k] = right[j];
                    j++;
                }
                k++;
            }

            while (i < n1)
            {
                input[k] = left[i];
                i++;
                k++;
            }

            while (j < n2)
            {
                input[k] = right[j];
                j++;
                k++;
            }
        }

        static void SortMerge(int[] input, int l, int r)
        {
            if (l < r)
            {
                int m = l + (r - l) / 2;
                SortMerge(input, l, m);
                SortMerge(input, m + 1, r);
                Merge(input, l, m, r);
            }
        }

        public static int[] Main(int[] input)
        {
            SortMerge(input, 0, input.Length - 1);
            return input;
        }
    }
```

Section 30.4: Merge Sort Implementation in Java

Below there is the implementation in Java using a generics approach. It is the same algorithm, which is presented above.

```
public interface InPlaceSort<T extends Comparable<T>> {
void sort(final T[] elements); }


public class MergeSort < T extends Comparable < T >> implements InPlaceSort < T > {


@Override
public void sort(T[] elements) {
    T[] arr = (T[]) new Comparable[elements.length];
    sort(elements, arr, 0, elements.length - 1);
}

// We check both our sides and then merge them
private void sort(T[] elements, T[] arr, int low, int high) {
    if (low >= high) return;
    int mid = low + (high - low) / 2;
    sort(elements, arr, low, mid);
    sort(elements, arr, mid + 1, high);
    merge(elements, arr, low, high, mid);
}


private void merge(T[] a, T[] b, int low, int high, int mid) {
    int i = low;
    int j = mid + 1;

    // We select the smallest element of the two. And then we put it into b
    for (int k = low; k <= high; k++) {

        if (i <= mid && j <= high) {
            if (a[i].compareTo(a[j]) >= 0) {
                b[k] = a[j++];
            } else {
                b[k] = a[i++];
            }
        } else if (j > high && i <= mid) {
            b[k] = a[i++];
        } else if (i > mid && j <= high) {
            b[k] = a[j++];
        }
    }

    for (int n = low; n <= high; n++) {
        a[n] = b[n];
    }}}
```

Section 30.5: Merge Sort Implementation in Python

```
def merge(X, Y):
    " merge two sorted lists "
    p1 = p2 = 0
    out = []
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
            out.append(X[p1])
            p1 += 1
        else:
            out.append(Y[p2])
            p2 += 1
    out += X[p1:] + Y[p2:]
    return out

def mergeSort(A):
    if len(A) <= 1:
        return A
    if len(A) == 2:
        return sorted(A)

    mid = len(A) / 2
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:]))

if __name__ == "__main__":
    # Generate 20 random numbers and sort them
    A = [randint(1, 100) for i in xrange(20)]
    print mergeSort(A)
```

Section 30.6: Bottoms-up Java Implementation

```
public class MergeSortBU {
    private static Integer[] array = { 4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70,
60,80,0,9,67,54,51,52,24,54,7 };

    public MergeSortBU() {
    }

    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid, int hi) {

        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }

        int i = lo;
        int j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid)
                arrayToSort[k] = aux[j++];
            else if (j > hi)
                arrayToSort[k] = aux[i++];
            else if (isLess(aux[i], aux[j])) {
                arrayToSort[k] = aux[i++];
            } else {
                arrayToSort[k] = aux[j++];
            }

        }
    }

    public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
        int N = arrayToSort.length;
        for (int sz = 1; sz < N; sz = sz + sz) {
            for (int low = 0; low < N; low = low + sz + sz) {
                System.out.println("Size:"+ sz);
                merge(arrayToSort, aux, low, low + sz -1 ,Math.min(low + sz + sz - 1, N - 1));
                print(arrayToSort);
            }
        }

    }

    public static boolean isLess(Comparable a, Comparable b) {
        return a.compareTo(b) <= 0;
    }

    private static void print(Comparable[] array)
{http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        StringBuffer buffer = new
StringBuffer();http://stackoverflow.com/documentation/algorithm/5732/merge-sort#
        for (Comparable value : array) {
            buffer.append(value);
            buffer.append(' ');
        }
        System.out.println(buffer);
    }

    public static void main(String[] args) {
        Comparable[] aux = new Comparable[array.length];
        print(array);
        MergeSortBU.sort(array, aux, 0, array.length - 1);
    }
}
```

Chapter 31: Insertion Sort

Section 31.1: Haskell Implementation

```haskell
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
insertSort (x:xs) = insert x (insertSort xs)

insert :: Ord a => a-> [a] -> [a]
insert n [] = [n]
insert n (x:xs) | n <= x    = (n:x:xs)
                | otherwise = x:insert n xs
```

Chapter 32: Bucket Sort

Section 32.1: C# Implementation

```
public class BucketSort
{
    public static void SortBucket(ref int[] input)
    {
        int minValue = input[0];
        int maxValue = input[0];
        int k = 0;

        for (int i = input.Length - 1; i >= 1; i--)
        {
            if (input[i] > maxValue) maxValue = input[i];
            if (input[i] < minValue) minValue = input[i];
        }

        List<int>[] bucket = new List<int>[maxValue - minValue + 1];

        for (int i = bucket.Length - 1; i >= 0; i--)
        {
            bucket[i] = new List<int>();
        }

        foreach (int i in input)
        {
            bucket[i - minValue].Add(i);
        }

        foreach (List<int> b in bucket)
        {
            if (b.Count > 0)
            {
                foreach (int t in b)
                {
                    input[k] = t;
                    k++;
                }
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortBucket(ref input);
        return input;
    }
}
```

Chapter 33: Quicksort

Section 33.1: Quicksort Basics

Quicksort is a sorting algorithm that picks an element ("the pivot") and reorders the array forming two partitions such that all elements less than the pivot come before it and all elements greater come after. The algorithm is then applied recursively to the partitions until the list is sorted.

1. Lomuto partition scheme mechanism :

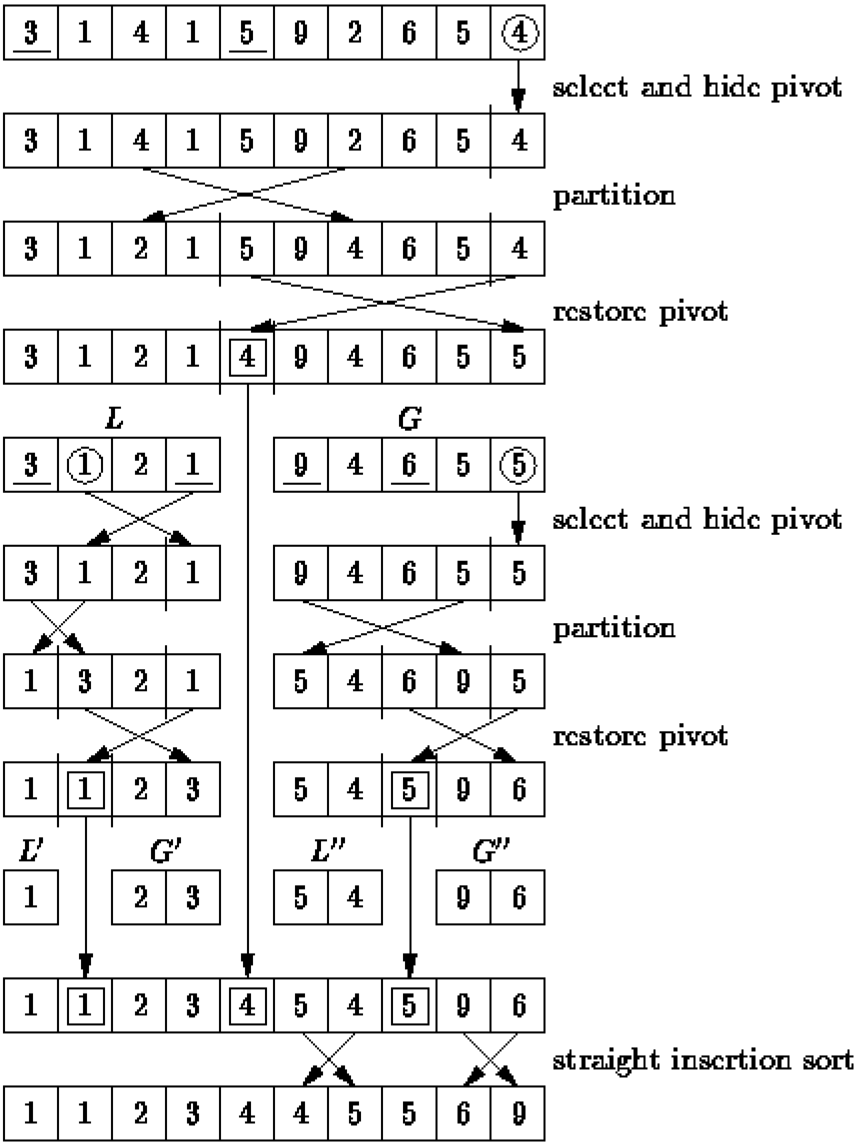
This scheme chooses a pivot which is typically the last element in the array. The algorithm maintains the index to put the pivot in variable i and each time it finds an element less than or equal to pivot, this index is incremented and that element would be placed before the pivot.

```
partition(A, low, high) is
pivot := A[high]
i := low
for j := low to high - 1 do
    if A[j] ≤ pivot then
        swap A[i] with A[j]
        i := i + 1
swap A[i] with A[high]
return i
```

Quick Sort mechanism :

```
quicksort(A, low, high) is
if low < high then
    p := partition(A, low, high)
    quicksort(A, low, p - 1)
    quicksort(A, p + 1, high)
```

Example of quick sort:



2. Hoare partition scheme:

It uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater or equal than the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index. Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.

```
quicksort(A, lo, hi) is
if lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Partition :

```
partition(A, lo, hi) is
pivot := A[lo]
i := lo - 1
j := hi + 1
loop forever
    do:
        i := i + 1
    while A[i] < pivot do

    do:
        j := j - 1
    while A[j] > pivot do

    if i >= j then
        return j

    swap A[i] with A[j]
```

Section 33.2: Quicksort in Python

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) / 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])
```

Prints "[1, 1, 2, 3, 6, 8, 10]"

Section 33.3: Lomuto partition java implementation

```
public class Solution {

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] ar = new int[n];
    for(int i=0; i<n; i++)
      ar[i] = sc.nextInt();
     quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high)
 {
    if(low<high)
    {
        int p = partition(ar, low, high);
        quickSort(ar, 0 , p-1);
        quickSort(ar, p+1, high);
    }
 }
public static int partition(int[] ar, int l, int r)
 {
    int pivot = ar[r];
    int i =l;
    for(int j=l; j<r; j++)
     {
        if(ar[j] <= pivot)
         {
            int t = ar[j];
            ar[j] = ar[i];
            ar[i] = t;
            i++;
        }
    }
    int t = ar[i];
    ar[i] = ar[r];
    ar[r] = t;

    return i;
}
```

Chapter 34: Counting Sort

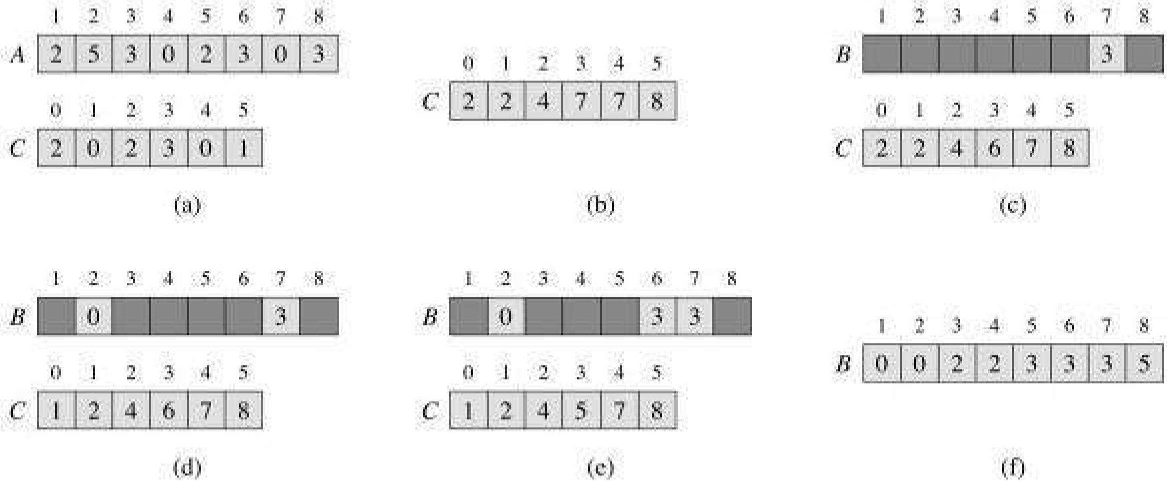
Section 34.1: Counting Sort Basic Information

Counting sort is an integer sorting algorithm for a collection of objects that sorts according to the keys of the objects.

Steps

1. Construct a working array *C* that has size equal to the range of the input array *A*.
2. Iterate through *A*, assigning *C*[x] based on the number of times x appeared in *A*.
3. Transform *C* into an array where *C*[x] refers to the number of values ≤ x by iterating through the array, assigning to each *C*[x] the sum of its prior value and all values in *C* that come before it.
4. Iterate backwards through *A*, placing each value in to a new sorted array *B* at the index recorded in *C*. This is done for a given *A*[x] by assigning *B*[*C*[*A*[x]]] to *A*[x], and decrementing *C*[*A*[x]] in case there were duplicate values in the original unsorted array.

Example of Counting Sort



Auxiliary Space: `O(n+k)`
Time Complexity: Worst-case: `O(n+k)`, Best-case: `O(n)`, Average-case `O(n+k)`

Section 34.2: Psuedocode Implementation

Constraints:

1. Input (an array to be sorted)
2. Number of element in input (n)
3. Keys in the range of *0..k-1* (k)
4. Count (an array of number)

Pseudocode:

```
for x in input:
    count[key(x)] += 1
total = 0
for i in range(k):
    oldCount = count[i]
    count[i] = total
    total += oldCount
```

```
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1
return output
```

Chapter 35: Heap Sort

Section 35.1: C# Implementation

```
public class HeapSort
{
    public static void Heapify(int[] input, int n, int i)
    {
        int largest = i;
        int l = i + 1;
        int r = i + 2;

        if (l < n && input[l] > input[largest])
            largest = l;

        if (r < n && input[r] > input[largest])
            largest = r;

        if (largest != i)
        {
            var temp = input[i];
            input[i] = input[largest];
            input[largest] = temp;
            Heapify(input, n, largest);
        }
    }

    public static void SortHeap(int[] input, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
            Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
            var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
        SortHeap(input, input.Length);
        return input;
    }
}
```

Section 35.2: Heap Sort Basic Information

Heap sort is a comparison based sorting technique on binary heap data structure. It is similar to selection sort in which we first find the maximum element and put it at the end of the data structure. Then repeat the same process for the remaining items.

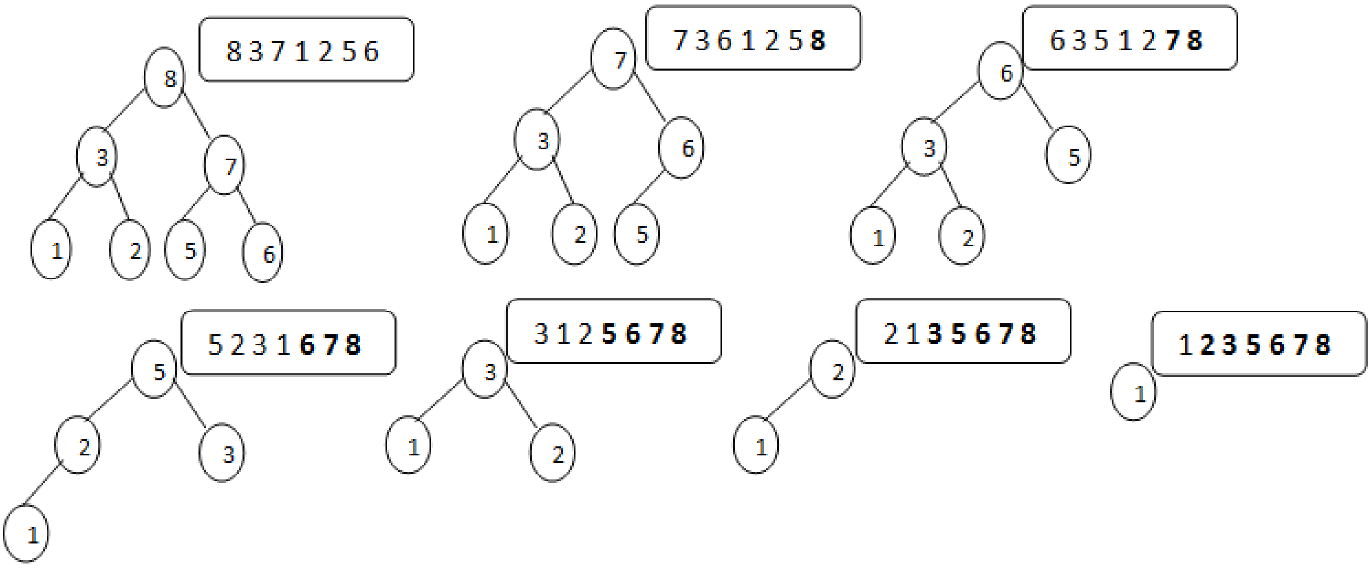
Pseudo code for Heap Sort:

```
function heapsort(input, count)
```

```
    heapify(a,count)
    end <- count - 1
    while end -> 0 do
    swap(a[end],a[0])
    end<-end-1
    restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

Example of Heap Sort:

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



Auxiliary Space: `O(1)`
Time Complexity: `O(nlogn)`

Chapter 36: Cycle Sort

Section 36.1: Pseudocode Implementation

```
(input)
output = 0
for cycleStart from 0 to length(array) - 2
    item = array[cycleStart]
    pos = cycleStart
    for i from cycleStart + 1 to length(array) - 1
        if array[i] < item:
            pos += 1
    if pos == cycleStart:
        continue
    while item == array[pos]:
        pos += 1
    array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
        pos = cycleStart
        for i from cycleStart + 1 to length(array) - 1
            if array[i] < item:
                pos += 1
        while item == array[pos]:
            pos += 1
        array[pos], item = item, array[pos]
        writes += 1
return outout
```

Chapter 37: Odd-Even Sort

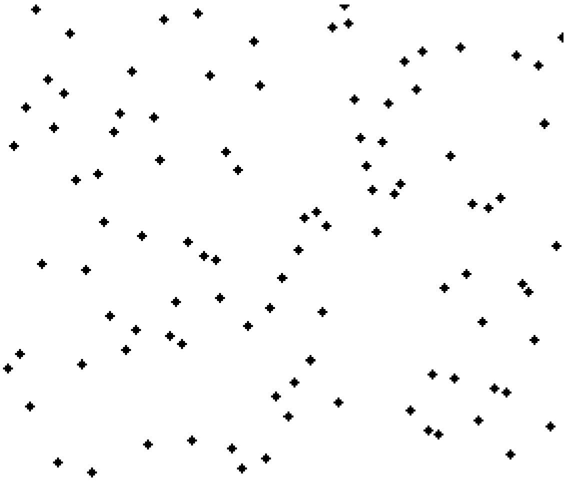
Section 37.1: Odd-Even Sort Basic Information

An Odd-Even Sort or brick sort is a simple sorting algorithm, which is developed for use on parallel processors with local interconnection. It works by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order the elements are switched. The next step repeats this for even/odd indexed pairs. Then it alternates between odd/even and even/odd steps until the list is sorted.

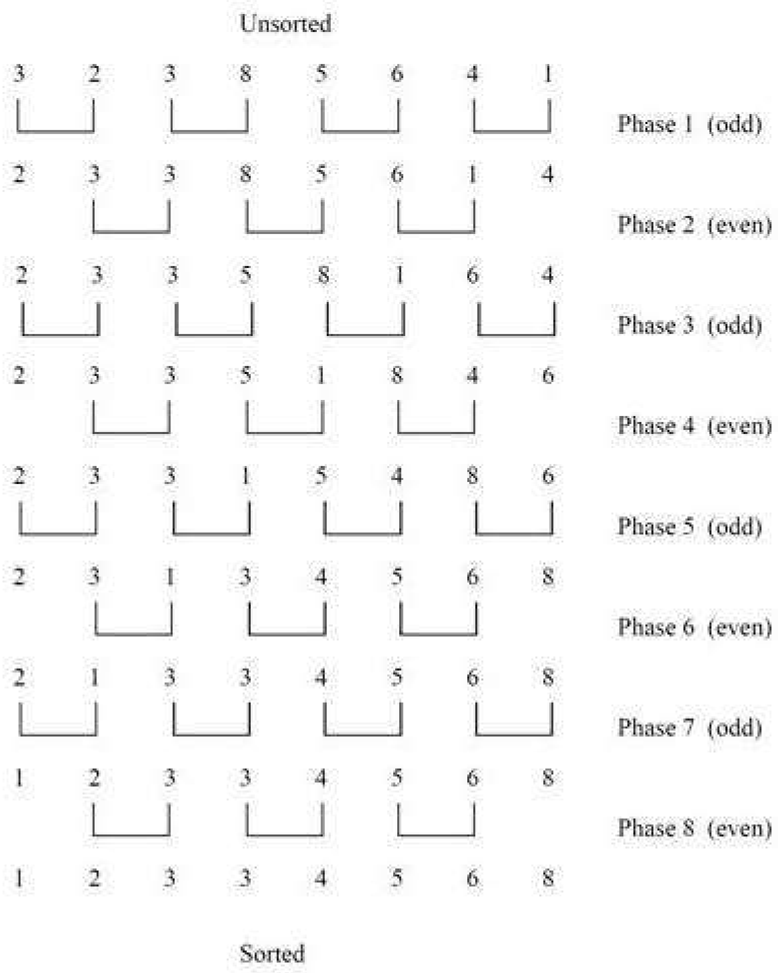
Pseudo code for Odd-Even Sort:

```
if n>2 then
    1. apply odd-even merge(n/2) recursively to the even subsequence a0, a2, ..., an-2 and to the
odd subsequence a1, a3, , ..., an-1
    2. comparison [i : i+1] for all i element {1, 3, 5, 7, ..., n-3}
else
    comparison [0 : 1]
```

Wikipedia has best illustration of Odd-Even sort:



Example of Odd-Even Sort:



Implementation:

I used C# language to implement Odd-Even Sort Algorithm.

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
        var sort = false;

        while (!sort)
        {
            sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
            for (var i = 0; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
                var temp = input[i];
                input[i] = input[i + 1];
                input[i + 1] = temp;
                sort = false;
            }
        }
    }

    public static int[] Main(int[] input)
    {
        SortOddEven(input, input.Length);
        return input;
    }
}
```

Auxiliary Space: `O(n)`
Time Complexity: `O(n)`

Chapter 38: Selection Sort

Section 38.1: Elixir Implementation

```
defmodule Selection do

  def sort(list) when is_list(list) do
    do_selection(list, [])
  end

  def do_selection([head|[]], acc) do
    acc ++ [head]
  end

  def do_selection(list, acc) do
    min = min(list)
    do_selection(:lists.delete(min, list), acc ++ [min])
  end

  defp min([first|[second|[]]]) do
    smaller(first, second)
  end

  defp min([first|[second|tail]]) do
    min([smaller(first, second)|tail])
  end

  defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    else
      e2
    end
  end
end

Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

Section 38.2: Selection Sort Basic Information

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

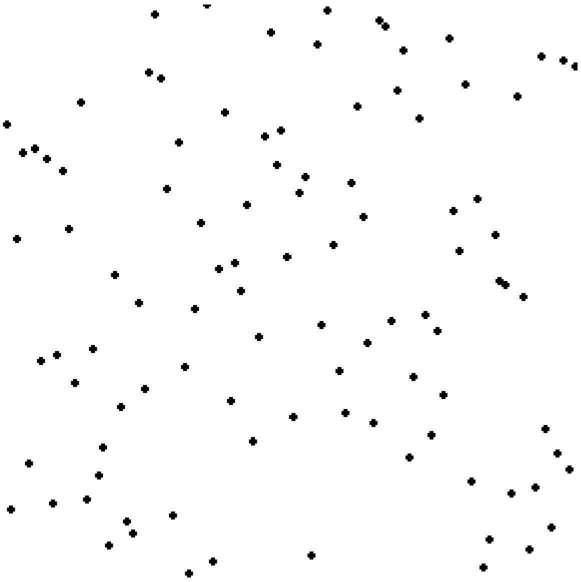
The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Pseudo code for Selection sort:

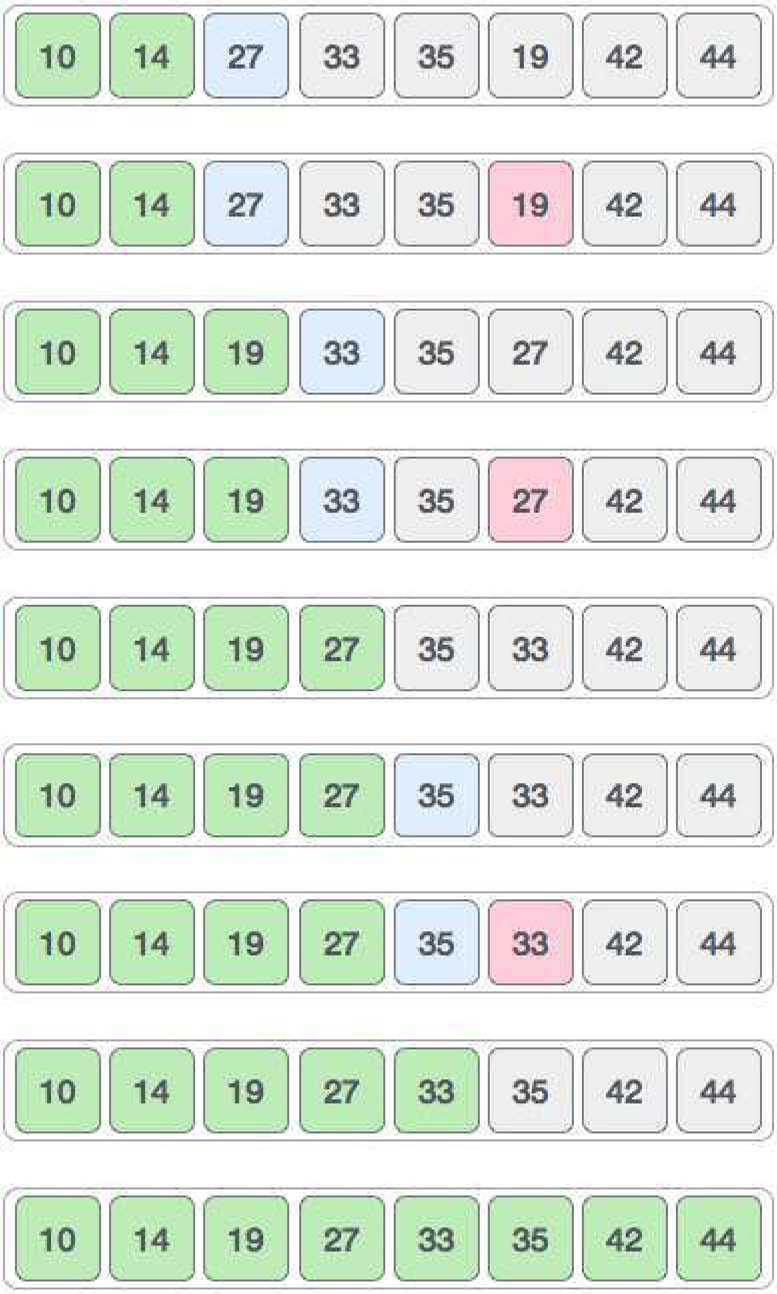
```
function select(list[1..n], k)
 for i from 1 to k
```

```
    minIndex = i
    minValue = list[i]
    for j from i+1 to n
        if list[j] < minValue
            minIndex = j
            minValue = list[j]
    swap list[i] and list[minIndex]
return list[k]
```

Visualization of selection sort:



Example of Selection sort:



10 14 27 33 35 19 42 44

10 14 27 33 35 19 42 44

10 14 19 33 35 27 42 44

10 14 19 33 35 27 42 44

10 14 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 35 33 42 44

10 14 19 27 33 35 42 44

10 14 19 27 33 35 42 44

Auxiliary Space: `O(n)`
Time Complexity: `O(n^2)`

Section 38.3: Implementation of Selection sort in C#

I used C# language to implement Selection sort algorithm.

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
            var temp = input[minId];
            input[minId] = input[i];
            input[i] = temp;
        }
    }

    public static int[] Main(int[] input)
    {
        SortSelection(input, input.Length);
        return input;
    }
}
```

Chapter 39: Searching

Section 39.1: Binary Search

Introduction

Binary Search is a Divide and Conquer search algorithm. It uses `O(log n)` time to find the location of an element in a search space where `n` is the size of the search space.

Binary Search works by halving the search space at each iteration after comparing the target value to the middle value of the search space.

To use Binary Search, the search space must be ordered (sorted) in some way. Duplicate entries (ones that compare as equal according to the comparison function) cannot be distinguished, though they don't violate the Binary Search property.

Conventionally, we use less than (<) as the comparison function. If a < b, it will return true. if a is not less than b and b is not less than a, a and b are equal.

Example Question

You are an economist, a pretty bad one though. You are given the task of finding the equilibrium price (that is, the price where supply = demand) for rice.

Remember the higher a price is set, the larger the supply and the lesser the demand

As your company is very efficient at calculating market forces, you can instantly get the supply and demand in units of rice when the price of rice is set at a certain price `p`.

Your boss wants the equilibrium price ASAP, but tells you that the equilibrium price can be a positive integer that is at most `10^17` and there is guaranteed to be exactly 1 positive integer solution in the range. So get going with your job before you lose it!

You are allowed to call functions `getSupply(k)` and `getDemand(k)`, which will do exactly what is stated in the problem.

Example Explanation

Here our search space is from `1` to `10^17`. Thus a linear search is infeasible.

However, notice that as the `k` goes up, `getSupply(k)` increases and `getDemand(k)` decreases. Thus, for any `x > y`, `getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)`. Therefore, this search space is monotonic and we can use Binary Search.

The following psuedocode demonstrates the usage of Binary Search:

```
high = 100000000000000000     <- Upper bound of search space
low = 1                       <- Lower bound of search space
while high - low > 1
    mid = (high + low) / 2    <- Take the middle value
    supply = getSupply(mid)
    demand = getDemand(mid)
    if supply > demand
        high = mid             <- Solution is in lower half of search space
    else if demand > supply
        low = mid                <- Solution is in upper half of search space
    else                         <- supply==demand condition
        return mid               <- Found solution
```

This algorithm runs in `~O(log 10^17)` time. This can be generalized to `~O(log S)` time where S is the size of the search space since at every iteration of the **`while`** loop, we halved the search space (*from [low:high] to either [low:mid] or [mid:high]*).

C Implementation of Binary Search with Recursion

```
int binsearch(int a[], int x, int low, int high) {
    int mid;

    if (low > high)
      return -1;

    mid = (low + high) / 2;

    if (x == a[mid]) {
        return (mid);
    } else
    if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    } else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Section 39.2: Rabin Karp

The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text.Its average and best case running time is O(n+m) in space O(p), but its worst-case time is O(nm) where n is the length of the text and m is the length of the pattern.

Algorithm implementation in java for string matching

```
void RabinfindPattern(String text,String pattern){
    /*
    q a prime number
    p hash value for pattern
    t hash value for text
    d is the number of unique characters in input alphabet
    */
    int d=128;
    int q=100;
    int n=text.length();
    int m=pattern.length();
    int t=0,p=0;
    int h=1;
    int i,j;
//hash value calculating function
    for (i=0;i<m-1;i++)
            h = (h*d)%q;
        for (i=0;i<m;i++){
        p = (d*p + pattern.charAt(i))%q;
        t = (d*t + text.charAt(i))%q;
        }
//search for the pattern
```

```
    for(i=0;i<end-m;i++){
        if(p==t){
//if the hash value matches match them character by character
            for(j=0;j<m;j++)
                if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
        }
        if(i<end-m){
            t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
            if(t<0)
                t=t+q;
        }
    }
}
```

While calculating hash value we are dividing it by a prime number in order to avoid collision.After dividing by prime number the chances of collision will be less, but still ther is a chance that the hash value can be same for two strings,so when we get a match we have to check it character by character to make sure that we got a proper match.

t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;

This is to recalculate the hash value for pattern,first by removing the left most character and then adding the new character from the text.

Section 39.3: Analysis of Linear search (Worst, Average and Best Cases)

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

/* Driver program to test above functions*/

```
int main()
```

```
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n)

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Average Case Time = $\frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$

= $\frac{\theta((n+1)*(n+2)/2)}{(n+1)}$

= Θ(n)

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1) Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort. Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Section 39.4: Binary Search: On Sorted Numbers

It's easiest to show a binary search on numbers using pseudo-code

```
int array[1000] = { sorted list of numbers };
int N = 100;  // number of entries in search space;
int high, low, mid; // our temporaries
int x; // value to search for

low = 0;
high = N -1;
while(low < high)
{
    mid = (low + high)/2;
    if(array[mid] < x)
        low = mid + 1;
    else
        high = mid;
}
if(array[low] == x)
    // found, index is low
else
    // not found
```

Do not attempt to return early by comparing array[mid] to x for equality. The extra comparison can only slow the code down. Note you need to add one to low to avoid becoming trapped by integer division always rounding down.

Interestingly, the above version of binary search allows you to find the smallest occurrence of x in the array. If the array contains duplicates of x, the algorithm can be modified slightly in order for it to return the largest occurrence of x by simply adding to the if conditional:

```
while(low < high)
    {
        mid = low + ((high - low) / 2);
        if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
            low = mid + 1;
        else
            high = mid;
    }
```

Note that instead of doing `mid = (low + high) / 2`, it may also be a good idea to try `mid = low + ((high - low) / 2)` for implementations such as Java implementations to lower the risk of getting an overflow for really large inputs.

Section 39.5: Linear search

Linear search is a simple algorithm. It loops through items until the query has been found, which makes it a linear algorithm - the complexity is O(n), where n is the number of items to go through.

Why O(n)? In worst-case scenario, you have to go through all of the n items.

It can be compared to looking for a book in a stack of books - you go through them all until you find the one that you want.

Below is a Python implementation:

```
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False

linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #returns True
```

Chapter 40: Substring Search

Section 40.1: Introduction To Knuth-Morris-Pratt (KMP) Algorithm

Suppose that we have a *text* and a *pattern*. We need to determine if the pattern exists in the text or not. For example:

```
+-------+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+
|  Text | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| Index   | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| Pattern | b | c | g | l |
+---------+---+---+---+---+
```

This *pattern* does exist in the *text*. So our substring search should return **3**, the index of the position from which this *pattern* starts. So how does our brute force substring search procedure work?

What we usually do is: we start from the **0th** index of the *text* and the **0th** index of our *pattern and we compare **Text[0]** with **Pattern[0]**. Since they are not a match, we go to the next index of our *text* and we compare **Text[1]** with **Pattern[0]**. Since this is a match, we increment the index of our *pattern* and the index of the *Text* also. We compare **Text[2]** with **Pattern[1]**. They are also a match. Following the same procedure stated before, we now compare **Text[3]** with **Pattern[2]**. As they do not match, we start from the next position where we started finding the match. That is index **2** of the *Text*. We compare **Text[2]** with **Pattern[0]**. They don't match. Then incrementing index of the *Text*, we compare **Text[3]** with **Pattern[0]**. They match. Again **Text[4]** and **Pattern[1]** match, **Text[5]** and **Pattern[2]** match and **Text[6]** and **Pattern[3]** match. Since we've reached the end of our *Pattern*, we now return the index from which our match started, that is **3**. If our *pattern* was: `bcgll`, that means if the *pattern* didn't exist in our *text*, our search should return exception or **-1** or any other predefined value. We can clearly see that, in the worst case, this algorithm would take `O(mn)` time where **m** is the length of the *Text* and **n** is the length of the *Pattern*. How do we reduce this time complexity? This is where KMP Substring Search Algorithm comes into the picture.

The Knuth-Morris-Pratt String Searching Algorithm or KMP Algorithm searches for occurrences of a "Pattern" within a main "Text" by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The algorithm was conceived in 1970 by Donuld Knuth and Vaughan Pratt and independently by James H. Morris. The trio published it jointly in 1977.

Let's extend our example *Text* and *Pattern* for better understanding:

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Index |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Text |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

At first, our *Text* and *Pattern* matches till index **2**. **Text[3]** and **Pattern[3]** doesn't match. So our aim is to not go backwards in this *Text*, that is, in case of a mismatch, we don't want our matching to begin again from the position that we started matching with. To achieve that, we'll look for a **suffix** in our *Pattern* right before our mismatch occurred (substring **abc**), which is also a **prefix** of the substring of our *Pattern*. For our example, since all the characters are unique, there is no suffix, that is the prefix of our matched substring. So what that means is, our next comparison will start from index **0**. Hold on for a bit, you'll understand why we did this. Next, we compare **Text[3]** with **Pattern[0]** and it doesn't match. After that, for *Text* from index **4** to index **9** and for *Pattern* from index **0** to index **5**, we find a match. We find a mismatch in **Text[10]** and **Pattern[6]**. So we take the substring from *Pattern* right before the point where mismatch occurs (substring **abcdabc**), we check for a suffix, that is also a prefix of this substring. We can see here **ab** is both the suffix and prefix of this substring. What that means is, since we've matched until **Text[10]**, the characters right before the mismatch is **ab**. What we can infer from it is that since **ab** is also a prefix of the substring we took, we don't have to check **ab** again and the next check can start from **Text[10]** and **Pattern[2]**. We didn't have to look back to the whole *Text*, we can start directly from where our mismatch occurred. Now we check **Text[10]** and **Pattern[2]**, since it's a mismatch, and the substring before mismatch (**abc**) doesn't contain a suffix which is also a prefix, we check **Text[10]** and **Pattern[0]**, they don't match. After that for *Text* from index **11** to index **17** and for *Pattern* from index **0** to index **6**. We find a mismatch in **Text[18]** and **Pattern[7]**. So again we check the substring before mismatch (substring **abcdabc**) and find **abc** is both the suffix and the prefix. So since we matched till **Pattern[7]**, **abc** must be before **Text[18]**. That means, we don't need to compare until **Text[17]** and our comparison will start from **Text[18]** and **Pattern[3]**. Thus we will find a match and we'll return **15** which is our starting index of the match. This is how our KMP Substring Search works using suffix and prefix information.

Now, how do we efficiently compute if suffix is same as prefix and at what point to start the check if there is a mismatch of character between *Text* and *Pattern*. Let's take a look at an example:

```
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

We'll generate an array containing the required information. Let's call the array **S**. The size of the array will be same as the length of the pattern. Since the first letter of the *Pattern* can't be the suffix of any prefix, we'll put **S[0]** = **0**. We take **i** = **1** and **j** = **0** at first. At each step we compare **Pattern[i]** and **Pattern[j]** and increment **i**. If there is a match we put **S[i]** = **j** + **1** and increment **j**, if there is a mismatch, we check the previous value position of **j** (if available) and set **j** = **S[j-1]** (if **j** is not equal to **0**), we keep doing this until **S[j]** doesn't match with **S[i]** or **j** doesn't become **0**. For the later one, we put **S[i]** = **0**. For our example:

```
            j   i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Pattern[j] and **Pattern[i]** don't match, so we increment **i** and since **j** is **0**, we don't check the previous value and put **Pattern[i]** = **0**. If we keep incrementing **i**, for **i** = **4**, we'll get a match, so we put **S[i]** = **S[4]** = **j** + **1** = **0** + **1** = **1** and increment **j** and **i**. Our array will look like:

```
                j               i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

Since **Pattern[1]** and **Pattern[5]** is a match, we put **S[i]** = **S[5]** = **j** + **1** = **1** + **1** = **2**. If we continue, we'll find a mismatch for **j** = **3** and **i** = **7**. Since **j** is not equal to **0**, we put **j** = **S[j-1]**. And we'll compare the characters at **i** and **j** are same or not, since they are same, we'll put **S[i]** = **j** + 1. Our completed array will look like:

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

This is our required array. Here a nonzero-value of **S[i]** means there is a **S[i]** length suffix same as the prefix in that substring (substring from **0** to **i**) and the next comparison will start from **S[i]** + **1** position of the *Pattern*. Our algorithm to generate the array would look like:

```
Procedure GenerateSuffixArray(Pattern):
i := 1
j := 0
n := Pattern.length
while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i] := j + 1
        j := j + 1
        i := i + 1
    else
        if j is not equal to 0
            j := S[j-1]
        else
            S[i] := 0
            i := i + 1
        end if
    end if
end while
```

The time complexity to build this array is `O(n)` and the space complexity is also `O(n)`. To make sure if you have completely understood the algorithm, try to generate an array for pattern `aabaabaa` and check if the result matches with this one.

Now let's do a substring search using the following example:

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   Text  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 |
+---------+---+---+---+---+---+---+
| Pattern | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

We have a *Text*, a *Pattern* and a pre-calculated array *S* using our logic defined before. We compare **Text[0]** and **Pattern[0]** and they are same. **Text[1]** and **Pattern[1]** are same. **Text[2]** and **Pattern[2]** are not same. We check the value at the position right before the mismatch. Since **S[1]** is **0**, there is no suffix that is same as the prefix in our substring and our comparison starts at position **S[1]**, which is **0**. So **Pattern[0]** is not same as **Text[2]**, so we move on. **Text[3]** is same as **Pattern[0]** and there is a match till **Text[8]** and **Pattern[5]**. We go one step back in the **S** array and find **2**. So this means there is a prefix of length **2** which is also the suffix of this substring (**abcab)** which is **ab**. That also means that there is an **ab** before **Text[8]**. So we can safely ignore **Pattern[0]** and **Pattern[1]** and start our next comparison from **Pattern[2]** and **Text[8]**. If we continue, we'll find the *Pattern* in the *Text*. Our procedure will look like:

```
Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)
m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
    if j is equal to n
        Return (j-i)
    else if i < m and Pattern[j] is not equal t Text[i]
        if j is not equal to 0
            j = S[j-1]
        else
            i := i + 1
        end if
    end if
end while
Return -1
```

The time complexity of this algorithm apart from the Suffix Array Calculation is `O(m)`. Since *GenerateSuffixArray* takes `O(n)`, the total time complexity of KMP Algorithm is: `O(m+n)`.

PS: If you want to find multiple occurrences of *Pattern* in the *Text*, instead of returning the value, print it/store it and set `j := S[j-1]`. Also keep a `flag` to track whether you have found any occurrence or not and handle it accordingly.

Section 40.2: Introduction to Rabin-Karp Algorithm

Rabin-Karp Algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any one of a set of pattern strings in a text.

A substring of a string is another string that occurs in. For example, *ver* is a substring of *stackoverflow*. Not to be confused with subsequence because *cover* is a subsequence of the same string. In other words, any subset of consecutive letters in a string is a substring of the given string.

In Rabin-Karp algorithm, we'll generate a hash of our *pattern* that we are looking for & check if the rolling hash of our *text* matches the *pattern* or not. If it doesn't match, we can guarantee that the *pattern* **doesn't exist** in the *text*.

However, if it does match, the *pattern* **can** be present in the *text*. Let's look at an example:

Let's say we have a text: **yeminsajid** and we want to find out if the pattern **nsa** exists in the text. To calculate the hash and rolling hash, we'll need to use a prime number. This can be any prime number. Let's take **prime** = **11** for this example. We'll determine hash value using this formula:

```
(1st letter) X (prime) + (2nd letter) X (prime)¹ + (3rd letter) X (prime)² X + ......
```

We'll denote:

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

The hash value of **nsa** will be:

```
 14 X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

Now we find the rolling-hash of our text. If the rolling hash matches with the hash value of our pattern, we'll check if the strings match or not. Since our pattern has **3** letters, we'll take 1st **3** letters **yem** from our text and calculate hash value. We get:

```
25 X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

This value doesn't match with our pattern's hash value. So the string doesn't exists here. Now we need to consider the next step. To calculate the hash value of our next string **emi**. We can calculate this using our formula. But that would be rather trivial and cost us more. Instead, we use another technique.

- We subtract the value of the **First Letter of Previous String** from our current hash value. In this case, **y**. We get, `1653 - 25 = 1628`.
- We divide the difference with our **prime**, which is **11** for this example. We get, `1628 / 11 = 148`.
- We add **new letter X (prime)**☐$^{-1}$, where **m** is the length of the pattern, with the quotient, which is **i** = **9**. We get, `148 + 9 X 11² = 1237`.

The new hash value is not equal to our patterns hash value. Moving on, for **n** we get:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
New String: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

It doesn't match. After that, for **s**, we get:

```
Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
New String: "ins"
1806 - 13 = 1793
1793 / 11 = 163
```

```
163 + 19 X 11² = 2462
```

It doesn't match. Next, for **a**, we get:

```
Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
New String: "nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

It's a match! Now we compare our pattern with the current string. Since both the strings match, the substring exists in this string. And we return the starting position of our substring.

The pseudo-code will be:

Hash Calculation:

```
Procedure Calculate-Hash(String, Prime, x):
hash := 0                                  // Here x denotes the length to be considered
for m from 1 to x                          // to find the hash value
    hash := hash + (Value of String[m])□⁻¹
end for
Return hash
```

Hash Recalculation:

```
Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]  //here Curr denotes First Letter of Previous String
Hash := Hash / Prime
m := String.length
New := Curr + m - 1
Hash := Hash + (Value of String[New])□⁻¹
Return Hash
```

String Match:

```
Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m - 1
    if Text[i] is not equal to Pattern[i]
        Return false
    end if
end for
Return true
```

Rabin-Karp:

```
Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length - m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for
```

```
Return -1
```

If the algorithm doesn't find any match, it simply returns **-1**.

This algorithm is used in detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical here. Again, **Knuth-Morris-Pratt algorithm** or **Boyer-Moore String Search algorithm** is faster single pattern string searching algorithm, than **Rabin-Karp**. However, it is an algorithm of choice for multiple pattern search. If we want to find any of the large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin-Karp algorithm.

For text of length **n** and **p** patterns of combined length **m**, its average and best case running time is **O(n+m)** in space **O(p)**, but its worst-case time is **O(nm)**.

Section 40.3: Python Implementation of KMP algorithm

Haystack: The string in which given pattern needs to be searched.
Needle: The pattern to be searched.

Time complexity: Search portion (strstr method) has the complexity O(n) where `n` is the length of haystack but as needle is also pre parsed for building prefix table O(m) is required for building prefix table where `m` is the length of the needle.
Therefore, overall time complexity for KMP is **O(n+m)**
Space complexity: **O(m)** because of prefix table on needle.

Note: Following implementation returns the start position of match in haystack (if there is a match) else returns -1, for edge cases like if needle/haystack is an empty string or needle is not found in haystack.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
        prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
                prefix_table[delimeter] = delimeter - j + 1
                break
            j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
    # m: denoting the position within S where the prospective match for W begins
    # i: denoting the index of the currently considered character in W.
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

Section 40.4: KMP Algorithm in C

Given a text *txt* and a pattern *pat*, the objective of this program will be to print all the occurance of *pat* in *txt*.

Examples:

Input:

```
  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"
```

output:

```
Pattern found at index 10
```

Input:

```
  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"
```

output:

```
   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

C Language Implementation:

```
// C program for implementation of KMP pattern searching
// algorithm
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // length of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // This is tricky. Consider the example
           // AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
```

```
          }
          else // if (len == 0)
          {
            lps[i] = 0;
            i++;
          }
        }
     }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Output:

```
Found pattern at index 10
```

Reference:

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

Chapter 41: Breadth-First Search

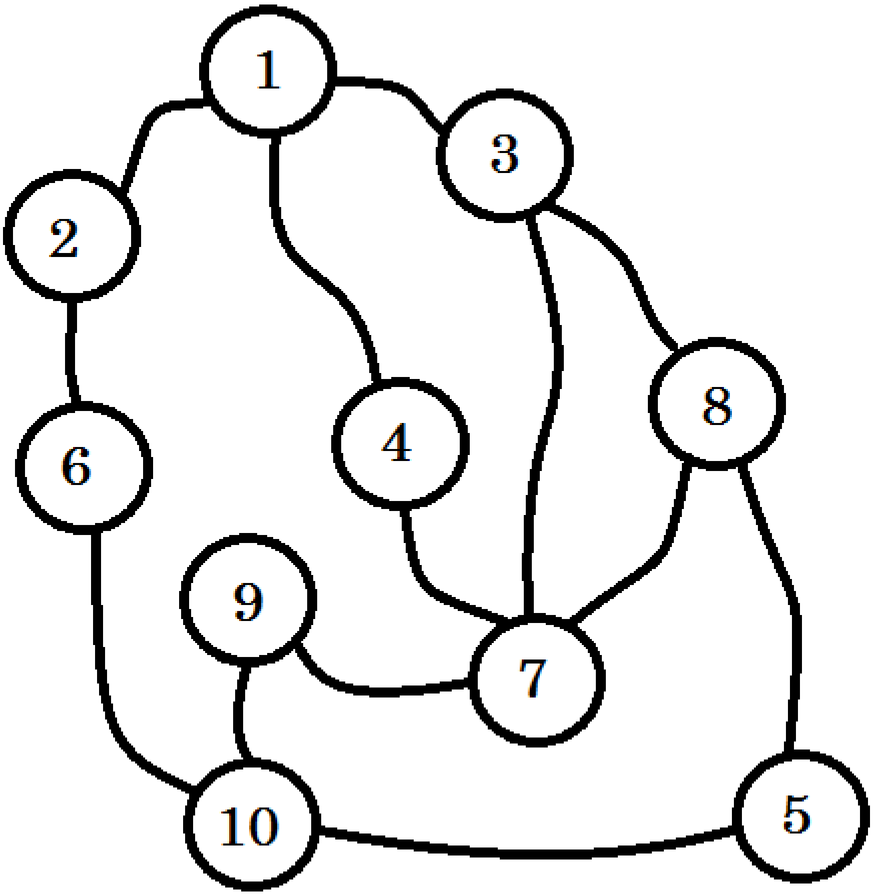
Section 41.1: Finding the Shortest Path from Source to other Nodes

Breadth-first-search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. BFS was invented in the late 1950s by Edward Forrest Moore, who used it to find the shortest path out of a maze and discovered independently by C. Y. Lee as a wire routing algorithm in 1961.

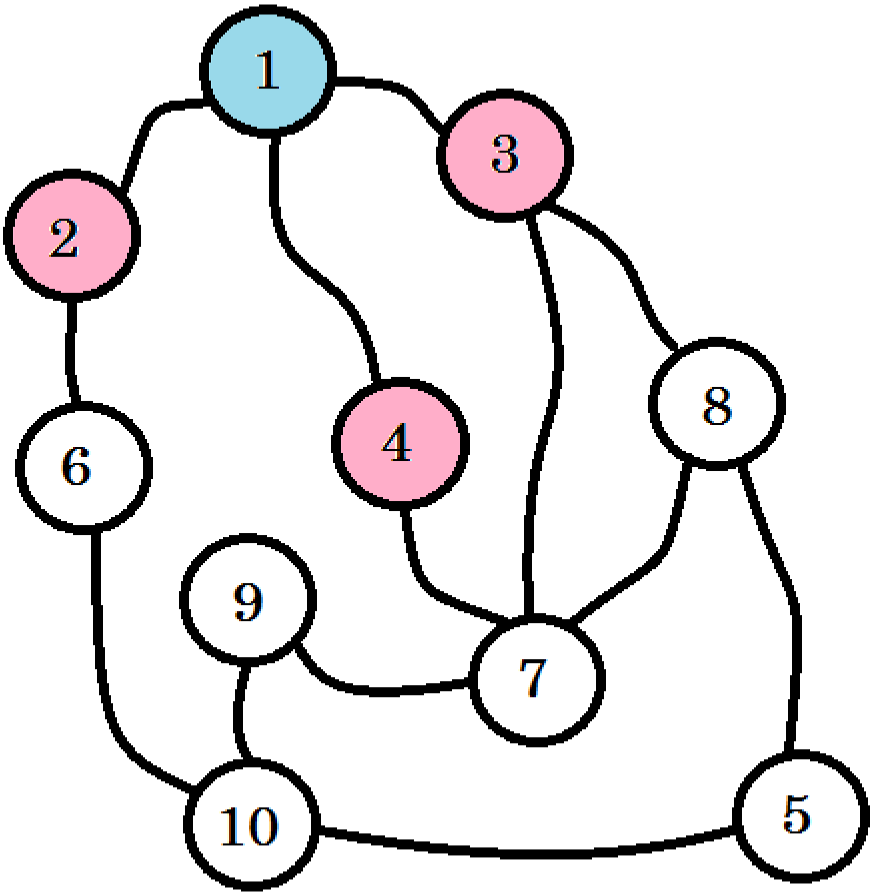
The processes of BFS algorithm works under these assumptions:

1. We won't traverse any node more than once.
2. Source node or the node that we're starting from is situated in level 0.
3. The nodes we can directly reach from source node are level 1 nodes, the nodes we can directly reach from level 1 nodes are level 2 nodes and so on.
4. The level denotes the distance of the shortest path from the source.

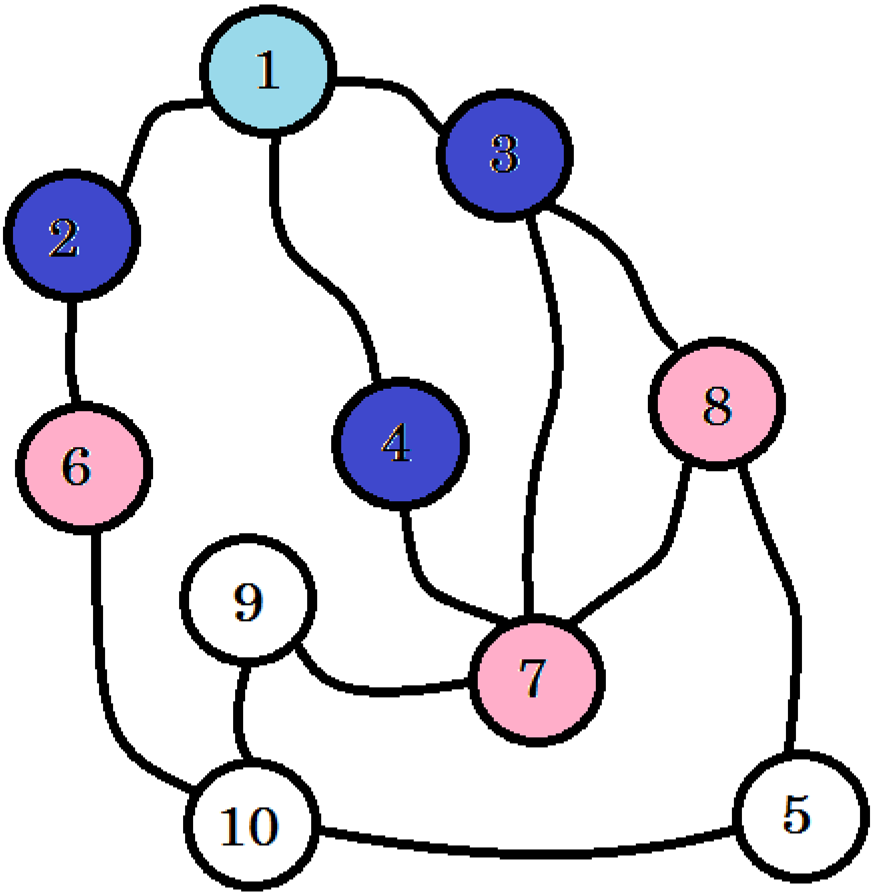
Let's see an example:



Let's assume this graph represents connection between multiple cities, where each node denotes a city and an edge between two nodes denote there is a road linking them. We want to go from **node 1** to **node 10**. So **node 1** is our **source**, which is **level 0**. We mark **node 1** as visited. We can go to **node 2**, **node 3** and **node 4** from here. So they'll be **level (0+1)** = **level 1** nodes. Now we'll mark them as visited and work with them.

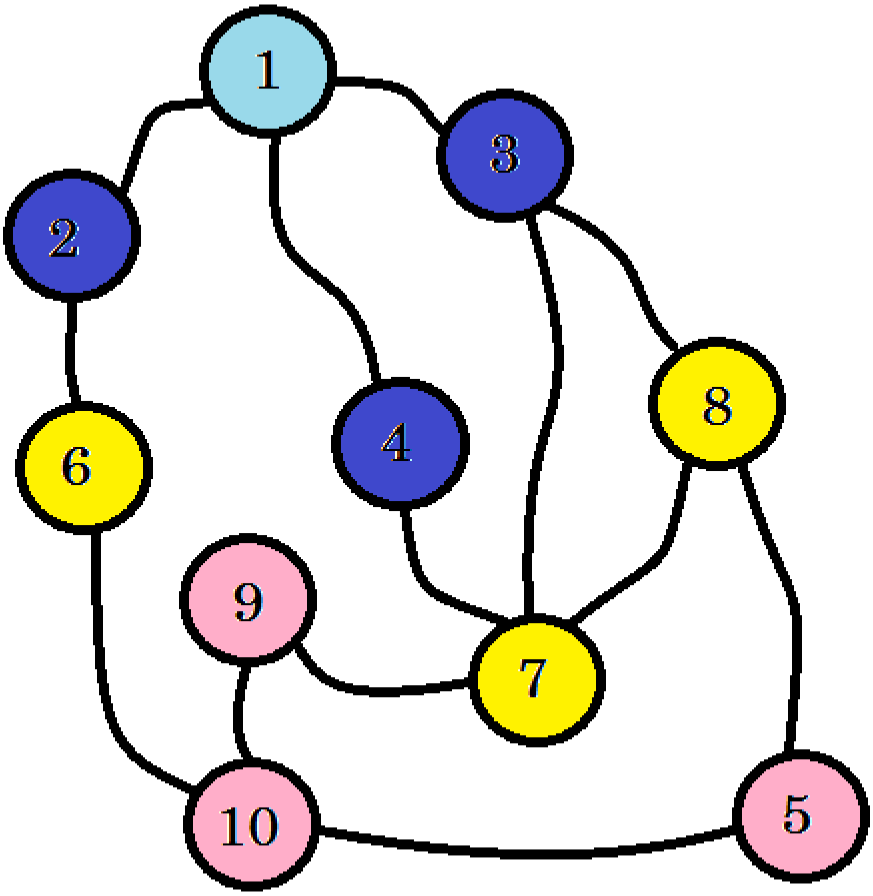


The colored nodes are visited. The nodes that we're currently working with will be marked with pink. We won't visit the same node twice. From **node 2**, **node 3** and **node 4**, we can go to **node 6, node 7** and **node 8**. Let's mark them as visited. The level of these nodes will be **level (1+1)** = **level 2**.

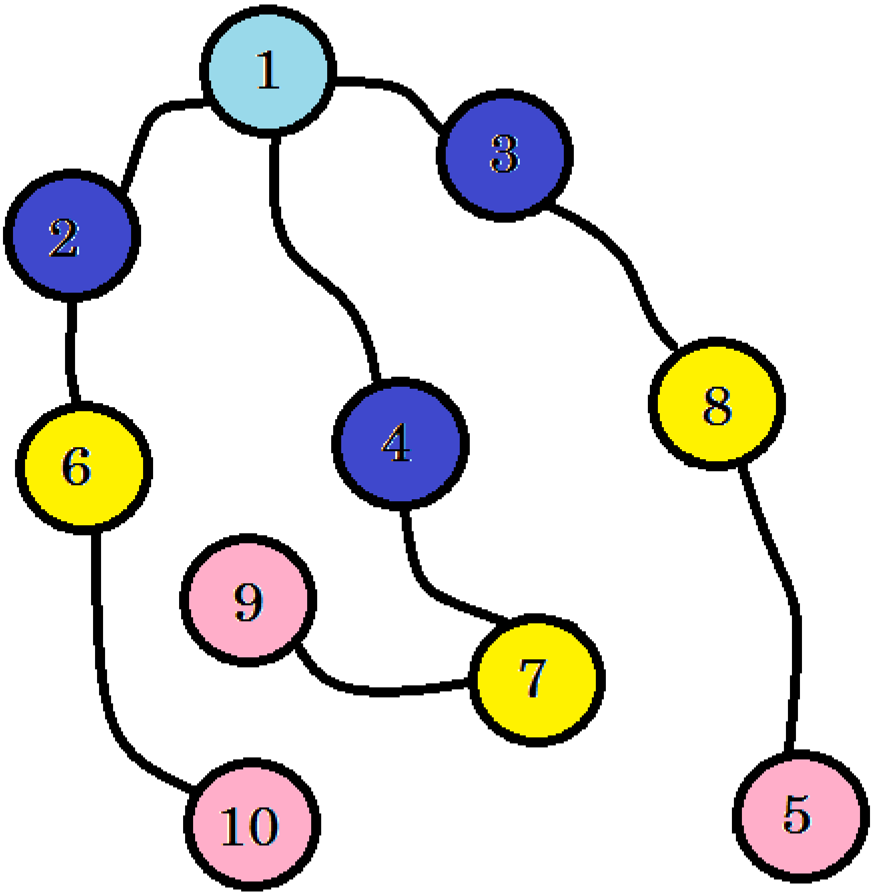


If you haven't noticed, the level of nodes simply denote the shortest path distance from the **source**. For example: we've found **node 8** on **level 2**. So the distance from **source** to **node 8** is **2**.

We didn't yet reach our target node, that is **node 10**. So let's visit the next nodes. we can directly go to from **node 6**, **node 7** and **node 8**.



We can see that, we found **node 10** at **level 3**. So the shortest path from **source** to **node 10** is **3.** We searched the graph level by level and found the shortest path. Now let's erase the edges that we didn't use:



After removing the edges that we didn't use, we get a tree called BFS tree. This tree shows the shortest path from **source** to all other nodes.

So our task will be, to go from **source** to **level 1** nodes. Then from **level 1** to **level 2** nodes and so on until we reach our destination. We can use *queue* to store the nodes that we are going to process. That is, for each node we're going to work with, we'll push all other nodes that can be directly traversed and not yet traversed in the queue.

The simulation of our example:

First we push the source in the queue. Our queue will look like:

```
 front
+-----+
|  1  |
+-----+
```

The level of **node 1** will be 0. **level[1]** = **0**. Now we start our BFS. At first, we pop a node from our queue. We get **node 1**. We can go to **node 4**, **node 3** and **node 2** from this one. We've reached these nodes from **node 1**. So **level[4]** = **level[3]** = **level[2]** = **level[1]** + **1** = **1**. Now we mark them as visited and push them in the queue.

```
                   front
+-----+  +-----+  +-----+
|  2  |  |  3  |  |  4  |
+-----+  +-----+  +-----+
```

Now we pop **node 4** and work with it. We can go to **node 7** from **node 4**. **level[7]** = **level[4]** + **1** = **2**. We mark **node 7** as visited and push it in the queue.

```
                   front
+-----+  +-----+  +-----+
|  7  |  |  2  |  |  3  |
+-----+  +-----+  +-----+
```

From **node 3**, we can go to **node 7** and **node 8**. Since we've already marked **node 7** as visited, we mark **node 8** as visited, we change **level[8]** = **level[3]** + **1** = **2**. We push **node 8** in the queue.

```
                   front
+-----+  +-----+  +-----+
|  6  |  |  7  |  |  2  |
+-----+  +-----+  +-----+
```

This process will continue till we reach our destination or the queue becomes empty. The **level** array will provide us with the distance of the shortest path from **source**. We can initialize **level** array with *infinity* value, which will mark that the nodes are not yet visited. Our pseudo-code will be:

```
Procedure BFS(Graph, source):
Q = queue();
level[] = infinity
level[source] := 0
Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
        end if
    end for
end while
Return level
```

By iterating through the **level** array, we can find out the distance of each node from source. For example: the distance of **node 10** from **source** will be stored in **level[10]**.

Sometimes we might need to print not only the shortest distance, but also the path via which we can go to our destined node from the **source**. For this we need to keep a **parent** array. **parent[source]** will be NULL. For each update in **level** array, we'll simply add `parent[v] := u` in our pseudo code inside the for loop. After finishing BFS, to find the path, we'll traverse back the **parent** array until we reach **source** which will be denoted by NULL value. The pseudo-code will be:

```
Procedure PrintPath(u):  //recursive    |   Procedure PrintPath(u):   //iterative
if parent[u] is not equal to null       |   S =  Stack()
    PrintPath(parent[u])                |   while parent[u] is not equal to null
end if                                  |       S.push(u)
print -> u                              |       u := parent[u]
                                        |   end while
                                        |   while S is not empty
                                        |       print -> S.pop
                                        |   end while
```

Complexity:

We've visited every node once and every edges once. So the complexity will be **O(V + E)** where **V** is the number of nodes and **E** is the number of edges.

Section 41.2: Finding Shortest Path from Source in a 2D graph

Most of the time, we'll need to find out the shortest path from single source to all other nodes or a specific node in a 2D graph. Say for example: we want to find out how many moves are required for a knight to reach a certain square in a chessboard, or we have an array where some cells are blocked, we have to find out the shortest path from one cell to another. We can move only horizontally and vertically. Even diagonal moves can be possible too. For these cases, we can convert the squares or cells in nodes and solve these problems easily using BFS. Now our **visited**, **parent** and **level** will be 2D arrays. For each node, we'll consider all possible moves. To find the distance to a specific node, we'll also check whether we have reached our destination.

There will be one additional thing called direction array. This will simply store the all possible combinations of directions we can go to. Let's say, for horizontal and vertical moves, our direction arrays will be:

```
+----+-----+-----+-----+-----+
| dx |  1  |  -1 |  0  |  0  |
+----+-----+-----+-----+-----+
| dy |  0  |   0 |  1  |  -1 |
+----+-----+-----+-----+-----+
```

Here *dx* represents move in x-axis and *dy* represents move in y-axis. Again this part is optional. You can also write all the possible combinations separately. But it's easier to handle it using direction array. There can be more and even different combinations for diagonal moves or knight moves.

The additional part we need to keep in mind is:

- If any of the cell is blocked, for every possible moves, we'll check if the cell is blocked or not.
- We'll also check if we have gone out of bounds, that is we've crossed the array boundaries.
- The number of rows and columns will be given.

Our pseudo-code will be:

```
Procedure BFS2D(Graph, blocksign, row, column):
for i from 1 to row
    for j from 1 to column
        visited[i][j] := false
    end for
end for
visited[source.x][source.y] := true
level[source.x][source.y] := 0
Q = queue()
Q.push(source)
m := dx.size
while Q is not empty
    top := Q.pop
    for i from 1 to m
        temp.x := top.x + dx[i]
        temp.y := top.y + dy[i]
        if temp is inside the row and column and top doesn't equal to blocksign
            visited[temp.x][temp.y] := true
            level[temp.x][temp.y] := level[top.x][top.y] + 1
            Q.push(temp)
        end if
    end for
end while
Return level
```

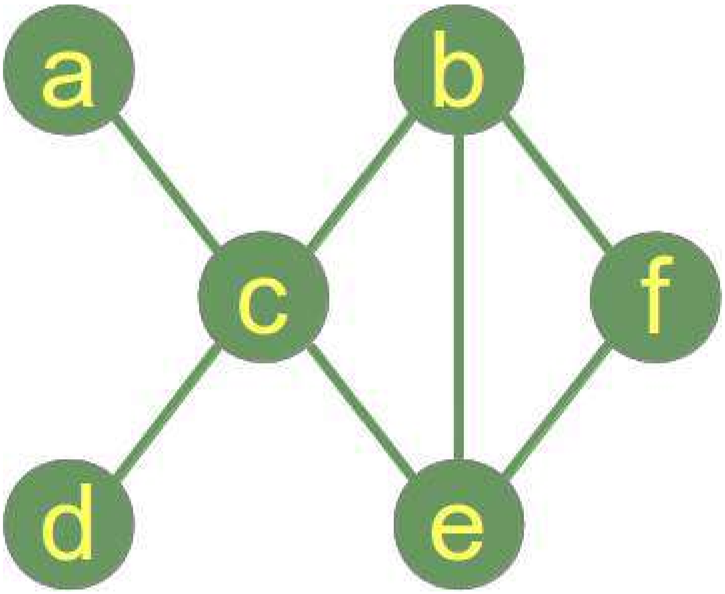
As we have discussed earlier, BFS only works for unweighted graphs. For weighted graphs, we'll need Dijkstra's algorithm. For negative edge cycles, we need Bellman-Ford's algorithm. Again this algorithm is single source shortest path algorithm. If we need to find out distance from each nodes to all other nodes, we'll need Floyd-Warshall's algorithm.

Section 41.3: Connected Components Of Undirected Graph Using BFS

BFS can be used to find the connected components of an undirected graph. We can also find if the given graph is connected or not. Our subsequent discussion assumes we are dealing with undirected graphs.The definition of a connected graph is:

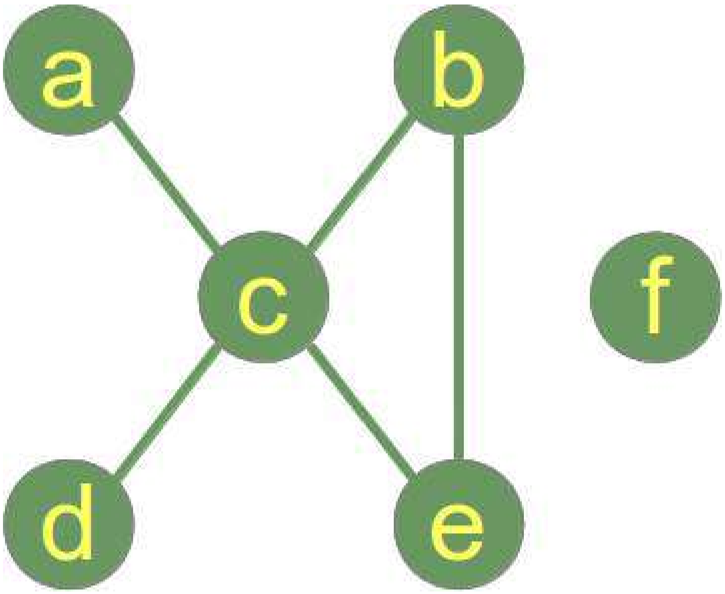
> A graph is connected if there is a path between every pair of vertices.

Following is a **connected graph**.



Following graph is **not connected** and has 2 connected components:

1. Connected Component 1: {a,b,c,d,e}
2. Connected Component 2: {f}



BFS is a graph traversal algorithm. So starting from a random source node, if on termination of algorithm, all nodes are visited, then the graph is connected,otherwise it is not connected.

PseudoCode for the algorithm.

```
boolean isConnected(Graph g)
{
 BFS(v)//v is a random source node.
 if(allVisited(g))
 {
  return true;
 }
 else return false;
}
```

C implementation for finding the whether an undirected graph is connected or not:

```
#include<stdio.h>
#include<stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph,int noOfVertices);
void BFS(char **graph,int vertex,int noOfVertices);
int count = 0;
//Queue node depicts a single Queue element
//It is NOT a graph node.
struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;//array that keeps track of visited vertices.

int main()
```

```
{
    int n,e;//n is number of vertices, e is number of edges.
    int i,j;
    char **graph;//adjacency matrix

    printf("Enter number of vertices:");
    scanf("%d",&n);

    if(n < 0 || n > MAXVERTICES)
    {
     fprintf(stderr, "Please enter a valid positive integer from 1 to %d",MAXVERTICES);
     return -1;
    }

    graph = malloc(n * sizeof(char *));
    visited = malloc(n*sizeof(char));

    for(i = 0;i < n;++i)
    {
        graph[i] = malloc(n*sizeof(int));
        visited[i] = 'N';//initially all vertices are not visited.
        for(j = 0;j < n;++j)
            graph[i][j] = 0;
    }

    printf("enter number of edges and then enter them in pairs:");
    scanf("%d",&e);

    for(i = 0;i < e;++i)
    {
        int u,v;
        scanf("%d%d",&u,&v);
        graph[u-1][v-1] = 1;
        graph[v-1][u-1] = 1;
    }

    if(isConnected(graph,n))
        printf("The graph is connected");
    else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if(Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int deque()
{
    if(Qfront == NULL)
    {
        printf("Q is empty , returning -1\n");
        return -1;
    }
    else
    {
        int v = Qfront->v;
        Nodeptr temp= Qfront;
        if(Qfront == Qrear)
        {
            Qfront = Qfront->next;
            Qrear = NULL;
        }
        else
            Qfront = Qfront->next;

        free(temp);
        return v;
    }
}

int isConnected(char **graph,int noOfVertices)
{
    int i;

    //let random source vertex be vertex 0;
    BFS(graph,0,noOfVertices);

    for(i = 0;i < noOfVertices;++i)
        if(visited[i] == 'N')
         return 0;//0 implies false;

    return 1;//1 implies true;
}

void BFS(char **graph,int v,int noOfVertices)
{
    int i,vertex;
    visited[v] = 'Y';
    enqueue(v);
    while((vertex = deque()) != -1)
    {
        for(i = 0;i < noOfVertices;++i)
            if(graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}
```

For Finding all the Connected components of an undirected graph, we only need to add 2 lines of code to the BFS function. The idea is to call BFS function until all vertices are visited.

The lines to be added are:

```
printf("\nConnected component %d\n",++count);
//count is a global variable initialized to 0
//add this as first line to BFS function
```

AND

```
printf("%d ",vertex+1);
add this as first line of while loop in BFS
```

and we define the following function:

```
void listConnectedComponents(char **graph,int noOfVertices)
{
    int i;
    for(i = 0;i < noOfVertices;++i)
    {
        if(visited[i] == 'N')
            BFS(graph,i,noOfVertices);

    }
}
```

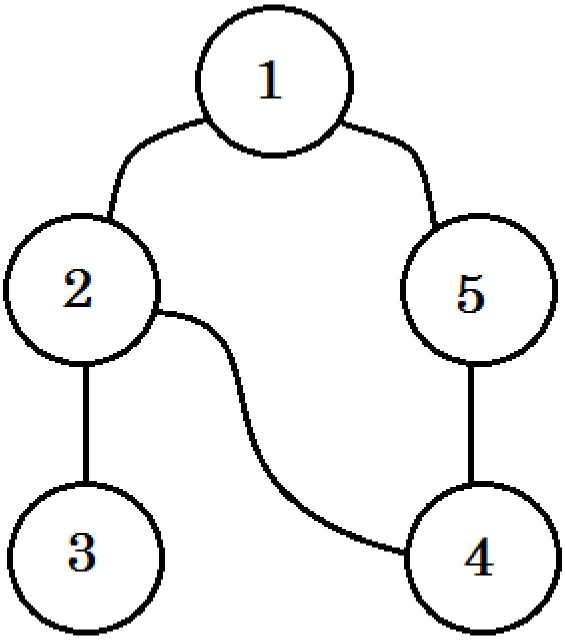
Chapter 42: Depth First Search

Section 42.1: Introduction To Depth-First Search

Depth-first search is an algorithm for traversing or searching tree or graph data structures. One starts at the root and explores as far as possible along each branch before backtracking. A version of depth-first search was investigated in the 19th century French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

Depth-first search is a systematic way to find all the vertices reachable from a source vertex. Like breadth-first search, DFS traverse a connected component of a given graph and defines a spanning tree. The basic idea of depth-first search is methodically exploring every edge. We start over from a different vertices as necessary. As soon as we discover a vertex, DFS starts exploring from it (unlike BFS, which puts a vertex on a queue so that it explores from it later).

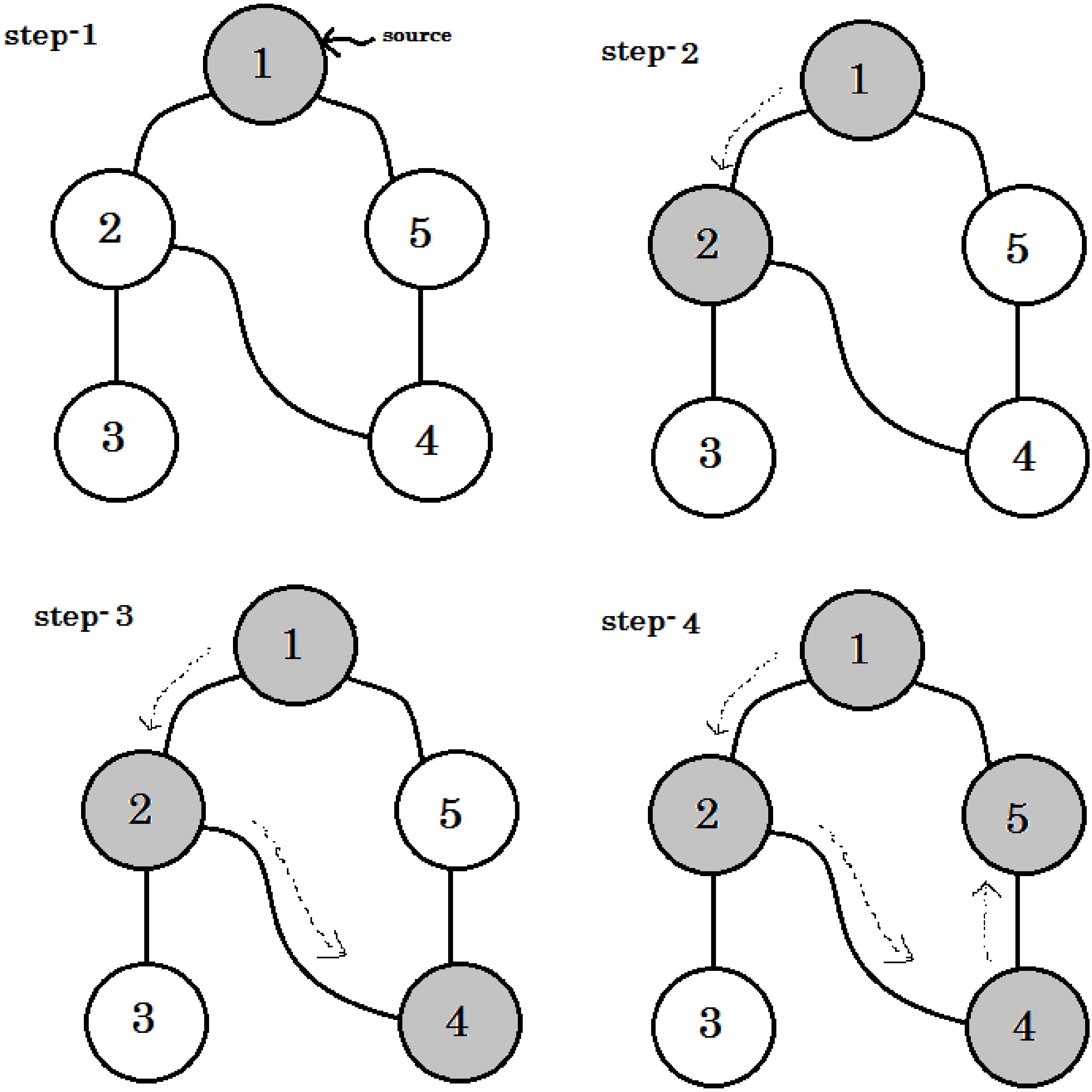
Let's look at an example. We'll traverse this graph:

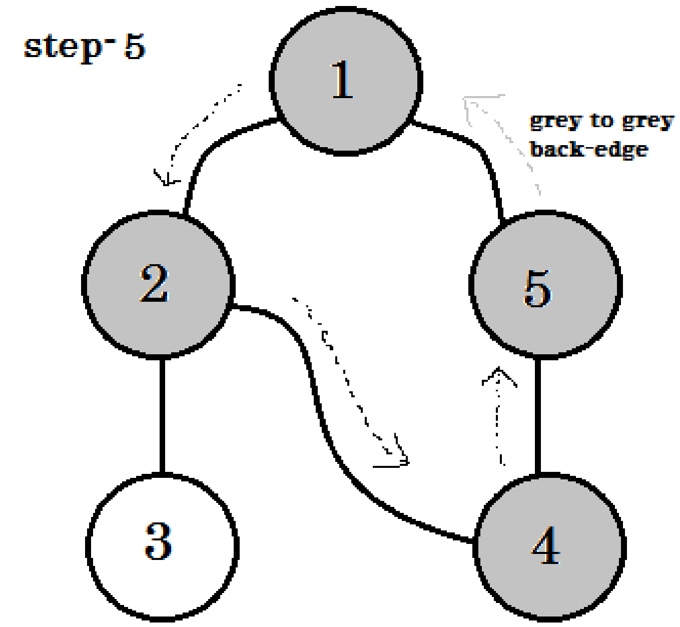


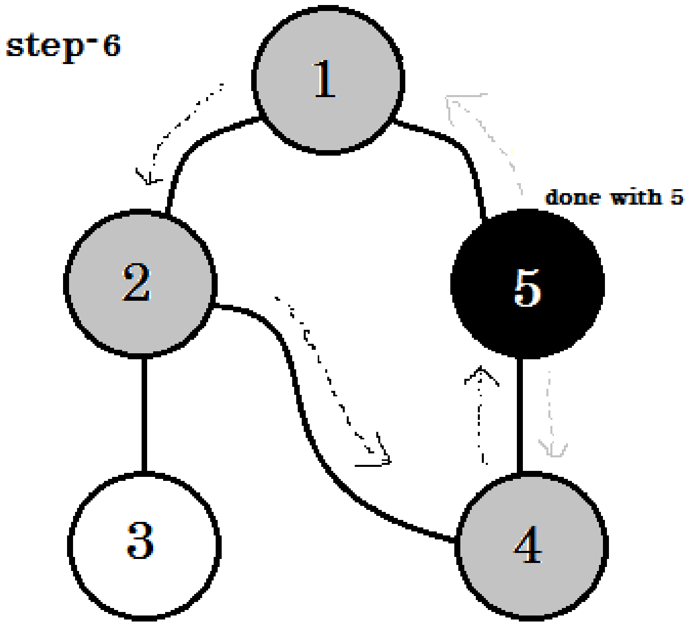
We'll traverse the graph following these rules:

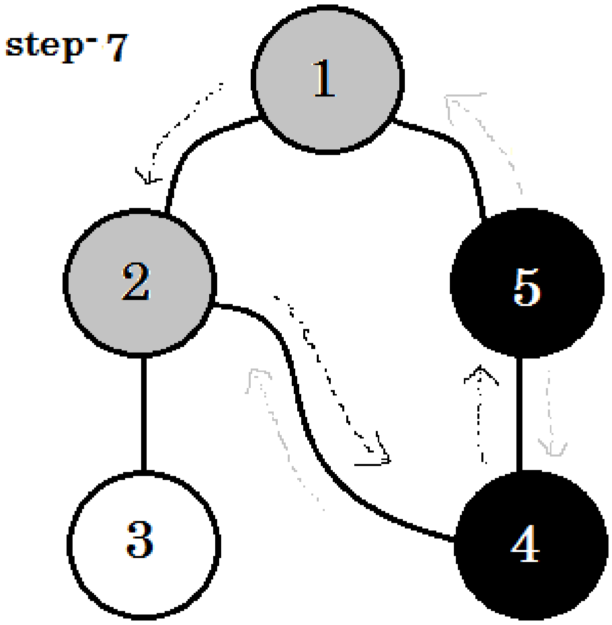
- We'll start from the source.
- No node will be visited twice.
- The nodes we didn't visit yet, will be colored white.
- The node we visited, but didn't visit all of its child nodes, will be colored grey.
- Completely traversed nodes will be colored black.

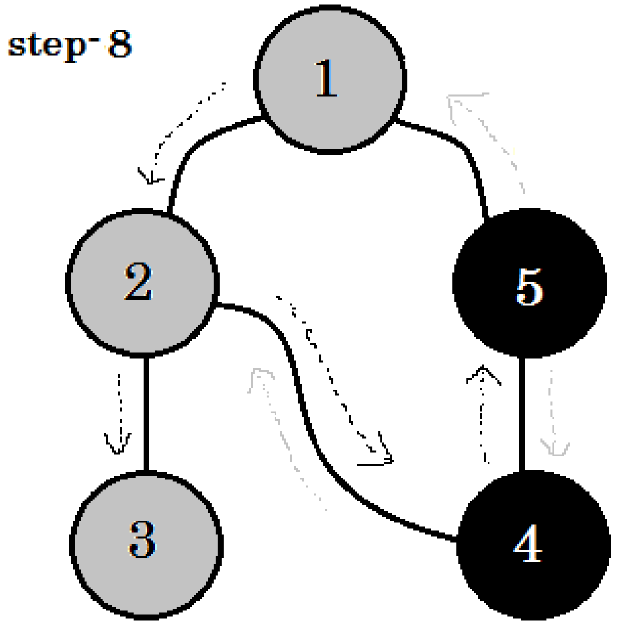
Let's look at it step by step:

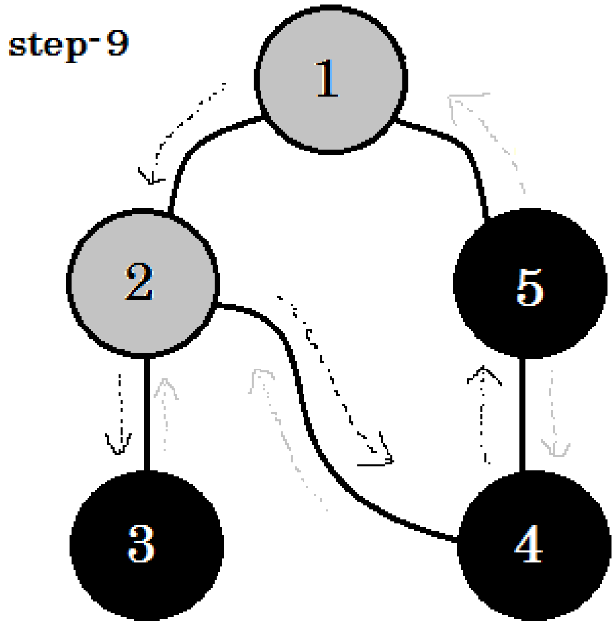

step-1
1
source
2
5
3
4
step-2
1
2
5
3
4
step-3
1
2
5
3
4
step-4
1
2
5
3
4

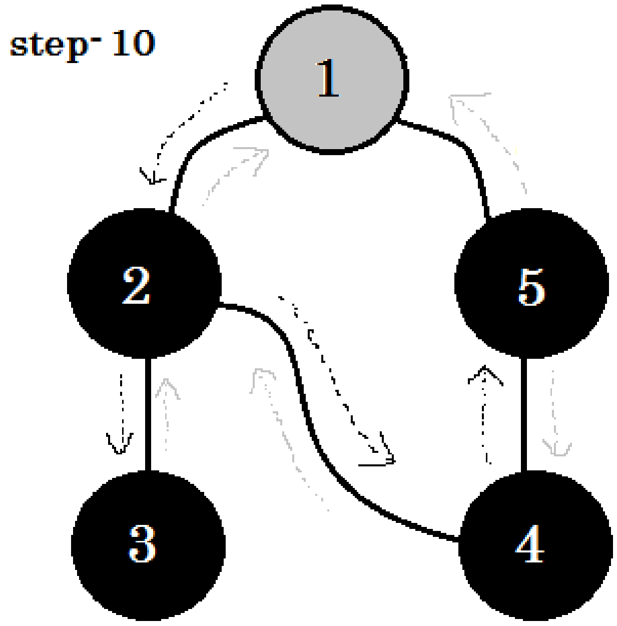

step-5
1
2
5
3
4
grey to grey
back-edge

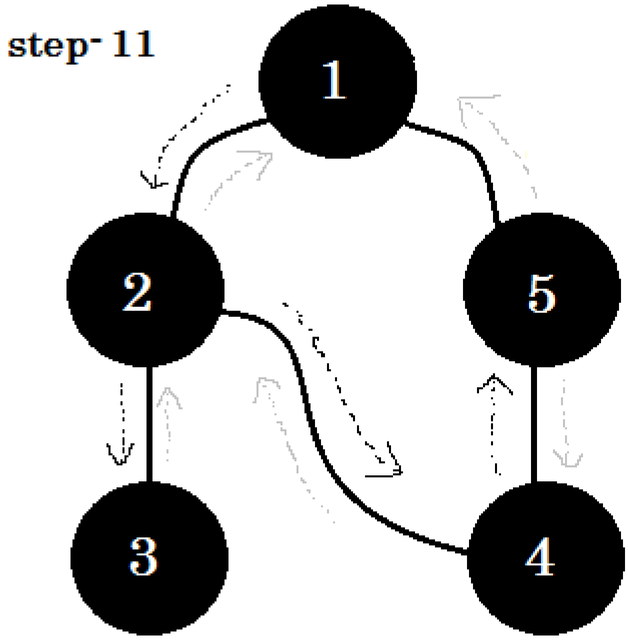

step-6
1
2
5
3
4
done with 5


step-7
1
2
5
3
4


step-8
1
2
5
3
4







We can see one important keyword. That is **backedge**. You can see. **5-1** is called backedge. This is because, we're not yet done with **node-1**, so going from another node to **node-1** means there's a cycle in the graph. In DFS, if we can go from one gray node to another, we can be certain that the graph has a cycle. This is one of the ways of detecting cycle in a graph. Depending on **source** node and the order of the nodes we visit, we can find out any edge in a cycle as **backedge**. For example: if we went to **5** from **1** first, we'd have found out **2-1** as backedge.

The edge that we take to go from gray node to white node are called **tree edge**. If we only keep the **tree edge**'s and remove others, we'll get **DFS tree**.

In undirected graph, if we can visit a already visited node, that must be a **backedge**. But for directed graphs, we must check the colors. *If and only if we can go from one gray node to another gray node, that is called a backedge*.

In DFS, we can also keep timestamps for each node, which can be used in many ways (e.g.: Topological Sort).

1. When a node **v** is changed from white to gray the time is recorded in **d[v]**.

2. When a node **v** is changed from gray to black the time is recorded in **f[v]**.

Here **d[]** means *discovery time* and **f[]** means *finishing time*. Our pesudo-code will look like:

```
Procedure DFS(G):
for each node u in V[G]
    color[u] := white
    parent[u] := NULL
end for
time := 0
for each node u in V[G]
    if color[u] == white
        DFS-Visit(u)
    end if
end for

Procedure DFS-Visit(u):
color[u] := gray
time := time + 1
d[u] := time
for each node v adjacent to u
    if color[v] == white
        parent[v] := u
        DFS-Visit(v)
    end if
end for
color[u] := black
time := time + 1
f[u] := time
```

Complexity:

Each nodes and edges are visited once. So the complexity of DFS is **O(V+E)**, where **V** denotes the number of nodes and **E** denotes the number of edges.

Applications of Depth First Search:

- Finding all pair shortest path in an undirected graph.
- Detecting cycle in a graph.
- Path finding.
- Topological Sort.
- Testing if a graph is bipartite.
- Finding Strongly Connected Component.
- Solving puzzles with one solution.