# CSC2032 Revision Notes

Sahas Talasila

# Contents

## Block I

This block will cover the first two weeks of content, looking at basic concepts needed for algorithm design and analysis

## 1.1 Algorithms Introduction

1. Clear definition of the steps required to solve a particular problem.
2. Not hardwired to a particular programming language, e.g. use pseudo code to describe.
3. Should be abstract:
    - Ignores unimportant details
    - Allows clear presentation of key idea.
    - Simplifies analysis of a proposed solution.
4. Normally straightforward to implement as a program.

### 1.1.1 What Is The Point Of An Algorithm?

To put it simply, it is a way of solving a problem by creating a blueprint for the code.

### 1.1.2 Pseudocode

We can use this template:

```
Algorithm Name
Inputs: List inputs e.g., $A$: Array of Integers; $x$: Real
Returns: Return type
Variables: Local Variables used

Begin
    Code + English
End
```

We have some more control flow examples below:

| **1) Assignments** | **2) Outputs** |
|---|---|
| `age := 31`<br>`age := age + 1` | `display(age, A)` |
| **3) Conditionals** | **4) Loops** |
| `if (BExp) then`<br>`-----`<br>`else`<br>`-----` | `for i := 1 to 100 do`<br>`-----`<br>`while (BExp) do`<br>`-----` |

**Note:** Indentation is used to show code belongs.

**Algorithm 1** Linear Search

```
procedure LinearSearch(array, target)
    for $i = 0$ to length(array) - 1 do
        if array[$i$] = target then
            Return $i$                          ▷ Target found at index $i$
        end if
    end for
    Return -1                                   ▷ Target not found
end procedure
```

# Block II

We will focus on complexity, algorithm performance and performance cases.

## 2.1 Algorithm Performance

Two algorithms for solving a problem may perform very differently. Need to measure performance of different algorithms to allow us to compare them.
Interested in the efficiency of an algorithm:

- **Time**: how fast does the algorithm run;
- **Space**: how much memory (RAM) is required.

We focus on time here (normally seen as the most important nowadays). We normally analyse an algorithm's performance for different sizes of inputs.

Experimental approach based on implementing algorithm and then running tests to measure time and space needed.
This can be problematic as testing possible is limited and results influenced by programming language, compiler, hardware, etc.
Alternative is to use a theoretical approach that calculates approximate performance bounds based on input size.
This is referred to as Complexity Analysis and allows algorithms classified by their efficiency.

### 2.1.1 Big-O Notation

We will mostly look at time complexity. To begin we choose which basic operations in an algorithm we want to count.

- We then analyse the algorithm to derive an expression that relates the number of operations required to the input size.
- **Example**: Consider a simple algorithm for displaying an array of numbers. For an input array of size N, how many display operations are required?

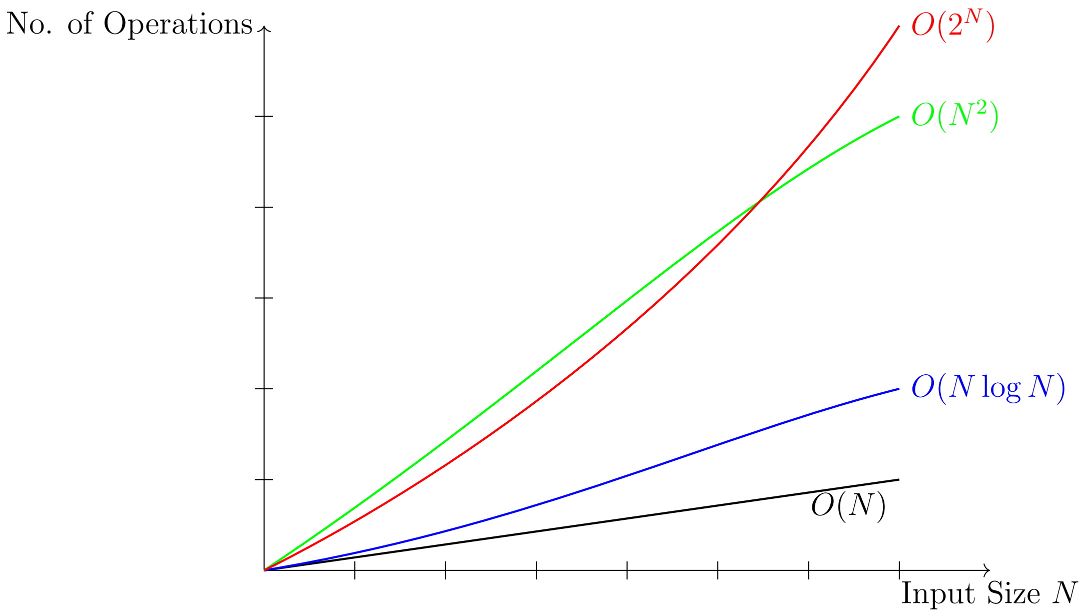
A problem here is that calculating exact performance bounds using this approach can be very difficult.

- Often only require an approximation to the complexity of an algorithm.
- Use Asymptotic Order Notations to approximate how the work required by an algorithm increases as the input size grows to infinity.

- **Example**:
  1. Suppose an algorithm has exact complexity $N^2 - 15$ for input size N
  2. As the input size N grows to infinity subtracting 15 becomes insignificant.
  3. In other words, as $N \to \infty$, $N^2$ and $N^2 - 15$ can be viewed as being approximately the same.
- We use the **Big O** asymptotic order notation which gives an **approximate upper bound** on an algorithm's complexity.
- We write $O(f(N))$ to indicate the approximate number of operations required by an algorithm for input size $N$.
- Here $f(N)$ is an expression (function) which takes the input size $N$ as a parameter.
- Big O only considers **dominant** arithmetic terms as input size N approaches infinity:
- Some common Big O expressions include:

| | | |
|---|---|---|
| $O(1)$ | $O(N \log_2 N)$ | $O(2^N)$ |
| $O(\log_2 N)$ | $O(N^2)$ | $O(N!)$ |
| $O(N)$ | $O(N^3)$ | |

- Consider the following graph of growth rates:



| $N$ | $N^2$ | $N^3$ | $2^N$ |
|---|---|---|---|
| 20 | 400 | 8000 | $10^6$ |
| 40 | 1600 | 64000 | $10^{12}$ |

We can see the change in operation count for each specific complexity.

### 2.1.2 Best, Worst, Average Case Performance

- When analysing an algorithm, consider Big O performance bounds for a range of cases:
  - **best case**: best possible performance?
  - **worst case**: what is the worst performance possible?
  - **average case**: average performance (random inputs)?
- Normally focus on **worst case** analysis as this gives us an upper bound on the time needed by an algorithm.
- **Average case** can be useful but can be hard to calculate and assumes random data.
- **Best case** is seen as least useful but can provide a good indication of when to use a particular algorithm.

- Recall the sequential search algorithm:

**Algorithm seqSearch**
**Inputs:** $k$: Integer; $A$: Array of Integers
**Returns:** Bool
**Variables:** $i$: Integer

**Begin**

```
for i := 0 to size(A) - 1 do
    if A[i] = k then
        return true
return false
End
```

1. **Best case?**
2. **Worst case?**
3. **Average case?**

- Suppose an algorithm consists of two parts performed one after the other.

**Algorithm 1**
$O(f(N))$
**Algorithm 2**
$O(g(N))$

**Sequential Composition**

- What is the overall big O performance?

$$O(f(N)) + O(g(N)) = O(\max(f(N), g(N)))$$

- Suppose an algorithm consists of one algorithm within a loop.

**For (loop $O(f(N))$ times) do:**
**Algorithm 1**
$O(g(N))$

- What is the overall Big-O performance?

$$O(f(N)) \times O(g(N)) = O(f(N) \cdot g(N))$$

## 3.1 Sorting

### 3.1.1 Introduction To Sorting

Sorting is a fundamental problem which involves rearranging a list of objects into ascending order.

- We assume we have an array `A` of integers.
- Array of size `N` starts at position `0` and ends at position `N-1`.

Sorting is an important algorithmic problem:

1. needed in many applications (e.g. presenting ranked results).
2. makes solving other problems easier (e.g. searching for data).

- Studying sorting algorithms provides a great way to introduce important algorithmic concepts.
- Many different sorting algorithms exist and they all have their own advantages and disadvantages.
- We will consider a simple sorting method called **Insertion Sort** which works well on average for small arrays.
- Then consider **Quicksort**, a sorting algorithm based on divide and conquer that works well on larger arrays.

### 3.1.2 Insertion Sort

**Basic idea:** consider elements one at a time, each new element is inserted into its correct position with respect to the previous sorted elements. Let's look at the pseudocode below:

**Algorithm** INSERTIONSORT
**Inputs:** $A$: Array of Integers; $N$: Integer
**Variables:** $i, j,$ key: Integer

**Begin**

```
for $i := 1$ to $N - 1$ do
    key $:= A[i]$
    $j := i$
    while $j > 0$ and key $< A[j-1]$ do
        $A[j] := A[j-1]$
        $j := j - 1$
    end while
    $A[j] :=$ key
end for
```

**End**

We will use something called a 'trace', which helps us visualise the algorithm easily.

$$i = 1 \ \begin{array}{|c|c|c|c|c|}\hline 7 & 3 & 10 & 5 & 8 \\ \hline\end{array}$$

$$i = 1 \ \begin{array}{|c|c|c|c|c|}\hline 7 & 3 & 10 & 5 & 8 \\ \hline\end{array}$$

$$j = 1,\ key = 3 \ \begin{array}{|c|c|c|c|c|}\hline 7 & \text{▨} & 10 & 5 & 8 \\ \hline\end{array} \qquad 3 < 7?\ \text{Yes}$$

$$j = 0,\ key = 3 \ \begin{array}{|c|c|c|c|c|}\hline \text{▨} & 7 & 10 & 5 & 8 \\ \hline\end{array} \qquad j > 0?\ \text{No}$$

---

$$i = 2 \ \begin{array}{|c|c|c|c|c|}\hline 3 & 7 & \boxed{10} & 5 & 8 \\ \hline\end{array}$$

$$j = 2,\ key = 10 \ \begin{array}{|c|c|c|c|c|}\hline 3 & 7 & \text{▨} & 5 & 8 \\ \hline\end{array} \qquad 10 < 7?\ \text{No}$$

$$i = 3 \ \begin{array}{|c|c|c|c|c|}\hline 3 & 7 & 10 & 5 & 8 \\ \hline\end{array}$$

$j = 3,\ key = 5$

| | | | | |
|---|---|---|---|---|
| 3 | 7 | 10 | ▨ | 8 |

$j = 2, 1,\ key = 5$

| | | | | |
|---|---|---|---|---|
| 3 | ▨ | 7 | 10 | 8 |

---

$i = 4$

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | 10 | 8 |

$j = 4,\ key = 8$

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | 10 | ▨ |

$j = 3,\ key = 8$

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | ▨ | 10 |

---

Here is our final, sorted array, which used **Insertion Sort**

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 7 | 8 | 10 |

---

Let's analyse the performance cases:

- Analyse behaviour in terms of the number of comparisons $C_N$ needed for an array of size $N$.
- Always execute the outer loop $N - 1$ times.

**Best case:**

- Array is already sorted.
- Inner loop is never executed, so $C_N = N - 1$.
- Performance is $O(N)$.

**Worst case:**

- Array is in reverse order (i.e., descending).
- Inner loop is executed $i$ times for $i = 1, \ldots, N - 1$.
- So about $C_N = \frac{N^2}{2}$ comparisons.
- Performance is $O(N^2)$.

**Average case:**

- Average behaviour on random data.

- Inner loop is executed $i/2$ times for $i = 1, \ldots, N-1$.
- Approximately $C_N = \frac{N^2}{4}$ comparisons.
- Performance is $O(N^2)$.
- Consider the code for the inner while loop in insertion sort:

```
while $j > 0$ and key $< A[j-1]$ do
    $A[j] := A[j-1]$
    $j := j-1$
```

- Would like to remove the test $j > 0$ from the while loop:
  - This test will rarely be false.
  - Depending on implementation language, could get an array out-of-bounds exception when $j < 1$ for $A[j-1]$.

**Solution:**

- Extend the array by one place.
- Add a **sentinel** in $A[0]$, making it the smallest value possible.
- This will mean that for all $i = 1, \ldots, n$, we have $A[0] \leq A[i]$ and so removes the need for the $j > 0$ test.
- Does this improve the Big O performance of the algorithm?
- Using a **sentinel** (i.e., adding a value to signal an important situation) is a useful algorithmic technique.

### 3.1.3 Quicksort

This is the second form of sorting that we will look at.

Basic idea is as follows:

- Note that when a partition contains only one element then the recursion ends.

1. Choose right most value in array as pivot v.
2. Rearrange (partition) the array so that v is in its correct place, i.e.
   (i) Every element to left is smaller than v.
   (ii) Every element to right is greater than v.
3. Recursively apply above to left hand side and right hand side of array.

**Algorithm** PARTITION
**Inputs:** $A$: Array of Integers; $L, R$: Integer
**Returns:** $pL$: Integer
**Variables:** $pL, pR, v$: Integer

**Begin**

```
$v := A[R]$
$pL := L$, $pR := R$

while ($pL < pR$) do
    while ($A[pL] < v$) do $pL := pL + 1$
    while ($A[pR] \geq v$ and $pR > L$) do $pR := pR - 1$
    if ($pL < pR$) then swap($A[pL], A[pR]$)
swap($A[pL], A[R]$)
return $pL$
```

**End**

## Partitioning the Array

- First step is choose a pivot element
    - For simplicity choose rightmost element.

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 8 | 3 | 10 | 6 | 4 | 9 | 5 |

- Next need to partition array into two sections
    - **Left section:** all elements less than pivot
    - **Right section:** all elements greater or equal to pivot.

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 3 | 8 | 10 | 6 | 7 | 9 | 5 |

- Finally place pivot element in its correct position.

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 3 | 5 | 10 | 6 | 7 | 9 | 8 |

- Partitioning algorithm based on using two pointers: left pointer `pL` and right pointer `pR`.
- Scan from the left using `pL` until an element greater than the pivot element is found.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 8 | 3 | 10 | 6 | 4 | 9 | 5 |

- Scan from the right using `pR` until an element less than the pivot element is found.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 8 | 3 | 10 | 6 | 4 | 9 | 5 |

- Swap these elements over.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 8 | 3 | 10 | 6 | 7 | 9 | 5 |

- Repeat above process until pointers cross:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 8 | 3 | 10 | 6 | 7 | 9 | 5 |

- Finally, swap element at left pointer with pivot value

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 3 | 5 | 10 | 6 | 7 | 9 | 8 |

Here is a high-level trace below:

7 2 8 3 10 6 4 9 5

1) $L = 0, R = 8$: Pivot is 5

7 2 8 3 10 6 4 9 [5]
4 2 3 [5] 10 6 7 9 8

2) $L = 0, R = 2$: Pivot is 3

4 2 [3]
2 [3] 4

3) $L = 4, R = 8$: Pivot is 8

10 6 7 9 [8]
7 6 [8] 9 10

4) $L = 4, R = 5$: Pivot is 6

7 [6]

7

5) $L = 7, R = 8$: Pivot is 10

9 [10]

9 [10]

- Want to calculate the number of comparisons $C_N$ needed for an input array of size $N$.
- Analysis for recursive algorithms is based on using **Recurrence Relations**.
- Recurrence Relations allow us to cope with the complexity of recursive definitions.
- Solving the recurrence relation will give us a bound on the number of comparisons.
- Will consider recurrence relations for QuickSort in worst, best, and average cases.

$$C_0 = C_1 = 1 \quad \text{(Base Case)}$$

$$C_N = \text{Expression involving } C_k \text{ for some } k < N \quad \text{(Recursive Case for } N > 1\text{)}.$$

- Formulate the following **Recurrence Relation** to calculate $C_N$:

$$C_0 = C_1 = 1, \quad C_N = C_{N-1} + N$$

- Partition algorithm has performance $O(N)$: scanning pointers never backtrack, and so the pivot is compared to $N$ other elements till pointers cross.
- By solving the recurrence relation, we derive the worst-case performance to be $O(N^2)$, which is poor.
- Have the following **Recurrence Relation** to calculate $C_N$:

$$C_0 = C_1 = 1, \quad C_N = 2C_{N/2} + N$$

- By solving the recurrence relation, we derive the best-case performance to be $O(N \log_2 N)$.
- If partition is formed at $k$-th element, then partitions are of size $k-1$ and $N-k$.
- **What is the average performance of all possible partitions?**

$$C_0 + C_{N-1}, \quad C_1 + C_{N-2}, \quad C_2 + C_{N-3}, \ldots$$

$$C_{N-2} + C_1, \quad C_{N-1} + C_0$$

- **So the average performance is given by:**

$$C_N = \frac{1}{N}\sum_{k=1}^{N}(C_{k-1} + C_{N-k}) + N$$

- **Recurrence Relation for Average Case:**
  - $C_0 = C_1 = 1$,
  - $C_N = \frac{1}{N}\sum_{k=1}^{N}(C_{k-1} + C_{N-k}) + N$
- Solving this recurrence relation gives:

$$O(N \log_2 N)$$

  as the average-case performance.

- **Formulate following Recurrence Relation** to calculate $C_N$:
  - **Base Case:** $C_0 = C_1 = 1$,
  - **Recursive Case (N ¿ 1):** $C_N = C_{N-1} + N$
- **Partition algorithm:**
  - Performance $O(N)$: scanning pointers never backtrack.
  - Pivot is compared to $N$ other elements until pointers cross.
- By solving the recurrence relation, we derive the worst-case performance as:

$$O(N^2), \text{ which is poor.}$$

- Quicksort is not optimal for sorting small arrays.
- However, since Quicksort is recursive, it will sort many small arrays.
- **Refinement:** Change the termination condition:

```
if (R - L) > few then
    $p :=$ partition$(A, L, R)$
    quickSort(A, L, p-1)
    quickSort(A, p+1, R)
```

- Leaves the array nearly sorted. Then efficiently complete sorting using **Insertion Sort**.
- Experimentation suggests **"few"** is between 5 and 25.

## 3.2 Searching

Key part of most algorithms, databases and nearly everything we use

### 3.2.1 Introduction To Searching

- Searching involves retrieving a piece of information from a large collection of data.
- Fundamental task needed in most computing systems.
- Think of information as being stored as a collection of **records**, each with an associated **key**.

| | |
|---|---|
| **Key1** | **Associated Information** |
| **Key2** | **Associated Information** |
| ⋮ | ⋮ |
| **KeyN** | **Associated Information** |

- Searching involves finding record(s) that match a given search key.
- **Example:** A dictionary in which keys are the words looked up, and the records store their associated meanings.

| | | |
|---|---|---|
| **Student** | **noun** | **a person engaged in studying** |

- A discussion of searching must consider the underlying data structure used.
- In particular, interested in operations for:
  - Setting up the data structure.
  - Inserting records.
  - Deleting records.
- **Note:** We assume that keys are unique (refining algorithms to cope with duplicate keys is normally straightforward).

### 3.2.2 Binary Search

- A search method based on the **divide and conquer** approach.
- Works on data stored in a **sorted array**.
- **Idea:** Compare search key with the key in the middle of the array:

$$m := \frac{(L+R)}{2} \quad \text{(middle key is } A[m]\text{)}.$$

**If search key = middle key** $A[m]$**:** Found record.
**If search key > middle key** $A[m]$**:** Search upper half.
**If search key < middle key** $A[m]$**:** Search lower half.

**Algorithm** *binSearch*
**Inputs:** $A$: Array of Integers; key: Integer; $L, R : Integer$
**Variables:** $m : Integer$
**Returns:** Integer

**Begin**

```
if R < L then return -1
m := (R + L) / 2
if key = A[m] then return m
if key > A[m] then
    return binSearch(A, key, m+1, R)
else
    return binSearch(A, key, L, m-1)
```

**End**

- **Best case:** $O(1)$ (Key found immediately).
- **Worst case:** When you don't find the value in the array.
- Represented by the following recurrence relation:

$$C_0 = C_1 = 1, \quad C_N = C_{\frac{N}{2}} + 2$$

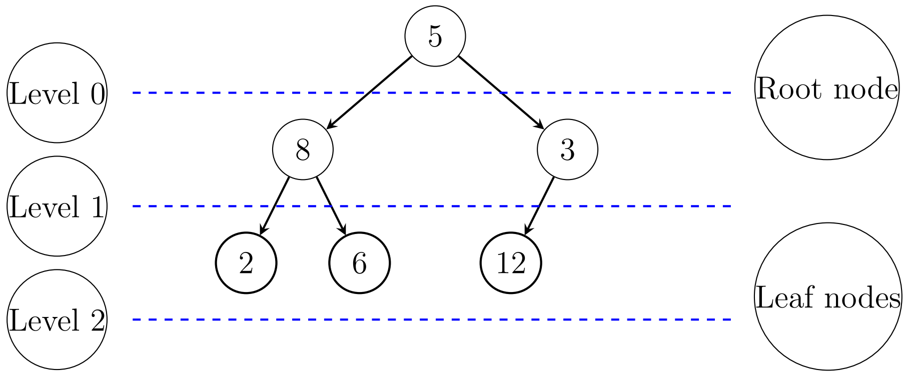
- Solving this tells us that binary search has $O(\log_2 N)$ performance in the worst case (and average case).
- Effective search method if array data remains fixed.
- However, the overhead of keeping the array sorted after insertion can be prohibitive for dynamic data.

### 3.2.3 Binary Search Trees

- A **tree** is a pointer structure consisting of **nodes**, where each node contains data and pointers to other nodes.
- A (rooted) **tree** is characterized by:

- A **root node** which is not pointed to by any other node.
- All other nodes are pointed to by exactly one other node.

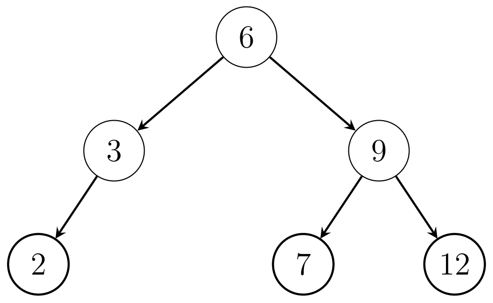
- A tree has a unique path from root to any node in the tree.
- A **binary tree** is one where every node points to at most two other nodes.



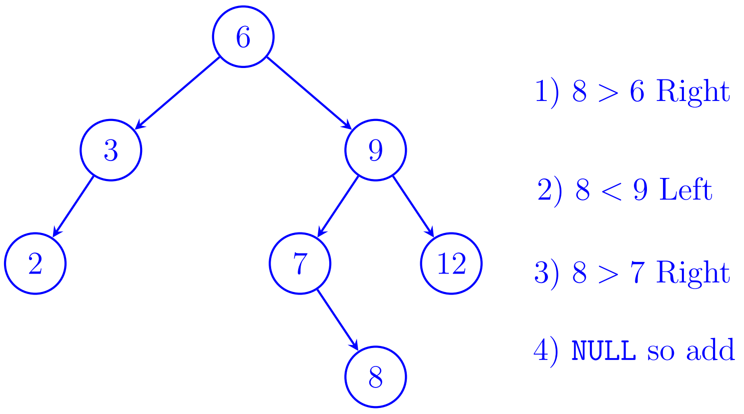
**Binary Search Tree:** Is a binary tree in which at **any** node we have:

- All nodes in left subtree have smaller keys.
- All nodes in right subtree have larger keys.

- **Example:**



- How do we insert a new node into a binary search tree?
  - Recursively move down through the tree.
  - At each node, if the new key is greater than the current node, go right; if it is less, go left.
  - Insert the new key when you reach a `NULL` pointer.
- If the key is already in the tree, then a strategy is needed to cope with this (e.g., abort insertion or overwrite).
- **Example:** Insert 8 into the following binary search tree.



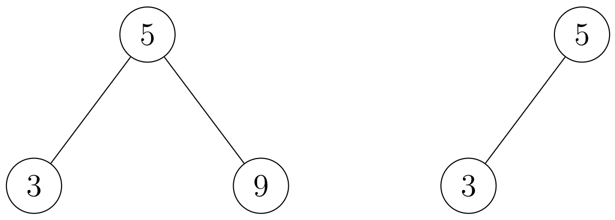
## Finding a Key

**Algorithm:** findBST

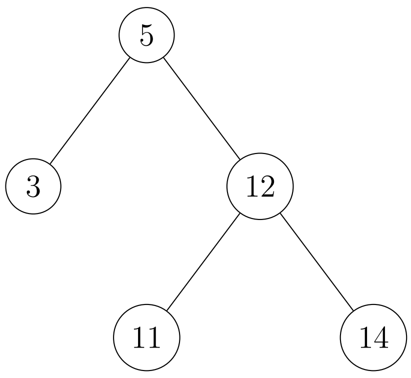
```
Inputs: c: Pointer; key: Integer
Returns: Pointer
Begin:
    if c = NULL then return NULL
    else if c.key() = key then return c
         if key > c.key() then
            return findBST(c.rightPTR(), key)
         else
            return findBST(c.leftPTR(), key)
End
```

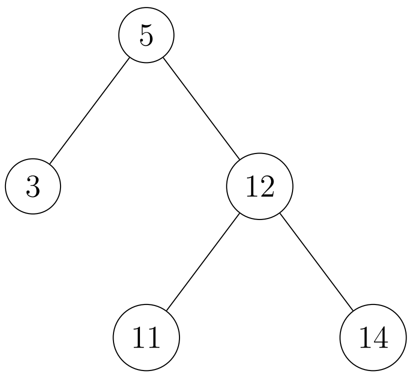
## Node Deletion:

- Deleting a node is the most complex operation since we need to preserve the binary search tree property.
- Consider some examples (deleting node 9):



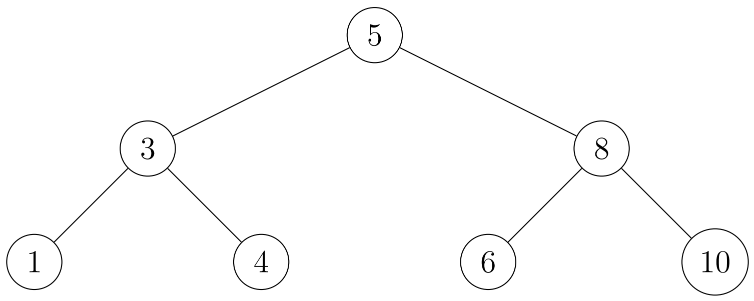
Simple as no subtrees.





Simple as only one subtree.

- Consider deleting node 5 in the following case:

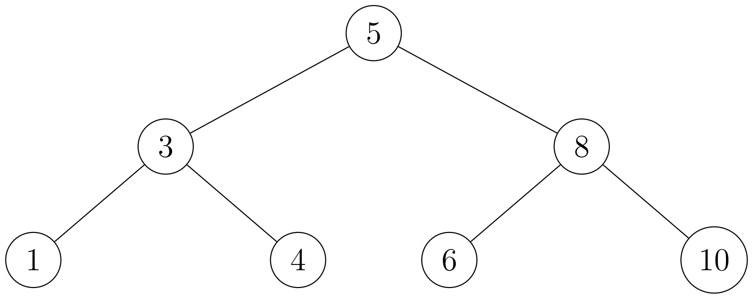


- **Problem:** Node 5 has two subtrees.
- Basic idea is to replace deleted node $D$ with next highest node $N$ (e.g., in the above, this is 6).
- Node $N$'s left link must be `NULL`, so it is easy to delete.
- Alternative approach?

- **Problem:** Node 5 has two subtrees.
- Basic idea is to replace deleted node $D$ with next highest node $N$ (e.g., in the above, this is 6).
- Node $N$'s left link must be `NULL`, so it is easy to delete.
- Alternative approach?

- **Problem:** Node 5 has two subtrees.
- Basic idea is to replace deleted node $D$ with next highest node $N$ (e.g., in the above, this is 6).
- Node $N$'s left link must be `NULL`, so it is easy to delete.
- Alternative approach?

- Delete node 5 in the following case:



- Node 5 has two subtrees, so:
    1. Find the next highest node $N$, i.e., go right, then left as far as you can.
    2. Delete $N$ (simple as it has at most one subtree).
    3. Replace node being deleted by node $N$.
    4. In this case, node 5 is replaced by node 6, deleting the node 6 sub branch.

### 3.2.4 Hashing

Final search technique we consider is called Hashing.

- Based on refining following idea:
- Suppose key values within range 0 to N-1 and use an array of size N to store records.
- Then a key can correspond directly to the array location of its associated record. E.g. record for key 12 is in array location 12.
- Searching and insertion would require only a single array access and thus be O(1).
- Above known as **Perfect Hashing**.
- However, in reality range of keys normally very large compared to the number of actual expected records.
- So perfect hashing is rarely practical to implement

In hashing, we refine the idea of using a **hash function** $H$, which maps a key to its corresponding array address.

$$H(k) = k \mod M$$

For example, given an array of size 11, we can use:

$$H(k) = k \mod 11$$

To insert or find a key in the array, simply apply the hash function to obtain the corresponding address.

Consider an array of size 11 and the hash function:

$$H(k) = k \mod 11$$

$$\text{Array: } [33, 24, 3, \_, 5, \_, 17, 40, 74, \_, 31]$$

- Insert 24: $H(24) = 24 \mod 11 = 2$
- Insert 74: $H(74) = 74 \mod 11 = 8$
- Search for 17: $H(17) = 17 \mod 11 = 6 \rightarrow$ Found
- Search for 37: $H(37) = 37 \mod 11 = 4 \rightarrow$ Not Found

**Problem: Key Collision**

When inserting 47:

$$H(47) = 47 \mod 11 = 3$$

Index 3 is already in use, leading to a **key collision**.

In the best case, hashing requires $O(1)$ comparisons. However, it is normal for a hash function to **not** be one-to-one. That is, two different keys may hash to the same address.

$$H(24) = 2, \quad H(35) = 2$$

This is called a **key collision**. Key collisions are common because the range of possible keys is often much greater than the size of the array.

**Example:** If keys are 2-byte integers (0 to $65,535$) and we only expect 1000 records, collisions are likely.

To reduce key collisions, use a modular hash function with a prime number for the array size $M$:

$$H(k) = \text{int}(k) \mod M$$

For example, if $M$ is a large prime, it reduces the risk of coincidental patterns in keys.

**Why Prime Numbers?**

Using a prime number minimizes patterns in the hashed keys:

$$\text{Array Size } M = 31 \quad \text{(Prime)} \quad \text{vs. } M = 64 \quad \text{(Not Prime)}$$

Linear Probing: When a key collides with a previously inserted key we simply place it in the next available unused location to the right.

To search for a key we:

- Find its position using the hash function.
- Search from here upwards through array looking for key (loop round to start at end of array).
- Search is deemed unsuccessful if we reach an empty location or end up back where we started.

Suppose N is number of values to insert and M is size of array.
• Linear probing works well even if N starts to approach M.
• Been shown that when N=2/3M on average the number of comparisons for an unsuccessful search is just 5 (only 2 for a successful search).
• Thus searching takes approximately constant time O(1) (i.e. independent of N).
• Linear probing also has an advantage: you make use of the spare space within an array (memory efficient!).
• However, deletion is a little problematic with linear probing.

1) How to define a good hash function.
– Need to ensure hash function is reasonably random.
– This means keys will be evenly distributed and so reduces key collisions.
2) Mechanisms for resolving key collisions.
– What happens when two keys hash to same location?
– Lots of approaches, we will consider one to illustrate idea.

When a collision occurs, linear probing checks the next available index.

**Array:** $[11, 41, 3, 36, 5, 17, 29, 58, 20, 32]$

- Insert 33: $H(36) = 3$
- Insert 20: $H(20) = 9$
- Insert 58: $H(58) = 3$ (Collision, move to next index: 4)
- Insert 41: $H(41) = 8$

Deleting a key in linear probing can create search problems. For example:

- Insert 36: $H(36) = 3$
- Insert 58: $H(58) = 3$ (moves to index 4)
- Delete 36: Index 3 becomes empty
- Search for 58: Stops at index 3, leading to failure

**Solutions:**

1. Use a special marker for deleted cells.
2. Rehash all neighboring keys after deletion.

Hashing provides O(1) performance for searching (but does have a theoretical worst case of O(N)).
• Very good method for implementing searches for specific key values (one of the standard methods used for both in-memory and disk-based searching).
• Can be used as a basis for solving many problems.
• However, for more general searches it does have some limitations:
- Cannot search for a value within a given range.
- Cannot find MIN/MAX values stored.
- Cannot retrieve values in sorted order.

## 4.1 String Searching

The problem is to find if one string (the **pattern** $p$) occurs as a substring in a larger string (the **text** $t$).

### Examples

1. $t =$ "trying to catch a train"
   $p =$ "cat" $\rightarrow$ Found!
2. $t =$ "ATTGCGAATGGGACACCTGGT"
   $p =$ "TGG" $\rightarrow$ Found!
   $p =$ "CCA" $\rightarrow$ Not Found.

### Applications

String searching is an important problem in:

- Information retrieval (e.g., Web, Databases)
- Linguistic analysis
- Bioinformatics (DNA analysis)

We can state the string searching problem more formally:

> **String Searching Problem:**
> Given text string $t$ and pattern string $p$, if $p$ occurs as a substring of $t$, then find the first position within $t$ where $p$ occurs.

### Key Properties

- Clearly, $\text{len}(t) \geq \text{len}(p)$.
- Normally, an **alphabet** $\mathcal{A}$ is given, which is the set of symbols the strings can contain, e.g., $\mathcal{A} = \{A, T, C, G\}$ for a DNA alphabet.
- Assume strings start at position 0 (like arrays).

#### Algorithm BruteForce

```
Algorithm BruteForce
Inputs t, p: String
Variables m, i: Integer
Returns Integer
```

```
Begin
    for m := 0 to (len(t) - len(p)) do
        i := 0
        while (i < len(p) and t[m+i] = p[i]) do
            i := i + 1
        if (i = len(p)) then return m
    return -1
End
```

The worst-case behavior for this algorithm on strings $t$ and $p$ is:

$$O(\text{len}(t) \cdot \text{len}(p))$$

## Example

$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & & & \end{array}$$

This requires $(\text{len}(t) - \text{len}(p) + 1) \times \text{len}(p)$ comparisons before the final match is found.

## Performance in Practice

- For large alphabets, the substring comparison should fail early (e.g., using the English alphabet).
- Expected performance: $O(\text{len}(t) + \text{len}(p))$ in most practical cases.

**Note:** The algorithm requires the text string to be buffered to allow backtracking after an unsuccessful match.

## The Problem

In this example, we are going to construct the **next table** for the string `ABCABD`, with each step explained in words and visualized.

## Entry 0 (Trivial)

Entry 0 (Trivial)

The first entry in the next table is always $-1$, regardless of the string.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Finding Entry 1

| Text | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |
|---|---|---|---|---|---|---|
| Pattern | $A$ | $B$ | | | | |

**Explanation**

We check if any information, if comparisons succeeded up to position 1 and then failed, would prevent a new comparison starting at position 0. It does not. Thus, we add 0 to our next table.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

## Finding Entry 2

| Text | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |
|---|---|---|---|---|---|---|
| Pattern | $A$ | $B$ | $C$ | | | |

**Explanation**

At position 0, the string would need to be $B$ to match, but it is $A$. Thus, we add 0 to the next table.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | 0 |
| 3 | |
| 4 | |
| 5 | |

## Finding Entry 3

| Text | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |
|---|---|---|---|---|---|---|
| Pattern | $A$ | $B$ | $C$ | $A$ | | |

Explanation

Comparisons fail across all possible placements. Thus, we add $-1$ to the next table.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | 0 |
| 3 | $-1$ |
| 4 | |
| 5 | |

## Finding Entry 4

| Text | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |
|---|---|---|---|---|---|---|
| Pattern | $A$ | $B$ | $C$ | $A$ | $B$ | |

Explanation

Comparisons succeed, so we add 0 to the next table.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | 0 |
| 3 | $-1$ |
| 4 | 0 |
| 5 | |

### Finding Entry 5

| Text | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |
|---|---|---|---|---|---|---|
| Pattern | $A$ | $B$ | $C$ | $A$ | $B$ | $D$ |

**Explanation**

The comparison succeeds across all positions. The final match aligns position 2 in the pattern string. Add 2 to the next table.

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | 0 |
| 3 | $-1$ |
| 4 | 0 |
| 5 | 2 |

## Final Next Table

| Position | Next |
|---|---|
| 0 | $-1$ |
| 1 | 0 |
| 2 | 0 |
| 3 | $-1$ |
| 4 | 0 |
| 5 | 2 |

This completes the construction of the next table for the string `ABCABD`.

Idea is when we have a mismatch we use the symbol in the text string that caused this to calculate a skip forward in the text string (i.e. shift text pointer). • So using the pattern string we calculate a skip value for each symbol in the alphabet we are using.
• Implement using a skip table from symbols to skip values.
• Combine this with a right-to-left next table which is similar to the one used in Knuth-Morris-Pratt.
• Algorithm either uses the skip or next value depending on which gives the greater shift.
• This produces a very effective algorithm as we'll now see in the analysis.

Boyer-Moore Analysis
• Generally, if we have a large alphabet and a small pattern string then we get a very good performance of O(len(t)/len(p)).
• This is because a mismatch here will allow us to use the skip table to disregard len(p) symbols.
• Note with this algorithm we don't need to look at every position in the text string to find a match.
• Better performance if we have a long, repetitive pattern, why?
• Worst case comparison is again O(len(t)+len(p)).
• Note that using this algorithm we do need to buffer the text string as we move back and forth in it.

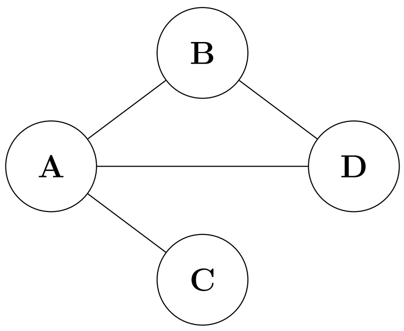
## 4.2 Graph Algorithms

Often want to model and analyse a collection of objects and the connections/relationships between them.
• Graphs provide a simple mathematical approach for doing this and are therefore very important in computing.
• A graph consists of a collection of objects (**nodes**) and connections between them (**edges**)

**Definition:** An **undirected graph** $G = (V, E)$ consists of:

- A set $V$ of nodes (objects we are interested in).
- A set $E$ of edges representing connections between nodes.

**Example:**



$$G = (V, E) \quad \text{where:}$$

$$V = \{A, B, C, D\}, \quad E = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}\}$$

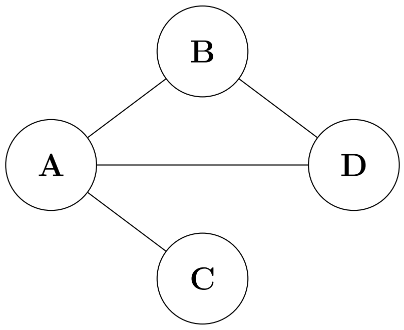
**Note:** Edges do not have direction, so we use a **set** of two nodes to represent an edge.

**Definition:** A **path** in a graph $G = (V, E)$ is a sequence of nodes:

$$(n_1, n_2, \ldots, n_k) \quad \text{such that } n_1, \ldots, n_k \in V \text{ and } \{n_i, n_{i+1}\} \in E \text{ for } i = 1, \ldots, k-1.$$

**Length of a Path:** The number of edges in the path (i.e., one less than the number of nodes).

**Example:**



Example Paths: $(C, A, B, D)$, $(A, D, B, A, C)$

**Procedure:**

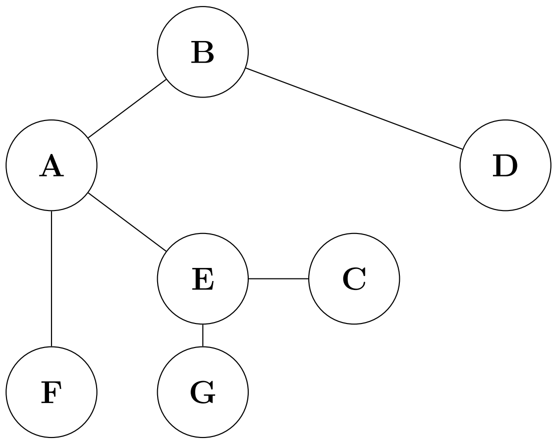
- Go as far as possible along a single path from a given node before backtracking to consider other paths.
- Once a node is visited, it is marked and not revisited.

## Depth-First Search (DFS)

**Procedure:**

- Go as far as possible along a single path from a given node before backtracking to consider other paths.
- Once a node is visited, it is marked as visited and not revisited.

**Example:**



**Trace:**

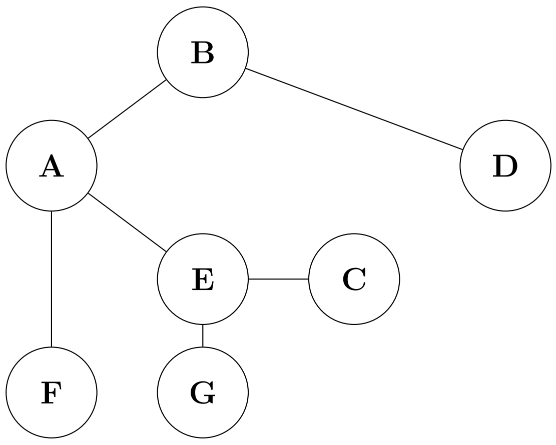
**A**
**B**
**C**
**D**
**E**
**F**
**G**

## Breadth-First Search (BFS)

**Procedure:**

- Explore all paths one step at a time from a given node before considering nodes further away.
- Visit all nodes at the same level before moving deeper.

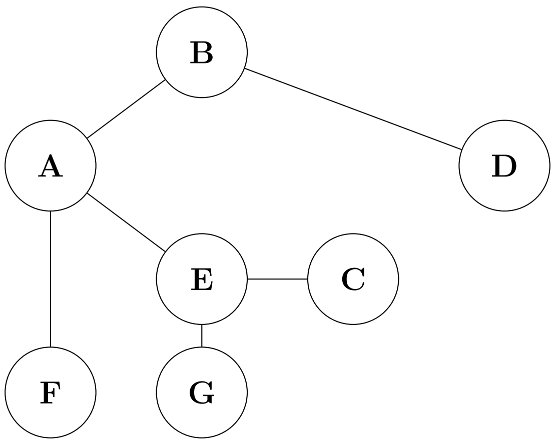
**Example:**



**Trace:**

**A**
**B**
**E**
**F**
**C**
**D**
**G**

## Comparison of DFS and BFS Traces



| **Depth-First Trace** | **Breadth-First Trace** |
|---|---|
| 1. A | 1. A |
| 2. B | 2. B |
| 3. C | 3. E |
| 4. D | 4. F |
| 5. E | 5. C |
| 6. F | 6. D |
| 7. G | 7. G |

Often want to traverse all nodes and edges in a graph.

• There are different approaches to traversing a graph but we will consider two important algorithms:

– Depth-first search: follows a path as far as it can before backtracking one node at a time.

– Breadth-first search: considers all the neighbor nodes one step away, then all two steps away, etc.

• These two algorithms are referred to as exhaustive search algorithms as they visit all nodes and edges in a graph.

• They are important algorithms for solving problems such as shortest path, checking for cycles, etc.

• Note: we always use alphabetical order when considering nodes.

A common problem that arises with networks is to connect together a group of objects in the most efficient way.
• Examples of this problem can be found in applications such as communication, transportation and electrical circuits.
• Analyzing this problem is also useful in a number of scientific areas such as biology, sociology, etc.
• This problem can be formulated using graph theory where it is referred to as the Minimal Spanning Tree (MST) Problem.
• We introduce MST problem in this section and then consider Prim's Algorithm, a greedy based algorithm for solving it.

**Prim's Algorithm** is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a weighted, connected, and undirected graph. It builds the MST by starting with an arbitrary vertex and repeatedly adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

—

## What is a Minimum Spanning Tree (MST)?

A **Minimum Spanning Tree (MST)** of a graph $G = (V, E)$:

- Is a subgraph that includes all vertices $V$.
- Contains exactly $|V| - 1$ edges (no cycles).
- Minimizes the sum of the edge weights.

The MST is unique if all edge weights are distinct.

—

## Steps of Prim's Algorithm

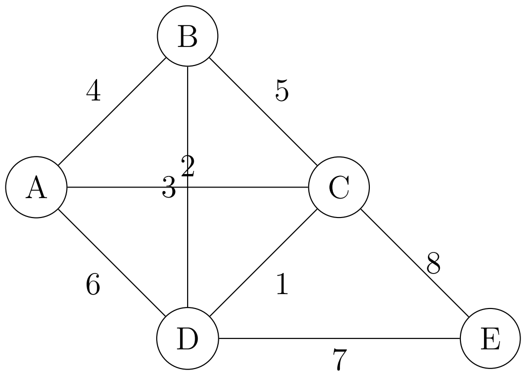
**Algorithm Steps**

1. Start with an arbitrary vertex and include it in the MST. 2. Repeatedly add the smallest edge that connects a vertex in the MST to a vertex outside the MST. 3. Stop when all vertices are included in the MST.

—

## Example of Prim's Algorithm

We will apply Prim's Algorithm to the following graph:

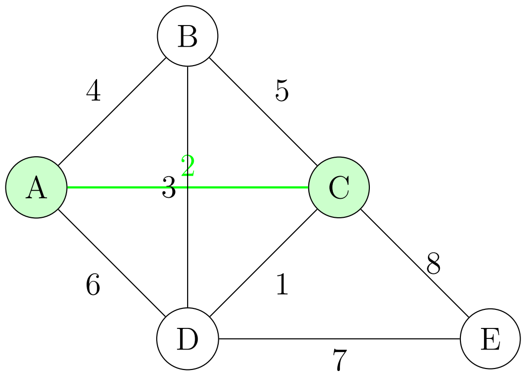


—

## Step-by-Step Execution

### Step 1: Start with Vertex A

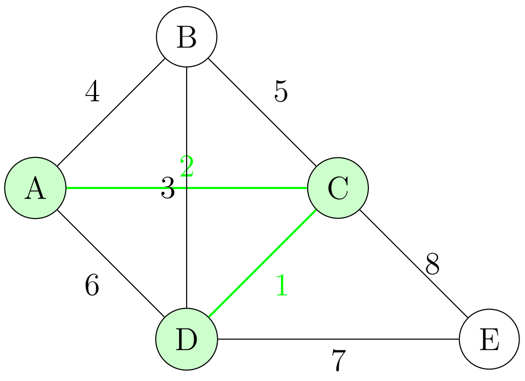
- Start with $A$. - Select the smallest edge connected to $A$: $A \to C$ (weight 2).



—

### Step 2: Add the Smallest Edge from the Tree

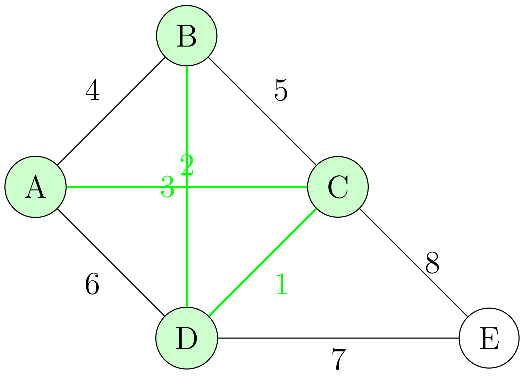
- Vertices in the MST: $\{A, C\}$. - Select the smallest edge connected to the MST: $C \to D$ (weight 1).



—

**Step 3: Add the Next Smallest Edge**

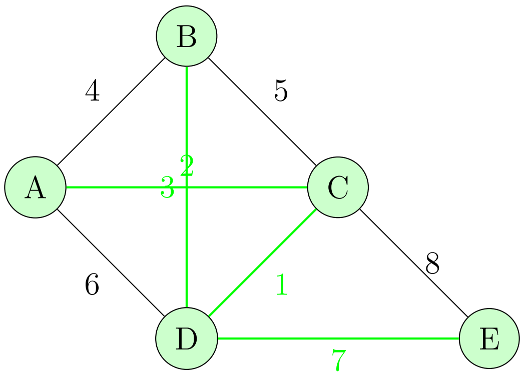
- Vertices in the MST: $\{A, C, D\}$. - Select the smallest edge connected to the MST: $D \rightarrow B$ (weight 3).



—

**Step 4: Add the Final Edge**

- Vertices in the MST: $\{A, B, C, D\}$. - Select the smallest edge connected to the MST: $D \rightarrow E$ (weight 7).



—

## Final Minimum Spanning Tree (MST)

The edges in the MST are:

$$\{A \rightarrow C, C \rightarrow D, D \rightarrow B, D \rightarrow E\}$$

with a total weight of:

$$2 + 1 + 3 + 7 = 13$$

- Prim's Algorithm always constructs the **Minimum Spanning Tree (MST)**. This can be shown by induction on the subtrees produced.
- A simple version of Prim's Algorithm would have a performance of:

$$O(|V|^2)$$

- However, you can improve this by being careful about the data structures used for representing the graph and performing the search for the next minimal edge.
- We end up with $|V| - 1$ deletions and $|E|$ verifications, each of which costs $\log |V|$.
- This gives:

$$((|V| - 1) + |E|) \times \log |V|$$

- Since in a connected graph $(|V| - 1) \leq |E|$, we get the following simplified performance:

$$O(|E| \log |V|)$$

## Block V

## 5.1 Problem Classifications

- Interesting to consider how difficult a problem is to solve.
- To do this, we classify problems by considering what we know about the algorithms for solving them.

### Easy Problems

- Have efficient algorithms with *polynomial* performance (e.g., $O(N)$, $O(N^2)$, $O(N^3)$).
- Can solve large instances of the problem.

### Hard Problems

- Only known algorithms have non-polynomial performance (e.g., $O(2^N)$, $O(N!)$).
- Only small instances of such problems can be solved.

—

## Decision Problems

- Focus on problems that have a **yes/no** answer, called *decision problems*.
- Examples:
  - Does an array contain a given value?
  - Is a number prime?
  - Does a graph have a cycle?
- Many general problems can be converted to an equivalent decision problem. For example:

**Example**

Find Shortest Path in Graph $\rightarrow$ Does a Path exist of size $K$?

—

## Subset Sum Problem

**Subset Sum Problem:** Given a set $\{i_1, \ldots, i_N\}$ of $N$ integers, does it contain a (non-empty) subset that sums to 0?

- **Example:** Consider applying the subset sum problem to the set $\{2, -5, 7, -1, 3\}$.
- Answer?
- **Question:** How difficult is the subset sum problem to solve?

—

## Classifying Problems

- Notions of **easy** and **hard** problems are formally classified in complexity theory.
- The following are some of the key classes of problems:
  - $P$: Polynomial time.
  - $NP$: Non-deterministic, Polynomial time.
  - $NP$ **Complete**: Hardest problems in $NP$.
  - $NP$ **Hard**: General problems at least as hard as $NP$ complete problems.
  - **Undecidable**: Problems that cannot be solved.

—

## P Class of Decision Problems

- We start by focusing on **easy** decision problems.
- The class $P$ represents all decision problems with algorithms that have polynomial time.
- Examples:
  - Is an array sorted?
  - Is a given value in an array?
  - Does a spanning tree with cost less than $K$ exist for a graph?
  - Does a string contain a given substring?

—

## NP Class of Decision Problems

- $NP$ stands for **Non-deterministic Polynomial time**.
- Decision problems in $NP$ satisfy:
  - **Non-deterministic**: Possible solutions can be guessed.

  - **Polynomial time**: A solution can be checked in polynomial time.
- **Easy to Check**: For example, checking a solution for the subset sum problem.
- Most researchers believe $P \neq NP$.

—

## Relationship Between $P$ and $NP$

- By definition, all decision problems in $P$ must also be in $NP$ ($P \subseteq NP$).
- $NP$ contains problems with no known polynomial algorithm, such as the subset sum problem.
- Important question: Does $P = NP$?
- So far, it has not been proven whether $P = NP$ or $P \neq NP$.

—

## NP Complete Problems

- Hardest problems in $NP$ are called **NP Complete**.
- A problem is $NP$ Complete if:
  - All problems in $NP$ can be reduced to it in polynomial time.
- Example: The subset sum problem is $NP$ Complete.

—

## NP Hard Problems

- Problem classes $P$ and $NP$ only cover decision problems.
- General problems that are as hard to solve are called $NP$ Hard.
- All $NP$ Complete problems are also $NP$ Hard.

—

## Undecidable Problems

- Some problems cannot be solved by any algorithm, e.g., the **Halting Problem**.
- Such problems are said to be **undecidable**.

—

## Traveling Salesperson Problem (TSP)

**Definition:** Given $N$ cities and the distance between each pair of cities, find the smallest distance required to visit all cities once and return to the starting city.

- Interesting optimization problem extensively studied.
- Example: Consider five cities $A, B, C, D, E$ with the following distances:

**Distance Matrix**

$A \to B = 10, B \to C = 8, \ldots$

—

## Approximation Algorithms

- Approximation algorithms are used to deal with hard problems.
- Example: The nearest neighbor algorithm for TSP.
- Performance: $O(N^2)$ for $N$ cities.

## Nearest Neighbour Algorithm for TSP

### Algorithm Steps

1. Start at an arbitrary city.
2. Find the nearest unvisited city and travel to it.
3. Mark the visited city as "visited."
4. Repeat until all cities are visited.
5. Return to the starting city.

This is a **greedy algorithm** and does not guarantee an optimal solution.

—

### Example

We are given the following cities and distances:

| City Pair | Distance |
|---|---|
| $A \to B$ | 10 |
| $A \to C$ | 5 |
| $A \to D$ | 15 |
| $A \to E$ | 9 |
| $B \to C$ | 8 |
| $B \to D$ | 3 |
| $B \to E$ | 7 |
| $C \to D$ | 12 |
| $C \to E$ | 6 |
| $D \to E$ | 2 |

—

## Step-by-Step Solution

1. **Start at City $A$:**

   Nearest city to $A$ is $C$ (distance: 5).

2. **Move to City $C$:**

   Nearest unvisited city to $C$ is $E$ (distance: 6).

3. **Move to City $E$:**

   Nearest unvisited city to $E$ is $D$ (distance: 2).

4. **Move to City $D$:**

   Nearest unvisited city to $D$ is $B$ (distance: 3).

5. **Move to City $B$:**

   Return to the starting city $A$ (distance: 10).

—

## Total Distance

The path is:

$$A \to C \to E \to D \to B \to A$$

with a total distance of:

$$5 + 6 + 2 + 3 + 10 = 26$$

I have included Python implementations of all of the algorithms and data structures we have used.

## 6.1 Python Implementations

### 6.1.1 Insertion Sort

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def main():
    arr = [12, 11, 13, 5, 6]
    print("Original array:", arr)
    insertion_sort(arr)
    print("Sorted array:", arr)

if __name__ == "__main__":
    main()
```

Listing 1: Insertion Sort (Python)

### 6.1.2 Quicksort

```python
def partition(arr, low, high):
    pivot = arr[high]  # Choose the last element as the pivot
    i = low - 1  # Index of smaller element

    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]  # Swap elements

    arr[i + 1], arr[high] = arr[high], arr[i + 1]  # Swap pivot to its
   correct position
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)  # Partition the array
        quick_sort(arr, low, pi - 1)  # Sort the left partition
        quick_sort(arr, pi + 1, high)  # Sort the right partition

def main():
    arr = [10, 7, 8, 9, 1, 5]
    print("Original array:", arr)
    quick_sort(arr, 0, len(arr) - 1)
    print("Sorted array:", arr)

if __name__ == "__main__":
    main()
```

Listing 2: Quicksort Algorithm (Python)

### 6.1.3 Binary Search

```python
def binary_search(arr, low, high, target):
    while low <= high:
        mid = low + (high - low) // 2  # Calculate the middle index
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half
    return -1  # Target not found

def main():
    arr = [2, 3, 4, 10, 40]  # Array must be sorted
    target = 10

    print("Array:", arr)
    print("Target:", target)

    result = binary_search(arr, 0, len(arr) - 1, target)

    if result != -1:
        print(f"Element found at index {result}")
    else:
        print("Element not found in the array")

if __name__ == "__main__":
    main()
```

Listing 3: Binary Search (Python)

### 6.1.4 Binary Search Trees

Contains binary search tree operations, from node insertion, deletion and key finding.

```python
class TreeNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, root, key):
        if root is None:
            return TreeNode(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        return root

    def search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self.search(root.left, key)
        return self.search(root.right, key)

    def delete(self, root, key):
        if root is None:
            return root
        if key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            # Node with only one child or no child
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left
            # Node with two children: Get the inorder successor
            temp = self._min_value_node(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)
        return root
    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

def main():
    bst = BinarySearchTree()
    root = None

    # Insert nodes
    values_to_insert = [50, 30, 70, 20, 40, 60, 80]
    print("Inserting values:", values_to_insert)
    for value in values_to_insert:
        root = bst.insert(root, value)

    # Display the BST (inorder traversal)
    print("Inorder traversal of the BST:")
    bst.inorder(root)
    print()

    # Search for a node
    value_to_search = 40
    print(f"Searching for value {value_to_search} in the BST:")
    result = bst.search(root, value_to_search)
    if result:
        print(f"Value {value_to_search} found in the BST.")
    else:
        print(f"Value {value_to_search} not found in the BST.")

    # Delete a node
    value_to_delete = 50
    print(f"Deleting value {value_to_delete} from the BST:")
    root = bst.delete(root, value_to_delete)

    # Display the BST after deletion
    print("Inorder traversal of the BST after deletion:")
    bst.inorder(root)
    print()
```

```
if __name__ == "__main__":
    main()
```

Listing 4: Binary Search Tree Operations

### 6.1.5 Hashing and Linear Probing

```
class LinearProbingHashTable:
    def __init__(self, capacity=10):
        """
        Initialize the hash table with a given capacity.
        The table will store tuples of (key, value) or None if empty.
        """
        self.capacity = capacity
        self.size = 0  # number of elements actually stored
        self.table = [None] * capacity

    def _hash(self, key):
        """
        A simple hash function (for integer keys).
        For integer keys, we can use key % capacity.
        """
        return key % self.capacity

    def insert(self, key, value):
        """
        Insert a (key, value) pair into the hash table using linear
   probing.
        If the key already exists, update its value.
        """
        idx = self._hash(key)

        # Probe linearly until we find an empty slot or a slot with the
    same key
        while self.table[idx] is not None:
            stored_key, _ = self.table[idx]
            if stored_key == key:
                # Update existing key
                self.table[idx] = (key, value)
                return
            idx = (idx + 1) % self.capacity

        # Insert the new key-value pair
        self.table[idx] = (key, value)
        self.size += 1

        # (Optional) Could implement resizing/rehashing here if load
```

```
factor is too high

 def search(self, key):
     """
     Search for a key in the hash table and return its value if
found,
     or None if not found.
     """
     idx = self._hash(key)

     # Probe up to 'capacity' times to avoid infinite loops
     for _ in range(self.capacity):
         if self.table[idx] is None:
             # If we hit an empty slot, key does not exist
             return None
         stored_key, stored_value = self.table[idx]
         if stored_key == key:
             return stored_value
         idx = (idx + 1) % self.capacity
     return None

 def delete(self, key):
     """
     Delete a key from the hash table by setting its slot to None
and then
     re-inserting any keys that follow to avoid breaking the probe
sequence.
     """
     idx = self._hash(key)

     for _ in range(self.capacity):
         if self.table[idx] is None:
             # Key not found
             return False
         stored_key, _ = self.table[idx]
         if stored_key == key:
             # Remove this entry
             self.table[idx] = None
             self.size -= 1
             # Rehash subsequent items to ensure proper lookups
             self._rehash_from_index(idx)
             return True
         idx = (idx + 1) % self.capacity
     return False

 def _rehash_from_index(self, start_index):
     """
```

```
        Re-hash items following 'start_index' until an empty slot is
    found.
        This prevents 'breaking' the linear probe chain.
        """
        idx = (start_index + 1) % self.capacity

        while self.table[idx] is not None:
            key_to_rehash, value_to_rehash = self.table[idx]
            self.table[idx] = None
            self.size -= 1  # Will re-increment on insert
            self.insert(key_to_rehash, value_to_rehash)
            idx = (idx + 1) % self.capacity

    def __str__(self):
        """
        Returns a string representation of the hash table contents.
        """
        result = []
        for i, entry in enumerate(self.table):
            if entry is None:
                result.append(f"Index {i}: Empty")
            else:
                k, v = entry
                result.append(f"Index {i}: {k} -> {v}")
        return "\n".join(result)


def main():
    # Create a hash table with a small capacity
    ht = LinearProbingHashTable(capacity=7)

    # Insert some key-value pairs
    ht.insert(10, "ten")
    ht.insert(17, "seventeen")
    ht.insert(3, "three")
    ht.insert(24, "twenty-four")

    print("Hash table after inserts:")
    print(ht)
    print()

    # Search for a key
    print("Searching for key 17:", ht.search(17))  # Should print '
   seventeen'
    print("Searching for key 5:", ht.search(5))    # Should print 'None
   '
    print()
    # Delete a key
    print("Deleting key 17...")
    ht.delete(17)

    print("Hash table after deletion:")
    print(ht)
    print()

    # Try deleting a non-existent key
    print("Deleting key 5 (non-existent)...")
    result = ht.delete(5)
    print("Delete result:", result)
    print("Hash table after attempting to delete key 5:")
    print(ht)


if __name__ == "__main__":
    main()
```

### 6.1.6 Brute Force Searching

```
def brute_force_search(text, pattern):
    """
    Return the index of the first occurrence of 'pattern' in 'text'
    or -1 if not found. This is the naive O(n*m) approach.
    """
    n = len(text)
    m = len(pattern)

    if m == 0:
        return 0  # empty pattern is at index 0
    if n == 0:
        return -1

    for i in range(n - m + 1):
        # Check if pattern matches text at position i
        match = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                match = False
                break
        if match:
            return i
    return -1

def main():
    text = "ABCDEFABCDEF"
    pattern = "CDE"
    result = brute_force_search(text, pattern)

    if result != -1:
        print(f"Brute Force: Pattern '{pattern}' found at index {result
    } in text.")
    else:
        print(f"Brute Force: Pattern '{pattern}' not found in text.")

if __name__ == "__main__":
    main()
```

### 6.1.7 KMP

```
def build_lps(pattern):
    """
    Build the Longest Prefix-Suffix (LPS) array for KMP.
    lps[i] = the longest proper prefix of pattern[:i+1]
             which is also a suffix of pattern[:i+1].
    """
    lps = [0] * len(pattern)
    prefix_index = 0  # length of the previous longest prefix suffix
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[prefix_index]:
            prefix_index += 1
            lps[i] = prefix_index
            i += 1
        else:
            if prefix_index != 0:
                # This is tricky: consider the example "AAACAAAA"
                prefix_index = lps[prefix_index - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    """
    Returns the index of the first occurrence of 'pattern' in 'text',
    or -1 if the pattern is not found.
    """
    if not pattern:
        return 0  # Convention: empty pattern appears at index 0
    if not text:
        return -1

    lps = build_lps(pattern)

    i = 0  # index for text
    j = 0  # index for pattern

    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
            if j == len(pattern):
                return i - j  # match found, return starting index
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return -1

def main():
    text = "ABABDABACDABABCABAB"
    pattern = "ABABCABAB"
    result = kmp_search(text, pattern)

    if result != -1:
        print(f"KMP: Pattern '{pattern}' found at index {result} in
   text.")
    else:
        print(f"KMP: Pattern '{pattern}' not found in text.")

if __name__ == "__main__":
    main()
```

### 6.1.8 Boyer-Moore

```
def build_bad_char_table(pattern):
    """
    Constructs the 'bad character' table.
    This table holds, for each possible character, the rightmost
   occurrence
    index in the pattern. If a character is not in the pattern, store
   -1.
    """
    # For simplicity, assume ASCII extended set of 256 chars
    # or you can adapt to a larger set if needed (e.g., Unicode).
    table_size = 256
```

```python
    bad_char_table = [-1] * table_size

    for i, char in enumerate(pattern):
        bad_char_table[ord(char)] = i

    return bad_char_table

def boyer_moore_search(text, pattern):
    """
    Returns the index of the first occurrence of 'pattern' in 'text',
    or -1 if not found, using the Boyer-Moore "bad character" heuristic
   .
    """
    if not pattern:
        return 0
    if not text or len(pattern) > len(text):
        return -1

    bad_char_table = build_bad_char_table(pattern)
    shift = 0
    while shift <= len(text) - len(pattern):
        j = len(pattern) - 1
        # Compare from the end of the pattern
        while j >= 0 and pattern[j] == text[shift + j]:
            j -= 1
        if j < 0:
            # Pattern found
            return shift
        else:
            # Calculate the shift based on the bad character
            bad_char_index = ord(text[shift + j])
            bad_char_pos_in_pattern = bad_char_table[bad_char_index]
            shift += max(1, j - bad_char_pos_in_pattern)

    return -1

def main():
    text = "HERE IS A SIMPLE EXAMPLE"
    pattern = "EXAMPLE"
    result = boyer_moore_search(text, pattern)

    if result != -1:
        print(f"Boyer-Moore: Pattern '{pattern}' found at index {result
   } in text.")
    else:
        print(f"Boyer-Moore: Pattern '{pattern}' not found in text.")
```

```
if __name__ == "__main__":
    main()
```

### 6.1.9 BFS Graph Traversal example

```
from collections import deque

def bfs(graph, start):
    """
    Perform Breadth-First Search on the graph from the 'start' node.
    Return the order in which the nodes are visited.

    graph: Adjacency list representation, e.g.:
           {
             'A': ['B', 'C'],
             'B': ['A', 'D'],
             ...
           }
    start: the starting node for BFS
    """
    visited = set()
    queue = deque([start])
    visited.add(start)
    order = []

    while queue:
        current = queue.popleft()
        order.append(current)

        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return order

def main():
    # Example graph (undirected) as adjacency list
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }

    start_node = 'A'
    result = bfs(graph, start_node)
    print(f"BFS traversal starting at '{start_node}': {result}")

if __name__ == "__main__":
    main()
```

### 6.1.10 DFS

```
def dfs(graph, current, visited, depth=0):
    """
    Recursive Depth-First Search that prints each step.

    :param graph: dict representing adjacency list, e.g.:
                  {
                    'A': ['B', 'C'],
                    'B': ['A', 'D', 'E'],
                    ...
                  }
    :param current: the current node we are visiting
    :param visited: set of visited nodes
    :param depth: used for indentation in printed output (to visualize
   depth)
    """
    # Mark current as visited
    visited.add(current)
    print("  " * depth + f"Visiting node: {current}")

    # Explore each neighbor that hasn't been visited
    for neighbor in graph[current]:
        if neighbor not in visited:
            print("  " * depth + f" -> Going deeper from {current} to {
   neighbor}")
            dfs(graph, neighbor, visited, depth + 1)
        else:
            print("  " * depth + f" -> Already visited {neighbor},
   skipping.")


def main():
    # Example graph (undirected) using adjacency list
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']
    }

    start_node = 'A'
    visited = set()
    print(f"Starting DFS from node '{start_node}'")
    dfs(graph, start_node, visited)

    print("\nFinal visited set:", visited)


if __name__ == "__main__":
    main()
```

### 6.1.11 Prim's Algorithm

```
import heapq

def prim_mst(graph, start):
    """
    Prim's algorithm to find MST.
    graph: adjacency list with edge weights in the form:
           {
             0: [(1, weight1), (2, weight2), ...],
             1: [(0, weight1), (3, weight3), ...],
             ...
           }
    start: starting node (e.g., 0)
    Returns a list of (u, v, weight) edges that are in the MST.
    """
    visited = set()
    mst_edges = []
    min_heap = []

    # Push edges from start node
    visited.add(start)
    for (neighbor, weight) in graph[start]:
        heapq.heappush(min_heap, (weight, start, neighbor))

    # While there are edges to explore
    while min_heap:
        weight, u, v = heapq.heappop(min_heap)
        if v in visited:
            # If we've already visited this node, skip
            continue
        # Otherwise, this edge is part of the MST
        visited.add(v)
        mst_edges.append((u, v, weight))
        # Push all edges from 'v' to the heap
        for (next_neighbor, w) in graph[v]:
            if next_neighbor not in visited:
                heapq.heappush(min_heap, (w, v, next_neighbor))

    return mst_edges

def main():
    """
    Example usage of Prim's algorithm on a small graph.
    """
    # Define a weighted undirected graph as adjacency list
    graph = {
        0: [(1, 4), (7, 8)],
        1: [(0, 4), (2, 8), (7, 11)],
        2: [(1, 8), (3, 7), (5, 4), (8, 2)],
        3: [(2, 7), (4, 9), (5, 14)],
        4: [(3, 9), (5, 10)],
        5: [(2, 4), (3, 14), (4, 10), (6, 2)],
        6: [(5, 2), (7, 1), (8, 6)],
        7: [(0, 8), (1, 11), (6, 1), (8, 7)],
        8: [(2, 2), (6, 6), (7, 7)]
    }

    start_node = 0
    mst = prim_mst(graph, start_node)
    print("Prim's MST result (edges in the MST):")
    for u, v, w in mst:
        print(f"{u} -- {v} (weight {w})")

if __name__ == "__main__":
    main()
```

### 6.1.12 TSP (Brute Force)

```
import itertools

def calculate_route_cost(distance_matrix, route):
    """
    Given a distance matrix and a route (list of city indices),
    return the total travel cost of visiting the cities in the order of
    'route',
    including returning to the start at the end.

    distance_matrix[i][j] = distance from city i to city j
    """
    cost = 0
    for i in range(len(route) - 1):
        cost += distance_matrix[route[i]][route[i+1]]
    # Add cost to return to the start city
    cost += distance_matrix[route[-1]][route[0]]
    return cost

def tsp_brute_force(distance_matrix):
    """
    Solve the Traveling Salesperson Problem by brute force.
    Prints each permutation (route) and its total cost.

    :param distance_matrix: 2D list or matrix, distance_matrix[i][j]
                            = distance from city i to city j
    :return: (best_route, best_cost)
    """
    n = len(distance_matrix)  # Number of cities
    cities = range(n)

    best_route = None
    best_cost = float('inf')

    # Try all permutations of the cities (except fixing city 0 as start
   )
    # Another approach is to include all permutations of all cities
    # and then interpret the first city as start; but commonly we fix
    # one city to reduce equivalent rotations.
    for perm in itertools.permutations(cities):
        # We could fix city 0 as the start by ensuring perm[0] == 0
        # But for demonstration, let's just try all permutations.
        cost = calculate_route_cost(distance_matrix, perm)

        # Print the route and cost at each step
        print(f"Trying route {perm} -> cost = {cost}")

        if cost < best_cost:
            best_cost = cost
            best_route = perm

    return best_route, best_cost

def main():
    """
    Example usage of brute-force TSP on a small 4-city distance matrix.

    Let's label the cities as 0, 1, 2, 3 for simplicity.
    """
    # Example distance matrix for 4 cities (0,1,2,3)
    # Symmetric matrix (undirected). Zero on diagonals.
    distance_matrix = [
        [0, 10, 15, 20],  # Distances from city 0
        [10, 0, 35, 25],  # Distances from city 1
        [15, 35, 0, 30],  # Distances from city 2
        [20, 25, 30, 0]   # Distances from city 3
    ]

    print("Brute Force TSP on 4 cities:\n")
    best_route, best_cost = tsp_brute_force(distance_matrix)

    print("\nBest route found:", best_route)
    print("Best route cost: ", best_cost)


if __name__ == "__main__":
    main()
```

### 6.1.13 TSP (Greedy)

```
def tsp_greedy(distance_matrix, start=0):
    """
    Greedy (nearest neighbor) approach to solve TSP.

    :param distance_matrix: 2D list or matrix of distances, where
                            distance_matrix[i][j] = distance from city i
     to city j
    :param start:            index of the starting city
    :return:                 (route, total_cost)
                             route = list of visited city indices in
   order
                             total_cost = sum of traveling that route (
   returning to start at the end)
    """
    n = len(distance_matrix)
    visited = [False] * n  # Keep track of which cities have been
   visited
    route = [start]
    visited[start] = True
    current_city = start
    total_cost = 0

    # Visit all other cities
    for _ in range(n - 1):
        # Find the nearest unvisited city
        next_city = None
        min_distance = float('inf')

        for city in range(n):
            if not visited[city]:
                dist = distance_matrix[current_city][city]
                if dist < min_distance:
                    min_distance = dist
                    next_city = city

        # Move to the chosen next city
        route.append(next_city)
        visited[next_city] = True
        total_cost += min_distance
        current_city = next_city

    # Finally, return to the start city
    total_cost += distance_matrix[current_city][start]
    route.append(start)  # Append the start city again to show the
   complete cycle

    return route, total_cost


def main():
    # Example distance matrix for 5 cities labeled 0..4
    distance_matrix = [
        [0, 10, 8,  19, 12],
        [10, 0, 20, 6,  3 ],
        [8,  20, 0, 18, 14],
        [19, 6,  18, 0,  8 ],
        [12, 3,  14, 8,  0 ]
    ]

    start_city = 0
    route, cost = tsp_greedy(distance_matrix, start=start_city)

    print(f"Greedy TSP starting at city {start_city}:")
    print(f"Route (with return): {route}")
    print(f"Total cost: {cost}")


if __name__ == "__main__":
    main()
```