Week 3 Report: Search Algorithms

Topics Covered

The key topics were:

- - Linear Search
- - Binary Search
- - Exponential Search
- - BFS (Breadth-First Search)
- - DFS (Depth-First Search)

Linear Search

What it does: Goes through each element in the list one-by-one to find the target.

Use Case: Small lists, unsorted data.

Pros: Simple to implement.

Cons: Slow for large lists.

Time Complexity: O(n)

Example: Checking if a specific name exists in an unsorted list of student names.

C++ Template:

```
int linear_search(vector<int>& arr, int target) {
   for (int i = 0; i < arr.size(); i++) {
      if (arr[i] == target)
         return i;
   }
   return -1;
}
```

Binary Search

What it does: Searches a sorted array by repeatedly dividing the search interval in half.

Use Case: Efficient searching in sorted data.

Pros: Very fast for large data sets.

Cons: Requires data to be sorted.

Time Complexity: O(log n)

Example: Finding if a particular number exists in a sorted list of integers.

C++ Template:

```
int binary_search(vector<int>& arr, int target) {
  int left = 0, right = arr.size() - 1;
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target)
      return mid;
    else if (arr[mid] < target)
      left = mid + 1;
    else
      right = mid - 1;
  }
  return -1;
}
```

Exponential Search

What it does: Rapidly expands the search range exponentially, then applies binary search within the found range.

Use Case: Searching in infinite or unbounded sorted lists.

Pros: Efficient for unknown size lists.

Time Complexity: O(log n)

Example: Searching for an element in a huge sorted file where size isn't known.

C++ Template:

```
int exponential_search(vector<int>& arr, int target) {
  if (arr[0] == target) return 0;
  int bound = 1;
  while (bound < arr.size() && arr[bound] < target)
    bound *= 2;
  int left = bound / 2;
  int right = min(bound, (int)arr.size() - 1);
  while (left <= right) {
    int mid = left + (right - left) / 2;
```

```
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Breadth-First Search (BFS)

What it does: Explores all nodes at one depth level before moving to the next level.

Use Case: Shortest path problems in unweighted graphs.

Pros: Guarantees shortest path.

Cons: Requires more memory (queue).

Time Complexity: O(V + E)

Example: Finding the shortest number of moves in a maze.

C++ Template:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void bfs(int start, vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> visited(n, false);
    queue<int> q;

    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        for (int neighbor : graph[node]) {
            if (!visited[neighbor]) {
```

```
            visited[neighbor] = true;
            q.push(neighbor);
        }
    }
  }
}
```

Depth-First Search (DFS)

What it does: Explores as deep as possible in each branch before backtracking.

Use Case: Cycle detection, connected components, topological sorting.

Pros: Easy to implement recursively.

Cons: May unnecessarily explore deep paths.

Time Complexity: O(V + E)

Example: Detecting cycles in a directed graph.

C++ Template:

```
void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited) {
  visited[node] = true;
  cout << node << " ";
  for (int neighbor : graph[node]) {
    if (!visited[neighbor]) {
      dfs(neighbor, graph, visited);
    }
  }
}
// Call like this:
// vector<bool> visited(n, false);
// dfs(start_node, graph, visited);
```