Algorithms Notes

Farhan Sadeek

Last Updated: August 17, 2025

This is the place where I will keep track of the all the algorithms that I learnt in my LeetCode and Competitive Programming journery.

1 Graph Algorithms

1.1 Breadth-First Search (BFS)

BFS is a traversal algorithm for graphs that explores all neighbors at the present depth prior to moving on to nodes at the next depth level. It is often used for finding the shortest path in unweighted graphs. Here is how the algorithm actually works:

1. Start from a given node (the source).
2. Mark the node as visited and enqueue it.
3. While the queue is not empty:
   (a) Dequeue a node from the front of the queue.
   (b) Process the node (e.g., print or store it).
   (c) Enqueue all unvisited neighbors of the node, marking them as visited.
4. Repeat until all reachable nodes are processed.

Here is a simple implementation of BFS in C++:

```cpp
void bfs(int start, const vector<vector<int>>& adj, vector<bool>& visited) {
    queue<int> q;
    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        // Process node here (e.g., print or store)
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

1.2 Depth-First Search (DFS)

DFS is another traversal algorithm for graphs that explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack. Here is how the algorithm works:

1. Start from a given node (the source).
2. Mark the node as visited.
3. For each unvisited neighbor, recursively call DFS on the neighbor.
4. Repeat until all reachable nodes are processed.

Here is a simple implementation of DFS in C++:

```cpp
void dfs(int node, const vector<vector<int>>& adj, vector<bool>& visited) {
    visited[node] = true;
    // Process node here (e.g., print or store)
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited);
        }
    }
}
```

1.3 Topological Sorting

Topological sorting is a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge $u \to v$, vertex u comes before vertex v in the ordering. It is often used in scheduling problems and can be implemented using either DFS or Kahn's algorithm. Here is how the algorithm works:

1. Compute he in-degree of each vertex.
2. Initialize a queue with all vertices that have in-degree 0.
3. While the queue is not empty:
   (a) Dequeue a vertex from the front of the queue.
   (b) Add the vertex to the topological order.
   (c) For each neighbor of the vertex, decrease its in-degree by 1. If the in-degree becomes 0, enqueue the neighbor.
4. If the topological order contains all vertices, return it; otherwise, the graph has a cycle.

Here is a simple implementation of topological sorting in C++:

```
vector<int> topologicalSort(const vector<vector<int>>& adj, int n) {
    vector<int> inDegree(n, 0);
    for (const auto& neighbors : adj) {
        for (int neighbor : neighbors) {
            inDegree[neighbor]++;
        }
    }
    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }
    vector<int> topOrder;
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        topOrder.push_back(node);
        for (int neighbor : adj[node]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }
    if (topOrder.size() == n) {
        return topOrder;
    } else {
        return {};  // Graph has a cycle
    }
}
```

2 Dynamic Programming

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems and has optimal substructure. The key idea is to store the results of subproblems to avoid redundant calculations.

2.1 Fibonacci Sequence

The Fibonacci sequence is a classic example of a problem that can be solved using dynamic programming. The sequence is defined as follows:

$$F(n) = F(n-1) + F(n-2)$$

with base cases

$$F(0) = 0, F(1) = 1$$

Here is a simple implementation of Fibonacci using dynamic programming in C++:

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    vector<int> dp(n + 1);
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

2.2 Kadane's Algorithm

Kadane's Algorithm is used to find the maximum sum of a contiguous subarray in an array. It works by iterating through the array and maintaining a running sum of the maximum subarray ending at the current position. If the running sum becomes negative, it is reset to zero. The maximum sum is updated whenever a larger sum is found. Here is how the algorithm works:

1. Initialize two variables: `max_so_far` to negative infinity and `current_sum` to 0.
2. Iterate through each element in the array:
   (a) Add the current element to `current_sum`.
   (b) If `current_sum` is greater than `max_so_far`, update `max_so_far`.
   (c) If `current_sum` becomes negative, reset it to 0.
3. The final value of `max_so_far` will be the maximum sum of a contiguous subarray.

Here is a simple implementation of Kadane's Algorithm in C++:

```
int kadane (const vector<int>& arr) {
    int max_so_far = INT_MIN;
    int current_sum = 0;
    for (int num : arr) {
        current_sum += num;
        if (current_sum > max_so_far) {
            max_so_far = current_sum;
        }
        if (current_sum < 0) {
            current_sum = 0;
        }
    }
    return max_so_far;
}
```

2.3 0/1 Knapsack Problem

The 0/1 Knapsack problem is a classic optimization problem where you have a set of items, each with a weight and a value, and you want to maximize the total value in a knapsack of a given capacity. The problem can be solved using dynamic programming as follows:

1. Create a 2D array `dp[i][j]` where `i` is the number of items and `j` is the capacity of the knapsack.
2. Initialize the first row and column to 0.
3. For each item `i` and capacity `j`:
   (a) If the weight of the item is less than or equal to `j`:

   ```
   dp[i][j] = max(dp[i-1][j], dp[i-1][j - weight[i]] + value[i])
   ```

   (b) If the weight of the item is greater than `j`:

   ```
   dp[i][j] = dp[i-1][j]
   ```

4. The maximum value will be found in `dp[n][W]` where `n` is the number of items and `W` is the capacity of the knapsack.

Here is a simple implementation of the 0/1 Knapsack problem in C++:

```cpp
int knapsack(int W, const vector<int>& weights, const vector<int>& values) {
    int n = weights.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= W; j++) {
            if (weights[i - 1] <= j) {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] + values[i - 1]);
            }
            else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][W];
}
```

2.4 Unbounded Knapsack Problem

2.5 Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) problem is a classic problem in computer science where you want to find the longest subsequence that is common to two sequences. The problem can be solved using dynamic programming as follows:

1. Create a 2D array `dp[i][j]` where `i` is the length of the first sequence and `j` is the length of the second sequence.
2. Initialize the first row and column to 0.
3. For each character in the first sequence and each character in the second sequence:
   (a) If the characters match, set `dp[i][j] = dp[i-1][j-1] + 1`.
   (b) If the characters do not match, set `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`.
4. The length of the longest common subsequence will be found in `dp[n][m]` where `n` is the length of the first sequence and `m` is the length of the second sequence.

Here is a simple implementation of LCS in C++:

```cpp
int lcs(const string& s1, const string& s2) {
    int n = s1.size(), m = s2.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[n][m];
}
```

2.6 Longest Increasing Subsequence (LIS)

The Longest Increasing Subsequence (LIS) problem is a classic problem in computer science where you want to find the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. The problem can be solved using dynamic programming as follows:

1. Create a 1D array `dp[i]` where `i` is the length of the input sequence.
2. Initialize all elements of `dp` to 1, since the minimum length of LIS ending at each element is 1 (the element itself).
3. For each element in the sequence, compare it with all previous elements:
   (a) If the current element is greater than a previous element, update `dp[i]` to be the maximum of its current value and `dp[j] + 1`, where `j` is the index of the previous element.
4. The length of the longest increasing subsequence will be the maximum value in the `dp` array.

Here is a simple implementation of LIS in C++:

```cpp
int lis(const vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1);
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }
    return *max_element(dp.begin(), dp.end());
}
```