Algorithms Notes

Farhan Sadeek

Last Updated: May 21, 2026

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

Contents

1 Graph Algorithms

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
#include <bits/stdc++.h>
using namespace std;

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);
      }
    }
  }
}
```

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
#include <bits/stdc++.h>
using namespace std;

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);
    }
  }
}
```

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 the 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++:

```cpp
#include <bits/stdc++.h>
using namespace std;

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.

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++:

```
#include <bits/stdc++.h>
using namespace std;

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 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++:

```cpp
#include <bits/stdc++.h>
using namespace std;

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;
}
```

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
#include <bits/stdc++.h>
using namespace std;

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];
}
```

4 Unbounded Knapsack Problem

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
#include <bits/stdc++.h>
using namespace std;

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];
}
```

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
#include <bits/stdc++.h>
using namespace std;

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());
}
```

7 Palindromic Subsequence

8 Edit Distance

Edit Distance, also known as Levenshtein distance, is a measure of how dissimilar two strings are by counting the minimum number of operations required to transform one string into the other. The allowed operations are insertion, deletion, and substitution of a single character. 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 string and `j` is the length of the second string.
2. Initialize the first row and column: `dp[i][0] = i` for all `i`, since it takes `i` deletions to convert a string of length `i` to an empty string. `dp[0][j] = j` for all `j`, since it takes `j` insertions to convert an empty string to a string of length `j`. `dp[0][0] = 0`, since no operations are needed to convert an empty string to another empty string.
3. For each character in the first string and each character in the second string: If the characters match, set `dp[i][j] = dp[i-1][j-1]`. If the characters do not match, set `dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1)`.
4. The minimum edit distance will be found in `dp[n][m]` where `n` is the length of the first string and `m` is the length of the second string.

```cpp
#include <bits/stdc++.h>
using namespace std;

int editDistance(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 = 0; i <= n; i++) {
    dp[i][0] = i;
  }
  for (int j = 0; j <= m; j++) {
    dp[0][j] = j;
  }
  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];
      } else {
        dp[i][j] = min({dp[i - 1][j] + 1, dp[i][j - 1] + 1, dp[i - 1][j - 1] + 1});
      }
    }
  }
  return dp[n][m];
}
```

3 Tree Algorithms

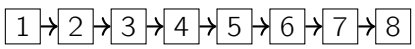
1 Fenwick Tree (Binary Indexed Tree)

A Fenwick Tree, or Binary Indexed Tree (BIT), is a data structure that efficiently supports prefix sum and point update operations in logarithmic time. It is commonly used in scenarios involving prefix sums, such as counting inversions, cumulative frequencies, or range sum queries with updates.

Key Operations:

- `update(i, v)`: Add v to index i (point update)
- `query(i)`: Compute prefix sum up to index i

Tree Structure Visualized (for $n = 8$ elements):



Each index stores a sum over some range defined by the least significant bit of the index.

C++ Implementation:

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> bit;  // binary indexed tree, 1-based indexing
int n;

void init_fenwick(int size) {
  n = size;
  bit.assign(n + 1, 0);
}

// Add val to index i
void update(int i, int val) {
  for (; i <= n; i += i & -i)
  bit[i] += val;
}

// Get prefix sum up to i
int query(int i) {
  int ans = 0;
  for (; i > 0; i -= i & -i)
  ans += bit[i];
  return ans;
}

// Get sum of range [l, r]
int query_range(int l, int r) {
  return query(r) - query(l - 1);
}
```

Quick facts:

- Construction: $O(n \log n)$ or $O(n)$ using special build
- **Supports:** Point updates, prefix/range sums
- **Does NOT support:** Range updates, range queries (unless advanced tricks used)

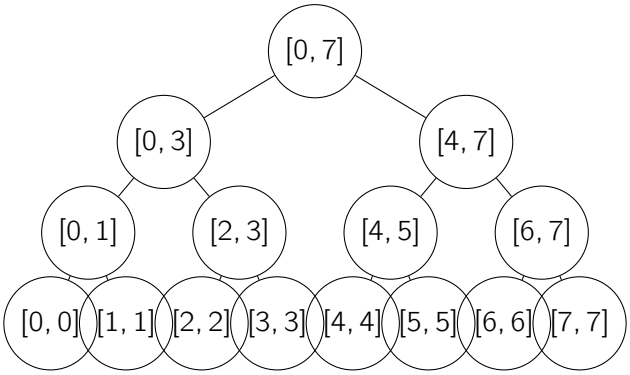
2 Segment Tree (Standard)

A Segment Tree efficiently supports range queries and point updates, often outperforming Fenwick Trees for more complex operations than just sums.

Key Features:

- Point update: Add or set a value at index i
- Range query: Compute sum, min, max, etc., over range $[l, r]$

Segment Tree Visualization (Perfect binary segmentation):



C++ Implementation (Range Sum, 0-based):

```cpp
#include <bits/stdc++.h>
using namespace std;

int seg_size;
vector<int> seg_tree;

void init_segment_tree(int n) {
  seg_size = 1;
  while (seg_size < n) seg_size <<= 1;
  seg_tree.assign(2 * seg_size, 0);
}

void set_val(int i, int v, int x, int lx, int rx) {
  if (rx - lx == 1) {
    seg_tree[x] = v;
    return;
  }
  int m = (lx + rx) / 2;
  if (i < m) set_val(i, v, 2*x+1, lx, m);
  else       set_val(i, v, 2*x+2, m, rx);
  seg_tree[x] = seg_tree[2*x+1] + seg_tree[2*x+2];
}

void set_val(int i, int v) {
  set_val(i, v, 0, 0, seg_size);
}

int seg_sum(int l, int r, int x, int lx, int rx) {
  if (lx >= r || rx <= l) return 0; // no overlap
  if (lx >= l && rx <= r) return seg_tree[x]; // complete overlap
  int m = (lx + rx) / 2;
  return seg_sum(l, r, 2*x+1, lx, m) + seg_sum(l, r, 2*x+2, m, rx);
}

int seg_sum(int l, int r) {
  return seg_sum(l, r, 0, 0, seg_size);
}
```

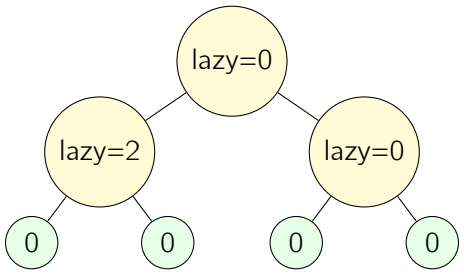
Supports: point update and range query in $O(\log n)$.

3 Segment Tree with Lazy Propagation

When you have to perform range updates (e.g., add a value to all elements in range $[l, r]$) along with range queries, a standard segment tree isn't enough: you need Lazy Propagation.

Lazy Propagation Idea: Postpone ("lazily") the update to children until needed, marking segments as needing to be updated later.

Visualization:



Here, yellow nodes have nonzero "lazy" values pending. When queried or split, values are pushed down.

C++ Implementation (Range add, range sum):

```cpp
#include <bits/stdc++.h>
using namespace std;

int lazy_seg_size;
vector<long long> lazy_tree, lazy;

void init_lazy_segment_tree(int n) {
  lazy_seg_size = 1;
  while (lazy_seg_size < n) lazy_seg_size <<= 1;
  lazy_tree.assign(2 * lazy_seg_size, 0LL);
  lazy.assign(2 * lazy_seg_size, 0LL);
}

void push(int x, int lx, int rx) {
  if (lazy[x] != 0 && rx - lx > 1) {
    lazy_tree[2*x+1] += lazy[x] * ((rx - lx) / 2);
    lazy[2*x+1] += lazy[x];
    lazy_tree[2*x+2] += lazy[x] * ((rx - lx) / 2);
    lazy[2*x+2] += lazy[x];
    lazy[x] = 0;
  }
}

void range_add(int l, int r, long long v, int x, int lx, int rx) {
  if (lx >= r || rx <= l) return; // no overlap
  if (lx >= l && rx <= r) {
    lazy_tree[x] += v * (rx - lx);
    lazy[x] += v;
    return;
  }
  push(x, lx, rx);
  int m = (lx + rx) / 2;
  range_add(l, r, v, 2*x+1, lx, m);
  range_add(l, r, v, 2*x+2, m, rx);
  lazy_tree[x] = lazy_tree[2*x+1] + lazy_tree[2*x+2];
}

void range_add(int l, int r, long long v) {
  range_add(l, r, v, 0, 0, lazy_seg_size);
}

long long range_sum(int l, int r, int x, int lx, int rx) {
  if (lx >= r || rx <= l) return 0;
  if (lx >= l && rx <= r) return lazy_tree[x];
  push(x, lx, rx);
  int m = (lx + rx) / 2;
  return range_sum(l, r, 2*x+1, lx, m) + range_sum(l, r, 2*x+2, m, rx);
}
```

Summary Table:

Structure	Point Update	Range Query	Range Update
Fenwick Tree	$O(\log n)$	$O(\log n)$	No
Segment Tree	$O(\log n)$	$O(\log n)$	No
Lazy Segment Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$