Algorithms Example Sheet 2: Core Questions
Solution Notes

These notes have not be fully proofread. So if you find any typos/mistakes or have any suggestions (even if very small), please do let me know.

1 Dynamic programming

Recommended reading/useful links:

- CLRS Chapter 15 (except for 15.5)
- LCS visualisation
- Collection of videos explaining some DP problems
- Various DP visualisations
- Jeff Erickson's Algorithms Chapter 3
- Algorithms' Illuminated: Part 3
- An article on knapsack problems.

Exercise 2.C.1 [Fibonacci Numbers]

(a) Explain how dynamic programming can be used to efficiently compute the Fibonacci numbers.

(b) Write pseudocode and explain the difference between the bottom-up and top-down approach.

(c) What is the time and space complexity of each approach?

(d) (optional) Implement one of the two DP algorithms.

(a) The recurrence relation for the Fibonacci numbers is given by $F(n) = F(n-1) + F(n-2)$ with $F(0) = 0$ and $F(1) = 1$. Dynamic programming suggests memorising $F(n)$ instead of calling $F(n-1) + F(n-2)$.

(b) See implementation below.

(c) The DP algorithms lead to an improvement from exponential to $\mathcal{O}(n)$ time. The direct direct implementation requires $\mathcal{O}(n)$ time. By noticing that we only need the last two values we can get an iterative implementation with $\mathcal{O}(1)$ space.

(d) Here is the recursive (top-down implementation):

```
/* Iterative with O(1) space. */
class Solution {
    public int fib(int n) {
        if (n <= 1) return n;
        int prev = 1, prev_prev = 0;
        for (int i = 2; i <= n; ++i) {
            int tmp = prev + prev_prev;
            prev_prev = prev;
            prev = tmp;
        }
        return prev;
    }
}
```

```java
/* Iterative with O(n) space. */
class Solution {
    public int fib(int n) {
        int dp[] = new int[n+1];
        dp[0] = 0;
        if (n >= 1) dp[1] = 1;
        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[n];
    }
}
```

Here is the iterative (bottom-up) implementations:

```java
/* Recursive with O(n) space. */
class Solution {
    int dp[];
    public int fib(int n) {
        dp = new int[n+1];
        return get(n);
    }
    public int get(int n) {
        if (n <= 1) return n;
        if (dp[n] > 0) return dp[n];
        int tmp = get(n-1) + get(n-2);
        return dp[n] = tmp;
    }
}
```

Exercise 2.C.2 [Rod cutting problem]

(a) Read section 15.1 from CLRS and define the *rod cutting problem*.

(b) Design an algorithm to solve it.

(c) (optional) Implement the algorithm. You can test your implementation on **[GeeksForGeeks Rod Cutting]**.

(a) In the *rod cutting problem* a rod of length n is given and a matrix p giving the price $p[\ell]$ for a rod of length ℓ. You have to find the optimal splitting of the rod (with smallest subdivision of 1, such that the price is maximised.

(b) We define $dp[i]$ to be the optimal splitting of a rod of length i.

$$dp[i] = \begin{cases} 0 & \text{if } i = 0 \\ \max_{1 \le j \le i}(p[j] + dp[i-j]) & \text{otherwise} \end{cases}$$

(c)
```java
class GFG {
    public int cutRod(int price[], int n) {
        int dp[] = new int[n+1];
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= i; ++j) {
                dp[i] = Math.max(dp[i], price[j-1] + dp[i-j]);
            }
        }
        return dp[n];
    }
}
```

Note: This is the $(\max, +)$ convolution, which we encountered in $0/1$ knapsack.

Exercise 2.C.3 [0/1 Knapsack]

(a) Define the *0/1-knapsack* problem.

(b) Provide a small counterexample that proves that the greedy strategy of choosing the item with the highest £/kg ratio is not guaranteed to yield the optimal solution.

(c) Describe a DP algorithm to solve 0/1 knapsack.

(d) What is the time and space complexity of the algorithm?

(e) (optional) Implement the DP algorithm. You can test your implementation on **[GeeksForGeeks 0/1 Knapsack]**.

(f) Explain how you could retrieve an optimal solution.

(a) We are given a knapsack of capacity C and a collection of n items where the i-th item has weight w_i and value v_i. The 0/1 knapsack problem is about finding the subset of items of maximum total value such that their total weight is $\leq C$.

(b) If the items have weights $w_1 = 100, w_2 = 60, w_3 = 50$, value 1 and $C = 120$, then the greedy algorithm chooses $w_1 = 100 \leq C$, but the optimal algorithm chooses $w_2 + w_3 = 60 + 50 = 110 \leq C$.

(c) Let $dp[i][w]$ be the maximum value for choosing a subset of $w_1, \ldots, w_i$ with total capacity w. Then,

$$dp[i][w] = \begin{cases} 0 & \text{if } i = 0 \wedge w = 0 \\ -\infty & \text{if } i = 0 \wedge w > 0 \\ \max(dp[i-1][w], v_i + dp[i-1][w - w_i]) & \text{otherwise} \end{cases}$$

The answer is the largest value in $dp[n][\cdot]$.

(d) There are $n \cdot C$ states and computing the recurrence relation takes constant time. Hence, this takes $\Theta(n \cdot C)$ time. By noticing that when computing $dp[i][\cdot]$ we are using only values in $dp[i-1][\cdot]$, the algorithm can be implemented with $\Theta(C)$ memory.

(e) The following implementation uses $\Theta(C)$ memory and actually only one array. By iterating through the weights in decreasing order, it is guaranteed that a particular item is used at most once.

```
class Solution {
    // Returns the maximum value that can be put in a knapsack of capacity W
    static int knapSack(int W, int wt[], int val[], int n) {
        int dp[] = new int[W + 1];
        for (int i = 1; i <= W; ++i) dp[i] = Integer.MIN_VALUE;
        int mx = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = W; j >= wt[i]; --j) {
                if (dp[j - wt[i]] != Integer.MIN_VALUE)
                    dp[j] = Math.max(dp[j], val[i] + dp[j - wt[i]]);
                mx = Math.max(mx, dp[j]);
            }
        }
        return mx;
    }
}
```

(f) We could retrieve the optimal solution, by keeping for each $dp[i][n]$ whether we used the i-th item to the optimal value or not. The following code implements this:

```
package sort2;

public class Knapsack01 {

   // Returns the maximum value that can be put in a knapsack of capacity W
   static int knapSack(int W, int wt[], int val[], int n) {
        int dp[][] = new int[n+1][W + 1];
        boolean used[][] = new boolean[n+1][W + 1];
        for (int i = 1; i <= W; ++i) dp[0][i] = Integer.MIN_VALUE;
```

```
        int mx = 0, mxWeight = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = W; j >= wt[i]; --j) {
                if (dp[i][j - wt[i]] != Integer.MIN_VALUE && val[i] + dp[i][j -
                    wt[i]] > dp[i][j]) {
                  dp[i+1][j] = val[i] + dp[i][j - wt[i]];
                  used[i+1][j] = true;
                } else {
                  dp[i+1][j] = dp[i][j];
                  // used[i+1][j] = false;
                }
                if (mx < dp[i+1][j]) {
                  mx = dp[i+1][j];
                  mxWeight = j;
                }
            }
            for (int j = 0; j < wt[i] && j < W; ++j) dp[i+1][j] = dp[i][j];
        }
        int curWeight = mxWeight, sm = 0;
        for (int i = n - 1; i >= 0; --i) {
         if (used[i+1][curWeight]) {
            System.out.println("(" + wt[i] + ", " + val[i] + ")");
            curWeight -= wt[i];
         }
        }

        return mx;
    }
}
```

Note: It is possible to use a variant of Hirschberg's algorithm to find the optimal subset using $\mathcal{O}(C)$ space and $\mathcal{O}(n \cdot C)$ time.

Note 1: The structure found in the $0/1$ knapsack problem is called a $(\max, +)$ convolution (which is equivalent to a $(\min, +)$ convolution if we negate the values). It is an open problem if this convolution can be computed in $\mathcal{O}\left(n^{2-\epsilon}\right)$ for constant $\epsilon > 0$. Several problems reduce to this as was proven in [**?**] (see here).

Note 2: This solution is not truly polynomial in the input size, since C could be very large (and we only need around $\Theta(\log_2 C)$ bits to represent it. Actually, as you will probably see in Part IB Complexity Theory, the knapsack problem is NP-Complete, so it is unknown whether it is solvable in truly polynomial time. In Part II Algorithms, you will approximations to the knapsack problem.

Exercise 2.C.4 [Matrix Chain Multiplication]

(a) Define the *matrix chain multiplication* problem.

(b) Why can we choose the order of the multiplications?

(c) Why may we want to choose the order of the multiplications? Demonstrate this with an example.

(d) Explain how dynamic programming can be used to solve this problem.

(e) What is the time complexity for this approach?

(f) (optional) Attempt **[LeetCode 312]**.

(a) In the matrix multiplication problem, we are given a sequence of k matrices $M_1, M_2, \ldots, M_k$ to be multiplied and we want to choose the order to perform the matrix multiplications, so that it takes the least time.

(b) We can choose the order because matrix multiplication is associative, i.e. $A(BC) = (AB)C$ for every matrix A, B, C (with the appropriate dimensions). This means (see Discrete Maths for a more rigorous proof) that we can choose any of the possible bracketings and they will give the same result.

(c) We can dramatically reduce the number of arithmetic operations required. A simple example to see this is consider having three matrices $A : \mathbb{R}^{n \times n}$, $B : \mathbb{R}^{n \times n}$ and $C : \mathbb{R}^{n \times 1}$. Then to compute $(AB)C$ we need $n \cdot n \cdot n$ for $R_1 = A \cdot B : \mathbb{R}^{n \times n}$ and $n \cdot n \cdot 1$ for $R_1 \cdot C$. This gives a total of $n^3 + n^2$ operations.

Then to compute $A(BC)$ we need $n \cdot n \cdot 1$ for $R_1 = B \cdot C : \mathbb{R}^{n \times 1}$ and $n \cdot n \cdot 1$ for $A \cdot R_1$. This gives a total of $n^2 + n^2$ operations.

Comparing the two we see an asymptotic decrease from $\Theta(n^3)$ to $\Theta(n^2)$.

Question: *Is it the case that the best reduction is by a multiplicative factor of N, where N is the largest dimension of one of the matrices? (It might be easier to consider the case where k, the number of matrices is fixed.)*

(d) Let $dp[i][j]$ be the minimum number of operations to multiply matrices $M_i, \ldots, M_j$. Then, given a sequence $M_i, \ldots, M_j$, we can search over all possible "last matrix multiplications". In the last matrix multiplication, a continuous sequence of matrices on the left will have been multiplied and a continuous sequence on the right will have been multiplied. So, we can search for all possible ways to split $M_i, \ldots, M_j$ into $M_i, \ldots, M_k$ and $M_{k+1}, \ldots, M_j$, by iterating over all $k \in [i, j]$. This gives rise to the following recurrence relation:

$$dp[i][j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k \in [i,j)} dp[i][k] + dp[k+1][j] & \text{otherwise} \end{cases}.$$

The time complexity for calculating $dp[i][j]$ is roughly $j - i + 1$. So there will be $n - \ell + 1$ pairs $(i, i + \ell)$, which gives a total of

$$\sum_{\ell=1}^{n} (n-\ell+1) \cdot \ell = \sum_{\ell=1}^{n} (n+1) \cdot \ell - \ell^2 = (n+1) \cdot \sum_{\ell=1}^{n} \ell - \sum_{\ell=1}^{n} \ell^2 = \frac{1}{2}(n+1) \cdot n \cdot (n+1) - \frac{1}{6} \cdot n(n+1)(2n+1) = \Theta(n^3).$$

(e) We can transform the problem into matrix chain multiplication by constructing the dimensions $(1, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, 1)$. The following code implements the algorithm:

```
class Solution {
    public int maxCoins(int[] nums2) {
        // Convert to a matrix chain multiplication problem.
        int nums[][] = new int[nums2.length + 1][2];
        for (int i = 0; i < nums2.length; ++i) {
            nums[i][1] = nums2[i];
            if (i + 1 < nums.length) nums[i+1][0] = nums2[i];
        }
        nums[0][0] = nums[nums.length - 1][1] = 1;

        int[][] dp = new int[nums.length][nums.length];
        for (int len = 1; len < nums.length; ++len) {
            // dp[i][i] = 0;
            for (int i = 0; i + len < nums.length; ++i) {
                // dp[i][j] = 0
               int j = i + len;
                for (int k = i; k < j; ++k) {
                    dp[i][j] = Math.max(dp[i][j],
                        dp[i][k] + dp[k+1][j] + nums[i][0] * nums[k][1] *
                            nums[j][1]);
                }
            }
        }
        return dp[0][nums.length - 1];
    }
}
```

Further reading: There is an algorithm by Hu and Shing that reduces the matrix chain multiplication problem into that of triangulating the a weighted convex polygon, which is then solved in $\mathcal{O}(n \log n)$ time (Warning: I have not verified the proof of the algorithm). In case you are interested, the algorithm was introduced in [?] and [?], and a correction to the core Lemma was suggested in [?].

Exercise 2.C.5 [Longest Common Subsequence]

(a) Define the *longest common subsequence* problem.

(b) Formulate the recurrence relation and explain how dynamic programming helps to solve it.

(c) Show the DP table for input sequences `BDCABA` and `ABCBDAB`.

(d) (optional) Implement the LCS algorithm (either bottom up or top-down). (You may want to submit your solution to **[LeetCode 1143]**)

(e) What is the time complexity of your implementation? How does it compare with the brute force approach?

(f) What is the space complexity of your implementation? How can you reduce this?

(g) Explain how you can recover a longest common subsequence. Draw the corresponding table for the example above. What is the time complexity of this algorithm?

(h) Demonstrate a pair of sequences that have more than one LCSs.

(a) A subsequence of a string s is $s' = s_{i_1}s_{i_2}\dots s_{i_k}$, where $0 < i_1 < i_2 < \dots < i_{k-1} < i_k \leq |S|$ are indices in the string. A common subsequence between strings a and b is a subsequence s' of a that is also a subsequence of b. The longest common subsequence is that with the most characters (breaking ties, e.g. lexicographically).

(b) Let $dp[i][j]$ be the longest subsequence ending at i in string a and ending at j in string b.

- The base case $dp[i][0] = 0$ for every i, since the empty string has no subsequences.
- Similarly, the base case $dp[0][i] = 0$.
- If $a[i] = b[j]$, it means that we can

(c) The following table corresponds to the above recursive definition.

	∅	**A**	**B**	**C**	**B**	**D**	**A**	**B**
∅	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

(d) Here is the recursive implementation:

```
class Solution {
    private int dp[][];
    public int longestCommonSubsequence(String A, String B) {
        dp = new int[A.length() + 1][B.length() + 1];
        for (int i = 0; i < A.length() + 1; ++i) {
            for (int j = 0; j < B.length() + 1; ++j) {
                dp[i][j] = -1;
            }
        }
        return lcs(A, B, A.length(), B.length());
    }

    public int lcs(String A, String B, int i, int j) {
        if (dp[i][j] != -1) return dp[i][j];
        if (i == 0 || j == 0) return 0;
        dp[i][j] = Math.max(lcs(A, B, i-1, j), lcs(A, B, i, j-1));
        if (A.charAt(i-1) == B.charAt(j-1))
            dp[i][j] = Math.max(lcs(A, B, i, j), lcs(A, B, i-1, j-1) + 1);
        return dp[i][j];
    }
}
```

Question: *Why do we have to initialise to -1? Can we not just check if an entry is 0?* If we recomputed an entry every time it is 0, then for the case where the strings have all characters different, we would need exponential running time, since all entries will be zero.

Here is the iterative implementation that uses memoisation (leading to linear memory).

```
class Solution {
    public int longestCommonSubsequence(String A, String B) {
        if (B.length() > A.length()) { String tmp = A; A = B; B = tmp; }
        int dp[][] = new int[B.length() + 1][2];
        int cur = 0, prev;
        for (int i = 0; i < A.length(); ++i) {
            prev = 1 - cur;
            for (int j = 0; j < B.length(); ++j) {
                // Skip either character.
                dp[j+1][cur] = Math.max(dp[j+1][prev], dp[j][cur]);
                // If they are equal, then extend previous matching.
                if (A.charAt(i) == B.charAt(j)) dp[j+1][cur] =
                    Math.max(dp[j+1][cur], 1 + dp[j][prev]);
            }
            cur = 1 - cur;
        }
        return dp[B.length()][1 - cur];
    }
}
```

(e) Both the recursive and the iterative implementation take $\mathcal{O}(n \cdot m)$ time where n and m are the lengths of the two strings. The iterative implementation using memoisation requires $\mathcal{O}(\min(n, m))$ memory.

(f) At each (i, j) we can keep a pointer to the (i', j') which lead to the minimum value. When traversing a direction $(i-1, j-1) \to (i, j)$ we should append character $A[i]$ to the subsequence. The following code implements this (giving an encoding $0, 1, 2$ to each of the 3 possible arrows):

```
public class LCS {
   private final int dp[][];
   private final int par[][];
   private final String A, B;

   public LCS(String A, String B) {
      this.A = A;
      this.B = B;
      dp = new int[A.length() + 1][B.length() + 1];
      par = new int[A.length() + 1][B.length() + 1];
   }

   public int longestCommonSubsequence() {
        for (int i = 0; i < A.length(); ++i) {
            for (int j = 0; j < B.length(); ++j) {
                // Skip either character.
               if (dp[i][j+1] < dp[i+1][j]) {
                  dp[i+1][j+1] = dp[i+1][j];
                  par[i+1][j+1] = 0;
               } else {
                  dp[i+1][j+1] = dp[i][j+1];
                  par[i+1][j+1] = 1;
               }
                // If they are equal, then extend previous matching.
                if (A.charAt(i) == B.charAt(j) && dp[i+1][j+1] < 1 + dp[i][j]) {
                  dp[i+1][j+1] = 1 + dp[i][j];
                  par[i+1][j+1] = 2;
                }
            }
        }
        int cur_i = A.length(), cur_j = B.length();
        StringBuilder builder = new StringBuilder();
```

```
            while (cur_i > 0 && cur_j > 0) {
             if (par[cur_i][cur_j] == 0) --cur_j;
             else if (par[cur_i][cur_j] == 1) --cur_i;
             else {
                --cur_i; --cur_j;
                builder.append(A.charAt(cur_i));
             }
            }
            builder.reverse();
            System.out.println(builder.toString());
            return dp[A.length()][B.length()];
        }

        public static void main(String[] args) {
            LCS lcs = new LCS("BDCABA", "ABCBDAB");
            System.out.println(lcs.longestCommonSubsequence());
        }
    }
```

The parent table for the two example strings is given below and one possible path to obtain the LCS is shown in red:

	∅	**A**	**B**	**C**	**B**	**D**	**A**	**B**
∅	←	←	←	←	←	←	←	←
B	←	↑	↖	←	←	←	←	←
D	←	↑	↑	↑	↑	↖	←	←
C	←	↑	↑	↖	←	↑	↑	↑
A	←	↖	↑	↑	↑	↑	↖	←
B	←	↑	↖	↑	↖	←	↑	↖
A	←	↑	↑	↑	↑	↑	↖	↑

Further reading: See Project **??** for how to recover the LCS in $\mathcal{O}(\min(n, m))$ space.

(g) Consider the sequences `ABC` and `ACB`, these have two LCSs, namely `AB` and `AC` of length 2.

2 Greedy algorithms

Recommended reading/useful links:

- CLRS Chapter 16
- Jeff Erickson's Algorithms Chapter 4

Exercise 2.C.6 [Huffman encoding]

(a) What problem does Huffman encoding solve?

(b) Describe Huffman's encoding algorithm.

(c) Show the steps of the algorithm on Figure 3.1 of the lecture notes.

(d) Describe how you would implement this algorithm. What is the time complexity of your implementation? (*Hint:* You may find it beneficial to use min-heap)

(e) (optional) Implement the Huffman encoding algorithm.

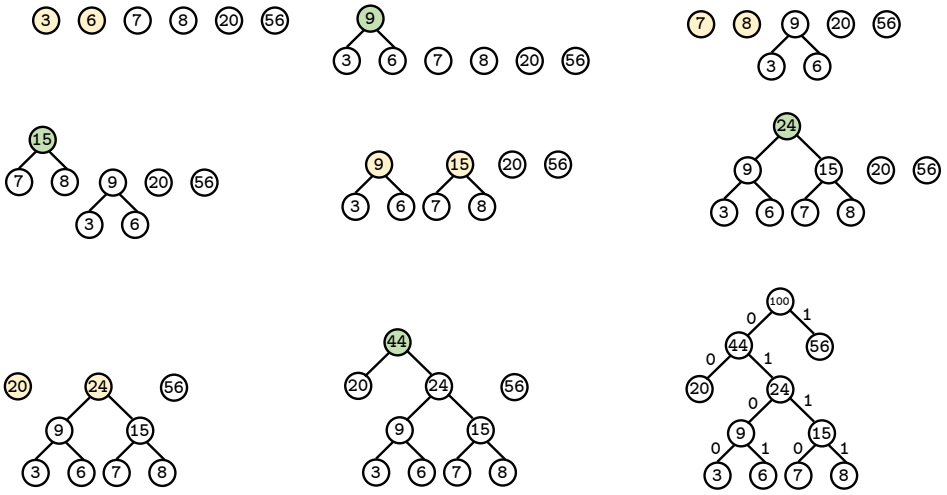
(f) (++) Prove that Huffman's algorithm solves the problem you described in (a).

(g) (optional) How can you implement Huffman's encoding algorithm in $\mathcal{O}(n)$ time if you are given the frequencies in sorted order?

(a) You are given N items and each one has a frequency of occurrence f_i. Assign 0/1 codewords (which are *prefix-free*) such that the expected length of an encoded word (where the probability of picking the i-th word is $p_i = f_i / \sum_{j=1}^{N} f_j$) is minimised.

(b) In Huffman's algorithm, every item is a group on its own with weight f_i. Then the smallest two values f_i and f_j are merged and create a group with weight $f_i + f_j$. Then repeat until there is a single element (of weight 1). This process of merging creates a binary tree. By assigning a value of 0 to the edges between the parent and the left child and a value of 1 to the edges between the parent and the right child, we define the codeword of each leaf and hence of each original item, by the path from the root to the leaf.

(c) Here is one possible execution and codeword mapping;



(d) Store the candidate items inside a min priority queue. While there are more than two elements in the data structure pick the two with the smallest weight and merge them, i.e. insert a new item with weight equal to their sum. Alongside with the weight it is helpful to keep a pointer to the root of the partial binary tree that is being formed.

(e) Here is an implementation in C++.

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <string>
#include <vector>

namespace sorting {

struct HuffmanTreeNode {

  virtual void print(const std::string&) = 0;

  virtual void generateTable(std::vector<std::string>& table, const
      std::string& pref = "") = 0;
};

struct HuffmanLeafNode : public HuffmanTreeNode {
  size_t word_id;

  HuffmanLeafNode(size_t word_id) : word_id(word_id) {}

  void print(const std::string& s) override {
    std::cout << s << " : " << word_id << std::endl;
  }

  void generateTable(std::vector<std::string>& table, const std::string& pref =
      "") override {
    if (table.size() <= word_id) table.resize(word_id + 1);
```

```
    table[word_id] = pref;
  }
};

struct HuffmanTreeInternalNode : public HuffmanTreeNode {
  HuffmanTreeNode* left, *right;

  HuffmanTreeInternalNode(HuffmanTreeNode* left, HuffmanTreeNode* right) :
      left(left), right(right) {}

  void print(const std::string& s) override {
    left->print(s + "0");
    right->print(s + "1");
  }

  void generateTable(std::vector<std::string>& table, const std::string& pref =
      "") override {
    left->generateTable(table, pref + "0");
    right->generateTable(table, pref + "1");
  }

  ~HuffmanTreeInternalNode() {
    delete left;
    delete right;
  }
};

template<class T>
using min_priority_queue = std::priority_queue<T, std::vector<T>,
    std::greater<T>>;

HuffmanTreeNode* generateHuffmanTree(const std::vector<size_t>&
    frequency_table) {
  min_priority_queue<std::pair<size_t, HuffmanTreeNode*>> pq;
  for (size_t i = 0; i < frequency_table.size(); ++i) {
    pq.push({ frequency_table[i], new HuffmanLeafNode(i) });
  }
  while (pq.size() != 1) {
    auto tp1 = pq.top(); pq.pop();
    auto tp2 = pq.top(); pq.pop();

    std::cout << "Merging : " << tp1.first << " + " << tp2.first << std::endl;
    pq.push({ tp1.first + tp2.first, new HuffmanTreeInternalNode(tp1.second,
        tp2.second) });
  }

  return pq.top().second;
}

}

int main() {
  std::vector<size_t> freqs({ 20, 3, 6, 7, 8, 56 });

  sorting::HuffmanTreeNode* tree = sorting::generateHuffmanTree(freqs);
  std::vector<std::string> table;
  tree->generateTable(table);

  // std::vector<std::string> expected({ "00", "0100", "0101", "0110", "0111",
      "1" });
  // EXPECT_EQ(table, expected);
  delete tree;
```

```
    return 0;
}
```

(f) We will prove by contradiction that the Huffman algorithm produces an encoding that minimises the expected length. We will do this in the following steps:

1. Consider the smallest (in terms of elements) instance where Huffman encoding does not produce the optimum.
2. Show that either x_1 x_2 is matched to a different element (x_is are assumed to be sorted)
3. Show that changing the mapping to (x_1, x_2) leads to an at least as good encoding.

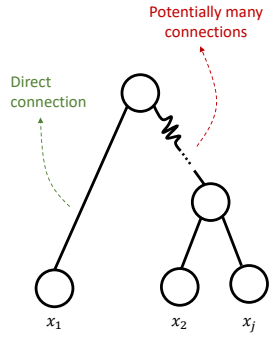
Step 1: Consider the smallest (in terms of elements) instance $x_1, x_2, \ldots, x_n$, where Huffman encoding does not produce the optimum.

Step 2: If x_1 is matched to x_2, then it means that the resulting set $(x_1 + x_2), x_3, \ldots, x_n$ has a different matching than the optimum. But this has $n-1$ elements, so it contradicts the assumption of it being the smallest counterexample (**step 1**).

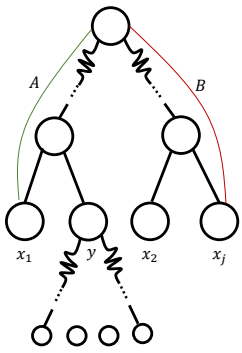
Step 3: So we can now assume that x_1 or x_2 is matched to a different element. The analysis below works for assuming either of these is mapped to a different value. So, let's assume that x_2 is matched to x_j ($j > 2$). Then there are two cases for x_1:

a it is matched to some ancestor of (x_2, x_j)

b it is matched to some other node y



Step 3 (a): x_1 is matched to some ancestor of (x_2, x_j).
The path to the root for x_j (and x_2) is at least as long as that of x_1. Hence, if we swap node x_1 with node x_j, then we get a encoding with smaller penalty (same reasoning as in re-arrangement inequality).



Step 3 (b): x_1 is matched to some node y
By assumption we know that $x_1 \leq x_j$ and $x_2 \leq y$. We consider three cases based on the path lengths:

- If $A > B$: swap y with x_2 and total decreases
- If $B > A$: swap x_1 with x_j and total decreases
- If $A = B$: swap x_1 with x_2 and Huffman encoding still leads to the same value

In either of these cases the matching (x_1, x_2) produces an at least as good solution