Contents

1 Time complexity

If the code has a time complexity of $\mathcal{O}(f(n))$ and the input size is m, then we should be okay if $f(m) < 100,000,000 \times time$.

2 Search

2.1 DFS

```
graph = {'A': set(['B', 'C']}

def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

2.2 BFS

```
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}
```

3 Grafer

3.1 Prefixträd

Prefixträd - eller Trie som de också kallas - är ett träd för snabb uppslagning och sökning bland ord. När man lägger till nya ord i en Trie så kan man säga att varje nod representerar en bokstav, och varje båge representerar att bokstavskombinationen är möjlig. Tack vare denna struktur så går det snabbt att kolla upp om ett ord finns i Trie:en (O(n), där n är längden på ordet) och det går snabbt att kolla upp om ett ord är ett giltigt prefix till något annat ord (O(m), där m är längden på prefix-ordet).

```
#include <string>
using std::string;

struct TrieNode {
        TrieNode* neighbors[26];
        bool endword;
};

void addWord(TrieNode* root, string word) {
        TrieNode* current = root;
        for(auto letter: word) {
                if( current->neighbors[letter-'a'] == nullptr ) {
                        TrieNode* newNode = new TrieNode;
                        for(int i=0; i<26; ++i) newNode->neighbors[i] = nullptr;
                        newNode->endword = false;
                        current->neighbors[letter-'a'] = newNode;
```

```
                        current = newNode;
                }
                else {
                        current = current->neighbors[letter-'a'];
                }
        }
        current->endword = true;
}

bool isPrefix(TrieNode* root, string word) {
        TrieNode* current = root;
        for(auto letter: word) {
                if( current->neighbors[letter-'a'] == nullptr ) return false;
                current = current->neighbors[letter-'a'];
        }
        return true;
}

int main() {
        TrieNode* root = new TrieNode;
        for(int i=0; i<26; ++i)
        {
        root->neighbors[i] = nullptr;
        root->endword = false;
        }
        addWord(root, "app");
        addWord(root, "apa");
        isPrefix(root, "ap"); //true
        return 0;
}
```

3.2 Minsta uppsrännande träd - Kruskal

```
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
    parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
            if rank[root1] == rank[root2]: rank[root2] += 1

def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)

    minimum_spanning_tree = set()
```

```
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            minimum_spanning_tree.add(edge)
    return minimum_spanning_tree

graph = {
'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
'edges': set([
(1, 'A', 'B'),
(5, 'A', 'C'),
(3, 'A', 'D'),
(4, 'B', 'C'),
(2, 'B', 'D'),
(1, 'C', 'D'),
])
}
minimum_spanning_tree = set([
(1, 'A', 'B'),
(2, 'B', 'D'),
(1, 'C', 'D'),
])
assert kruskal(graph) == minimum_spanning_tree
```

3.3 Bipartit

3.4 Bellman-ford

An algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.[1] It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

```
"""
The Bellman-Ford algorithm
Graph API:
iter(graph) gives all nodes
iter(graph[u]) gives neighbours of u
graph[u][v] gives weight of edge (u, v)
"""

# Step 1: For each node prepare the destination and predecessor
def initialize(graph, source):
    d = {} # Stands for destination
    p = {} # Stands for predecessor
    for node in graph:
        d[node] = float('Inf') # We start admiting that the rest of nodes are very very far
        p[node] = None
    d[source] = 0 # For the source we know how to reach
    return d, p

def relax(node, neighbour, graph, d, p):
    # If the distance between the node and the neighbour is lower than the one I have now
    if d[neighbour] > d[node] + graph[node][neighbour]:
        # Record this lower distance
        d[neighbour]  = d[node] + graph[node][neighbour]
        p[neighbour] = node

def bellman_ford(graph, source):
    d, p = initialize(graph, source)
```

```
    for i in range(len(graph)-1): #Run this until is converges
        for u in graph:
            for v in graph[u]: #For each neighbour of u
                relax(u, v, graph, d, p) #Lets relax it

    # Step 3: check for negative-weight cycles
    for u in graph:
        for v in graph[u]:
            assert d[v] <= d[u] + graph[u][v]

    return d, p


def test():
    graph = {
            'a': {'b': -1, 'c':  4},
            'b': {'c':  3, 'd':  2, 'e':  2},
            'c': {},
            'd': {'b':  1, 'c':  5},
            'e': {'d': -3}
            }

    d, p = bellman_ford(graph, 'a')

    assert d == {
                'a':  0,
                'b': -1,
                'c':  2,
                'd': -2,
                'e':  1
                }

    assert p == {
                'a': None,
                'b': 'a',
                'c': 'b',
                'd': 'e',
                'e': 'b'
                }
```

3.5 Dijkstra

3.6 Topological sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering.

```
from collections import defaultdict
from itertools import takewhile, count

def sort_topologically(graph):
    levels_by_name = {}
    names_by_level = defaultdict(set)

    def walk_depth_first(name):
        if name in levels_by_name:
            return levels_by_name[name]
        children = graph.get(name, None)
        level = 0 if not children else (1 + max(walk_depth_first(lname) for lname in children))
        levels_by_name[name] = level
        names_by_level[level].add(name)
```

```
        return level

    for name in graph:
        walk_depth_first(name)

    return list(takewhile(lambda x: x is not None, (names_by_level.get(i, None) for i in count())))


graph = {
        1: [2, 3],
        2: [4, 5, 6],
        3: [4,6],
        4: [5,6],
        5: [6],
        6: []
    }
```

4 Strings

4.1 Matching

Följande kod kan användas för att hitta alla index som en sträng förekommer i en annan sträng. Bara att kalla på KMPSearch("sträng", "sträng två"), så får man en vector med alla index i linjär tid (med avseende på strängen man söker i).

```
#include <vector>
#include <string>
#define let const auto
using namespace std;

vector<int> BuildKMPTable(const string& pattern) {
    let sz = (int) pattern.size();

    vector<int> table(pattern.size() + 1);
    table[0] = -1;
    int pos = 0, cnd = -1;

    while (pos < sz) {
        while (cnd >= 0 && pattern[cnd] != pattern[pos]) {
            cnd = table[cnd];
        } // Jump back until found match
        pos++; cnd++;
        table[pos] = (pattern[pos] == pattern[cnd]) ? table[cnd] : cnd;
    }

    return table;
}

vector<int> KMPSearch(const string& pattern, const string& text) {
    vector<int> indices;

    let table = BuildKMPTable(pattern);
    let sz = (int) pattern.size();
    int i = 0;

    for (int pos = 0; pos < (int) text.size(); ++pos) {
        while (i >= 0 && text[pos] != pattern[i]) {
            i = table[i];
        } // Jump back until text matches
        i++;
```

```
        if (i >= sz) {
            indices.push_back(pos - i + 1);
            i = table[i];
        }
    }

    return indices;
}
```

5 DP

5.1 Top down (no mutables (lists) as input - tuples?)

```
import functools

@functools.lru_cache(maxsize=None)
def fib(num):
    if num < 2:
        return num
    else:
        return fib(num-1) + fib(num-2)

# fib.cache_info()
```

5.2 Knapsack problem

```
# (Name, weight, value)
items = (
        ("map", 9, 150), ("compass", 13, 35), ("water", 153, 200), ("sandwich", 50, 160),
        ("glucose", 15, 60), ("tin", 68, 45), ("banana", 27, 60), ("apple", 39, 40)
        )

def knapsack01_dp(items, limit):
    table = [[0 for w in range(limit + 1)] for j in range(len(items) + 1)]

    for j in range(1, len(items) + 1):
        item, wt, val = items[j-1]
        for w in range(1, limit + 1):
            if wt > w:
                table[j][w] = table[j-1][w]
            else:
                table[j][w] = max(table[j-1][w], table[j-1][w-wt] + val)

    result = []
    w = limit
    for j in range(len(items), 0, -1):
        was_added = table[j][w] != table[j-1][w]

        if was_added:
            item, wt, val = items[j-1]
            result.append(items[j-1])
            w -= wt

    return result


bagged = knapsack01_dp(items, 40)
print("Bagged", bagged)
```

6 Misc

6.1 Coin changing

```
# T: an array containing the values of the coins
# L: integer wich is the total to give back
# Output: Minimal number of coins needed to make a total of L
def dynamicCoinChange( T, L ):
        Opt = [0 for i in range(0, L+1)]
        n = len(T)
        for i in range(1, L+1):
                smallest = float("inf")
                for j in range(0, n):
                        if (T[j] <= i):
                                smallest = min(smallest, Opt[i - T[j]])
                Opt[i] = 1 + smallest
        return Opt[L]
```

7 Useful libraries

7.1 Python

7.2 C++

8 Math

8.1 Combinatoric