# Algorithm Cheat Sheet (Pageless)

## Contents



## 1 QuickSelect

QuickSelect is an algorithm to find the **k-th smallest (or largest) element** in an unsorted array. It is related to QuickSort but only recurses on the side containing the $k$-th element.

- **Average time complexity:** $O(n)$
- **Worst-case time complexity:** $O(n^2)$
- **Space complexity:** $O(1)$ (in-place, recursive stack $O(\log n)$ on average)

### 1.1 Implementation

```cpp
// Partition function: rearranges elements around a pivot
// After partitioning:
// - elements <= pivot are on the left
// - elements > pivot are on the right
// Returns the final index of the pivot
int partition(vector<int>& arr, int left, int right) {
    // Randomly pick a pivot index to reduce worst-case
    int pivot_idx = left + rand() % (right - left + 1);
    int pivot = arr[pivot_idx];

    // Move pivot to the end temporarily
    swap(arr[pivot_idx], arr[right]);

    int i = left;  // i points to the next position for swapping smaller elements
    for (int j = left; j < right; j++) {
        if (arr[j] <= pivot) {    // If element <= pivot, move it to left
            swap(arr[i], arr[j]);
            i++;
        }
    }
    // Place pivot in its correct sorted position
    swap(arr[i], arr[right]);
    return i; // return the index of the pivot
}

// QuickSelect: finds the k-th smallest element (1-indexed)
int quickSelect(vector<int>& arr, int left, int right, int k) {
    if (left == right) return arr[left]; // only one element

    // Partition the array and get pivot index
    int pivot_idx = partition(arr, left, right);
    int count = pivot_idx - left + 1; // number of elements <= pivot

    if (count == k) {
        return arr[pivot_idx]; // pivot is the k-th smallest element
    } else if (k < count) {
        // k-th element lies in left partition
        return quickSelect(arr, left, pivot_idx - 1, k);
    } else {
        // k-th element lies in right partition
        // adjust k because we discard left partition
        return quickSelect(arr, pivot_idx + 1, right, k - count);
    }
}
```

### 1.2 Practice Problems

- K-th Largest Element in an Array (LeetCode)

## 2 Trie (Prefix Tree)

Efficient data structure for string retrieval problems (prefixes, alphabets, etc).

### 2.1 Implementation

```cpp
const int ALPHABET = 26; // number of lowercase letters

struct TrieNode {
    TrieNode *children[ALPHABET]; // pointers to child nodes
    int terminal;                 // number of words ending at this node
};

// Create and initialize a new Trie node
TrieNode *new_node() {
    TrieNode *node = new TrieNode;
    for(int i = 0; i < ALPHABET; ++i)
        node->children[i] = nullptr; // initialize all children to null
    node->terminal = 0;               // no word ends here yet
    return node;
}

// Insert a string into the trie
void insert(TrieNode *node, string s) {
    for (char c : s) {
        int idx = c - 'a';          // map char to index 0-25
        if (!node->children[idx])   // if child does not exist, create it
            node->children[idx] = new_node();
        node = node->children[idx]; // move to the child
    }
    node->terminal++;               // mark end of word
}

// Remove a string from the trie
void remove(TrieNode *node, string s) {
    for (char c : s) {
        int idx = c - 'a';
        if (!node->children[idx]) return; // word not found
        node = node->children[idx];
    }
    if (node->terminal > 0) node->terminal--; // unmark end of word
}

// Search for a string in the trie
bool search(TrieNode *node, string s) {
    for (char c : s) {
        int idx = c - 'a';
        if (!node->children[idx]) return false; // missing letter
        node = node->children[idx];
    }
    return node->terminal > 0; // true if word ends here
}
```

### 2.2 Practice Problems

- Codeforces 706D
- Word Search II (Leetcode)

## 3 Huffman Coding

Huffman Coding is a lossless data compression algorithm that assigns variable-length codes to characters based on their frequency. More frequent characters get shorter codes, while less frequent characters get longer codes.

- **Time complexity:** $O(n \log n)$ where $n$ is the number of unique characters
- **Space complexity:** $O(n)$ for the tree structure

### 3.1 Key Properties

- **Prefix-free codes:** The tree structure ensures that no character's code is a prefix of another character's code. This eliminates the need for separators between encoded characters and allows unambiguous decoding.
- **Optimal compression:** For a given set of character frequencies, Huffman coding produces the minimum possible average code length.

### 3.2 Algorithm Steps

1. Count frequency of each character
2. Create leaf nodes for each character and build a min-heap
3. While heap has more than one node:
   - Extract two nodes with minimum frequency
   - Create new internal node with these as children
   - Set frequency as sum of children frequencies
   - Insert back into heap
4. Root of remaining tree is the Huffman tree
5. Assign codes: left edge = 0, right edge = 1

### 3.3 Implementation

```cpp
struct HuffmanNode {
    char data;
    int freq;
    HuffmanNode* left;
    HuffmanNode* right;

    HuffmanNode(char d, int f) : data(d), freq(f), left(nullptr), right(nullptr) {}
    HuffmanNode(int f) : data('\0'), freq(f), left(nullptr), right(nullptr) {}
};

// Comparator for priority queue (min-heap based on frequency)
struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        if (a->freq == b->freq) {
            return a->data > b->data; // tie-breaker for consistency
        }
        return a->freq > b->freq;
    }
};

// Build Huffman tree from character frequencies
HuffmanNode* buildHuffmanTree(unordered_map<char, int>& freq) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> minHeap;

    // Create leaf nodes and add to heap
    for (auto& pair : freq) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }

    // Build tree bottom-up
    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top(); minHeap.pop();
        HuffmanNode* right = minHeap.top(); minHeap.pop();

        // Create internal node
        HuffmanNode* merged = new HuffmanNode(left->freq + right->freq);
        merged->left = left;
        merged->right = right;

        minHeap.push(merged);
    }

    return minHeap.top(); // root of Huffman tree
}

// Generate codes by traversing the tree
void generateCodes(HuffmanNode* root, string code, unordered_map<char, string>& codes) {
    if (!root) return;

    // Leaf node - store the code
    if (!root->left && !root->right) {
        codes[root->data] = code.empty() ? "0" : code; // handle single character case
        return;
    }

    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}

// Encode text using Huffman codes
string encode(string text, unordered_map<char, string>& codes) {
    string encoded = "";
    for (char c : text) {
        encoded += codes[c];
    }
    return encoded;
}

// Decode using Huffman tree
string decode(string encoded, HuffmanNode* root) {
    string decoded = "";
    HuffmanNode* current = root;

    for (char bit : encoded) {
        if (bit == '0') {
            current = current->left;
        } else {
            current = current->right;
        }

        // Reached leaf node
        if (!current->left && !current->right) {
            decoded += current->data;
            current = root; // reset to root
        }
    }

    return decoded;
}

// Complete Huffman coding example
void huffmanExample() {
    string text = "abracadabra";

    // Count frequencies
    unordered_map<char, int> freq;
    for (char c : text) {
        freq[c]++;
    }

    // Build tree and generate codes
    HuffmanNode* root = buildHuffmanTree(freq);
    unordered_map<char, string> codes;
    generateCodes(root, "", codes);

    // Print codes
    cout << "Huffman␣Codes:" << endl;
    for (auto& pair : codes) {
        cout << pair.first << ":␣" << pair.second << endl;
    }

    // Encode and decode
    string encoded = encode(text, codes);
    string decoded = decode(encoded, root);

    cout << "Original:␣" << text << endl;
    cout << "Encoded:␣" << encoded << endl;
    cout << "Decoded:␣" << decoded << endl;
}
```

### 3.4 Example

For text "abracadabra":

- Frequencies: a(5), b(2), r(2), c(1), d(1)
- Possible codes: a(0), b(10), r(110), c(1110), d(1111)
- Original: 88 bits (8 bits per char $\times$ 11 chars)
- Compressed: 27 bits ($5\times1 + 2\times2 + 2\times3 + 1\times4 + 1\times4$)
- Compression ratio: 69% reduction

### 3.5 Practice Problems

- Huffman Tree Construction
- Text Compression Problems

## 4 Unit Testing

How to write unit tests for your code.

### 4.1 Pytest

Simple Python program:

```python
# file: math_utils.py
def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot␣divide␣by␣zero")
    return a / b
```

```python
# file: test_math_utils.py
import pytest
from math_utils import add, divide

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(0, 0) == 0

def test_divide():
    assert divide(6, 3) == 2
    assert divide(5, 2) == 2.5

    # Check that dividing by zero raises an exception
    with pytest.raises(ValueError):
        divide(1, 0)
```

How to run the tests:

```
pytest test_math_utils.py
```

### 4.2 Catch2

```cpp
// file: math_utils.hpp
int add(int a, int b) {
    return a + b;
}

double divide(double a, double b) {
    if (b == 0) throw std::runtime_error("Cannot␣divide␣by␣zero");
    return a / b;
}
```

## 5 Disjoint Set Union (DSU)

## 6 Monotonic Stack

### 6.1 What is it and when to use it

A monotonic stack is a standard stack (LIFO), but the elements mantain a monotonic order at every time.

- **Increasing stack:** elements are in increasing order from bottom to top.
- **Decreasing stack:** elements are in decreasing order from bottom to top.

This allows us to efficiently answer the following questions:

- What is the **greater/smaller element**?
- Who **blocks visibility**? (think about the problem about heights and how many people can every person see on leetcode)
- How many elements are visibiel until I hit a blocker?

### 6.2 Principle

Whenever we want to push an element $x$, while the top of the stack violates the monotone property, pop it (and posibly update something related to the answer). After this, push $x$. This has $O(n)$ time complexity, because every element is pushed and popped at most once.

### 6.3 Problems

- Next Greater Element I (LeetCode)
- Next Greater Element II (LeetCode)
- Daily Temperatures (LeetCode)
- Largest Rectangle in Histogram (LeetCode)
- Maximal Rectangle (LeetCode)
- Number of Visible People in a Queue (LeetCode)