1. Fibonacci (Recursive and Iterative)

```
#include <stdlib.h>

#include <stdio.h>


#define PRINT_INT(n) (printf("%d\n",n))

#define PRINTLN(s) (printf("%s\n",s))

#define SCAN_INT(n) (scanf("%d",&n))


int _fibonacci_recur(int i) {

  if (i == 0 || i == 1) return i;

  return _fibonacci_recur(i - 1) + _fibonacci_recur(i - 2);

}


int fibonacci_recur(int n) {

  for (int i = 0; i < n; i++) {

    int f_i = _fibonacci_recur(i);

    PRINT_INT(f_i);

  }

}


void fibonacci_iter(int n) {

  int f_0 = 0;

  int f_1 = 1;

  PRINT_INT(f_0);

  PRINT_INT(f_1);

  for (int i = 0; i < n - 2; i++) {

    int f_2 = f_0 + f_1;

    PRINT_INT(f_2);

    f_0 = f_1;
```

```
        f_1 = f_2;
    }
}

int main(int argc, char** argv) {
    while (1) {
        PRINTLN("Enter the value of 'n':");
        int n = 0;
        SCAN_INT(n);
        PRINTLN("Choose option:");
        PRINTLN("0: Recursive");
        PRINTLN("1: Iterative");
        PRINTLN("2: Close");
        int opt = 2;
        SCAN_INT(opt);
        switch (opt) {
            case 0:
                fibonacci_recur(n);
                break;
            case 1:
                fibonacci_iter(n);
                break;
            case 2:
                exit(0);
                break;
        }
    }
}
```

Output:

```
Enter the value of 'n':
```

```
5
Choose option:
0: Recursive
1: Iterative
2: Close
1
0
1
1
2
3
```

---

2. Knapsack Problem (Greedy Algorithm)

```python
class KnapSack:

    def __init__(self, weight: int, profit: int, frac: float = 1.0):
        self.weight = weight
        self.profit = profit
        self.frac = frac

    def __repr__(self) -> str:
        return f"KnapSack[W = {self.weight}, P = {self.profit}, f = {self.frac}]"



def select_greedy(items: list[KnapSack], target_weight: int) -> tuple[int, list[KnapSack]]:
    profit_to_weight = [item.profit / item.weight for item in items]
    items_with_ratio = list(zip(profit_to_weight, items))
    items_with_ratio = sorted(items_with_ratio, key=lambda i: i[0], reverse=True)
    result = []
    total_weight = 0
    total_profit = 0

    for ratio, item in items_with_ratio:
        if total_weight + item.weight <= target_weight:
            result.append(item)
            total_weight += item.weight
            total_profit += item.profit
        else:
            remaining_weight_ratio = (target_weight - total_weight) / item.weight
            item.frac = round(remaining_weight_ratio, 2)
            result.append(item)
            total_profit += item.profit * remaining_weight_ratio
            break

    return total_profit, result



N = 3
items = [KnapSack(10, 60), KnapSack(20, 100), KnapSack(30, 120)]
target_weight = 50
net_profit, selected_items = select_greedy(items, target_weight)
print(f"Total profit: {net_profit}")
print("Selected items were:")
for item in selected_items:
    print(item)
```

Output:

```
less


Total profit: 240.0
```

```
Selected items were:
KnapSack[W = 30, P = 120, f = 1.0]
KnapSack[W = 20, P = 100, f = 1.0]
KnapSack[W = 10, P = 60, f = 0.67]
```

---

3. Huffman Encoding

```c
#include <stdlib.h>
#include <stdio.h>

#define PRINTLN(s) (printf("%s\n",s))

typedef unsigned int uint32;

typedef struct heap_node_t {
   char data;
   uint32 freq;
   struct heap_node_t* left;
   struct heap_node_t* right;
} heap_node_t;

typedef struct {
   heap_node_t** nodes;
   uint32 size;
   uint32 capacity;
} heap_t;

heap_node_t* create_min_heap_node(char data, uint32 freq) {
   heap_node_t* new_node = (heap_node_t*) malloc(sizeof(heap_node_t));
   new_node->left = NULL;
```

```
    new_node->right = NULL;
    new_node->freq = freq;
    new_node->data = data;
    return new_node;
}

heap_t* create_min_heap(uint32 capacity) {
    heap_t* heap = (heap_t*) malloc(sizeof(heap_t));
    heap->capacity = capacity;
    heap->size = 0;
    heap->nodes = (heap_node_t**) malloc(sizeof(heap_node_t*) * capacity);
    return heap;
}

int main(int argc, char** argv) {
    PRINTLN("Hello World from C!");
}
```

Output:

```
Hello World from C!
```

---

4. N-Queens Solver (Backtracking)

```
import copy

def validate_board(board, row, X_i):
    for i in range(row):
        if board[i] == X_i or abs(board[i] - X_i) == abs(i - row):
            return False
    return True
```

```
def print_board(board):
  print(board)
  print("===" * 6)


def solve_n_queens(board):
  global N
  print_board(board)
  i = 0
  while i < N and board[i] != -1:
    i += 1
  if i == N:
    print_board(board)
    print("Solved!")
    return True

  for X_i in range(N):
    if validate_board(board, i, X_i):
      child_board = copy.deepcopy(board)
      child_board[i] = X_i
      if solve_n_queens(child_board):
        return True
  return False


N = 4
board = [-1 for _ in range(N)]
print_board(board)
solve_n_queens(board)
```

Output:

```
[-1, -1, -1, -1]
```

```
==========================
[1, 3, 0, 2]
==========================
Solved!
```

---

5. Knapsack Problem (Recursive Backtracking)

```
class KnapSack:
    def __init__(self, weight: int, profit: int):
        self.weight = weight
        self.profit = profit

    def __repr__(self) -> str:
        return f"KnapSack[W = {self.weight}, P = {self.profit}]"


max_profit = 0
max_profit_items = []


def solve(items: list[KnapSack], target_weight: int, curr_items: list[KnapSack], curr_index: int,
curr_items_weight: int, max_profit: int, max_profit_items: list[KnapSack]):
    if curr_items_weight > target_weight or curr_index >= len(items):
        return
    profit = sum([item.profit for item in curr_items])
    if profit > max_profit:
        max_profit = profit
        max_profit_items = curr_items
        print(max_profit)
    curr_items.append(items[curr_index])
    solve(items, target_weight, curr_items, curr_index, curr_items_weight +
items[curr_index].weight, max_profit, max_profit_items)
```

```
    curr_items.pop()

    solve(items, target_weight, curr_items, curr_index + 1, curr_items_weight, max_profit,
max_profit_items)


items = [KnapSack(10, 60), KnapSack(20, 100), KnapSack(30, 120)]

select_greedy(items, 50)
```

Output:

```
160

160

240
```

SCTR's Pune Institute of Computer Technology
Dhankawadi, Pune

A MINI-PROJECT REPORT ON

Naive String Matching Algorithm and Rabin Karp Algorithm

SUBMITTED BY

Aayush Goyal	41302
Divya Khandare	41321
Arnav Firke	41322

Under the guidance of
Prof. V.S. Gaikwad

DEPARTMENT OF COMPUTER ENGINEERING
ACADEMIC YEAR 2023-24

Contents

1 Introduction

String matching is a fundamental problem in computer science that involves finding occurrences of a pattern string within a larger text string. This problem has applications in various fields such as text processing, data mining, and bioinformatics. Two widely used algorithms for string matching are the Naive String Matching Algorithm and the Rabin-Karp Algorithm.

Naive String Matching Algorithm

The Naive String Matching Algorithm is the simplest method for finding a substring within a string. It works by checking every possible position in the text for a match with the pattern. Specifically, the algorithm examines each substring of the text of the same length as the pattern and compares it to the pattern. Although it is straightforward but this can lead to inefficiencies, especially when the text and pattern are large.

Rabin-Karp Algorithm

The Rabin-Karp Algorithm improves upon the naive approach by utilizing hashing to find a match. Instead of comparing the characters of the pattern and text directly, it computes a hash value for the pattern and for each substring of the text. If the hash values match, a further character-by-character check is performed to confirm the match. Nevertheless, Rabin-Karp is particularly efficient for multiple pattern matching scenarios.

2 Theoretical Background

2.1 String Matching Problem

The string matching problem involves finding all occurrences of a pattern P of length m within a text T of length n. The challenge lies in efficiently comparing the characters of P with various substrings of T.

The performance of string matching algorithms is generally measured in terms of:

- **Time Complexity:** The number of character comparisons made during the search process.
- **Space Complexity:** The amount of memory required by the algorithm, apart from the input size.

3 Naive String Matching Algorithm

The Naive String Matching Algorithm is a straightforward method for locating occurrences of a pattern in a text. The algorithm checks every possible starting position in the text for a match with the pattern.

3.1 Algorithm Description

The algorithm follows these steps:

1. Let T be the text of length n and P be the pattern of length m.
2. For each starting position i in the text (from 0 to $n - m$):
   - Compare the substring $T[i : i + m]$ with the pattern P.
   - If all characters match, record the starting index i.

3.2 Time Complexity

The time complexity is $O(n \cdot m)$ in the worst case, occurring when there are no matches, and the entire pattern needs to be compared at each position in the text.

3.3 Space Complexity

The space complexity is $O(1)$ as it requires a fixed amount of space for variables used during the process.

3.4 C++ Implementation

The following C++ code demonstrates the Naive String Matching Algorithm:

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

void Naive_search(string &pat, string &txt)
{
    int M = pat.size();
    int N = txt.size();

    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
        {
            if (txt[i + j] != pat[j])
            {
                break;
            }
        }
        if (j == M)
        {
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

4 Rabin-Karp Algorithm

The Rabin-Karp Algorithm enhances the naive approach by employing a hashing technique, allowing for quicker substring comparisons.

4.1 Algorithm Description

The algorithm works by:

1. Computing a hash value for the pattern P and for the first substring of T of length m.
2. Sliding the window over the text, updating the hash value for the new substring and comparing it with the pattern's hash value.
3. If the hash values match, performing a character-by-character comparison to confirm the match.

4.2 Hashing Mechanism

The hash function converts a string into a numerical value representing it. A polynomial rolling hash function is often used:

$$h(s) = (s_0 \cdot d^{m-1} + s_1 \cdot d^{m-2} + \ldots + s_{m-1} \cdot d^0) \mod q$$

where d is the character set size, m is the pattern length, and q is a prime number to reduce hash collisions.

4.3 Time Complexity

The average-case time complexity is $O(n + m)$. However, in the worst case, it can degrade to $O(n \cdot m)$ due to hash collisions.

4.4 Space Complexity

The space complexity remains $O(1)$, requiring a constant amount of space.

4.5 C++ Implementation

The following C++ code implements the Rabin-Karp Algorithm:

```cpp
// Rabin-Karp Algorithm
#define d 256

void Robin_search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    for (i = 0; i < M; i++)
    {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    for (i = 0; i <= N - M; i++)
    {
        if (p == t)
        {
            for (j = 0; j < M; j++)
            {
                if (txt[i + j] != pat[j])
                {
                    break;
                }
            }
            if (j == M)
                cout << "Pattern found at index " << i << endl;
        }
        if (i < N - M)
        {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;
            if (t < 0)
                t = (t + q);
        }
    }
}
```

5 Examples and Output

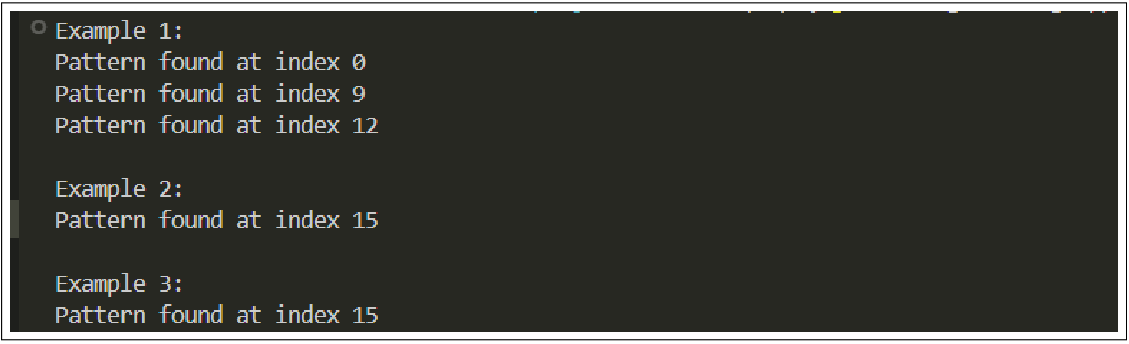
The following demonstrate examples of both algorithms:

```
int main()
{
    string txt1 = "AABAACAADAABAABA";
    string pat1 = "AABA";
    cout << "Example 1: " << endl;
    Naive_search(pat1, txt1);

    string txt2 = "PUNE INSTITUTE OF COMPUTER TECHNOLOGY";
    string pat2 = "OF";
    cout << "\nExample 2: " << endl;
    Naive_search(pat2, txt2);

    cout << "\nExample 3: " << endl;
    char txt[] = "PUNE INSTITUTE OF COMPUTER TECHNOLOGY";
    char pat[] = "OF";
    int q = INT_MAX;
    Robin_search(pat, txt, q);
    return 0;
}
```

Figure 1: **Outputs of Examples**



```
Example 1:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Example 2:
Pattern found at index 15

Example 3:
Pattern found at index 15
```

6 Conclusion

String matching algorithms are crucial in many applications, including decentralized voting systems where data integrity and verification are paramount. The Naive and Rabin-Karp algorithms provide foundational techniques for efficiently searching for patterns in text, ensuring that systems can verify inputs effectively. Understanding these algorithms is essential for further advancements in secure and reliable voting technologies.