

Stored Code Queries

Query:

Armstrong number

Response:

```
```python
def isArmstrong(num):
 num_str = str(num)
 n = len(num_str)
 sum = 0
 for digit in num_str:
 sum += int(digit) ** n
 return sum == num

number = int(input("Enter a number: "))

if isArmstrong(number):
 print(number, "is an Armstrong number")
else:
 print(number, "is not an Armstrong number")
```
```

Query:

Knapsack

Response:

```
```python
def knapsack(capacity, weights, values):
 n = len(values)
```

```
dp = [[0 for x in range(capacity + 1)] for y in range(n + 1)]
```

```
for i in range(1, n + 1):
```

```
 for w in range(1, capacity + 1):
```

```
 if weights[i-1] <= w:
```

```
 dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
```

```
 else:
```

```
 dp[i][w] = dp[i-1][w]
```

```
return dp[n][capacity]
```

```
values = [60, 100, 120]
```

```
weights = [10, 20, 30]
```

```
capacity = 50
```

```
max_value = knapsack(capacity, weights, values)
```

```
print(max_value)
```

```
values = [1,2,3,4,5]
```

```
weights = [1,2,3,4,5]
```

```
capacity = 5
```

```
max_value = knapsack(capacity, weights, values)
```

```
print(max_value)
```

```
values = [1,6,10,16]
```

```
weights = [1,2,3,5]
```

```
capacity = 7
```

```
max_value = knapsack(capacity, weights, values)
```

```
print(max_value)
```

```
```
```

Query:

knapsack in cpp

Response:

```
```cpp
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int knapsack(int capacity, const vector<int>& weights, const vector<int>& values) {
```

```
 int n = weights.size();
```

```
 if (n == 0 || capacity == 0) return 0;
```

```
 vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
```

```
 for (int i = 1; i <= n; ++i) {
```

```
 for (int w = 1; w <= capacity; ++w) {
```

```
 if (weights[i - 1] <= w) {
```

```
 dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]]);
```

```
 } else {
```

```
 dp[i][w] = dp[i - 1][w];
```

```
 }
 }
}
return dp[n][capacity];
}
```

```
int main() {
 int capacity;

 int numItems;

 cout << "Enter the knapsack capacity: ";
 cin >> capacity;

 cout << "Enter the number of items: ";
 cin >> numItems;

 vector<int> weights(numItems);
 vector<int> values(numItems);

 cout << "Enter the weights of the items: ";
 for (int i = 0; i < numItems; ++i) {
 cin >> weights[i];
 }

 cout << "Enter the values of the items: ";
 for (int i = 0; i < numItems; ++i) {
 cin >> values[i];
 }
}
```

```

}

int maxValue = knapsack(capacity, weights, values);

cout << "The maximum value that can be carried is: " << maxValue << endl;

return 0;
}
...

```

This C++ code implements a dynamic programming solution for the 0/1 knapsack problem. It's optimized by using a 2D vector for dynamic programming instead of recursion, which avoids redundant calculations and improves efficiency, especially for larger inputs. The time complexity is  $O(nW)$  where  $n$  is the number of items and  $W$  is the knapsack capacity. The space complexity is  $O(nW)$  as well. The code also includes user input for capacity and item details.

A Python solution using the same dynamic programming approach would be:

```

```python
def knapsack(capacity, weights, values):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:

```

```
dp[i][w] = dp[i - 1][w]
```

```
return dp[n][capacity]
```

```
capacity = int(input("Enter the knapsack capacity: "))
```

```
numItems = int(input("Enter the number of items: "))
```

```
weights = list(map(int, input("Enter the weights of the items: ").split()))
```

```
values = list(map(int, input("Enter the values of the items: ").split()))
```

```
maxValue = knapsack(capacity, weights, values)
```

```
print("The maximum value that can be carried is:", maxValue)
```

```
...
```

Query:

insomnia

Response:

Not a code problem. Insomnia

Query:

bfs

Response:

Please provide the code or problem statement you'd like me to review. I need the context to check if a BFS (Breadth-First Search) implementation is working correctly and to provide an optimized solution in Python.

Query:

No provide me complete code for breadth first search

Response:

```
```python
```

```
from collections import deque

def breadth_first_search(graph, start):

 visited = set()

 queue = deque([start])

 visited.add(start)

 while queue:

 vertex = queue.popleft()

 print(vertex, end=" ")

 for neighbor in graph[vertex]:

 if neighbor not in visited:

 visited.add(neighbor)

 queue.append(neighbor)

graph = {
 'A': ['B', 'C'],
 'B': ['D', 'E'],
 'C': ['F'],
 'D': [],
 'E': ['F'],
 'F': []
}

breadth_first_search(graph, 'A')
```

```

Query:

give bfs code in cpp

Response:

```cpp

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

vector<int> bfs(vector<vector<int>>& graph, int startNode) {

int n = graph.size();

vector<int> visited(n, 0);

vector<int> result;

queue<int> q;

q.push(startNode);

visited[startNode] = 1;

while (!q.empty()) {

int u = q.front();

q.pop();

result.push\_back(u);

for (int v : graph[u]) {



```

 if (!visited[v]) {
 visited[v] = 1;
 q.push(v);
 }
 }
}

return result;
}

int main() {
 vector<vector<int>> graph = {
 {1, 2},
 {2},
 {3},
 {}
 };

 vector<int> result = bfs(graph, 0);

 for (int node : result) {
 cout << node << " ";
 }

 cout << endl;

 return 0;
}
...

```