Heaven's Light is Our Guide


# Rajshahi University of Engineering and Technology

CSE-2102

Lab-6

---

# Discrete Mathematics Sessional

---

*Submitted To:*
Suhrid Shakhar Ghosh
Asst. Professor
Dept. of Computer Science & Engineering

*Submitted By :*
Kaif Ahmed Khan
ID: 2103163
Dept. of Computer Science & Engineering

February 25, 2024

# 1 Cashier's Algorithm

Given an integer n, use the greedy algorithm to find the change for n cents using quarters, dimes, nickels, and pennies.

## 1.1 Algorithm

**procedure** *change*$(c_1, c_2, \ldots, c_r$ *,where* $c_1 > c_2 > \ldots > c_r)$
**for** $i := 1$ **to** $r$ **do**
  $d_i := 0$
  **while** $n \geq c_i$ **do**
    $d_i := d_i + 1$
    $n := n - c_i$
  **end**
**end**

**Algorithm 1:** Cashier's Algorithm

## 1.2 Source Code

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

int main(){
  int coin[4] = {25, 10, 5, 1};
  int count[4] = {0};
  int n;
  cout << "n: ";
  cin >> n;
  for(int i = 0; i < 4; i++){
    while(n >= coin[i]){
      count[i] = count[i] + 1;
      n = n - coin[i];
    }
  }
  for(int i =0; i < 4; i++){
    cout << coin[i]<< ": " << count[i] << endl;
  }
}
```

## 1.3 Output

```
n: 31
25: 1
10: 0
5: 1
1: 1
```

Figure 1: Output showing change for 31 cents

### 1.4 Analysis

The code implements the algorithm 1. The values for the coins are stored in the `coin` array. For each coin the total count is stored in the `count` array. Then for each coin the coin count is increased by 1 if remaining cent is greater than the coin value. Finally in the end the counts are printed in the console.

## 2 Greedy Scheduling

Given the starting and ending times of n talks, use the appropriate greedy algorithm to schedule the most talks possible in a single lecture hall.

### 2.1 Source Code

```cpp
#include <bits/stdc++.h>
#include <utility>
using namespace std;
typedef pair<int,int> pii;

int main(){
  int n;
  cout << "Number of Talks: ";
  cin >> n;
  vector<pair<pii,pii>> time; // (endtime, starttime)
  vector<pair<pii,pii>> talks; // (endtime, starttime)

  for(int i = 0; i < n; i++){
    int hs,ms, he,me;
    cout << "Talk#" << i+1 << ": ";
    scanf("%d:%d %d:%d", &hs, &ms, &he, &me);
    pii x = make_pair(hs, ms);
    pii y = make_pair(he, me);
    time.push_back(make_pair(y, x));
  }
  sort(time.begin(), time.end()); // sort time by endtime
  talks.push_back(*time.begin());
  int j = 0;
  for(int i = 1; i < time.size(); i++){
    pii st = time[i].second;
    pii et = time[j].first;
    if(st < et) continue;
    else{
      talks.push_back(time[i]);
      j = i;
    }
  }
  cout << "Schedule: " << endl;
  for(int i =0; i < talks.size(); i++){
    pii st =  talks[i].second;
    pii et = talks[i].first;

    cout << st.first << ":";
    ((st.second == 0) ? cout << "00" : cout << st.second) << "\t";
    cout << et.first << ":";
    ((et.second == 0) ? cout << "00" : cout << et.second) << "\n";
  }
}
```

### 2.2 Output

```
Number of Talks: 11
Talk#1: 9:00 9:45
Talk#2: 9:30 10:00
Talk#3: 9:50 10:15
Talk#4: 10:00 10:30
Talk#5: 10:10 10:25
Talk#6: 10:30 10:55
Talk#7: 10:15 10:45
Talk#8: 10:30 11:00
Talk#9: 10:45 11:30
Talk#10: 10:55 11:25
Talk#11: 11:00 11:15
Schedule:
9:00    9:45
9:50    10:15
10:15   10:45
11:00   11:15
```

Figure 2: Schedules from 11 talks.

### 2.3 Analysis

According to the algorithm firstly the vector time is sorted on the basis of ending times. Then if the talk is compatible with the schedule the talk is pushed into the talks vector. The complexity of the code is $O(n)$ where n is the total number of talks.

# 3 Comparison between Bubble sort & Quick Sort

Determine the time taken by bubble sort and quick sort on a randomly generated list of numbers and plot the time in a graph.

## 3.1 Source Code

```cpp
#include <bits/stdc++.h>
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iomanip>
using namespace std;
using namespace std::chrono;


int main(){
  cout << "sample" << setw(18) << "quick sort" << setw(15) << "bubble sort" << endl;
  int sample[7] = {10000,20000,30000,40000,50000,60000,70000};
  for(int l = 0; l < 7; l++){
    vector<int> nums;
    vector<int> nums_for_bubble;
    srand(time(0));
    for(int k = 0; k < sample[l]; k++){
        int x = rand() % 100007 + 1;
        nums.push_back(x);
        nums_for_bubble.push_back(x);
    }
    auto st = high_resolution_clock::now();
    sort(nums.begin(), nums.end());
    auto et = high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(et - st).count();
    time_taken *= 1e-9;
    // cout << "Quick sort: " << ((double)(endt - startt)/CLOCKS_PER_SEC) << endl;
    //bubble sort
    auto st2 = high_resolution_clock::now();
    for(int i = 0; i < nums_for_bubble.size(); i++){
      for(int j = 0; j < nums_for_bubble.size() - i; j++){
        if(nums_for_bubble[j] > nums_for_bubble[j+1]){
          int temp = nums_for_bubble[j+1];
          nums_for_bubble[j+1] = nums_for_bubble[j];
          nums_for_bubble[j] = temp;
        }
      }
    }
    auto et2 = high_resolution_clock::now();
    double time_taken_2 = chrono::duration_cast<chrono::nanoseconds>(et2 - st2).count();
    time_taken_2 *= 1e-9;
    cout << nums.size()<< setw(18) << time_taken << setw(15) << time_taken_2<< endl;
  }
}
```

### 3.2 Output

| sample | quick sort | bubble sort |
|---|---|---|
| 10000 | 0.00213967 | 0.509572 |
| 20000 | 0.00388683 | 1.99083 |
| 30000 | 0.00618295 | 4.45463 |
| 40000 | 0.00817428 | 8.10198 |
| 50000 | 0.0103573 | 12.8605 |
| 60000 | 0.0127978 | 18.281 |
| 70000 | 0.0153016 | 24.8098 |

Figure 3: Output showing time taken by bubble sort and quick sort

### 3.3 Graph

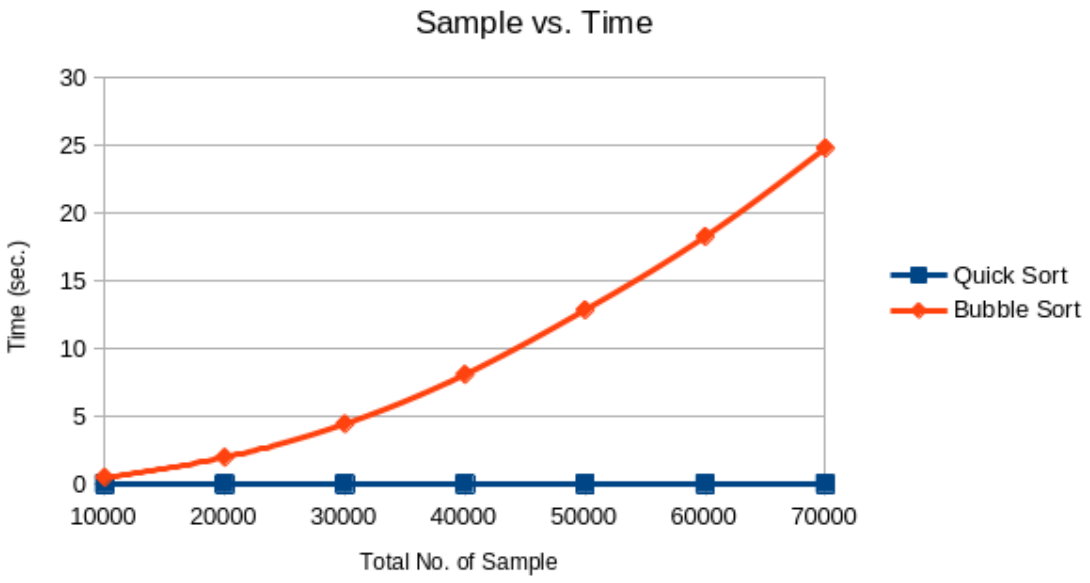


Figure 4: Output showing time taken by bubble sort and quick sort

### 3.4 Analysis

From the graph it is clear that bubble sort takes more time than quick sort. The time for quick sort is in millisecond range whereas time for bubble sort is in second range. The times are measured by the std::chrono API in C++.
According to the complexity, bubble sort has a complexity of $O(n^2)$. The graph also reflects this property following a quadratic trend. In case of quick sort the complexity is $O(n \cdot logn)$. The graph is very close to the horizontal axis line as the time is in milliseconds.