알고리즘분석

**HW#3 Report**

2011147068 김정환

2011147115 허재화

2012147562 최인호

1. **목차**
   1. **문제 분석**
   2. **알고리즘**
   3. **구현**
      1. **Class 설명**
      2. **알고리즘에 대한 세부구현**
   4. **토의 과정**
      1. **Big-O notation을 통한 분석**
      2. **구현 과정에서 겪은 어려움**

**문제 분석**

* 주어진 금액을 지불할 때, 지불한 동전의 개수 및 거스름돈으로 받는 동전의 개수가 최소로 되게 지불하는 방법을 찾는 것이다.
* 이 때, 가질 수 있는 동전 type과 지불 할 수 있는 동전의 개수는 정해져 있는 범위 내에 있어야 한다.
* 만약 지불 할 수 없는 금액이라면 -1을 출력한다. 예를 들어 6원을 지불해야 하는데 최소 동전이 5원인 경우가 있다.
* Input
  + 첫 줄은 test case number이 들어 오고, 그 다음부터, 세 줄씩 Input case가 들어온다.
  + Input case의 첫 줄엔 동전 type의 수와 지불해야 될 금액이 나온다.
  + 두 번째 줄엔 동전 type의 값으로 이뤄진 벡터가 나온다. // 이하 Type
  + 마지막엔 각 동전의 개수에 대한 벡터가 주어진다. // 이하 Range
* Output
  + 지불해야 할 동전의 min number 및 동전의 지불 현황 vector가 나온 뒤에, 거스름돈으로 받는 동전의 지불 현황 vector가 나온다.
  + 이 때, 지불하는 동전만이 Range의 범위를 만족하고 거스름돈 vector는 Range와 무관하다

1. **알고리즘**

* **이번 과제를 처음 봤을 때 우리가 생각한 큰 흐름은 다음과 같다.**
* 전체 알고리즘
  + 내가 만든 돈을 Money라고 하자.
  + T<=Money<=Type \* Range를 만족할건데 저 범위 내의 모든 Money 에 대해서 최소 동전을 갖는 경우를 찾아본다.
  + 즉, g(Money, Range, Type) 과 그 때의 Money 에 대한 f(Money-T. Type)을 구하여 일일이 모두 비교를 하는 방법.
  + \_f(Money, Type) 의 구현은 다음과 같다.
    - Type이 t0, t1… t (n-1) 이렇게 n개가 있을 때, 내림차순으로 저장했다고 하자
    - 이 때 t0 부터 썼을 때의 min 벡터
    - .t1 부터 썼을 때의 min벡터
    - … 이런 식으로 min 벡터를 구해 그 동전 개수의 합이 가장 작은 것을 찾는다.
    - 결국 ti 에 이르렀을 때 지금까지 구한 동전 개수의 합보다 Money/ti가 더 클 때가 올 것이고 그 때가 오면 종료를 한다.
  + \_g(Money, Range, Type)의 구현은 \_f에서 제약사항만 붙은 것이므로 그 방법은 유사하다.
* 여기서 중요한 것은, 돈을 지불하는 Customer는 각 동전에 대해 개수제한이 있고, 돈을 거슬러주는 사람(앞으로는 Clerk라 하겠다.)은 동전개수에 제한이 없다는 점이다. 위의 흐름에서 결국 함수 g(Money, Range, Type)는 Customer이 Money를 지불할 때, 가능한 가장 적은 동전의 개수를 구하는 함수이고, f(Money-T. Type)는 Customer이 T원을 지불해야 하는데 Money를 지불했다면 Money-T원을 거슬러 받아야 하고 그것은 곧 Clerk입장에서 Money-T원을 동전개수에 제한 없이 지불해야 한다는 것이므로 그때의 최적(최소) 동전개수를 구하는 함수가 된다. 그리고 여기서 T원을 지불해야 한다면 무조건 T원 이상의 돈을 지불하고 남는 금액을 거슬러 받아야 하고, Customer이 가지고 있는 모든 돈을 합친 것보다 더 많은 돈을 지불할 수는 없기 때문에T<=Money<=Type \* Range 라는 범위가 설정된다.
* g(Money, Range, Type)를 구현하는 과정에서 Dynamic Programming으로 g(Money, Range, Type)를 구현할 수 있다고 생각하였다. 그 식은 다음과 같다.
  + Coin[1~n]은 각 Coin의 가치에 따라 오름차순으로 정렬해 놓는다.
  + Coin[1]~Coin[n]중에서 Coin[1]부터 Coin[k]까지의 동전을 썼을 때, money를 지불하는 최소동전개수 Dy[money]는 **Dy[money]** = **MIN(**Dy[money], Dy[money-Coin[1]]+1, Dy[money-Coin[2]]+1, …. , Dy[money-Coin[k]]+1**)** (여기에서 Dy[money-Coin[i]]+1 의 의미는 i번째 코인을 마지막에 한번 사용해서 money를 지불한다는 의미이고, 따라서 무조건 MIN값이라고 갱신하지 않는다. 여기에서 가장 중요한 조건인 Coin의 개수제한을 한번 더 확인해야 한다. 따라서 저 값들 중에서 Coin의 개수제한을 봤을 때 아직 사용이 가능하면서 최소 동전개수를 만드는 Dy[money-Coin[1]]+1를 찾아서 Dy값을 갱신해 나간다.)
  + Function DP{

For i = 1 ~ Coin 종류 개수

For j = Coin[i] ~ MAX\_VALUE {

For k = 1 ~ i {

If(Coin[k]의 개수제한에 걸리지 않는 k에 대해서){

**Dy[j]** = **MIN(**Dy[money], Dy[money-Coin[k]]+1**)**

}

}

}

* 그리고 Coin[k]의 개수제한(range)를 확인하기 위해서 항상 각각의 Dy값에 Coin벡터(각 Coin들을 몇 개씩 사용해서 나온 Dy값인지)를 저장해야 한다.
* 이렇게 해서 동전 개수에 제한이 있는 Customer이 특정 값을 지불할 때 최소 동전개수를 구했다. 그리고 Clerk는 동전 개수에 제한이 없으므로 같은 함수를 쓰고 동전 개수의 제한을 Max\_integer로 해주면 될 거라고 생각했다. 하지만 동전 개수에 제한이 없다면 각 Dy[j]값에 대하여 그 당시 사용하는 모든 동전의 종류 Coin[k]를 고려하지 않아도 된다는 것을 발견하였다. 어차피 Coin배열이 오름차순으로 정렬되어 있으므로 i번째 Coin까지 사용할 때 최적해는 i번째 Coin을 사용하는 경우와 사용하지 않는 경우, 두 값만 비교하면 된다.
* Function DP\_change{

For i = 1 ~ Coin 종류 개수

For j = Coin[i] ~ MAX\_VALUE {

**Dy[j]** = **MIN(**Dy[money], Dy[money-Coin[i]]+1**)**

}

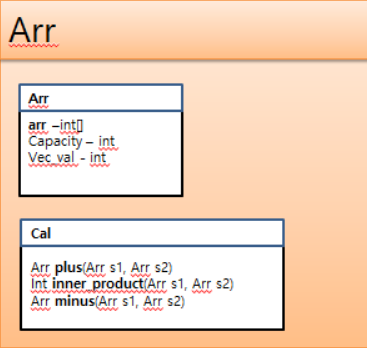
}

* 그리고 이렇게 구현하는 것이 그 시점에 사용하는 모든 Coin들에 대해서 고려하는 필요 없는 작업을 없애주므로 Performance 측면에서 더 유리하다.

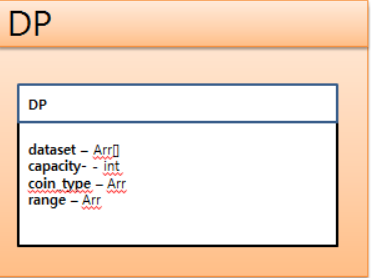
1. **구현**

**2.1 Class 설명**

* 프로그램은 Arr, DP, Run 총 3개의 Package로 구성 된다.
* **Arr** :알고리즘을 구현 시 vector를 표현하는 Arr 및 Arr의 계산을 위한 Cal이 있다.

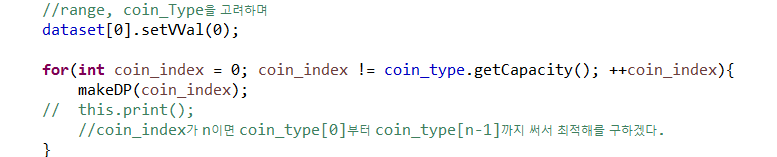


* + Arr : vector를 나타내기 위한 객체
  + Cal : 두 벡터의 내적을 구하거나 더하는 것 등 간단한 연산들을 위해 만든 객체
* **DP** :Dynamic Programming을 쓰기 위해 만든 객체



**2.2 알고리즘에 대한** 세**부 구현**

* [Customer이 돈을 지불하는 방법]

****

****

**if**(dataset[money].getVVal() > before.getVVal()+1){

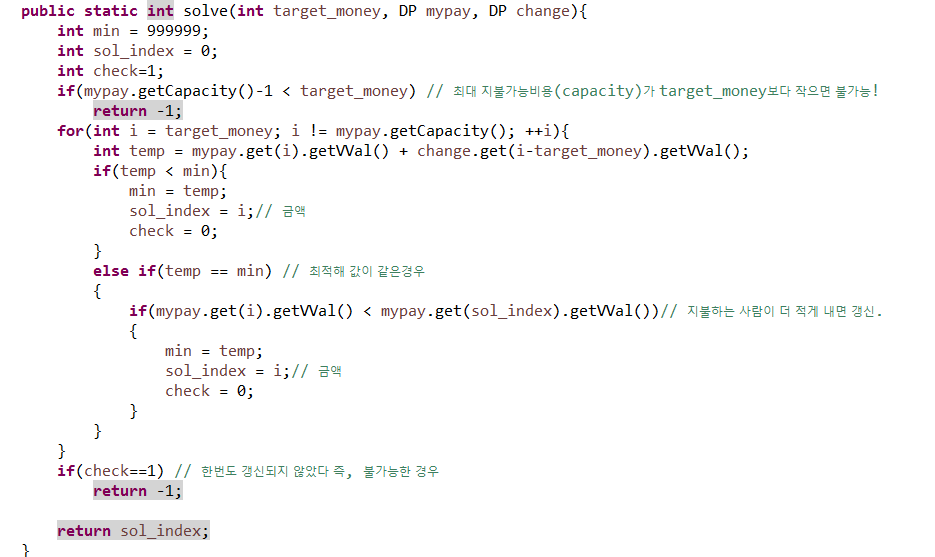
**if**(range.get(j) > before.get(j)){

**if**(min>before.getVVal()+1)

* + 제일 처음 나온 if문으로 일단 현재 dataset.getVVal()값보다 작은지를 판별한다. 그리고 그 값이 더 작아서 사용하기 위해서는 그 코인의 range에 걸리지 않아야 하므로, 두 번째 if문으로 j번째 코인의 range를 확인한다. 그리고 마지막 if문으로 그런 조건을 만족하는 모든 값 중 가장 작은 값을 찾아야 하므로 그 중에서 min값을 찾고 그 값으로 dataset의 val값을 갱신해 나간다. 갱신할 때 중요한 점은 참조한 이전 dataset의 각 Coin에 대한 사용 개수를 가지고 와서 거기에 이번에 새로 사용한 Coin종류를 1 더한 값으로 갱신해 줘야 한다는 점이다.
* [clerk가 돈을 지불하는 방법]



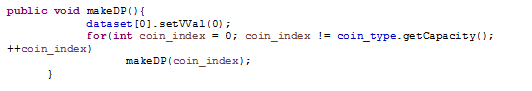
* Clerk는 동전 개수에 제한이 없기 때문에 Customer에 비해 간단하다. 단순히 coin\_index번째 코인을 그 순간에 한번 더 사용하는 것이 유리한지만 판단하면 된다. 그리고 그러는 것이 더 유리하다고 판단이 되면 값을 갱신하면서 이전의 Coin사용 개수에다가 coin\_index번째 코인을 하나 더 사용한 것으로 갱신해주면 된다.



* 마지막으로 solve함수에서는 구해진 Customer과 Clerk의 각 val에 대한 최적해를 이용하여 둘을 합쳤을 때 어떤 경우에 동전의 교환횟수가 최소로 되는지를 결정한다. 그런데 여기서 불가능한 경우를 처리해 주어야 한다. 불가능한 경우는 크게 두 가지로 나눌 수 있다.
  + Capacity가 target\_money보다 작은경우.
    - 이런 경우에는 일차적으로 Customer이 target\_money를 지불할 수 없는 상황이므로 불가능한 상황이다. 문제의 조건상 지불하는데 돈의 액수는 충분하다고 가정하므로 이런 경우는 없지만 우리가 사용한 알고리즘을 봤을 때 만약에 그런 조건이 없다면 이런 상황도 처리해야 한다.
  + 1번의 경우에 해당하지 않지만, 가능한 동전 조합으로 target\_money를 만들 수 없는 경우
    - 우리 알고리즘에서 처음에 0을 제외한 모든 money에 대한 dataset[]의 val을 MAX(999999)값으로 지정해 놓았는데, 이런 경우에는 그 i(customer이 지불)와 i-target\_money(clerk가 거슬러 줌)에 대해서 둘 중 하나의 dataset[]의 val값이 갱신되지 않는다. 따라서 모든 지불하고 거슬러주는 경우에 대해 Customer과 Clerk의 dataset[] 값이 둘 중 하나라도 갱신되지 않았으면 그것은 불가능한 경우로 봐줘야 한다. 그것은 처음에 min의 초기값을 MAX(999999)로 설정하고 그 min값보다 두 dataset의 val값을 합친 것이 작은 경우에 대해서만 sol\_index로 고려해 주므로 거기에서 걸러지고, 단 한번의 경우에도 sol\_index로 고려되지 않았다는 말은 지불 가능한 조합이 한번도 없었다는 의미이므로 그때는 불가능한 경우로 보고 -1을 출력하였다. 그 과정은 check변수를 추가하여 sol\_index로 고려되는 경우가 한번이라도 있었으면 check변수값을 바꿔주는 것으로 구현하였다.

1. **토의**

* **Big O-notation을 통한 분석**
  + 이번 과제에서 사용한 알고리즘은 크게 2가지 부분으로 나눌 수 있다. 코드상에서 각각 makeDP, makeDP\_change 로 정의된 메소드이다. 코드에서 makeDP와 makeDP\_change의 리턴값을 solve가 파라미터로 넘겨 받아서 결과를 출력하므로 서로 선형 관계로 동작하기 때문에 두 메소드만 비교한다 . makeDP는 코인의 종류를 오름차순으로 저장하고 있는 coin\_type배열의 인덱스를 나타내는 coin\_index를 전달받고서 동전의 종류 중 coin\_index만큼만 사용해서 money를 지불하는데 드는 최소 동전 갯수(min)를 출력한다. 위 과정을 money라는 변수를 통해 현재 가진 동전으로 만들 수 있는 최대 금액(capacity )만큼 루프를 돌린다. 루프를 돌리면서 마지막으로 해당 money를 만드는 데 사용한 동전의 개수가 range를 넘는지 coin\_type만큼 돌면서 비교하는 과정이 들어간다.

* 결국 capacity를 c, coin\_type의 수를 n이라 하고, 한번 makeDP(int coin\_index)를 c\*1+ c\*2+……+c\*n만큼, makeDP()를 n만큼 돌리므로 총 n(n+1)\*c/2만큼 수행하게 되고 따라서 빅오 표현은 O(c\*n^2)이다.
* 이제 makeDP\_change를 볼 텐데 이 메소드는 makeDP와 완전히 똑같이 동작하되 거스름돈을 주는 경우에 사용할 것이라 range에 대한 제한이 필요 없다는 것이 유일한 차이다. 이 점에서 보았을 때 위 코드에서 n(n+1)/2번 수행해야 했던 range비교 작업이 없어지면서 단순히 O(cn)이 되고 서로 독립적으로 동작하므로 알고리즘에서 지배적인 부분은 make\_DP의 O(cn^2)가 된다.

1. **구현과정에서 겪은 어려운 점**

* 처음에 알고리즘에 대해 생각을 할 때 직관적으로 가능한 모든 경우에 대해 새로이 계산을 하려고 했다. 이 경우는 전체 수행시간은 입력된 코인 종류를 c1 c2…. Cn 이라고 했을 떄 O(c1\*c2\*….cn)으로 너무나 비효율적인 방식이었다. 고민을 한 결과 위 경우에서 앞에서 계산한 결과를 그 다음 계산할 때 사용할 수 있을 것이라는 생각을 하였고, 결국 중복된 연산이 너무 많아 그것을 줄이기 위해 다이나믹 프로그래밍이라는 방법을 사용하게 되었다.
* 일반적인 동전 교환문제는 많이 알려져 있어서 처음 문제를 보고 그런 식으로 접근하면 되지 않을까 생각했다. 아직 수업시간에 Dynamic Programing을 배우지 않았고 따로 깊이 공부해본 경험이 없어서 처음에는 greedy algorithm을 사용하는 쪽으로 생각해보고, 화폐단위가 입력 값에 따라 다양하기 때문에 greedy로는 해결할 수 없다는 생각을 하게 되었다. 그래서 다음으로 recursive하게 구현하여 가능한 모든 조합을 찾아보는 경우를 생각해 봤다. 그렇게 하면 최적해를 찾을 수는 있겠지만 performance적으로 봤을 때 exponential한 시간이 걸리므로 개선할 방법이 없는지 생각해야 했다. 그래서 동전교환문제에 대한 알고리즘으로 많이 사용하는 Dynamic Programing에 대해 인터넷 검색을 통해 알아보고 이해하는 것부터 시작했다. 그리고 Dynamic Programing을 사용하여 해결하는 것으로 방향을 잡았는데 Customer의 동전 개수에 제한이 있다는 것이 가장 큰 걸림돌이었다. 이 문제는 결국 Customer이 지불해야 하는 money 이상의 돈을 지불하면 Clerk가 그 남는 돈을 거슬러주는 것으로 이해할 수 있는데, Clerk가 돈을 거슬러주는 방법의 최적해는 일반적인 Dynamic Programing으로 구할 수 있다. 그런데 Customer의 특정 money에 대한 최적해는 조건을 조금 추가하여 range검사를 하고 가능한 경우에 대해서만 값을 갱신하는 Dynamic Programing을 사용해야 했다. 만약 Clerk의 최적해를 구할 때 Customer의 Dynamic Programing을 그대로 쓰고 동전의 개수제한을 INF값으로 주는 방법을 취하면 고려하지 않아도 되는 경우를 매번 고려하면서 지나가기 때문에 performance가 많이 떨어진다. 그래서 Customer과 Clerk의 경우를 나눠서 DP함수를 작성하는 것으로 그 문제를 해결했다.