Coputer Algorithms
Project - 03

이름 김태연
학번 201511058
이메일 kim77ty@dgist.ac.kr

요 약

본 프로젝트를 통해서 몇가지의 예제 문제를 풀어보면서 다이나믹 프로그래밍과 Greedy 알고리즘에 대해서 이해하는 것을 목표로 한다. 각 문제들에 대해서 어떠한 알고리즘을 써서 문제를 해결하였는지 설명을 하고, 그 문제에서 사용한 메모리, 시간 복잡도 등에 대해서 설명을 한다.

1. 서론

컴퓨터 알고리즘 수업시간에 배운 다이나믹 프로그래밍의 방법을 실습해보는데 목적을 두고 있다. 다이나믹 프로그래밍이란, 어떠한 문제에 대해서 직접적으로 바로 답을 구하기에는 그 비효율성이 매우 커지지만, 이전 단계의 계산 결과가 바로 다음의 계산 결과와 관련이 있을 경우에 적용되어, 이전 단계의 계산 결과를 이용해서 답을 얻어내는 과정이다. 즉, 다이나믹 프로그래밍을 통해서 어떠한 문제의 답을 얻어낸다고 하면, 가장 낮은 단계부터 시작하여 모든 단계의 결과값을 저장하게 된다.

Greedy 알고리즘은 다이나믹 프로그래밍의 하나의 종류라고 생각할 수 있다. 이 알고리즘에서는 각 스텝에의 최선의 선택을 하는 것이 전체 문제의 최선의 선택이 된다는 조건이 성립할 때 사용되게 되며, 이 경우에서는 모든 단계마다 답에 가까운 가장 최선의 선택을 하게 된다.

2. 본문

2.1 problem 1 구현

문제 1 번의 경우는 수업시간에 다루었던 예제로, knapsack problem 의 대표적인 예시이다. 이 문제의 경우는 brute-force 방식으로 문제를 풀기에는 풀 수는 있지만, 시간 복잡도가 2^n 이 되어 running time 이 매우 길어진다는 문제가 있다. 따라서, 다이나믹 프로그래밍을 이용해서 문제를 푸는 것이 더 효율적일 것 이라고 생각을 할 수 있는데, 이를 위해 subproblem 으로 k 개의 labeled 된 item 들에 대한 솔루션을 생각 할 수 있을 것이다. 하지만 이 경우 역시, k-1 번째의 솔루션이 k 번째의 솔루션의 부분이 되지 않는다는 것을 확인 할 수 있다. 즉, 일반적인 다이나믹 프로그래밍에서의 subproblem 과는 조금 다르게 정의를 해줘야 하는데, subproblem 을 각 subset 의 item 들의 실제 무게를 저장해주는 새로운 parameter 인 w 를 추가하고 subproblem 을 B[k, w]로 풀어준다. 이러면 B[k, w]가 B[k-1, w]의 일부가 되어, B[k – 1, w] 또는 max{B[k – 1, w], B[k – 1, w – wk] + bk}가 B[k, w]가 된 다는 것을 알 수 있다. 이 알고리즘을 구현하기 위해서는 item 의 수 i 와 최대 weight, w 에 대해서 i*w 크기의 integer 형의 array 가 필요하게 된다.

자세한 구현 코드는 prob1.cpp 파일을 통해 확인해볼 수 있으므로 생략하고, 다음으로 실제로 이 알고리즘을 구현할 것을 분석해보면 다음과 같다.

일단 문제에 주어진 순서대로 input 을 받아서 max_weight, 아이템의 수를 저장해주는 max_num, 각각 아이템의 weight 와 가치를 저장해주는 weight[]와 value[]라는 array 에 저장을 해준다. 그리고 이번 문제의 경우는 간단한 문제풀이를 위한 코드이기 때문에, 각 경우에 따라서 메모리를 동적할당 해주지 않고 아이템의 수와 최대 weight 값을 이용해서 최대가 되는 경우인 1001*1001 의 다이나믹 프로그래밍을 위한 2 차원 array 를 선언해준다. 그리고 알고리즘이 동작하는 방법은 강의노트에 올라와 있는 pseudo code 와 같이 구성이 되어, weight 가 1 일 때부터 subproblem 을 풀어주며, array 가 모두 다 계산이 되었을 때 array 의 가장 마지막 원소가 우리가 구하고자 하는 답이 된다.

이 알고리즘의 시간 복잡도를 생각해보면, subproblem 의 array 들을 모두 계산해주는 만큼의 비용이 필요하므로, O(i*w)가 된다. (i 는 item 의 수, w 는 최대 weight)

다음으로 이 알고리즘이 사용하는 메모리를 생각해보면, 일단 간단한 변수 지정을 위한 메모리들은 고려하지 않고, data 들을 저장해주기 위해 선

언해준 array 와 같은 부분들의 메모리만을 고려하도록 하겠다. subproblem 을 위한 메모리와 (1001*1001*4(Byte)) 각 아이템들의 무게와 가치를 저장해주는 메모리(max_num * 2 * 4(Byte))가 필요하므로, 약 4MB + max_num * 2 * 4 의 메모리가 필요하게 된다. 하지만 이 중 문제의 조건을 저장해주는 부분은 필수적인 메모리이고, subproblem 을 위한 메모리 역시 필요하다면 input 으로 받은 max_num*max_weight 의 크기로 최적화해줄 수 있다.

2.2 problem 2 구현

문제 2 번의 경우는 비교적 쉽게 생각해줄 수 있는 경우이다. 문제에서 N 명의 사람이 있을 때 ATM 에서 돈을 인출하기 위해 걸리는 시간 P 의 총합이 가장 작은 경우를 구하라고 하였는데, 이 문제는 Greedy 알고리즘을 쓰면 쉽게 해결 할 수 있다.

돈을 인출하는데 오래 걸리는 사람일수록 앞에서 인출을 하면 더 많은 사람들이 오래 기다리게 되므로, 걸리는 시간이 짧은 사람이 먼저 인출을 하는 것이 좋다. 즉, N 명의 사람들 중에서 k 명만 인출을 해야 한다고 가정을 하면, k + 1 번째 사람까지 되었을 때의 최단 시간은 k + 1 번째의 사람이 남은 사람들 중 가장 적게 걸리는 사람이 나와야 되는 것 이므로, Greedy 알고리즘이 되어 첫번째부터 가장 최선의 선택, 즉 가장 시간이 적게 걸리는 사람을 선택해주면 답을 얻을 수 있다.

자세한 구현 코드는 prob2.cpp 파일을 통해 확인해볼 수 있으므로 생략하고, 다음으로 실제로 이 알고리즘을 구현할 것을 분석해보면 다음과 같다.

코드에서 main 함수만을 보면, 일단 input 을 받아준 후에, 각자 처리하는데 걸리는 시간에 대해서 적은 순서대로 sorting 을 해준 후 각각의 사람들이 기다리는 시간을 누적합으로 더해주었다. 그리고 sorting 을 해주는 부분 역시 그냥 cpp 의 algorithm 을 쓸 수 있겠지만, 지난 컴퓨터 알고리즘 프로젝트 1 에서 만들었던 quickSort 의 optimize 된 코드를 가져와서 사용하였다.

이 알고리즘의 시간 복잡도를 생각해보면, 일단 인풋으로 들어온 데이터들을 sorting 을 해주는 작업이 필요한데, quicksort_Op 를 사용하면 평균적으로 O(n*logn)의 시간이 필요하고, 계산을 해주는 부분은 총 수 n 에 대해서 두번의 연산을 해주게 되므로 O(2n) = O(n)의 시간이 필요하다. 따라서 총 필요한 시간의 시간 복잡도는 O(n*logn) + O(n) = O(n*logn)이 된다는 것을 알 수 있다.

다음으로 이 알고리즘이 동작하기 위해 필요한 메모리에 대해서 생각을 해보면, 마찬가지로, quicksort 에서의 pivot 이나 key 처럼 간단한 변수들을 저장해주기 위한 메모리들은 무시할 수 있다고 가정을 한다. 일단 인풋 데이터들을 저장해주기 위한 array(number*4(Btye))가 필요하고, 각 사람들의 대기시간을 누적합으로 저장해주기 위한 array(number*4(Byte))가 필요하다. 이 역시 인풋으로 받은 데이터들을 저장해주기 위한 array 는 문제를 위해 필수적이므로, 사실상 number*4(Byte) (number 는 총 사람의 수)의 메모리가 필요하다는 것을 알 수 있다.

2.3 problem 3 구현

문제 3 번의 경우는 수열 A 가 주어졌을 때 가장 긴 감소하는 부분 수열을 구하는 문제로, 다이나믹 프로그래밍을 활용하는 문제로서, 이 문제의 경우는 subproblem 을 어떻게 잡는지가 중요하다.

다이나믹 프로그래밍을 적용하기 위해 어떤 방식으로 subproblem 을 정의할지 고민을 하다가, 총 원소의 수를 n 이라고 할 때, (n-k)번째부터 n 번째까지의 수열에서 가장 긴 감소하는 부분 수열의 개수를 생각해주었다. 이를 count(n-k)라고 하면, 우리가 구하고자 하는 문제의 답은 count 라는 array 에 저장된 가장 큰 값이 될 것이다. 또한, count(n-(k+1)) 의 값은 항상 count(n-j) +1 (j < k) 또는 count(n-k) + 1 이 될 것이라는 것을 알 수 있다. 즉, n-(k+1)번째의 숫자가 이전의 부분 감소수열의 최대값보다 작을 때는 k+1 번째의 숫자보다 작은 숫자를 가지고 있는 j(j<k) 중 count(j)값이 가장 큰 j 에 대해서 count(n-j) + 1 이 되고, 그렇지 않을 경우는 최대 부분 감소수열이 한 개 늘어날 수 있기때문에 count(n-k) + 1 이 된다.

자세한 구현 코드는 prob3.cpp 파일을 통해 확인해볼 수 있으므로 생략하고, 다음으로 실제로 이 알고리즘을 구현할 것을 분석해보면 다음과 같다.

이번에는 위의 알고리즘으로 문제를 조금 더 효율적으로 풀기 위해 인풋 데이터를 받을 때 역순으로 받아준다. 역순으로 데이터를 받으면, 0 번째부터 시작하여 자신을 포함한 자신의 앞에서 부분 증가 수열을 찾아주면 되기 때문이다. 각각 k 번째에 대해서 수열의 개수를 저장해주기 위한 count 라는 함수를 선언하고, 최소 자신 하나만으로도 1 개의 부분 수열을 만들 수 있기 때문에 모두 1 으로 값을 초기화해준다. 다음으로 인덱스를 i = 0 부터 시작해서 하나씩 늘려주면서 i 보다 작은 값이 앞에 있으면 그 값에 저장된 count 의 값 + 1 과 i 값에 저장된 count 의 값 중 더 큰 값을 저장해주는 작업을 계속 반복한다. 이 반복작업이 끝나면 count 라는 array 에서 최대값을 뽑아주는 작업을 하고 그 값을 반환해준다.

이 알고리즘의 시간 복잡도를 생각해보면, count 를 for 문 안에서 하나씩 채워주고 i 번째를 채워줄 때 i 번의 반복문이 필요하므로, O(n^2)의 시간 복잡도가 걸리고, 결과를 뽑아줄 때 O(n)의 시간 복잡도가 걸린다. 따라서, 전체 시간 복잡도는 O(n^2) + O(n) = O(n^2)이 된다.

다음으로 이 알고리즘이 동작하기 위해 필요한 메모리에 대해서 생각해보면, 마찬가지로 간단한 변수들을 저장해주기 위한 메모리들은 무시할 수 있다고 가정을 하고, 알고리즘에서 정의한

subproblem 의 결과를 저장하기 위한 array(number(인풋 수열의 크기) * 4(Byte))와 인풋 데이터를 저장하기 위한 array(number * 4(Byte))가 필요하다. 이 역시 인풋으로 들어온 데이터를 저장하기 위한 메모리는 문제를 위해 필수적이므로, 사실상 number*4(Byte)의 메모리가 필요하다는 것을 알 수 있다.

2.4 결론

총 3 가지의 예시 문제를 통해서 다양한 방법으로 다이나믹 프로그래밍과 Greedy 알고리즘을 사용하는 것을 실습해보았다. 이러한 문제의 경우는 다이나믹 프로그래밍을 사용하지 않으면 시간복잡도가 거의 지수 scale 로 매우 커질 수 있는 문제이지만, 다이나믹 프로그래밍을 이용해서 subproblem 들을 정의해주고 그 결과를 저장을 하면서 풀면서 훨씬 더 효율적으로 문제를 해결할 수 있다는 것을 알 수 있었다.

참고문헌

[1] 본 레포트 양식은 국내 학술대회 논문을 참고하였습니다.
[2] Introduction to Algorithms (Thomas H. Cormen)
[3] 컴퓨터 알고리즘 강의노트