알고리즘 실습(12주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Activity Selection

Activity Selection 문제는 어떠한 일정의 스케줄링을 함에 있어서 최적의 일정을 찾아내는 문제이다. 최적의 스케줄링의 기준으로는 종료 시간 기준, 시작 시간 기준과 각 Activity들의 Value의 합이 기준이 된다.

Activity Selection 문제를 푸는 방법에는 Dynamic Programming과 Greedy Algorithm의 두 가지 방법이 있다.

① Dynamic Programming 기반 – # of Selected Activities

- 최적의 Activity의 수를 Dynamic Programming 방식으로 찾는다.

② Dynamic Programming 기반 – Sum of Selected Activites

- Activity의 Value의 합이 가장 큰 Selection을 Dynamic Programming 방법으로 찾는다.

③ Greedy 기반 – # of Selected Activities 종료 시간 기준

- Greedy Algorithms을 통해 종료 시간 기준으로 정렬되어 있는 Activity 중에서 최적의 Selection의 개수를 찾는다.

④ Greedy 기반 – # of Selected Activities 시간 시간 기준

- Greedy Algorithms을 통해 시작 시간 기준으로 정렬되어 있는 Activity 중에서 최적의 Selection의 개수를 찾는다.

⑤ Greedy 기반 – # of Selected Activities 임의 순서 기준

- Greedy Algorithms을 통해 시작 시간 기준으로 정렬되어 있는 Activity 중에서 최적의 Selection의 개수를 찾는다.

2. Knapsack Problem

정해진 무게량을 가진 가방 안에 얼마만큼의 물건을 넣어야 최대의 가치로 가장 많은 아이템을 넣을 수 있는지의 문제로, 여기서는 쪼갤 수 없는 0-1문제와 남은 공간에서 무게당 가치를 계산하여 물건을 쪼개어 넣는 Fractional 문제가 있다.

① 0-1 Knapsack Problem - 모든 값이 정수인 경우

- 가방의 무게와 아이템의 무게가 모두 정수인 경우

② 0-1 Knapsack Problem - 무게가 소수점 둘째 자리인 경우

- 가방의 무게와 아이템의 무게가 모두 소수점 둘째 자리까지 나오는 실수인 경우

③ Fractional Knapsack Problem - 모든 값이 정수인 경우

- 가방의 무게와 아이템의 무게가 모두 정수인 경우

④ Fractional Knapsack Problem - 무게가 소수점 둘째 자리인 경우

- 가방의 무게와 아이템의 무게가 모두 소수점 둘째 자리까지 나오는 실수인 경우

2. 주요 부분 코드 설명(알고리즘 부분)

1. Activity Selection

① Dynamic Programming 기반

```
/* Dynamic Activity Selector */
for(int l = 1; l<=length; l++) {
    for(int i = 0; i<=n-l+1; i++) {
        int j = i+l;
        c[i][j]=0; //스케줄링에 따른 activity의 개수를 적어나갈 매트릭스
        s[i][j]=0; // 어떠한 activity를 수행하게 될지 기록하는 매트릭스.
            for(int k = 2 ; k<=j-1; k++) {
                if(activity_select[k].start >= activity_select[i].finish &&
                   activity_select[k].finish <= activity_select[j].start) {
                    int q = c[i][k] + c[k][j]+1;
                    if(q > c[i][j]) {
                        c[i][j] = q;
                        s[i][j] = k;
                }
            }
        }
    }

}
/* Dynamic Activity Selector */
```

- Dynamic Programing 을 하기 위해 Activity의 개수를 적어갈 배열과 어떠한 Activity를 할 것인지를 저장할 배열을 선언한다.
- Dynamic Programming 방식에서는 i와 j사이에 있는 k값을 구하는 것이 목적이다.
- k값의 시작이 i값의 끝보다 크거나 같으면서, k값의 끝이 j값의 시작보다 작거나 같다면, i와 j사이에 공간이 있다는 것이므로, 해당 위치에 activity가 들어갈 수 있다. c[i][k]+c[k][j]+1을 저장하고 이 값이 c[i][j]값보다 크면 c[i][j]에 저장한다.
- k값을 따로 저장한다.

```
void print_activity(int ** c,int ** t, activitySelection * activity_select ,int s, int f) {
    /*
     Activity들의 출력
     */
    int k=0;
    if (c[s][f] > 0) {
        k = t[s][f];
        printf("%d, ", activity_select[k].number);
        print_activity(c, t, activity_select, s, k);
        print_activity(c, t, activity_select, k, f);
    }
}
```

- k값에 해당하는 Activity를 재귀적으로 호출한다.

② Greedy Algorithms 기반

```
void greedy_Activity_Selector( activitySelection * activity_select, int arrayLength) {
    int n= arrayLength;
    int total = 0; //액티비티의 총 갯수를 저장할 변수

    activitySelection a[n]; //해당되는 액티비티의 번호를 저장할 배열.
    a[1] = activity_select[1]; //비교를 위해 첫번째 스케줄을 넣고 시작.
    int index = 2; //해당되는 액티비티의 배열을 조절할 인덱스
    int k = 1;

    for (int m = 2; m<n; m++) {
        if (activity_select[m].start >= activity_select[k].finish) {
            // m값의 스타트가 k값의 피니쉬보다 크거나 같다면?
            a[index] = activity_select[m]; //selection에 추가
            k = m; //k값을 m값으로 변경.
            index++; //index 증가
            total++; //토탈 증가
        }
    }

    printf(" 총 스케줄의 개수 : %d\n" , total);
    for(int i=1; i<index; i++) {
        if(a[i].number == NULL) {                    ⚠ Comparison between NULL a
            break;
        }
        printf("%d, " , a[i].number);
    }
     printf("\n");

}
```

- Greedy Algorithms은 한 번 선택한 결과에 대해서 수정하지 않는다.
- 비교를 위해서 첫 번째 Activity를 결과값에 입력하고 시작한다.
- k값의 종료보다 m값의 시작값이 크거나 같다면 결과값에 m값을 추가한다.
- m값의 시간동안은 다른 activity가 선택되면 안되므로, 비교 값인 k값을 m값으로 바꾼다.

2. Knapsack Problem

① 0-1 Knapsack Problem

```
/* 0-1 Knapsack Problem */
for(int i=1; i<item; i++) {
    for(int w =1; w<bagWeight; w++) {
        if(intItem[i].weight <= w) { //현재 아이템의 무게가 현재 무게보다 작거나 같다면
            if (intItem[i].b + B[i-1][w-intItem[i].weight] > B[i-1][w]) {
                //현재 아이템의 가치 + 이전 아이템의 가치 가 이전아이템의 가치보다 크다면
                int q =  intItem[i].b + B[i-1][w-intItem[i].weight];
                B[i][w] = q; //해당 값을 저장
            } else {
                B[i][w] = B[i-1][w]; //그렇지 않다면 그냥 이전 아이템의 가치를 저장.
            }
        } else {
            B[i][w] = B[i-1][w];
        }

    }
        printf("\n");
}
/* 0-1 Knapsack Problem */
```

- 아이템의 번호와 최대무게까지의 무게를 열과 행으로 하는 매트릭스를 통해 계산한다.
- 현재 아이템의 무게가 현재 들어갈 수 있는 무게보다 작거나 같다면, 현재 아이템의 가치와 이전 아이템의 가치가 이전 아이템의 가치보다 크다면 해당 값을 매트릭스에 저장한다.
- 즉 해당 무게값에 들어갈 수 있는 아이템의 총 가치를 저장하는 것이다.

```
void print_knapsack(int ** B, int n, int w, Item intItem[]) {
    if(n == 0) {
        return;
    }
    if(B[n][w] > B[n-1][w] ){
        printf("%d ", n);
        return print_knapsack(B, n-1, w-intItem[n].weight,intItem);
    } else {
        return print_knapsack(B, n-1, w,intItem);
    }


}
```

- 이후 출력할 때에는 매트릭스의 최대값부터 출력해나간다.
- 매트릭스의 최대값의 위치에서 그 이전 행의 위치보다 값이 크다면, 해당 인덱스에 해당하는 아이템이 들어있다는 것으로, 인덱스를 출력하고, 해당 인덱스의 아이템 만큼의 무게를 빼고 아이템 개수가 1개 줄어든 위치의 값을 재귀적으로 호출한다.
- 그렇지 않으면 해당 위치에 n번째 아이템이 없다는 것으로, 아이템 개수를

하나 줄인채로 재귀적으로 호출한다.

② Fractional Knapsack Problem

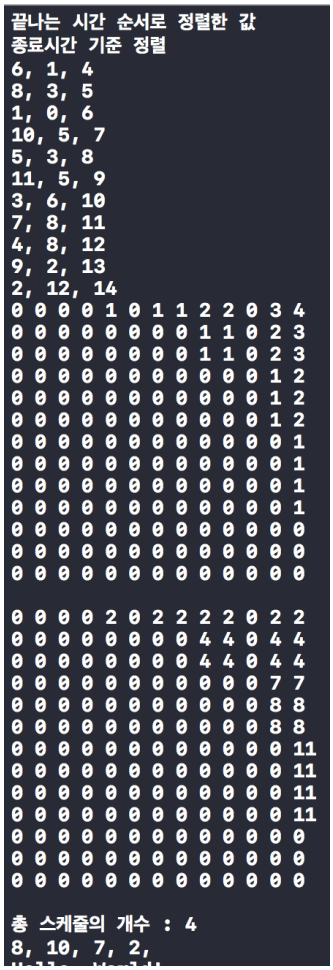
```
void fractional_knapsack(int bw, int itemLength, Item intItem[]) {
    int bagWeight = bw;
    int item = itemLength;

    int totalWeight =0;

    for(int i =1 ; i< itemLength; i++) {
        intItem[i].valuePerWeight =  float(intItem[i].b) / float(intItem[i].weight) ;
    }
    insertionSort(intItem, item); //무게당 가치를 큰값이 앞으로 오게끔 내림차순 정렬

    for(int i = 1; i<itemLength ; i++) {
        if(intItem[i].weight + totalWeight <= bagWeight) {//현재 아이템의 무게 + 총 무게가 가방무게보다 작을 경우
            totalWeight += intItem[i].weight; //무게당 가치가 큰 아이템들 부터 그냥 저장
            printf("value : %d, weight : %d, TotalWeight : %d\n", intItem[i].b, intItem[i].weight, totalWeight);
        } else { //가방 무게보다 클 경우
            int a = bagWeight - totalWeight; //현재 남은 무게 계산
            float value = a * (intItem[i].b / intItem[i].weight); //단위 무게당 가치에 남은 무게를 곱하여 가치 계산.
            totalWeight += a;
            printf("value : %d, Fractioanlvalue : %f, weight : %d, TotalWeight : %d\n", intItem[i].b, value, a,
                totalWeight);
            break; //더이상 가방에 아이템을 담으면 안되므로 반복문 종료
        }
    }
}
```

- 아이템을 쪼개어 넣을 수 있는 Fractional Knapsack Problem은 일단 단위 무게당 가치를 계산하고, 해당 가치를 내림차순으로 하여 아이템들을 정렬해준다.
- 현재 아이템의 무게와 현재까지 가방에 들어간 무게의 합이 가방의 무게보다 낮다면, 그냥 가방에 넣어준다.
- 가방의 무게보다 크다면, 무게당 가치를 계산하여 남은 가방의 무게만큼 아이템을 쪼개어서 가방에 넣어야한다.

3. 결과

1. Activity Selection

① Dynamic Programming 기반 – # of Selected Activities

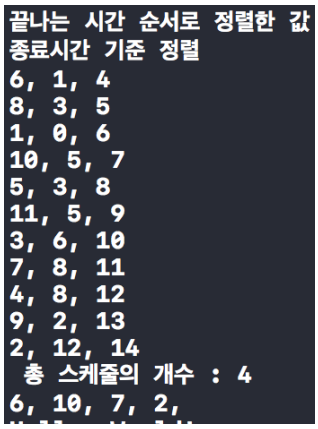


- 아이템을 종료시간 기준으로 오름차순을 정렬한 후 Activitiy selection을 진행한다.
- 선택된 스케줄의 갯수는 4개이며 8번, 10번, 7번, 2번이 선택되었다.

② Dynamic Programming 기반 – Sum of Selected Activites

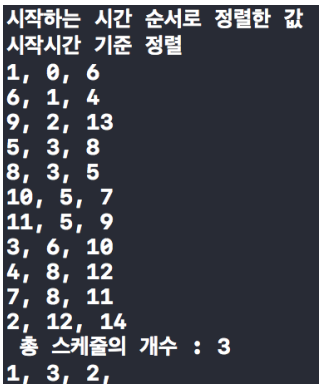
- 어려워서 풀지 못했다.

③ Greedy 기반 – # of Selected Activities 종료 시간 기준



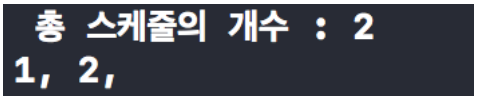
- Greedy Algorithms을 통해 Activity를 구했다.
- 총 개수는 4개이며 6번, 10번, 7번, 2번이 선택되었다.
- Dynamic Programming의 결과와 다른 이유에 대해선 알지 못했다.

④ Greedy 기반 – # of Selected Activities 시간 시간 기준



- Greedy Algorithms을 통해 Activity를 구했다.
- 총 개수는 3개이며 1번, 3번, 2번이 선택되었다.

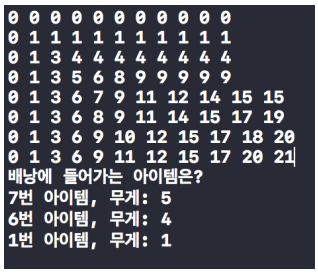
⑤ Greedy 기반 – # of Selected Activities 임의 순서 기준



- 정렬되지 않은 아이템을 이용하여 Greedy Algorithms으로 Activity를 선택했다.
- 총 개수는 2개이며 1번, 2번이 선택되었다.

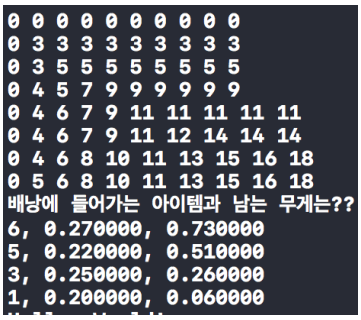
2. Knapsack Problem

① 0-1 Knapsack Problem - 모든 값이 정수인 경우



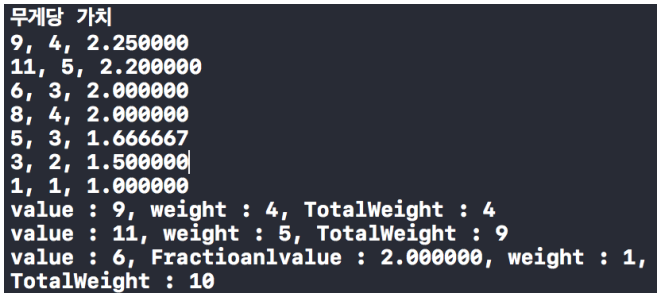
- 가방의 무게와 아이템의 무게가 모두 정수인 경우
- 아이템은 총 3개가 선택되었으며 7번, 6번, 1번이 선택되었다.

② 0-1 Knapsack Problem - 무게가 소수점 둘째 자리인 경우



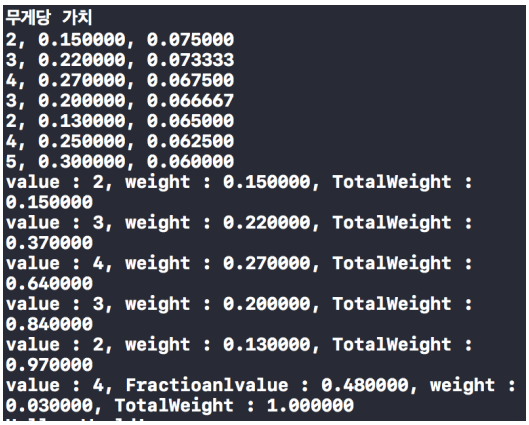
- 가방의 무게와 아이템의 무게가 모두 소수점 둘째 자리까지 나오는 실수인 경우
- 아이템은 총 4개가 선택되었으며 6번, 5번, 3번, 1번이 선택되었다.

③ Fractional Knapsack Problem - 모든 값이 정수인 경우



- 가방의 무게와 아이템의 무게가 모두 정수인 경우
- 아이템은 3개가 선택되었으며 9번, 11번, 6번이 선택되었다.

④ Fractional Knapsack Problem – 무게가 소수점 둘째 자리인 경우



- 가방의 무게와 아이템의 무게가 모두 소수점 둘째 자리까지 나오는 실수인 경우
- 아이템은 총 6개가 선택되었으며 2번, 5번, 6번, 1번, 4번, 3번 아이템이 선택되었다.

3. 결론

전체적으로 어려운 과제였으며 특히 Activity Selection의 Dynamic Programming 은 수도코드가 없어서 구현하기 어려웠다. 또한 Dynamic Programming과 Greedy Algorithms의 결과가 같아야 할텐데 다른 이유에 대해서는 잘 모르겠다.

과제 1-2인 Activity Selection을 Sum of value 기준으로 구하는 문제는 어려워서 풀지 못했다.