개요

배낭 채우기 알고리즘이란 보석 가게에 도둑이 들었다고 가정했을 때, 도둑은 보석을 가져온 가방에 담을 수 있다. 가방이 담을 수 있는 최대 용량을 W 라고 하자. 보석 방에 있는 보석의 종류는 I={i1, i2, i3, …}이 있고, 각 종류마다 보석은 하나씩 밖에 없다. 이 보석들의 각각 무게 w, 가치 p 를 갖는다. 이 때 한정된 최대 용량 W 내에서 최대의 가치를 내는 보석들을 담고 싶을 때 어떤 방식으로 접근해야 하는지에 대한 내용이 배낭 채우기 알고리즘의 핵심 내용이다. 이번 실험에서는 이러한 배낭 채우기 알고리즘을 깊이 우선 탐색(Depth-First Search), 너비 우선 탐색(Breath-First Search), 최고 우선 탐색(Best-fit Search)으로 나누어 알아보고자 한다.

1. 깊이 우선 탐색으로 배낭 채우기 문제 풀기

깊이 우선 탐색은 상태 공간 트리를 구축하여 백트래킹 기법으로 문제를 해결한다. 즉 값어치가 높은 물건을 순서대로 넣는다. 만약 상태 공간 트리의 루트에서 왼쪽으로 가면 첫 번째 물건을 배낭에 넣는 경우이고, 오른쪽으로 간다면 첫 번째 물건을 넣지 않는 경우이다. 트리에서 왼쪽으로 계속 넣다가 적절하지 않은 물건이면 트리 왼쪽 끝에서 한 단계 백트래킹 해서 먹지 않고 다른걸 먹고, 또 적절한 물건이 아니면 위로 올라가면서 다른 마디로 이동하여 계속 넣는다. 이러한 과정을 거쳐 최적의 해답을 구한다.

또한 유망성 검사를 해야 하는데, 상태 공간 트리의 어떠한 마디까지의 물건의 무게 합이 weight 이라고 할 때, 가방의 무게 W 보다 크거나 같으면 그 마디는 유망하지 않다고 판단한다.

마지막으로 아래의 식으로 bound 값을 구하여 maxprofit 과 비교하여 유망하지 않은지를 판단한다.

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{처음 } k-1\text{개 아이템의 값어치}} + \underbrace{(W - totweight)}_{k\text{번째 아이템에 가용한 용량}} \times \underbrace{\frac{p_k}{w_k}}_{k\text{번째 아이템의 단위무게 당 값어치}}$$

[그림 1] totalweight, bound 를 구하는 식

위에 작성된 유망성 검사하는 내용을 코드로 작성하면 다음과 같다.

```
int promising(int i, int profit, int weight) {
    int totweight;
    float _bound;
    if(weight >= W) return false;   // 자식 마디로 확장할 수 있을 때만 마디는 유망하다.
    else {                          // 자식 마디에 쓸 공간이 남아 있어야 한다.
        _bound = profit;
        totweight = weight;
        // 가능한 한 많은 아이템을 취한다.
        while(i <= MAX && totweight + w[i] <= W) {
            totweight = totweight + w[i];
            _bound = _bound + p[i++];
        }
        if(i <= MAX)  _bound = _bound + (W - totweight) * p[i]/w[i];
        return _bound > maxprofit;
    }
}
```

[코드 1] DFS_knapsack.promising( )

그리고 이 유망성 검사 함수를 사용하여 되추적 알고리즘으로 코드를 작성하면 다음과 같다.

```
// 깊이우선탐색
void DFS_knapsack(int i, int profit, int weight, int include[MAX +
1]) {
    if(weight <= W && profit > maxprofit) {  // 지금까지는 이 집합이 최고이다.
        maxprofit = profit;
        // numbest를 고려한 아이템의 개수로 놓고, bestset을 이 해답으로 놓는다.
        numbest = i;
```

```
        memcpy(bestset, include, (MAX + 1) * sizeof(int));
    }
    if(promising(i + 1, profit, weight)) {
        include[i + 1] = true;  // w[i + 1] 포함
        DFS_knapsack(i + 1, profit + p[i + 1], weight + w[i + 1],
include);
        include[i + 1] = false; // w[i + 1] 비포함
        DFS_knapsack(i + 1, profit, weight, include);
    }
}
```

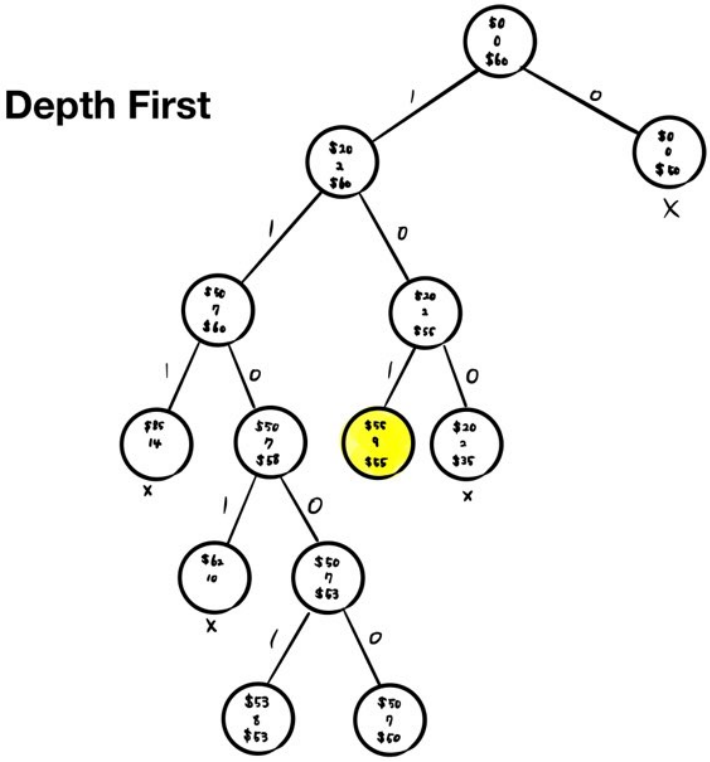
[코드 2] DFS_knapsack( )

이 코드를 실행해보면 다음과 같은 결과가 나온다.

```
========== DFS_knapsack() ==========

Select Item:    i1(2, 20)
profit: 20

Select Item:    i1(2, 20)   i2(5, 30)
profit: 50

Select Item:    i1(2, 20)   i2(5, 30)   i5(1, 3)
profit: 53

Select Item:    i1(2, 20)   i3(7, 35)
profit: 55

Best set:   i1(2, 20)   i3(7, 35)
maxprofit: 55
performance time : 37.000000ms
```

[그림 2] DFS_knapsack( ) 실행 결과

백트래킹 기법 대로 유망하다면 계속 내려가다가 유망하지 않으면 부모 마디로 되돌아와 다른 마디로 이동하는 모습을 볼 수 있고 이것을 그림으로 그리면 다음과 같이 표현할 수 있다.



[그림 3] DFS_knapsack() 실행 결과 상태 공간 트리

2. 너비 우선 탐색으로 배낭 채우기 문제 풀기

너비 우선 탐색은 분기 한정 가지치기 기법을 사용하여 문제를 해결한다. 그 외에는 백트래킹 기법과 똑같은 방법으로 진행된다. 즉, weight 과 profit 은 그 마디에 오기까지 포함되었던 아이템의 총 무게와 총 이익으로 한다. 마디가 유망한지 결정하기 위해서 totweight 과 bound 를 weight 과 profit 으로 각각 초기화하고, 그 다음 totweight 이 W 를 초과하게 하는 물건에 도달할 때까지 탐욕적으로 물건을 취하여 그 무게와 이익을 totweight 과 bound 에 각각 더한다. 이런 식으로 bound 는 그 마디에서 확장하여 얻을 수 있는 이익의 상한이 된다. 만약 그 마디가 수준 i 에 위치하고 있고, 수준 k 에 위치한 마디는 무게의 합이 W 가 넘어가는 마디라고 하자. 그러면 깊이 우선 탐색에서의 [그림 1]과 똑같은 수식을 얻게 된다.

지금까지 찾은 최고 해답에서의 값인 maxprofit 보다 이 한계값이 작거나 같으면 그 마디는 유망하지 않다. 다음 부등식이 성립해도 마디는 유망하지 않다.

$$weight \geq W$$

너비 우선 탐색은 뿌리 마디를 먼저 방문하고, 다음에 수준 1 의 마디를 모두 방문하고, 다음에 수준 2 의 마디를 모두 방문하고, 등등 이런식으로 계속한다. 깊이 우선 탐색과는 달리 너비우선 탐색을 하는 재귀알고리즘은 작성하기 어렵다. 그러나 대기열(Queue)을 사용하여 구현하면 가능하다. 먼저 대기열을 코드로 구현하면 다음과 같다.

```
typedef struct node{
     int level;
     int profit;
     int weight;
     int bound;
     void *next;
} node;

typedef struct queue{
    node *front;    // 맨 앞(꺼낼 위치)
    node *rear;     // 맨 뒤(보관할 위치)
    int count;      // 보관 개수
} queue;

void init_q(queue *q) {
    q->front = q->rear = NULL;  // front와 rear를 NULL로 설정
    q->count = 0;               // 보관 개수를 0으로 설정
}

void enqueue(node *v, queue *q) {
    if (q_is_empty(q))      //큐가 비어있을 때
       q->front = v;
    else q->rear->next = v; // 큐가 비어있지 않을 때
    q->rear = v;            // 맨 뒤를 삽입한 노드로 설정
    q->count++;             // 보관 개수를 1 증가
}

node *dequeue(queue *q) {
    node *temp;
    if (q_is_empty(q)) return NULL;

    temp = q->front;        //맨 앞의 노드를 기억
    q->front = temp->next;  //맨 앞은 front의 다음 노드로 설정
    q->count--;             //보관 개수를 1 감소
    return temp;
}

int q_is_empty(queue *q) { q->count == 0; }
```

[코드 3] 대기열(Queue)의 구현

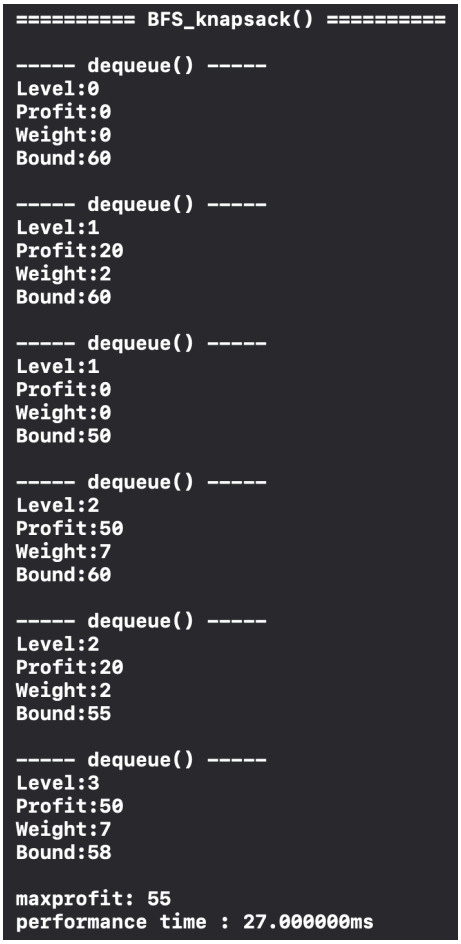
이제 이 대기열을 사용하여 너비 우선 탐색을 하는 코드를 구현하면 다음과 같다.

```
float bound(node *u) {
```

```
    int j, totweight;
    float _bound;
    if(u->weight >= W) return false;
    else {
        _bound = u->profit;
        j = u->level + 1;
        totweight = u->weight;
        // 가능한 한 많은 아이템을 취한다.
        while(j <= MAX && totweight + w[j] <= W) {
            totweight = totweight + w[j];
            _bound = _bound + p[j++];
        }
        if(j <= MAX)
            _bound = _bound + (W - totweight) * p[j] / w[j];
        return _bound;
    }
}

void BFS_knapsack(queue *q, node *v) {
    node *u;
    v->level = v->profit = v->weight = 0;
    v->bound = bound(v);
    enqueue(v, q);
    while(!q_is_empty(q)) {
        v = dequeue(q);
        u = malloc(sizeof(node));
        u->level = v->level + 1;     // u를 v의 자식마디로 놓음
        u->profit = v->profit + p[u->level]; // u를 다음 마디를 포함
        u->weight = v->weight + w[u->level]; // 한 자식 마디로 둠
        u->bound = bound(u);
        if(u->weight <= W && u->profit > maxprofit)
            maxprofit = u->profit;
        if(u->bound > maxprofit)
            enqueue(u, q);
        u = malloc(sizeof(node));
        u->level = v->level + 1;
        u->weight = v->weight;    // u를 다음 아이템을 포함하지 않는
        u->profit = v->profit;    // 자식마디로 놓음
        u->bound = bound(u);
        if(u->bound > maxprofit)
            enqueue(u, q);
    }
}
```

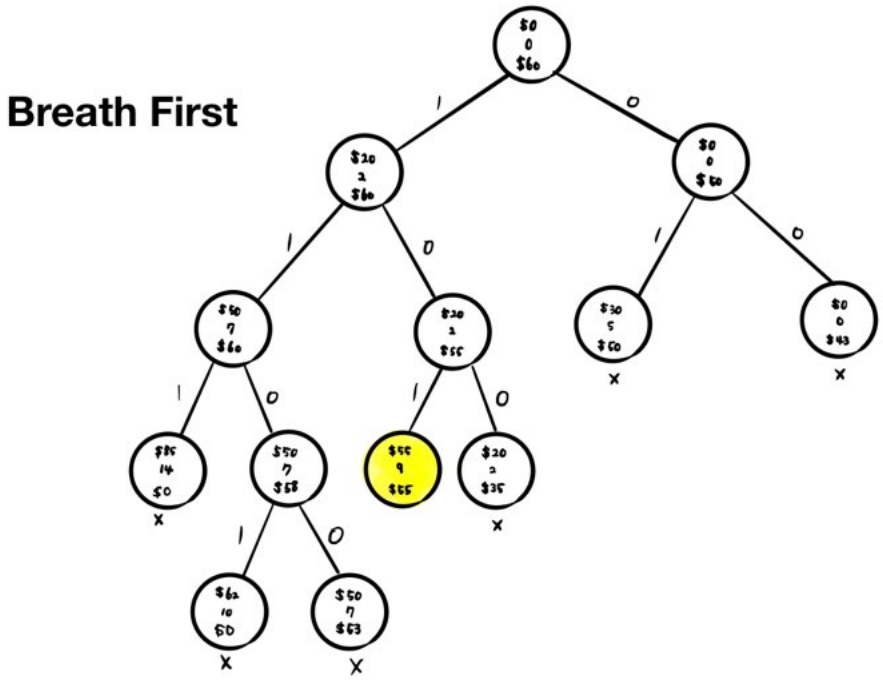
[코드 4] BFS_knapsack( )

이 코드를 실행하면 다음과 같은 결과가 나온다.



[그림 4] BFS_knapsack( ) 실행 결과

분기 한정 가지치기 기법 대로 같은 수준의 마디들을 순차적으로 계속 탐색하다가 bound 값이 maxprofit 보다 크면 탐색 대기열로 집어넣는다. 이것은 DFS 의 promising( )함수와 본질적으로 같다. 차이는 분기한정 알고리즘을 만들기 위한 지침에 따라서 bound( )를 작성했고, 따라서 bound( )는 정수를 넘겨주는 반면, promising( )은 되추적 지침을 따라서 작성했으므로 불(boolean)값을 넘겨준다. 우리의 분기 한정 기법에서 maxprofit 과의 비교는 호출하는 프로시저에 의해서 이루어진다. 이 경우 bound( )가 넘겨주는 값은 maxprofit 보다 작거나 같기 때문에, bound( )에서 i = n 조건을 검사할 필요는 없다. 이것의 상태 공간 트리를 그리면 다음과 같이 표현할 수 있다.



[그림 5] BFS_knapsack() 실행 결과 상태 공간 트리

3. 최고 우선 탐색으로 배낭 채우기 문제 풀기

일반적으로 너비 우선 탐색은 깊이 우선 탐색보다 좋은 점이 없다. 그러나 마디의 유망성 여부를 결정하는 것 이외에 추가적인 용도로 한계값을 사용하면 탐색을 향상시킬 수 있다. 주어진 어떤 마디의 모든 자식 마디를 방문한 후 유망하면서 확장하지 않은 마디들을 모두 살펴보고 그 중에서 한계값이 가장 좋은 마디를 우선적으로 확장한다. 지금까지 찾은 최고의 해답보다 그 한계값이 더 좋다면 그 마디는 유망하다. 미리 정해놓은 순서대로 무작정 검색을 진행하는 것보다 이런 식으로 하면 최적해를 더 빨리 찾게된다.

최고 우선 탐색은 너비 우선 탐색을 조금 변형시켜서 구현하면 된다. 대기열(Queue)을 사용하는 대신, 우선순위 대기열을 사용한다. 우선순위 대기열(Priority Queue)에서 우선순위가 높은 구성요소를 항상 먼저 제거한다. 최고 우선 탐색을 하면 우선순위가 높은 구성요소가 최고의 한계값을 가진 마디가 된다. 우선순위 대기열은 연결된 리스트로 구현할 수 있지만, 힙(Heap)으로 더 효율적으로 구현할 수 있다. 먼저 우선순위 대기열을 코드로 구현하면 다음과 같다.

```
typedef struct priority_queue{
    node **heap;
    int count;
} priority_queue;

void init_priority_q(priority_queue *p_q) {
    p_q->heap = calloc(1024, sizeof(node *));
    p_q->count = 0;
}

int pq_is_empty(priority_queue *p_q) { return p_q->count == 0; }

int get_parent(int i) { return i / 2; }

int get_left(int i) { return i * 2; }

int get_right(int i) { return i * 2 + 1; }

void insert(priority_queue *p_q, node *v) {
    int index;
    node *temp;
    p_q->heap[++p_q->count] = v;
    index = get_parent(p_q->count);
    while (index > 0) {
        if(p_q->heap[get_left(index)] != NULL)
            if (p_q->heap[index]->bound < p_q->heap[get_left(index)]-
>bound) {
                temp = p_q->heap[index];
                p_q->heap[index] = p_q->heap[get_left(index)];
                p_q->heap[get_left(index)] = temp;
            }
        if(p_q->heap[get_right(index)] != NULL)
            if (p_q->heap[index]->bound < p_q->heap[get_right(index)]-
>bound) {
                temp = p_q->heap[index];
                p_q->heap[index] = p_q->heap[get_right(index)];
                p_q->heap[get_right(index)] = temp;
            }
        index = get_parent(index);
    }
}

node *delete(priority_queue *p_q) {
    int i, tempNo;
    node *answer = p_q->heap[1];           // 부모 노드를 가져옴
    p_q->heap[1] = p_q->heap[p_q->count--]; // 마지막 노드를 가져옴
    tempNo = p_q->count;    // Heap 화
    p_q->count = 0;
    for (i = 1; i <= tempNo; i++)
        insert(p_q, p_q->heap[i]);
    return answer;
}
```

[코드 5] 우선순위 대기열(Priority Queue)의 구현

여기에서도 마찬가지로 너비 우선 탐색 때 사용되었던 마디(node)를 사용한다. 또한 대기열 대신에 우선순위 대기열을 사용하는 이외에 우선순위 대기열에서 마디를 제거한 후 마디의 한계값이 best 보다 아직 좋은지를 결정하는 검사를 추가하였다. 이제 이 우선순위 대기열을 사용하여 코드를 구현하면 다음과 같다.

```
void BF_knapsack(priority_queue *p_q, node *v) {
    node *u;
    v->level = v->profit = v->weight = 0;
    v->bound = bound(v);
    insert(p_q, v);
    while(!pq_is_empty(p_q)) {  // 최고의 한계값을 가진 마디 제거
        v = delete(p_q);
        if(v->bound > maxprofit) {  // 마디가 아직 유망한지 검사
            u = malloc(sizeof(node));
            u->level = v->level + 1;
            u->profit = v->profit + p[u->level]; // u를 다음 아이템을 포함하는
            u->weight = v->weight + w[u->level]; // 자식마디로 놓음
            if(u->weight <= W && u->profit > maxprofit)
                maxprofit = u->profit;
            u->bound = bound(u);
            if(u->bound > maxprofit)
                insert(p_q, u);
            u = malloc(sizeof(node));
            u->level = v->level + 1;
            u->weight = v->weight;    // u를 다음 아이템을 포함하지 않는
            u->profit = v->profit;    // 자식마디로 놓음
            u->bound = bound(u);
            if(u->bound > maxprofit)
                insert(p_q, u);
        }
    }
}
```

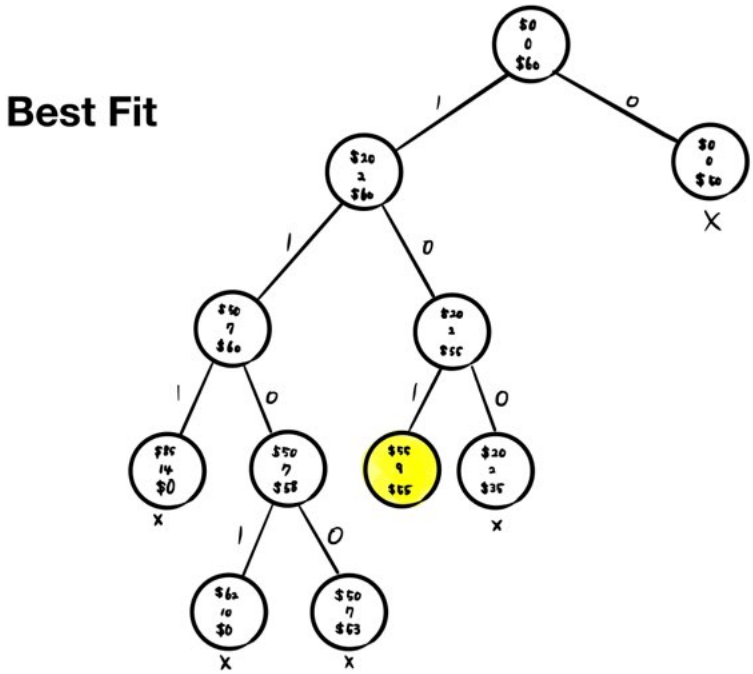
[코드 6] BF_knapsack( )

이 코드를 실행하면 다음과 같은 결과가 나온다.

```
========== BF_knapsack() ==========

----- delete() -----
Level:0
Profit:0
Weight:0
Bound:60

----- delete() -----
Level:1
Profit:20
Weight:2
Bound:60

----- delete() -----
Level:2
Profit:50
Weight:7
Bound:60

----- delete() -----
Level:3
Profit:50
Weight:7
Bound:58

----- delete() -----
Level:2
Profit:20
Weight:2
Bound:55

----- delete() -----
Level:4
Profit:50
Weight:7
Bound:53

----- delete() -----
Level:1
Profit:0
Weight:0
Bound:50

maxprofit: 55
performance time : 33.000000ms
```

[그림 6] BF_knapsack( ) 실행 결과

[그림 6]에서 보는 것과 같이 bound 가 가장 큰 마디부터 대기열에서 빠져나와 탐색해 나가는 것을 알 수 있다. 이제 이것을 상태 공간 트리로 나타내면 다음과 같다.



[그림 7] BF_knapsack() 실행 결과 상태 공간 트리

4. Source

```
/** Header **/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/** Const variable **/
#define MAX 5            // Item의 개수
#define W 9              // 주어진 문제의 가방 용량
#define true 1
#define false 0
const int p[MAX + 1] = {0, 20, 30, 35, 12, 3};     // item.Profit
const int w[MAX + 1] = {0, 2, 5, 7, 3, 1};          // item.Weight

/** Struct variable **/
typedef struct node{
    int level;
    int profit;
    int weight;
    int bound;
    void *next;
} node;

typedef struct queue{
    node *front;      // 맨 앞(꺼낼 위치)
    node *rear;       // 맨 뒤(보관할 위치)
    int count;        // 보관 개수
} queue;

typedef struct priority_queue{
    node **heap;
    int count;
} priority_queue;

/** Common variable **/
int totalweight = 0;
int maxprofit = 0;

/** Common function **/
void _pause(void);
float bound(node *u);

void _pause(void) {
    printf("\tPress any key ...\n");
    getchar();
}

float bound(node *u) {
    int j, totweight;
    float _bound;

    if(u->weight >= W) return false;
    else {
        _bound = u->profit;
        j = u->level + 1;
        totweight = u->weight;
        while(j <= MAX && totweight + w[j] <= W) {   // 가능한 한 많은 아이템을 취한다.
            totweight = totweight + w[j];
            _bound = _bound + p[j++];
        }
        if(j <= MAX)
            _bound = _bound + (W - totweight) * p[j] / w[j];
        return _bound;
    }
}

/** DFS_knapsack **/
int numbest;
```

```
int bestset[MAX + 1];
int promising(int i, int profit, int weight);
void DFS_knapsack(int i, int profit, int weight, int include[MAX + 1]);
int promising(int i, int profit, int weight) {
    int totweight;
    float _bound;
    if(weight >= W) return false;    // 자식 마디로 확장할 수 있을 때만 마디는 유망하다.
    else {                           // 자식 마디에 쓸 공간이 남아 있어야 한다.
        _bound = profit;
        totweight = weight;
        while(i <= MAX && totweight + w[i] <= W) {  // 가능한 한 많은 아이템을 취한다.
            totweight = totweight + w[i];
            _bound = _bound + p[i++];
        }
        if(i <= MAX)
            _bound = _bound + (W - totweight) * p[i]/w[i];
        return _bound > maxprofit;
    }
}

// 깊이우선탐색
void DFS_knapsack(int i, int profit, int weight, int include[MAX + 1]) {
    if(weight <= W && profit > maxprofit) {          // 지금까지는 이 집합이 최고이다.
        maxprofit = profit;
        numbest = i;                                 // numbest를 고려한 아이템의 개수로
        memcpy(bestset, include, (MAX + 1) * sizeof(int));  // 놓고, bestset을 이 해답으로 놓는다.
        printf("\tSelect Item: ");
        for(int i = 1; i <= MAX; i++) {
            if(bestset[i] == true)
                printf("\ti%d(%d, %d)", i, w[i], p[i]);
        }
        printf("\n\tprofit: %d\n\n", profit);
    }
    if(promising(i + 1, profit, weight)) {
        include[i + 1] = true;   // w[i + 1] 포함
        DFS_knapsack(i + 1, profit + p[i + 1], weight + w[i + 1], include);
        include[i + 1] = false; // w[i + 1] 비포함
        DFS_knapsack(i + 1, profit, weight, include);
    }
}
/** BFS_knapsack **/
void init_q(queue *q);
void enqueue(node *v, queue *q);
node *dequeue(queue *q);
int q_is_empty(queue *q);
void BFS_knapsack(queue *q, node *v);
void init_q(queue *q) {
    q->front = q->rear = NULL;   // front와 rear를 NULL로 설정
    q->count = 0;                // 보관 개수를 0으로 설정
}

void enqueue(node *v, queue *q) {
    if (q_is_empty(q))          //큐가 비어있을 때
        q->front = v;
    else q->rear->next = v; // 큐가 비어있지 않을 때
    q->rear = v;                // 맨 뒤를 now로 설정
    q->count++;                 // 보관 개수를 1 증가
}

node *dequeue(queue *q) {
    node *temp;
    if (q_is_empty(q)) return NULL;

    temp = q->front;            //맨 앞의 노드를 now에 기억
```

```
	q->front = temp->next;   //맨 앞은 now의 다음 노드로 설정
	q->count--;                    //보관 개수를 1 감소
	printf("\t----- dequeue() -----\n\tLevel:%d\n\tProfit:%d\n\tWeight:%d\n\tBound:%d\n\n", temp->level, temp->profit, temp->weight, temp->bound);
	return temp;
}

int q_is_empty(queue *q) {
	return q->count == 0;
}

void BFS_knapsack(queue *q, node *v) {
	node *u;
	v->level = v->profit = v->weight = 0;
	v->bound = bound(v);
	enqueue(v, q);

	while(!q_is_empty(q)) {
		v = dequeue(q);
		u = malloc(sizeof(node));
		u->level = v->level + 1;                    // u를 v의 자식마디로 놓음
		u->profit = v->profit + p[u->level]; // u를 다음 아이템을 포함하는
		u->weight = v->weight + w[u->level]; // 자식마디로 놓음
		u->bound = bound(u);
		if(u->weight <= W && u->profit > maxprofit)
			maxprofit = u->profit;
		if(u->bound > maxprofit)
			enqueue(u, q);

		u = malloc(sizeof(node));
		u->level = v->level + 1;
		u->weight = v->weight;       // u를 다음 아이템을 포함하지 않는
		u->profit = v->profit;       // 자식마디로 놓음
		u->bound = bound(u);
		if(u->bound > maxprofit)
			enqueue(u, q);
	}
}

/** BF_knapsack **/
void init_priority_q(priority_queue *p_q);
int pq_is_empty(priority_queue *p_q);
void insert(priority_queue *p_q, node *v);
node *delete(priority_queue *p_q);
void BF_knapsack(priority_queue *p_q, node *v);
int get_parent(int i);
int get_left(int i);
int get_right(int i);

void init_priority_q(priority_queue *p_q) {
	p_q->heap = calloc(1024, sizeof(node *));
	p_q->count = 0;
}

int pq_is_empty(priority_queue *p_q) {
	return p_q->count == 0;
}

int get_parent(int i) {
	return i / 2;
}

int get_left(int i) {
	return i * 2;
}

int get_right(int i) {
	return i * 2 + 1;
```

```
}

void insert(priority_queue *p_q, node *v) {
	int index;
	node *temp;
	p_q->heap[++p_q->count] = v;
	index = get_parent(p_q->count);
	while (index > 0) {
		if(p_q->heap[get_left(index)] != NULL)
			if (p_q->heap[index]->bound < p_q->heap[get_left(index)]->bound) {
				temp = p_q->heap[index];
				p_q->heap[index] = p_q->heap[get_left(index)];
				p_q->heap[get_left(index)] = temp;
			}
		if(p_q->heap[get_right(index)] != NULL)
			if (p_q->heap[index]->bound < p_q->heap[get_right(index)]->bound) {
				temp = p_q->heap[index];
				p_q->heap[index] = p_q->heap[get_right(index)];
				p_q->heap[get_right(index)] = temp;
			}
		index = get_parent(index);
	}
}

node *delete(priority_queue *p_q) {
	int i, tempNo;

	node *answer = p_q->heap[1];                    // 부모 노드를 가져옴
	p_q->heap[1] = p_q->heap[p_q->count--]; // 마지막 노드를 가져옴

	printf("\t----- delete() -----\n\tLevel:%d\n\tProfit:%d\n\tWeight:%d\n\tBound:%d\n\n", answer->level, answer->profit, answer->weight,
answer->bound);

	tempNo = p_q->count;       // Heap 화
	p_q->count = 0;
	for (i = 1; i <= tempNo; i++)
		insert(p_q, p_q->heap[i]);
	return answer;
}


void BF_knapsack(priority_queue *p_q, node *v) {
	node *u;
	v->level = v->profit = v->weight = 0;
	v->bound = bound(v);
	insert(p_q, v);
	while(!pq_is_empty(p_q)) {   // 최고의 한계값을 가진 마디 제거
		v = delete(p_q);
		if(v->bound > maxprofit) {   // 마디가 아직 유망한지 검사
			u = malloc(sizeof(node));
			u->level = v->level + 1;
			u->profit = v->profit + p[u->level]; // u를 다음 아이템을 포함하는
			u->weight = v->weight + w[u->level]; // 자식마디로 놓음
			if(u->weight <= W && u->profit > maxprofit)
				maxprofit = u->profit;
			u->bound = bound(u);
			if(u->bound > maxprofit)
				insert(p_q, u);

			u = malloc(sizeof(node));
			u->level = v->level + 1;
			u->weight = v->weight;       // u를 다음 아이템을 포함하지 않는
			u->profit = v->profit;       // 자식마디로 놓음
			u->bound = bound(u);
			if(u->bound > maxprofit) {
				insert(p_q, u);
			}
```

```
            }
        }
}

int main(void) {
        while (1) {
                int select_num = 0;
                clock_t s_time, e_time;
                double res_time;
                printf("\t1. DFS_knapsack()\n");
                printf("\t2. BFS_knapsack()\n");
                printf("\t3. BF_knapsack()\n");
                printf("\t0. Exit\n");
                printf("\tInput Number: ");
                scanf("%d", &select_num);
                getchar();
                totalweight = 0;
                maxprofit = 0;
                switch (select_num) {
                        case 0:
                                return 0;
                        case 1: // Solve Knapsack Problem by using Dynmaic Programming.
                                printf("\n\t========== DFS_knapsack() ==========\n\n");
                                int include[MAX + 1] = {0};
                                s_time = clock();
                                DFS_knapsack(0, 0, 0, include);
                                e_time = clock();
                                res_time = (double)e_time - (double)s_time;
                                printf("\tBest set: ");
                                for(int i = 1; i <= MAX; i++) {
                                        if(bestset[i] == true)
                                                printf("\ti%d(%d, %d)", i, w[i], p[i]);
                                }
                                printf("\n\tmaxprofit: %d\n", maxprofit);
                                printf("\tperformance time : %fms\n\n\n", res_time);
                                _pause();
                                break;
                        case 2: // Solve Knapsack Problem by using Dynmaic Programming.
                                printf("\n\t========== BFS_knapsack() ==========\n\n");
                                node *v1 = malloc(sizeof(node));
                                queue *q = malloc(sizeof(queue));
                                init_q(q);
                                s_time = clock();
                                BFS_knapsack(q, v1);
                                e_time = clock();
                                res_time = (double)e_time - (double)s_time;
                                printf("\tmaxprofit: %d\n", maxprofit);
                                printf("\tperformance time : %fms\n\n\n", res_time);
                                _pause();
                                break;
                        case 3:
                                printf("\n\t========== BF_knapsack() ==========\n\n");
                                node *v2 = malloc(sizeof(node));
                                priority_queue *p_q = malloc(sizeof(priority_queue));
                                init_priority_q(p_q);
                                s_time = clock();
                                BF_knapsack(p_q, v2);
                                e_time = clock();
                                res_time = (double)e_time - (double)s_time;
                                printf("\tmaxprofit: %d\n", maxprofit);
                                printf("\tperformance time : %fms\n\n\n", res_time);
                                _pause();
                                break;
                        default:
                                break;
                }
        }
        return 0;
}
```