Knapsack 형태 문제를 구성하고,
해당 문제를 다양한 알고리즘으로 수행한 결과의 비교 분석 보고서

IT소프트웨어학과 / 201932030 / 송현교

제출일자 : 2023/12/10

목차

1. 문제)

당신은 유명한 보석 상인입니다. 당신은 출장 판매를 위해 가게에 존재하는 보석 중 몇 가지를 선택하여 가방에 넣어야 합니다. 당신의 시대에는 아직 자동차 같은 편리한 이동수단이 존재하지 않기 때문에, 매우 먼 거리를 도보로 이동해야만 합니다.

당신은 경험적으로 보았을 때, 가방에 약 8kg정도의 보석을 담는 것이 도보로 이동하기에 가장 적당할 것 같다고 판단했습니다. 어떤 보석들을 담아야 8kg을 초과하지 않고 가치를 최대화시킬 수 있을까요 ?

보석 목록)

다이아몬드 - 무게: 2kg, 가치: 10
에메랄드 - 무게: 3kg, 가치: 12
루비 - 무게: 4kg, 가치: 15
사파이어 - 무게: 5kg, 가치: 20
가방의 최대 허용 무게는 8kg입니다.

전제) 각 보석은 한 개씩만 존재하며, 한 번에 하나의 보석만 가방에 넣을 수 있습니다.
또한, 보석은 손상될 시 그 가치가 현저히 줄어들기 때문에, 각 품목은 분할이 불가능합니다.

서론)
시작에 앞서, 욕심쟁이 알고리즘부터 동적 프로그래밍, 너비 우선탐색, … , 분기 한정 가지치기 최고우선 알고리즘으로 구성해가며, 가시적으로 조금 더 나은 프로그램을 작성하기 위해 고민한 결과, 매 알고리즘의 출력 방식이 조금씩 달라져 온 점 양해 부탁드립니다.

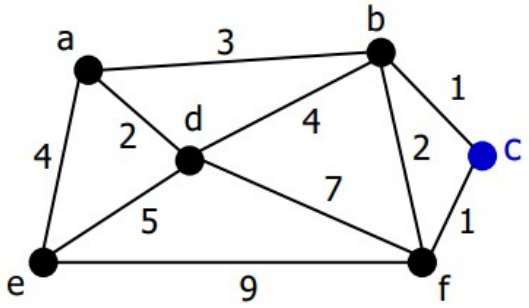
2) 욕심쟁이 알고리즘 (Greedy Algorithm)

욕심쟁이 알고리즘이란, “매 시점에 선택할 수 있는 모든 상황 중 당장 가장 최적인 답을 선택하여 적합한 결과를 도출하자” 라는 모토를 가지는 알고리즘 선택 기법이다.
욕심쟁이 기법, 그리디 기법, 탐욕 기법 등으로 불리운다.
즉, 여러 경우 중 하나를 결정해야 할 때마다 그 순간에 최적이라 생각되는 것을 선택해 나가는 방식이다.

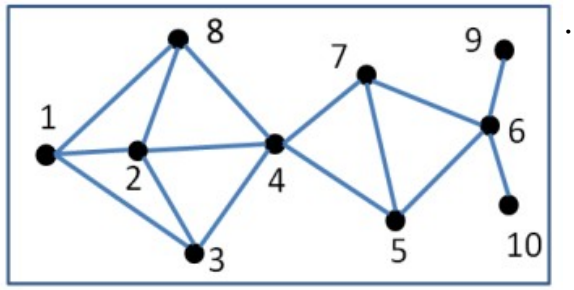
욕심쟁이 알고리즘은 문제 전체가 아닌 현 상황만으로 판단하고, 현 시점에서 보이지 않는 다음 상황을 고려하지 때문에 전체의 과정에서 최적해를 보장하지는 않는다. 그러나, 알고리즘의 구현이 쉽고, 시간이 적게 걸린다는 장점이 있다.

예를 들어, 마시멜로 실험에 비유할 수 있다. 그리디 알고리즘을 사용한다는 것은 지금 당장 눈 앞에 있는 마시멜로를 먹는 것이다. 하지만 이 방법을 사용하는 것은 “기다렸다가 2개를 먹는다” 라는 최적해를 보장해주지는 못한다.

욕심쟁이 알고리즘을 사용할 수 있는 문제들로는 대표적으로,

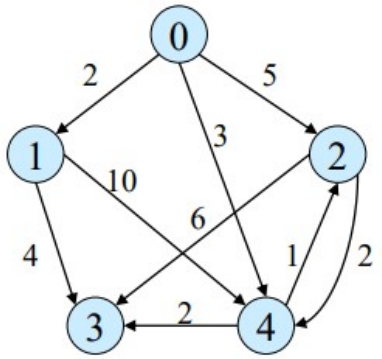


Kuruskul / Prim / Solin 최소 신장 트리



.

집합 커버 (Set Cover) 문제



Dijkstra 최단 경로 알고리즘 등이 존재한다.

2-1) 욕심쟁이 알고리즘 소스 코드

```
#include <stdio.h>

struct Jewel {
	int weight;
	int value;
	int valuePerWeight;
};

int CurrentMaxWeight = 8;

int ValuePerWeight(struct Jewel* jewel) {
	jewel->valuePerWeight = jewel->value / jewel->weight;
	return jewel->valuePerWeight;
}

bool isOverMaxWeight(struct Jewel jewel) {
	return jewel.weight <= CurrentMaxWeight;
}

void sortByValuePerWeight(struct Jewel* Jewels[], int size) {
	for (int i = 0; i < size - 1; i++) {
		int idx = i, max = (*Jewels[i]).valuePerWeight;
		for (int j = i; j < size; j++) {
			if (max < (*Jewels[j]).valuePerWeight) {
				max = (*Jewels[j]).valuePerWeight;
				idx = j;
			}
		}
		struct Jewel* temp;
		temp = Jewels[i];
		Jewels[i] = Jewels[idx];
		Jewels[idx] = temp;
	}
}

int Greedy(struct Jewel* jewels[], int size) {
	int totalValue = 0;
	for (int i = 0; i < size; i++) {
		if (isOverMaxWeight((*jewels[i]))) {
			printf("%d) 무게 : %d 가치 : %d 단위 무게당 가치 : %d₩n", i + 1,
jewels[i]->weight, jewels[i]->value, jewels[i]->valuePerWeight);
			CurrentMaxWeight -= (*jewels[i]).weight;
			totalValue += (*jewels[i]).value;
		}
	}
	return totalValue;
}

void main() {
	struct Jewel Diamond = { 2,10,0 };
	struct Jewel Emerald = { 3,12,0 };
	struct Jewel Ruby = { 4,15,0 };
	struct Jewel Sapphire = { 5,20,0 };
	struct Jewel* Jewels[] = { &Diamond,&Emerald,&Ruby,&Sapphire };
```

```
	struct Jewel* Selected_Jewels[] = { 0,0,0,0 };

	int size = sizeof(Jewels) / sizeof(Jewels[0]);
	int Ssize = sizeof(Selected_Jewels) / sizeof(Selected_Jewels[0]);

	for (int i = 0; i < size; i++) {
		Jewels[i]->valuePerWeight = ValuePerWeight(Jewels[i]);
	}

	sortByValuePerWeight(Jewels, size);

	printf("정렬된 보석 순서 : ₩n");
	for (int i = 0; i < size; i++) {
		printf("무게 : %d 가치 : %d 단위 무게당 가치 : %d₩n", Jewels[i]->weight,
Jewels[i]->value, Jewels[i]->valuePerWeight);
	}

	printf("₩n");

	printf("가방에 담은 보석 : ₩n");
	printf("₩n최대 가치 %d", Greedy(Jewels, size));
}
```

2-2) 예상 수행

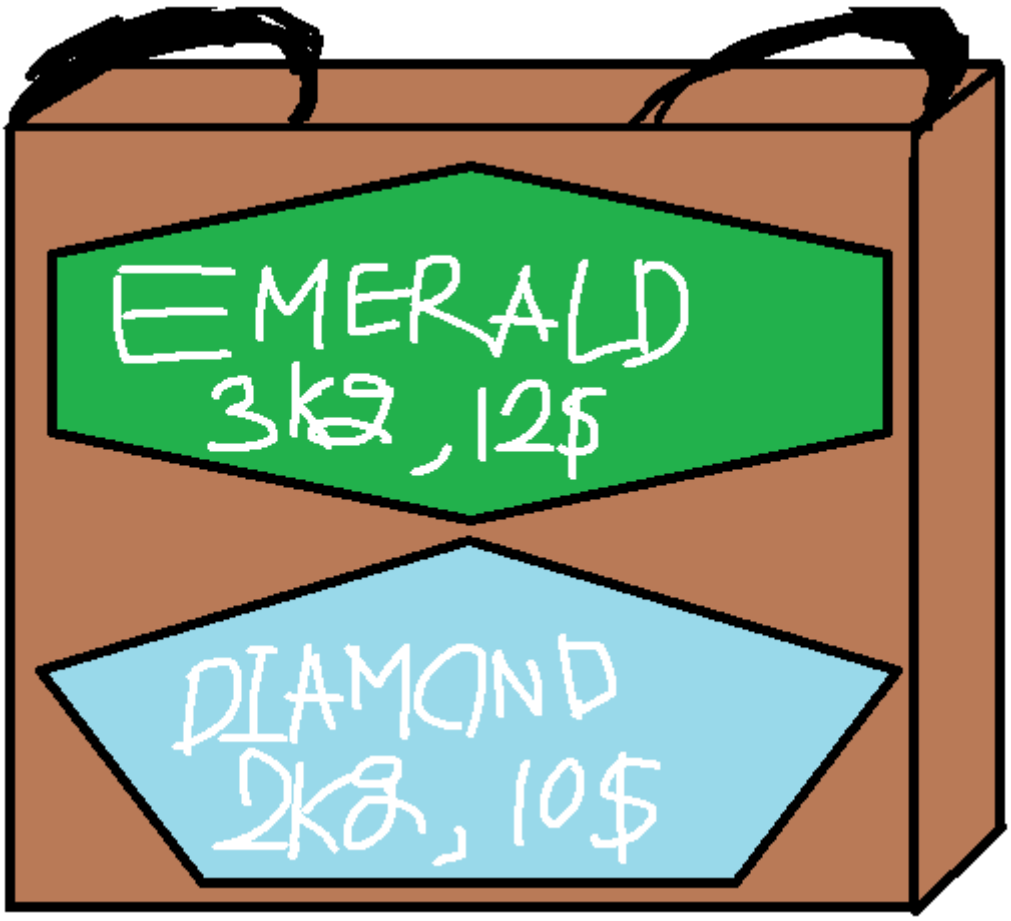
우선 보석들을 단위 무게당 가치가 높은 보석 순으로 정렬한 후 ,
정렬된 보석 중 가치가 높은 보석부터 차례로 가방에 넣게 됩니다.

단위 무게당 가치는
다이아몬드가 5,
에메랄드가 4,
루비가 약 3.7,
사파이어가 4로,

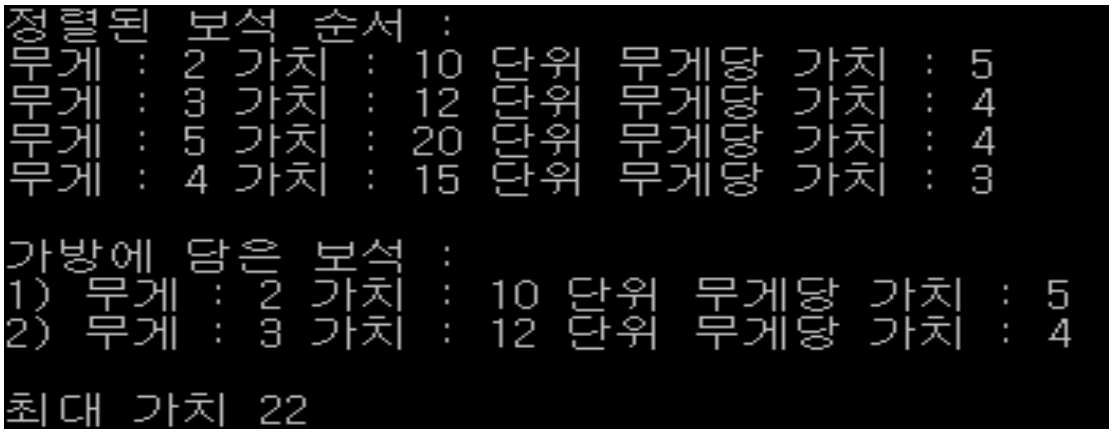
정렬하게 되면
다이아몬드,
에메랄드 (혹은 사파이어)
사파이어 (또는 에메랄드)
루비 의 순서가 됩니다.

욕심쟁이 알고리즘은 당장 최선의 경우를 선택하므로,
첫번째로 다이아몬드를, 두번째로 에메랄드를 선택하게 됩니다.

따라서, 가방에 들어가는 보석은 총합 5kg, 가치는 22일 것입니다.



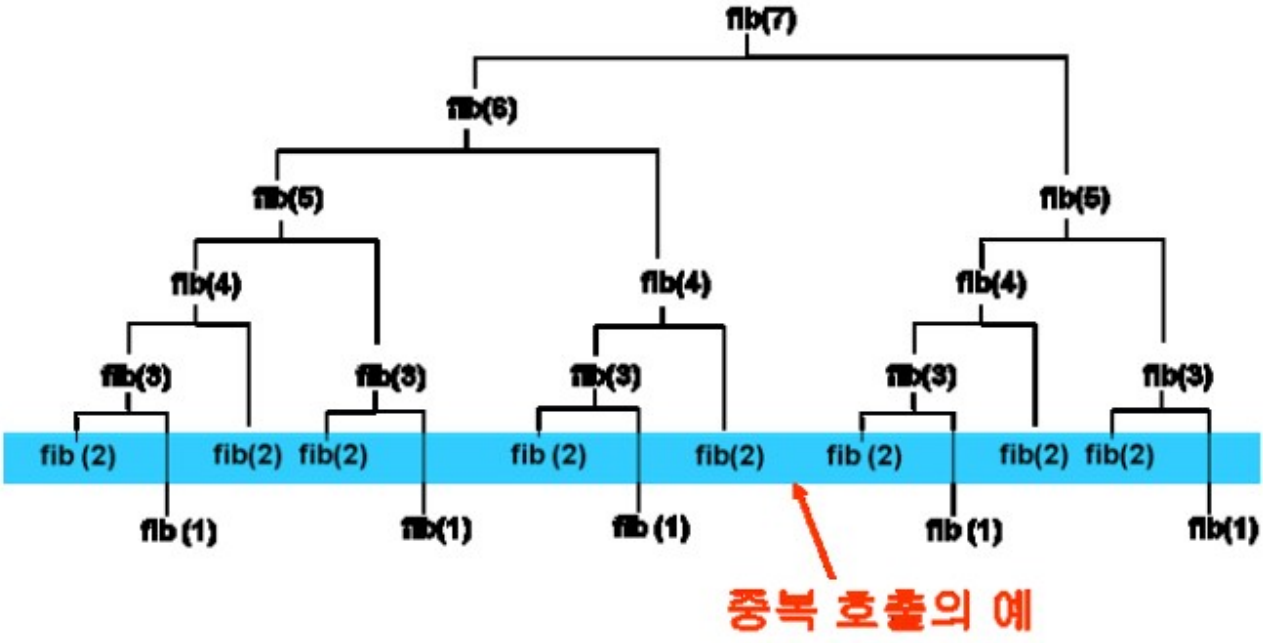
2-3) 소스 코드 실행 결과


정렬된 보석 순서 :
무게 : 2 가치 : 10 단위 무게당 가치 : 5
무게 : 3 가치 : 12 단위 무게당 가치 : 4
무게 : 5 가치 : 20 단위 무게당 가치 : 4
무게 : 4 가치 : 15 단위 무게당 가치 : 3
가방에 담은 보석 :
1) 무게 : 2 가치 : 10 단위 무게당 가치 : 5
2) 무게 : 3 가치 : 12 단위 무게당 가치 : 4
최대 가치 22

3) 동적 프로그래밍 (Dynamic Programming, DP)

동적 프로그래밍은 최적화 이론의 한 기술이며, 특정 범위까지의 값을 구하기 위하여 그것과 다른 범위까지의 값을 이용하여 효율적으로 답을 구하는 알고리즘 설계 기법이다.

답을 구하기 위해서 했던 계산을 또 하고 또 하고 계속해야 하는 종류의 문제 구조를 최적 부분 구조(Optimal Substructure)라고 부른다. 동적 계획법은 이런 문제들에서 주로 효과를 발휘한다. Subproblem들을 풀 때마다 배열에 기록해 두었다가 같은 계산을 수행할 필요가 있을 때 기록을 찾아 해로 이용할 수 있기 때문이다.



예) 피보나치 수열 구하기 문제에서 반복해서 계산되는 fib(2)

$$(1+x)^n = \sum_{r=0}^{n} {}_nC_r \circ x^r = {}_nC_0 + {}_nC_1 x + {}_nC_2 x^2 + \cdots + {}_nC_n x^n$$

예) 이항계수 구하기 문제

동적 계획 알고리즘은 최단 경로 문제, 행결의 제곱 문제 등의 최적화에 사용된다. 이것은 동적 계획법이 문제를 해결하기 위한 모든 방법을 검토하고, 그 중에 최적의 풀이법을 찾아내기 때문이다. 문제가 가능한 모든 방법을 충분히 빠른 속도로 처리할 수 있는 경우, 동적 계획법은 최적의 해법이라고 말할 수 있다.

3-1) 동적 프로그래밍 구현 소스 코드

```
#include <stdio.h>
#include <minmax.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
};

void main() {
  struct Jewel Diamond = { "Diamond ",2, 10 };
  struct Jewel Emerald = { "Emerald ", 3, 12 };
  struct Jewel Ruby = { "Ruby    ",4, 15 };
  struct Jewel Sapphire = { "Sapphire",5, 20 };
  struct Jewel Jewels[] = { Diamond, Emerald, Ruby, Sapphire };
  int size = sizeof(Jewels) / sizeof(Jewels[0]);

  int maxWeight = 8;
  int valueArr[5][9] = { 0 };
  int selectedItems[5] = { 0 };
  int i = size;
  int j = maxWeight;

  for (int j = 1; j <= size; j++) {
    printf("₩n%s %dkg %d$ ", Jewels[j - 1].name, Jewels[j - 1].weight, Jewels[j -
1].value);
    for (int y = 1; y <= maxWeight; y++) {
      if (y >= Jewels[j - 1].weight) {
        valueArr[j][y] = max(Jewels[j - 1].value + valueArr[j - 1][y - Jewels[j -
1].weight], valueArr[j - 1][y]);
      }
      else { valueArr[j][y] = valueArr[j - 1][y]; }
      printf("%2d ", valueArr[j][y]);
    }
  }

  printf("₩n₩n최대 가치: %d₩n₩n", valueArr[size][maxWeight]);

  printf("₩n가방에 담은 보석 : ");

  while (i > 0 && j > 0) {
    if (valueArr[i][j] != valueArr[i - 1][j]) {
      selectedItems[i - 1] = 1;
      j -= Jewels[i - 1].weight;
    }
    i--;
  }

  for (int i = 0; i < size; i++) {
    if (selectedItems[i]) { printf("%s ", Jewels[i].name); }
  }
}
```

3-2) 예상 수행 결과

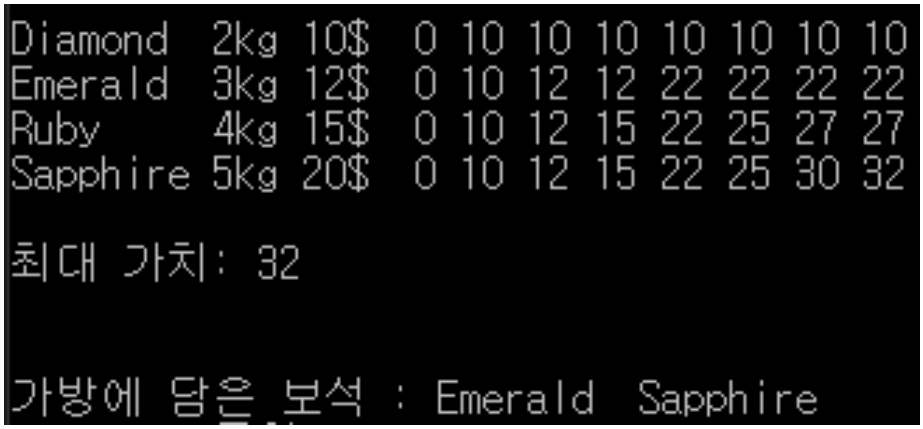
동적 계획법을 사용하면, 이전 항에 있던 결과값들을 통하여 다음 값을 도출하므로, 결과를 예상하는 것은 어렵지 않습니다. 결과는 다음과 같습니다.

	1	2	3	4	5	6	7	8
Diamond 2kg 10$	0	10	10	10	10	10	10	10
Emerald 3kg 12$	0	10	12	12	22	22	22	22
Ruby 4kg 15$	0	10	12	15	22	25	27	27
Sapphire 5kg 20$	0	10	12	15	15	25	30	32

표를 기반으로 소스 코드를 작성하게 되었습니다.

결과적으로, 표의 우측 최하단에 입력된, 32를 최대값으로 출력하게 될 것입니다.

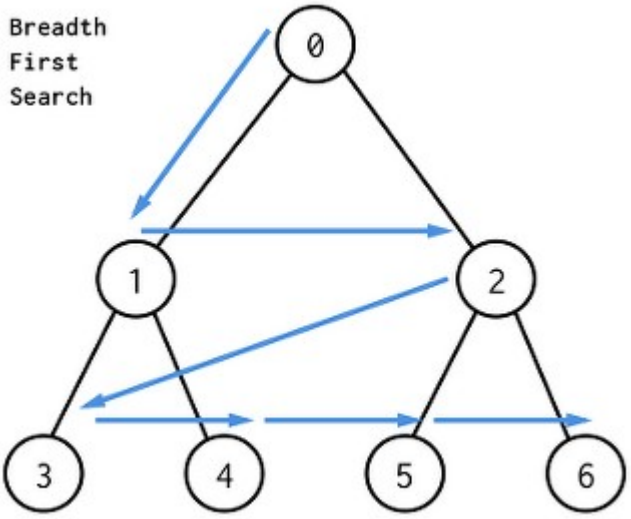
3-3) 실행 결과


Diamond 2kg 10$ 0 10 10 10 10 10 10 10
Emerald 3kg 12$ 0 10 12 12 22 22 22 22
Ruby 4kg 15$ 0 10 12 15 22 25 27 27
Sapphire 5kg 20$ 0 10 12 15 22 25 30 32
최대 가치: 32
가방에 담은 보석 : Emerald Sapphire

4) 너비 우선 탐색 (Breath – First Search, BFS) 알고리즘

너비 우선 탐색은 맹목적 탐색 방법의 하나로, 시작 정점을 방문한 후 시작 정점에 인접한 모든 정점들을 우선 방문하는 방법이다. 더 이상 방문하지 않은 정점이 없을 때 까지 방문하지 않은 모든 정점들에 대해서도 너비 우선 탐색을 적용한다.

출발 노드에서 목표 노드까지의 최단 길이 경로를 보장한다는 장점이 존재하나, 경로가 매우 길 경우에는 탐색 가지가 급격히 증가함에 따라 보다 많은 기억 공간을 필요로 하게 되며, 해가 존재하지 않는 유한 그래프의 경우 모든 그래프를 탐색한 후에 실패로 끝난다는 단점이 존재한다.



사진은 너비 우선 탐색의 탐색 순서이다.
출처 : https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om

깊이가 가장 얕은 노드부터 모두 탐색한 뒤 깊이가 깊은 노드를 탐색하게 된다.

알고리즘의 구현은,

1. 루트 노드에서 시작한다.
2. 루트노드와 인접하고, 방문된 적 없고, 큐에 저장되지 않은 상태의 노드를 Queue에 삽입한다.
3. Queue에서 dequeue 하여 가장 먼저 큐에 저장한 노드를 방문한다.

의 순서에 따라 이루어진다.

4-1) 너비 우선 탐색 알고리즘 구현 소스 코드

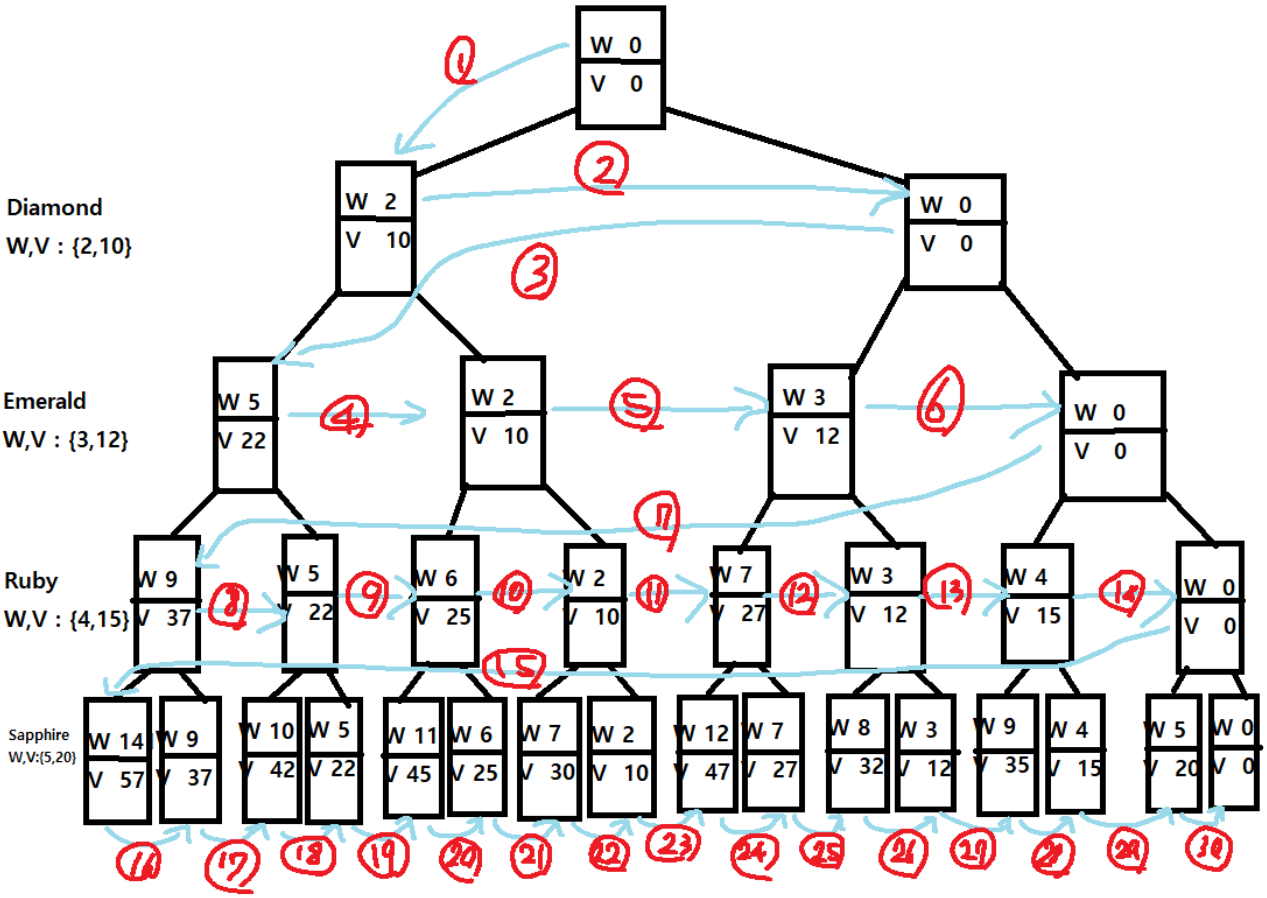
```
#include <stdio.h>
#include <stdlib.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
};

struct JewelArray {
  int value;
  int weight;
};

typedef struct Node {
  struct JewelArray data;
  struct Node* next;
} Node;

typedef struct Queue {
  Node* front;
  Node* rear;
  int count;
} Queue;

void init(Queue* queue) {
  queue->front = queue->rear = NULL;
  queue->count = 0;
}

int isEmpty(Queue* queue) {
  return queue->count == 0;
}

void enqueue(Queue* queue, struct JewelArray data) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  newNode->data = data;
  newNode->next = NULL;

  if (isEmpty(queue)) {
    queue->front = newNode;
  }
  else {
    queue->rear->next = newNode;
  }
  queue->rear = newNode;
  queue->count++;
}

struct JewelArray dequeue(Queue* queue) {
  struct JewelArray data;
  Node* ptr;
  if (isEmpty(queue)) {
```

```
    return JewelArray{ 0, 0 };
  }
  ptr = queue->front;
  data = ptr->data;
  queue->front = ptr->next;
  free(ptr);
  queue->count--;
  return data;
}

struct JewelArray showQueue(Queue* queue, int i) {
  struct JewelArray data;
  Node* ptr;
  if (isEmpty(queue)) {
    return JewelArray{ 0,0 };
  }
  ptr = queue->front;
  data = ptr->data;

  printf("%d번째 연산 후 큐 상태₩n", i);
  printf("[Value : %d, Weight : %d] ₩n", ptr->data.value, ptr->data.weight);
  while (ptr != NULL) {
    ptr = ptr->next;
  }
  printf("₩n");
}

void main() {
  Queue queue;
  init(&queue);

  struct Jewel Diamond = { "Diamond ", 2, 10 };
  struct Jewel Emerald = { "Emerald ", 3, 12 };
  struct Jewel Ruby = { "Ruby    ", 4, 15 };
  struct Jewel Sapphire = { "Sapphire", 5, 20 };
  struct Jewel Jewels[] = { Diamond, Emerald, Ruby, Sapphire };
  int size = sizeof(Jewels) / sizeof(Jewels[0]);
  struct JewelArray JewelArray[31] = { 0, 0 };
  int JewelArraySize = sizeof(JewelArray) / sizeof(JewelArray[0]);

  int maxWeight = 8;
  int currentWeight = 0;
  int maxValue = 0;
  int index = 0;
  int LEVEL = 0;
  int j = 0;

  for (int i = 0; i < 15; i++) {
    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 1].value = JewelArray[j].value + Jewels[LEVEL].value;
    JewelArray[i * 2 + 1].weight = JewelArray[j].weight + Jewels[LEVEL].weight;

    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 2].value = JewelArray[j].value;
```

```
    JewelArray[i * 2 + 2].weight = JewelArray[j].weight;

    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  LEVEL = 0;

  for (int i = 0; i < 31; i++) {
    printf("%d) [value:%d, weight:%d] {%d, %d}₩n", i, JewelArray[i].value,
JewelArray[i].weight, LEVEL, i);
    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }
  printf("₩n이진 트리 구성 완료 !₩n₩n");

  enqueue(&queue, JewelArray[0]);

  while (!isEmpty(&queue)) {
    struct JewelArray currentNode = dequeue(&queue);

    if (2 * index + 1 < JewelArraySize) {
      enqueue(&queue, JewelArray[2 * index + 1]);
    }

    if (2 * index + 2 < JewelArraySize) {
      enqueue(&queue, JewelArray[2 * index + 2]);
    }

    if ((currentNode.value > maxValue) && (currentNode.weight <= maxWeight)) {
      maxValue = currentNode.value;
      currentWeight = currentNode.weight;
      printf("최대 가치 갱신 !₩n₩
n---------------------------------------------------------------------------- ₩
n₩n");
    }

    showQueue(&queue, index);
    index++;
  }
  printf("최대 가치) Value : %d, Weight : %d₩n", maxValue, currentWeight);
}
```

4-2) 예상 수행 결과

너비 우선 알고리즘의 실행 과정 예상도입니다.



이 예상도를 기반으로 소스 코드를 작성하게 되었습니다.

결과적으로, 25번째 과정을 거치며 얻은 최대값인 {W:8,V:32}을 저장해 두었다가 탐색을 마친 후 출력하게 될 것입니다.

4-3) 실행 결과

```
0) [value:0, weight:0] {0, 0}
1) [value:10, weight:2] {1, 1}
2) [value:0, weight:0] {1, 2}
3) [value:22, weight:5] {2, 3}
4) [value:10, weight:2] {2, 4}
5) [value:12, weight:3] {2, 5}
6) [value:0, weight:0] {2, 6}
7) [value:37, weight:9] {3, 7}
8) [value:22, weight:5] {3, 8}
9) [value:25, weight:6] {3, 9}
10) [value:10, weight:2] {3, 10}
11) [value:27, weight:7] {3, 11}
12) [value:12, weight:3] {3, 12}
13) [value:15, weight:4] {3, 13}
14) [value:0, weight:0] {3, 14}
15) [value:57, weight:14] {4, 15}
16) [value:37, weight:9] {4, 16}
17) [value:42, weight:10] {4, 17}
18) [value:22, weight:5] {4, 18}
19) [value:45, weight:11] {4, 19}
20) [value:25, weight:6] {4, 20}
21) [value:30, weight:7] {4, 21}
22) [value:10, weight:2] {4, 22}
23) [value:47, weight:12] {4, 23}
24) [value:27, weight:7] {4, 24}
25) [value:32, weight:8] {4, 25}
26) [value:12, weight:3] {4, 26}
27) [value:35, weight:9] {4, 27}
28) [value:15, weight:4] {4, 28}
29) [value:20, weight:5] {4, 29}
30) [value:0, weight:0] {4, 30}

이진 트리 구성 완료 !

<0> [Value : 10, Weight : 2]
최대 가치 갱신 !

--------------------------------
```

```
<1> [Value : 0, Weight : 0]
<2> [Value : 22, Weight : 5]
최대 가치 갱신 !

--------------------------------

<3> [Value : 10, Weight : 2]
<4> [Value : 12, Weight : 3]
<5> [Value : 0, Weight : 0]
<6> [Value : 37, Weight : 9]
<7> [Value : 22, Weight : 5]
<8> [Value : 25, Weight : 6]
최대 가치 갱신 !

--------------------------------

<9> [Value : 10, Weight : 2]
<10> [Value : 27, Weight : 7]
최대 가치 갱신 !

--------------------------------

<11> [Value : 12, Weight : 3]
<12> [Value : 15, Weight : 4]
<13> [Value : 0, Weight : 0]
<14> [Value : 57, Weight : 14]
<15> [Value : 37, Weight : 9]
<16> [Value : 42, Weight : 10]
<17> [Value : 22, Weight : 5]
<18> [Value : 45, Weight : 11]
<19> [Value : 25, Weight : 6]
<20> [Value : 30, Weight : 7]
최대 가치 갱신 !
```

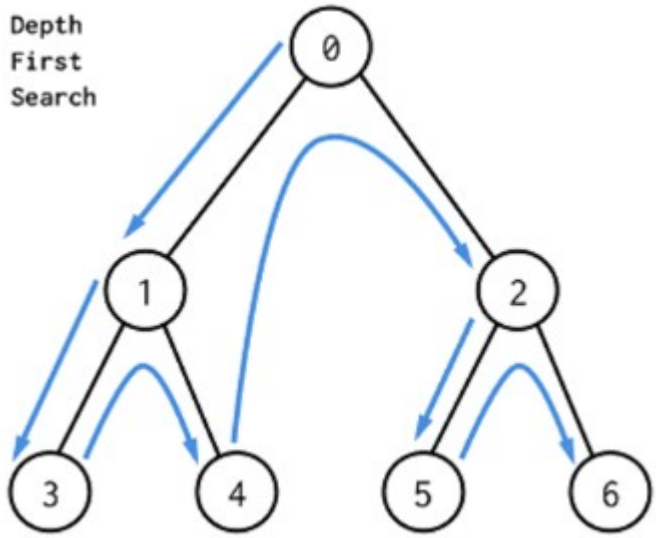
```
<21> [Value : 10, Weight : 2]
<22> [Value : 47, Weight : 12]
<23> [Value : 27, Weight : 7]
<24> [Value : 32, Weight : 8]
최대 가치 갱신 !

--------------------------------

<25> [Value : 12, Weight : 3]
<26> [Value : 35, Weight : 9]
<27> [Value : 15, Weight : 4]
<28> [Value : 20, Weight : 5]
<29> [Value : 0, Weight : 0]
최대 가치) Value : 32, Weight : 8
```

5) 깊이 우선 탐색 (Depth – First Search, DFS) 알고리즘

깊이 우선 탐색은 맹목적 탐색 방법의 하나로 탐색트리의 최근에 첨가된 노드를 선택하고, 이 노드에 적용 가능한 동작자 중 하나를 적용하여 트리에 다음 수준(*level*)의 한 개의 자식노드를 첨가하며, 첨가된 자식 노드가 목표노드일 때까지 앞의 자식 노드의 첨가 과정을 반복해 가는 방식이다.

현 경로상의 노드들만을 기억하면 되므로 저장공간의 수요가 비교적 적고, 목표노드가 깊은 단계에 있을 경우 해를 빨리 구할 수 있다는 장점이 있으나,

목표에 이르는 경로가 다수인 문제에 대해 깊이 우선 탐색은 해에 다다르면 탐색을 끝내버리므로 얻어진 해가 최단 경로가 된다는 보장이 없다는 점과, 해가 없는 경로에 깊이 빠질 수도 있다는 점 등 단점 또한 존재한다.



사진은 깊이 우선 탐색의 탐색 순서이다.
출처 : https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om

깊이 우선 탐색의 구현은 스택으로 이루어진다.

1. 시작 정점을 스택에 삽입한다.
2. 스택에서 하나의 정점을 꺼낸다.
3. 스택에서 꺼낸 정점이 아직 방문하지 않은 정점이라면, 방문 표시 후 이웃 정점들을 스택에 삽입한다.
4. 스택에 담긴 정점이 없을 때 까지 과정을 반복한다.

의 과정으로 이루어지게 된다.

5-1) 깊이 우선 탐색 알고리즘 구현 소스 코드

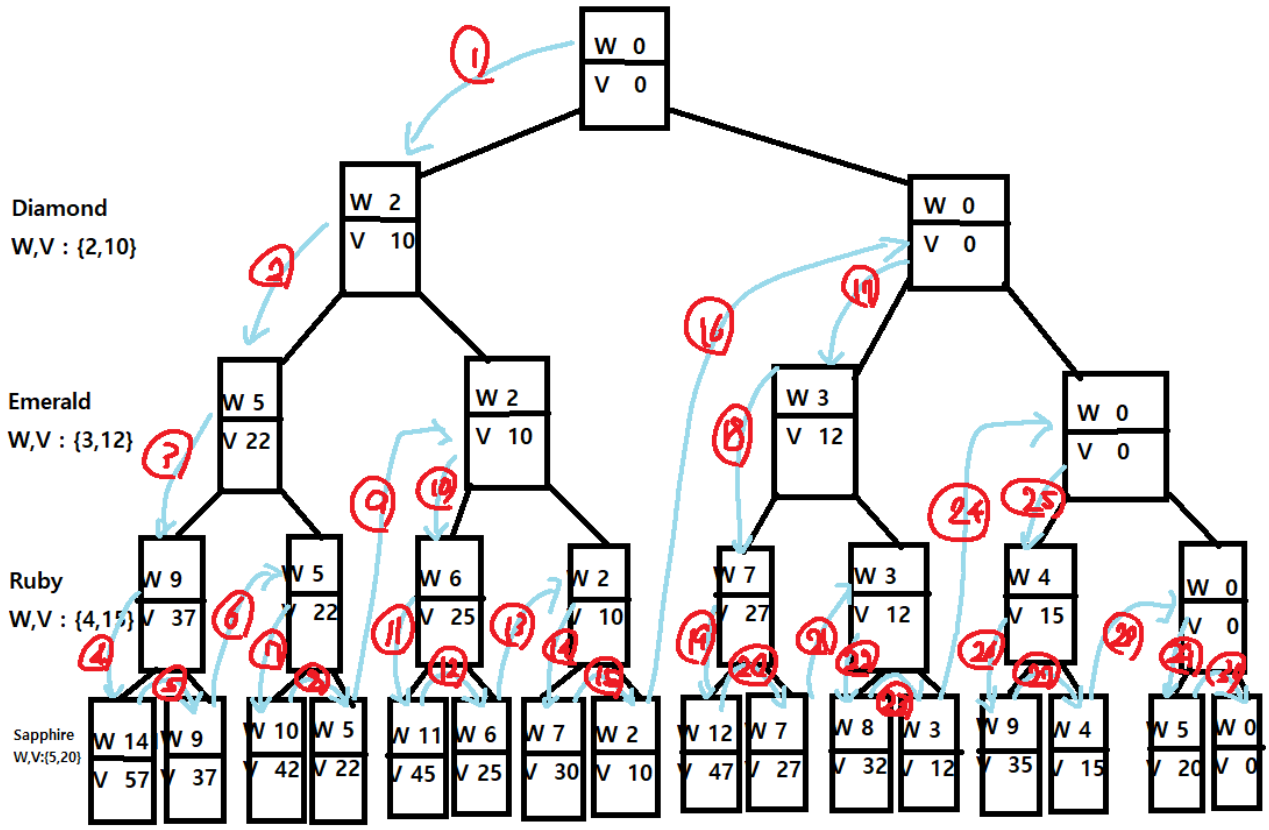
```
#include <stdio.h>
#include <stdlib.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
};

struct JewelArray {
  int value;
  int weight;
};

typedef struct Node {
  struct JewelArray data;
  struct Node* left;
  struct Node* right;
} Node;

typedef struct {
  Node* top;
} Stack;

void init(Stack* stack) {
  stack->top = NULL;
}

int isEmpty(Stack* stack) {
  return stack->top == NULL;
}

void push(Stack* stack, struct JewelArray data, Node* left, Node* right) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  if (newNode == NULL) { return; }
  newNode->data = data;
  newNode->left = left;
  newNode->right = right;

  if (isEmpty(stack)) {
    stack->top = newNode;
  }
  else {
    newNode->left = stack->top;
    stack->top = newNode;
  }
}

struct JewelArray pop(Stack* stack, Node** left, Node** right) {
  if (isEmpty(stack)) {
    return JewelArray{ 0, 0 };
  }

  Node* temp = stack->top;
  struct JewelArray popItem = temp->data;
  *left = temp->left;
  *right = temp->right;
  stack->top = stack->top->left;
  free(temp);
```

```
  return popItem;
}

struct JewelArray DFS(Node* CurrentNode, int maxWeight) {
  struct JewelArray static result = { 0,0 };
  static int index = 0;
  static int maxValue = 0;
  static int currentWeight = 0;

  if (CurrentNode == NULL) {
    return result;
  }

  index++;
  printf("<%d> : [value:%d, weight:%d]₩n", index, CurrentNode->data.value, CurrentNode->data.weight);

  if (CurrentNode->data.value > maxValue && CurrentNode->data.weight <= maxWeight) {
    maxValue = CurrentNode->data.value;
    currentWeight = CurrentNode->data.weight;
    result = { maxValue, currentWeight };
    printf("최대 가치 갱신 !₩n₩
n---------------------------------------------------------------------------------- ₩n₩n");
  }

  if (CurrentNode->left != NULL) { DFS(CurrentNode->left, maxWeight); }
  if (CurrentNode->right != NULL) { DFS(CurrentNode->right, maxWeight); }

  return result;
}

Node* createNode(struct JewelArray data) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  if (newNode == NULL) { return { 0 }; }
  newNode->data = data;
  newNode->left = newNode->right = NULL;
  return newNode;
}

void buildTree(Node** root, struct JewelArray* JewelArray, int i, int n) {
  if (i < n) {
    *root = createNode(JewelArray[i]);

    buildTree(&((*root)->left), JewelArray, 2 * i + 1, n);
    buildTree(&((*root)->right), JewelArray, 2 * i + 2, n);
  }
}

void main() {
  struct Jewel Diamond = { "Diamond ", 2, 10 };
  struct Jewel Emerald = { "Emerald ", 3, 12 };
  struct Jewel Ruby = { "Ruby    ", 4, 15 };
  struct Jewel Sapphire = { "Sapphire", 5, 20 };
  struct Jewel Jewels[] = { Diamond, Emerald, Ruby, Sapphire };
  struct JewelArray Result = { 0,0 };
  int size = sizeof(Jewels) / sizeof(Jewels[0]);
  struct JewelArray JewelArray[31] = { 0, 0 };
  int JewelArraySize = sizeof(JewelArray) / sizeof(JewelArray[0]);
  int maxWeight = 8;
  int index = 0;
  int LEVEL = 0;
  int j = 0;

  for (int i = 0; i < 15; i++) {
```

```
    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 1].value = JewelArray[j].value + Jewels[LEVEL].value;
    JewelArray[i * 2 + 1].weight = JewelArray[j].weight + Jewels[LEVEL].weight;

    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 2].value = JewelArray[j].value;
    JewelArray[i * 2 + 2].weight = JewelArray[j].weight;

    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  LEVEL = 0;

  for (int i = 0; i < 31; i++) {
    printf("%d) [value:%d, weight:%d] {%d, %d}₩n", i, JewelArray[i].value, JewelArray[i].weight,
LEVEL, i);
    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  printf("₩n이진 트리 구성 완료 !₩n₩n");

  Node* root = NULL;
  buildTree(&root, JewelArray, 0, JewelArraySize);

  Result = DFS(root, maxWeight);
  printf("최대 가치: %d, 무게 : %d₩n", Result.value, Result.weight);
}
```

5-2) 예상 수행 결과

깊이 우선 알고리즘의 실행 과정 예상도입니다.



이 예상도를 기반으로 소스 코드를 작성하게 되었습니다.

결과적으로, 22번째 과정을 거치며 얻은 최대값인 {W:8,V:32}을 저장해 두었다가 탐색을 마친 후 출력하게 될 것입니다.

5-3) 실행 결과

```
0) [value:0, weight:0] {0, 0}
1) [value:10, weight:2] {1, 1}
2) [value:0, weight:0] {1, 2}
3) [value:22, weight:5] {2, 3}
4) [value:10, weight:2] {2, 4}
5) [value:12, weight:3] {2, 5}
6) [value:0, weight:0] {2, 6}
7) [value:37, weight:9] {3, 7}
8) [value:22, weight:5] {3, 8}
9) [value:25, weight:6] {3, 9}
10) [value:10, weight:2] {3, 10}
11) [value:27, weight:7] {3, 11}
12) [value:12, weight:3] {3, 12}
13) [value:15, weight:4] {3, 13}
14) [value:0, weight:0] {3, 14}
15) [value:57, weight:14] {4, 15
16) [value:37, weight:9] {4, 16}
17) [value:42, weight:10] {4, 17
18) [value:22, weight:5] {4, 18}
19) [value:45, weight:11] {4, 19
20) [value:25, weight:6] {4, 20}
21) [value:30, weight:7] {4, 21}
22) [value:10, weight:2] {4, 22}
23) [value:47, weight:12] {4, 23
24) [value:27, weight:7] {4, 24}
25) [value:32, weight:8] {4, 25}
26) [value:12, weight:3] {4, 26}
27) [value:35, weight:9] {4, 27}
28) [value:15, weight:4] {4, 28}
29) [value:20, weight:5] {4, 29}
30) [value:0, weight:0] {4, 30}

이진 트리 구성 완료 !

<1> : [value:0, weight:0]
<2> : [value:10, weight:2]
최대 가치 갱신 !
```

```
<3> : [value:22, weight:5]
최대 가치 갱신 !

------------------------------

<4> : [value:37, weight:9]
<5> : [value:57, weight:14]
<6> : [value:37, weight:9]
<7> : [value:22, weight:5]
<8> : [value:42, weight:10]
<9> : [value:22, weight:5]
<10> : [value:10, weight:2]
<11> : [value:25, weight:6]
최대 가치 갱신 !

------------------------------

<12> : [value:45, weight:11]
<13> : [value:25, weight:6]
<14> : [value:10, weight:2]
<15> : [value:30, weight:7]
최대 가치 갱신 !

------------------------------

<16> : [value:10, weight:2]
<17> : [value:0, weight:0]
<18> : [value:12, weight:3]
<19> : [value:27, weight:7]
<20> : [value:47, weight:12]
<21> : [value:27, weight:7]
<22> : [value:12, weight:3]
<23> : [value:32, weight:8]
최대 가치 갱신 !
```

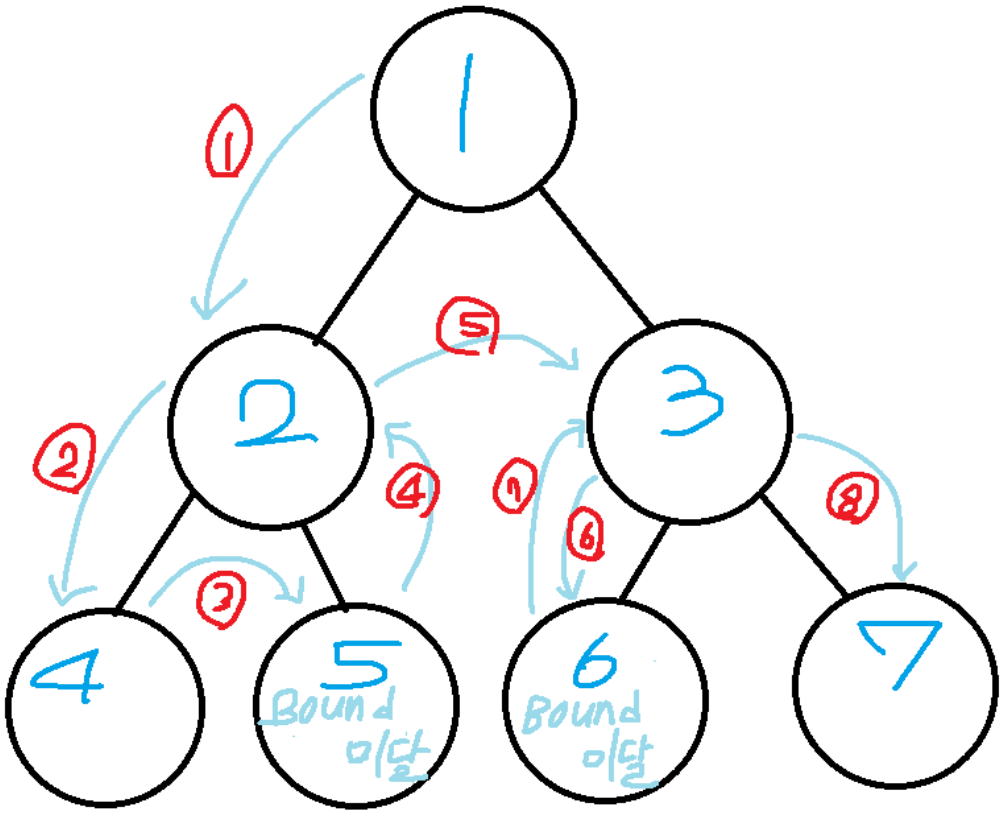
```
<24> : [value:12, weight:3]
<25> : [value:0, weight:0]
<26> : [value:15, weight:4]
<27> : [value:35, weight:9]
<28> : [value:15, weight:4]
<29> : [value:0, weight:0]
<30> : [value:20, weight:5]
<31> : [value:0, weight:0]
최대 가치: 32, 무게 : 8
```

6) 되추적 (Backtracking) 알고리즘

되추적 알고리즘은 깊이 우선 탐색 / 너비 우선 탐색을 기반으로 한다. 트리를 탐색하는 도중 오답을 만나면 이전 분기점으로 돌아간다. 시도해보지 않은 다른 해결 방법이 있으면 시도한다. 해결 방법이 더 없으면 이전의 분기점으로 돌아간다. 모든 트리의 노드를 검사해도 답을 찾지 못할 경우, 이 문제의 해결책은 없는 것이다.

즉, 어떤 노드의 유망성을 점검한 후에 유망하지 않다고 판단되면 그 노드의 부모로 되돌아가, 다음 자식 노드로 이동하는 기법이다.

유망하다 함은, 어떤 노드를 방문하였을 때 그 노드를 포함한 경로가 해답이 될 수 있다는 의미이고, 반대로 해답의 가능성이 없으면 해당 경로는 유망하지 않다.



사진은 되추적 기법의 예시이다. 만약 5번 노드의 유망성을 판단하여 유망하지 않다고 판단되었을 경우, 그의 부모 노드인 2번 노드로 다시 돌아가 다음 탐색을 진행하는 것이다. 6번 노드도 마찬가지이다. 6번 노드가 유망하지 않다고 판단될 경우 부모 노드인 3번 노드로 돌아가 탐색을 재개하게 된다.

6-1) 되추적 알고리즘 구현 소스 코드

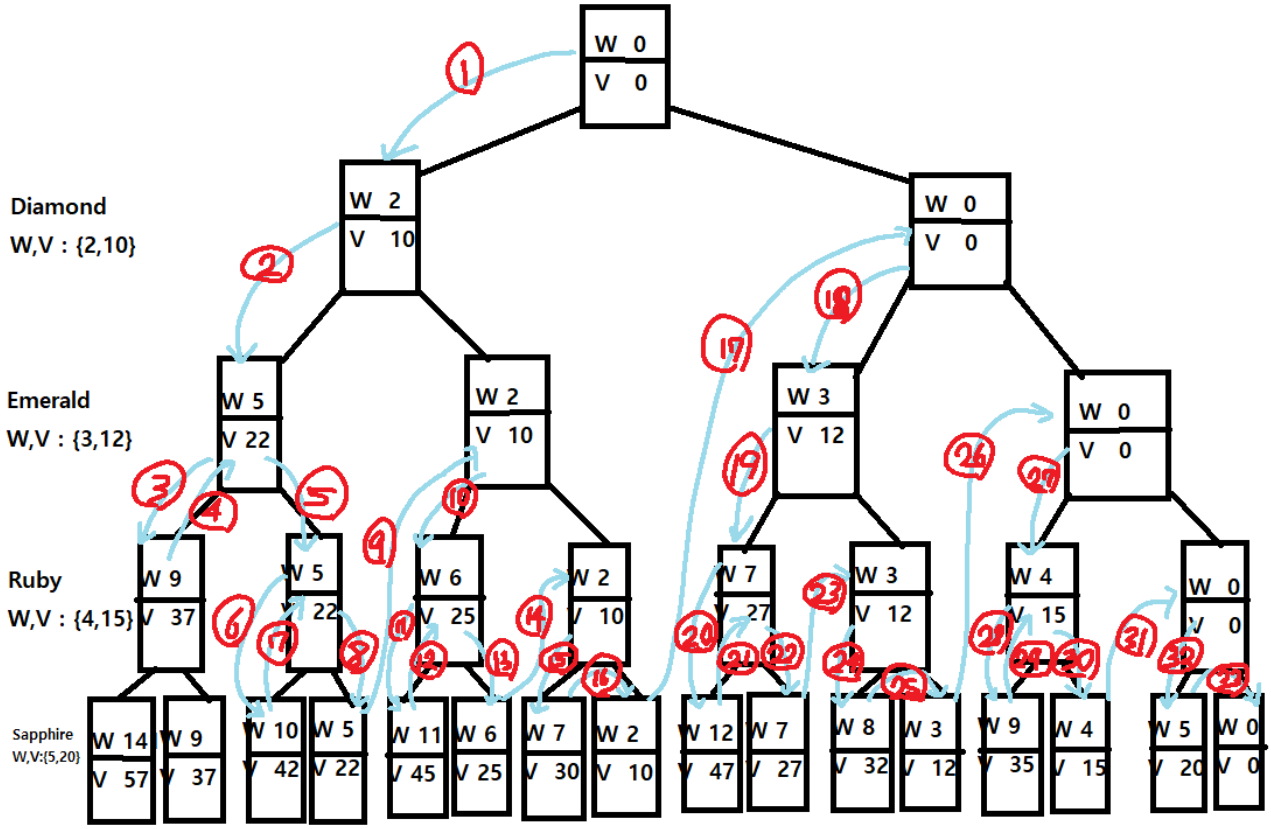
```
#include <stdio.h>
#include <stdlib.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
};

struct JewelArray {
  int value;
  int weight;
};

typedef struct Node {
  struct JewelArray data;
  struct Node* left;
  struct Node* right;
} Node;

typedef struct {
  Node* top;
} Stack;

void init(Stack* stack) {
  stack->top = NULL;
}

int isEmpty(Stack* stack) {
  return stack->top == NULL;
}

void push(Stack* stack, struct JewelArray data, Node* left, Node* right) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  if (newNode == NULL) { return; }
  newNode->data = data;
  newNode->left = left;
  newNode->right = right;

  if (isEmpty(stack)) {
    stack->top = newNode;
  }
  else {
    newNode->left = stack->top;
    stack->top = newNode;
  }
}

struct JewelArray pop(Stack* stack, Node** left, Node** right) {
  if (isEmpty(stack)) {
    return JewelArray{ 0, 0 };
  }

  Node* temp = stack->top;
  struct JewelArray popItem = temp->data;
  *left = temp->left;
  *right = temp->right;
  stack->top = stack->top->left;
  free(temp);

  return popItem;
```

```
}

struct JewelArray DFS(struct Node* CurrentNode, int maxWeight) {
  struct JewelArray static result = { 0, 0 };
  static int index = 0;
  static int maxValue = 0;
  static int currentWeight = 0;

  if (CurrentNode == NULL || CurrentNode->data.weight > maxWeight) {
    index++;
    printf("<%d> 무게 초과이므로 되추적하여 탐색합니다.₩n", index);
    return result;
  }

  index++;
  printf("<%d> : [value:%d, weight:%d]₩n", index, CurrentNode->data.value, CurrentNode->data.weight);

  if (CurrentNode->data.value > maxValue && CurrentNode->data.weight <= maxWeight) {
    maxValue = CurrentNode->data.value;
    currentWeight = CurrentNode->data.weight;
    result = { maxValue, currentWeight };
    printf("최대 가치 갱신 !₩n₩
n---------------------------------------------------------------------------------------- ₩n₩n");
  }

  if (CurrentNode->left != NULL) { DFS(CurrentNode->left, maxWeight); }
  if (CurrentNode->right != NULL) { DFS(CurrentNode->right, maxWeight); }

  return result;
}

Node* createNode(struct JewelArray data) {
  Node* newNode = (Node*)malloc(sizeof(Node));
  if (newNode == NULL) { return { 0 }; }
  newNode->data = data;
  newNode->left = newNode->right = NULL;
  return newNode;
}

void buildTree(Node** root, struct JewelArray* JewelArray, int i, int n) {
  if (i < n) {
    *root = createNode(JewelArray[i]);

    buildTree(&((*root)->left), JewelArray, 2 * i + 1, n);
    buildTree(&((*root)->right), JewelArray, 2 * i + 2, n);
  }
}

void main() {
  struct Jewel Diamond = { "Diamond ", 2, 10 };
  struct Jewel Emerald = { "Emerald ", 3, 12 };
  struct Jewel Ruby = { "Ruby    ", 4, 15 };
  struct Jewel Sapphire = { "Sapphire", 5, 20 };
  struct Jewel Jewels[] = { Diamond, Emerald, Ruby, Sapphire };
  struct JewelArray Result = { 0,0 };
  int size = sizeof(Jewels) / sizeof(Jewels[0]);
  struct JewelArray JewelArray[31] = { 0, 0 };
  int JewelArraySize = sizeof(JewelArray) / sizeof(JewelArray[0]);
  int maxWeight = 8;
  int index = 0;
  int LEVEL = 0;
  int j = 0;
```

```
  for (int i = 0; i < 15; i++) {
    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 1].value = JewelArray[j].value + Jewels[LEVEL].value;
    JewelArray[i * 2 + 1].weight = JewelArray[j].weight + Jewels[LEVEL].weight;

    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 2].value = JewelArray[j].value;
    JewelArray[i * 2 + 2].weight = JewelArray[j].weight;

    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  LEVEL = 0;

  for (int i = 0; i < 31; i++) {
    printf("%d) [value:%d, weight:%d] {%d, %d}₩n", i, JewelArray[i].value, JewelArray[i].weight,
LEVEL, i);
    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  printf("₩n이진 트리 구성 완료 !₩n₩n");

  Node* root = NULL;
  buildTree(&root, JewelArray, 0, JewelArraySize);

  Result = DFS(root, maxWeight);
  printf("최대 가치: %d, 무게 : %d₩n", Result.value, Result.weight);
}
```

6-2) 예상 수행 결과

제가 작성한 되추적 기법 코드의 예상 수행도를 첨부합니다



이 예상도를 기반으로 소스 코드를 작성하게 되었습니다.

결과적으로, 14번째 과정을 거치며 얻은 최대값인 {W:8,V:32}을 저장해 두었다가 탐색을 마친 후 출력하게 될 것입니다.

6-3) 되추적 알고리즘 실행 결과

```
0) [value:0, weight:0] {0, 0}
1) [value:10, weight:2] {1, 1}
2) [value:0, weight:0] {1, 2}
3) [value:22, weight:5] {2, 3}
4) [value:10, weight:2] {2, 4}
5) [value:12, weight:3] {2, 5}
6) [value:0, weight:0] {2, 6}
7) [value:37, weight:9] {3, 7}
8) [value:22, weight:5] {3, 8}
9) [value:25, weight:6] {3, 9}
10) [value:10, weight:2] {3, 10}
11) [value:27, weight:7] {3, 11}
12) [value:12, weight:3] {3, 12}
13) [value:15, weight:4] {3, 13}
14) [value:0, weight:0] {3, 14}
15) [value:57, weight:14] {4, 15}
16) [value:37, weight:9] {4, 16}
17) [value:42, weight:10] {4, 17}
18) [value:22, weight:5] {4, 18}
19) [value:45, weight:11] {4, 19}
20) [value:25, weight:6] {4, 20}
21) [value:30, weight:7] {4, 21}
22) [value:10, weight:2] {4, 22}
23) [value:47, weight:12] {4, 23}
24) [value:27, weight:7] {4, 24}
25) [value:32, weight:8] {4, 25}
26) [value:12, weight:3] {4, 26}
27) [value:35, weight:9] {4, 27}
28) [value:15, weight:4] {4, 28}
29) [value:20, weight:5] {4, 29}
30) [value:0, weight:0] {4, 30}

이진 트리 구성 완료 !

<1> : [value:0, weight:0]
<2> : [value:10, weight:2]
최대 가치 갱신 !

-----------------------------------

<3> : [value:22, weight:5]
최대 가치 갱신 !
```

```
<4> 무게 초과이므로 되추적하여 탐색합니다.
<5> : [value:22, weight:5]
<6> 무게 초과이므로 되추적하여 탐색합니다.
<7> : [value:22, weight:5]
<8> : [value:10, weight:2]
<9> : [value:25, weight:6]
최대 가치 갱신 !

-----------------------------------

<10> 무게 초과이므로 되추적하여 탐색합니다.
<11> : [value:25, weight:6]
<12> : [value:10, weight:2]
<13> : [value:30, weight:7]
최대 가치 갱신 !

-----------------------------------

<14> : [value:10, weight:2]
<15> : [value:0, weight:0]
<16> : [value:12, weight:3]
<17> : [value:27, weight:7]
<18> 무게 초과이므로 되추적하여 탐색합니다.
<19> : [value:27, weight:7]
<20> : [value:12, weight:3]
<21> : [value:32, weight:8]
최대 가치 갱신 !

-----------------------------------

<22> : [value:12, weight:3]
<23> : [value:0, weight:0]
<24> : [value:15, weight:4]
<25> 무게 초과이므로 되추적하여 탐색합니다.
<26> : [value:15, weight:4]
<27> : [value:0, weight:0]
<28> : [value:20, weight:5]
<29> : [value:0, weight:0]
최대 가치: 32, 무게 : 8
```

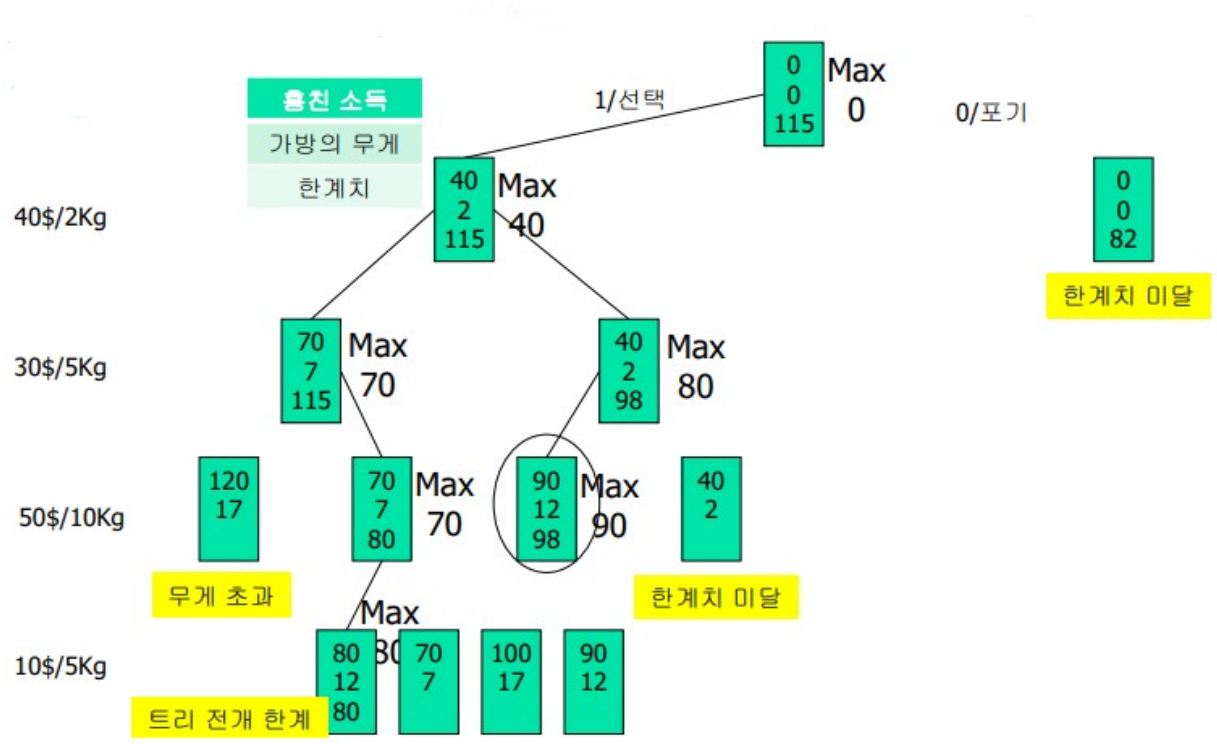
7) 분기 한정 가지치기 (Branch and Bound) 알고리즘

분기 한정법 (Branch and Bound)은 가능한 모든 해를 검사하지 않고도 최적화 할 수 있는 효율적인 방법을 제공하는 알고리즘이다.

분기 한정법에서는 분기를 한정시켜 쓸데없는 시간 낭비를 줄이게 된다.
즉, 이 기법은 되추적 기법 (Backtracking)에서 부가적으로 발생한 알고리즘이다. 되추적이 허용되는 탐색에서, 더 이상 탐색할 필요가 없는 지점을 판단하여 잘라내는 것이다. 이를 나무에서 필요없는 가지를 잘라내는 가지치기와 유사하다고 보아 가지치기 기법이라고 부르기도 한다.

분기 한정 가지치기에서는, 각 노드를 방문할 때마다 그 노드가 유망한지의 여부를 결정하기 위하여 한계치(Bound)를 계산하게 된다. 한계치는 그 노드로부터 가지를 뻗어나가 얻을 수 있는 해답의 한계를 나타낸다. 이 때, 한계치가 이전까지 찾은 최고의 해답보다 더 좋다면 그 마디를 유망하다고 보게 된다. 반대로 한계치가 기존의 최고 해답보다 좋지 않다면 더 이상 가지를 뻗어 검색할 필요가 없으므로 그 노드는 유망하지 않다고 보고, 해당 서브트리를 가지치기 하게 된다.

Bound를 계산한다는 점에서 얼핏 보면 되추적 기법과 같아 보일수도 있겠으나, 차이가 존재한다. 되추적 기법은 가보고 진행이 되지 않을 시 돌아오는 기법이고, 분기 한정법은 최적해를 찾을 가능성을 판단하여 없으면 아예 분기를 하지 않는 기법이다.



사진은 분기 한정 가지치기의 예시이다. 루트 노드에서 시작하고, 탐색을 시작하면 노드의 Bound를 계산한 후 Bound가 최고의 해답에 미치지 않는다면 가지를 뻗지 않는다. 다시 말해 가지를 친 경로는 이후 탐색하지 않게 된다.

7-1) 분기 한정 가지치기 알고리즘 구현 소스 코드

```
#include <stdio.h>
#include <stdlib.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
  int valuePerWeight;
};

struct JewelArray {
  int value;
  int weight;
  int bound;
};

struct Node {
  struct JewelArray data;
  struct Node* next;
};

struct Queue {
  struct Node* front;
  struct Node* rear;
  int count;
};

void init(struct Queue* queue) {
  queue->front = queue->rear = NULL;
  queue->count = 0;
}

int isEmpty(struct Queue* queue) {
  return queue->count == 0;
}

void enqueue(struct Queue* queue, struct JewelArray data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = NULL;

  if (isEmpty(queue)) {
    queue->front = newNode;
  }
  else {
    queue->rear->next = newNode;
  }
  queue->rear = newNode;
  queue->count++;
}

struct JewelArray dequeue(struct Queue* queue) {
  struct JewelArray data;
  struct Node* ptr;
  if (isEmpty(queue)) {
    return JewelArray{ 0, 0, 0 };
  }
  ptr = queue->front;
  data = ptr->data;
  queue->front = ptr->next;
  free(ptr);
```

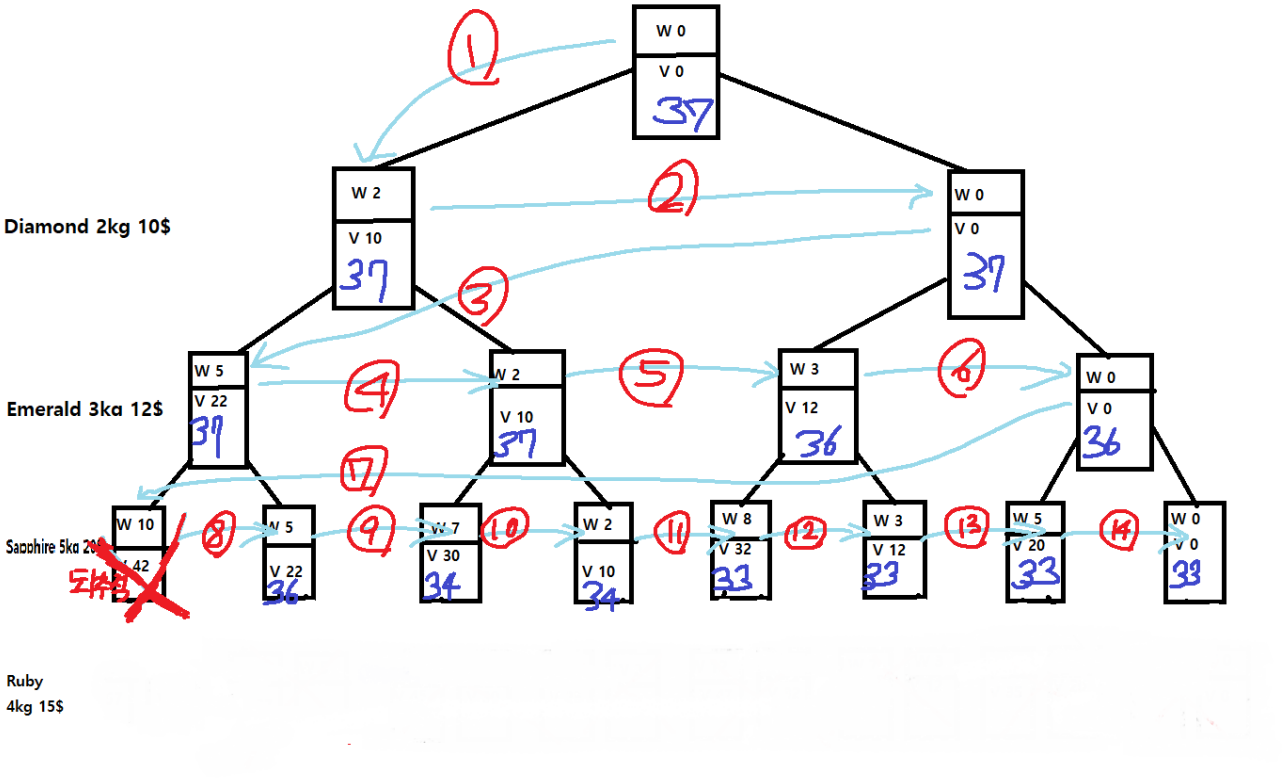
```
  queue->count--;
  return data;
}

struct JewelArray showQueue(struct Queue* queue, int i) {
  int maxweight = 8;
  struct JewelArray data;
  struct Node* ptr;
  if (isEmpty(queue)) {
    return JewelArray{ 0, 0, 0 };
  }
  ptr = queue->front;

  if (ptr->data.weight > maxweight) {
    printf("₩n<%d> 무게 초과 ! 자식 노드는 탐색하지 않습니다.₩n", i);
  }
  else {
    printf("₩n%d번째 연산 후 큐 상태₩n", i + 1);
    data = ptr->data;
    printf("[Value : %d, Weight : %d, Bound : %d] ₩n", data.value, data.weight, data.bound);
  }

  while (ptr != NULL) {
    ptr = ptr->next;
  }
  printf("₩n");
  return data;
}

int calculateBound(struct JewelArray* currentNode, struct JewelArray* JewelArray, int maxWeight, int
JewelArraySize, int index) {
  int totweight = currentNode->weight;
  int bound = currentNode->value;
  int j = index + 1;

  while (j < JewelArraySize && totweight + JewelArray[j].weight <= maxWeight) {
    totweight += JewelArray[j].weight;
    bound += JewelArray[j].value;
    j++;
  }

  if (j < JewelArraySize) {
    bound += (maxWeight - totweight) * (JewelArray[j].value / (double)JewelArray[j].weight);
  }

  return bound;
}

int ValuePerWeight(struct Jewel* jewel) {
  jewel->valuePerWeight = jewel->value / jewel->weight;
  return jewel->valuePerWeight;
}

bool isOverMaxWeight(struct Jewel jewel) {
  return jewel.weight <= 8;
}

void sortByValuePerWeight(struct Jewel Jewels[], int size) {
  for (int i = 0; i < size - 1; i++) {
    int idx = i, max = (Jewels[i]).valuePerWeight;
    for (int j = i; j < size; j++) {
      if (max < (Jewels[j]).valuePerWeight) {
        max = (Jewels[j]).valuePerWeight;
```

```
            idx = j;
          }
        }
        struct Jewel* temp;
        temp = &Jewels[i];
        Jewels[i] = Jewels[idx];
        Jewels[idx] = *temp;
    }
}

void main() {
    Queue queue;
    init(&queue);

    struct Jewel Diamond = { "Diamond ", 2, 10 };
    struct Jewel Emerald = { "Emerald ", 3, 12 };
    struct Jewel Ruby = { "Ruby    ", 4, 15 };
    struct Jewel Sapphire = { "Sapphire", 5, 20 };
    struct Jewel Jewels[] = { Diamond, Emerald, Sapphire, Ruby };
    int size = sizeof(Jewels) / sizeof(Jewels[0]);
    struct JewelArray JewelArray[31] = { 0 };
    int JewelArraySize = sizeof(JewelArray) / sizeof(JewelArray[0]);

    int maxWeight = 8;
    int currentWeight = 0;
    int maxValue = 0;
    int index = 0;
    int LEVEL = 0;
    int j = 0;

    sortByValuePerWeight(Jewels, 4);


    for (int i = 0; i < 15; i++) {
        j = (i > 0) ? (i) : 0;
        JewelArray[i * 2 + 1].value = JewelArray[j].value + Jewels[LEVEL].value;
        JewelArray[i * 2 + 1].weight = JewelArray[j].weight + Jewels[LEVEL].weight;

        j = (i > 0) ? (i) : 0;
        JewelArray[i * 2 + 2].value = JewelArray[j].value;
        JewelArray[i * 2 + 2].weight = JewelArray[j].weight;

        if (i == 0 || i == 2 || i == 6 || i == 14) {
            LEVEL++;
        }
    }

    LEVEL = 0;

    int bound = 0;
    enqueue(&queue, JewelArray[0]);
    int maxprofit = 0;

    for (int i = 0; i < 31; i++) {
        printf("%d) [value:%d, weight:%d] {%d, %d}₩n", i, JewelArray[i].value, JewelArray[i].weight,
LEVEL, i);
        if (i == 0 || i == 2 || i == 6 || i == 14) {
            LEVEL++;
        }
    }
    printf("₩n이진 트리 구성 완료 !₩n₩n");

    while (!isEmpty(&queue)) {
```

```
    struct JewelArray currentNode = dequeue(&queue);
    bound = calculateBound(&currentNode, JewelArray, maxWeight, JewelArraySize, index);

    printf("₩n현재 MaxProfit : %d, Bound : %d₩n", maxprofit, bound);

    for (int i = 0; i < 31; i++) {
      JewelArray[i].bound = bound;
    }

    if (currentNode.value > maxprofit) {
      maxprofit = currentNode.value;
    }

    if (2 * index + 1 < JewelArraySize && JewelArray[index * 2 + 1].bound >= maxprofit) {
      enqueue(&queue, JewelArray[2 * index + 1]);
    }
    else (printf("₩n한계치 미달로 자식 노드는 탐색하지 않습니다.₩n"));

    if (2 * index + 2 < JewelArraySize && JewelArray[index * 2 + 2].bound >= maxprofit) {
      enqueue(&queue, JewelArray[2 * index + 2]);
    }
    else (printf("₩n한계치 미달로 자식 노드는 탐색하지 않습니다.₩n"));

    if ((currentNode.value > maxValue) && (currentNode.weight <= maxWeight)) {
      maxValue = currentNode.value;
      currentWeight = currentNode.weight;
    }

    showQueue(&queue, index);
    index++;
  }

  index = 0;

  printf("최대 가치) Value : %d, Weight : %d₩n", maxValue, currentWeight);
}
```

7-2) 예상 수행 결과

분기 한정 가지치기 알고리즘의 소스 코드를 실행하였을 시 동작할 예상도를 첨부합니다.
이 예상도를 기반으로 소스 코드를 작성하게 되었습니다.



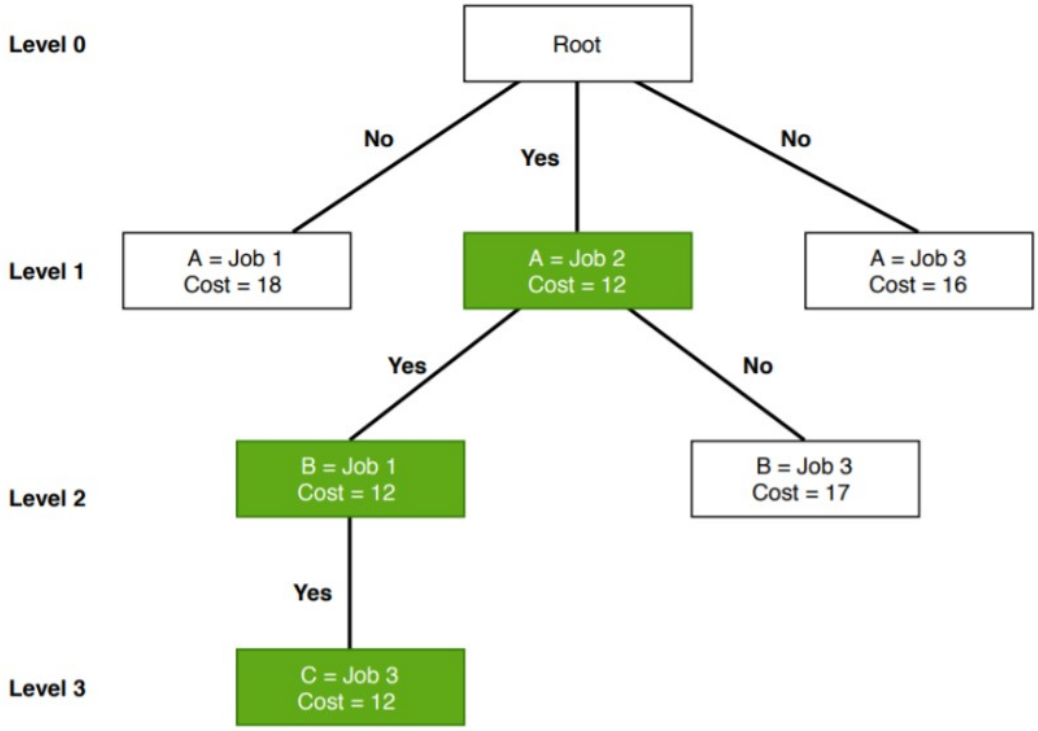
결과적으로, 11번째 과정을 거치며 얻은 최대값인 {W:8,V:32}을 저장해 두었다가 탐색을 마친 후 출력하게 될 것입니다.

7-3) 분기 한정 가지치기 알고리즘 실행 결과

```
0) [value:0, weight:0] {0, 0}
1) [value:10, weight:2] {1, 1}
2) [value:0, weight:0] {1, 2}
3) [value:22, weight:5] {2, 3}
4) [value:10, weight:2] {2, 4}
5) [value:12, weight:3] {2, 5}
6) [value:0, weight:0] {2, 6}
7) [value:42, weight:10] {3, 7}
8) [value:22, weight:5] {3, 8}
9) [value:30, weight:7] {3, 9}
10) [value:10, weight:2] {3, 10}
11) [value:32, weight:8] {3, 11}
12) [value:12, weight:3] {3, 12}
13) [value:20, weight:5] {3, 13}
14) [value:0, weight:0] {3, 14}
15) [value:57, weight:14] {4, 15}
16) [value:42, weight:10] {4, 16}
17) [value:37, weight:9] {4, 17}
18) [value:22, weight:5] {4, 18}
19) [value:45, weight:11] {4, 19}
20) [value:30, weight:7] {4, 20}
21) [value:25, weight:6] {4, 21}
22) [value:10, weight:2] {4, 22}
23) [value:47, weight:12] {4, 23}
24) [value:32, weight:8] {4, 24}
25) [value:27, weight:7] {4, 25}
26) [value:12, weight:3] {4, 26}
27) [value:35, weight:9] {4, 27}
28) [value:20, weight:5] {4, 28}
29) [value:15, weight:4] {4, 29}
30) [value:0, weight:0] {4, 30}

이진 트리 구성 완료 !

1번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 37]

2번째 연산 후 큐 상태
[Value : 0, Weight : 0, Bound : 37]

3번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 37]

4번째 연산 후 큐 상태
[Value : 22, Weight : 5, Bound : 37]

5번째 연산 후 큐 상태
[Value : 22, Weight : 5, Bound : 36]

<5> 무게 초과이므로 되추적하여 탐색합니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

9번째 연산 후 큐 상태
[Value : 12, Weight : 3, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

10번째 연산 후 큐 상태
[Value : 32, Weight : 8, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

11번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

12번째 연산 후 큐 상태
[Value : 30, Weight : 7, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.

13번째 연산 후 큐 상태
[Value : 0, Weight : 0, Bound : 0]

한계치 미달로 자식 노드는 탐색하지 않습니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.
최대 가치) Value : 32, Weight : 8
```

8) 분기 한정 가지치기 최고 우선 (Branch and Bound Best – First Search)

분기 한정 가지치기 최고 우선 알고리즘은, 주어진 마디의 모든 자식 마디를 검색하고, 유망하면서 확장되지 않은 마디를 살펴보고, 그 중에서 가장 좋은 한계치를 가진 마디를 확장하는 방식이다. 최고의 bound를 가진 마디를 우선적으로 검색하는 것이 최고 우선 알고리즘의 핵심이다.



출처 : https://www.baeldung.com/cs/branch-and-bound

즉 루트 노드에서 시작하여, 다음 노드의 유망성을 판단하고 유망하지 않다고 판단될 경우 가지를 쳐내고 유망하다고 판단되는 가지로 취사 선택하는 것을 반복하여 탐색하는 기법이다. 최적의 해를 구하기 위해서는 어차피 모든 해를 다 고려해 보아야 하나, 가치 평가 기준 중 최고로 전개하여 깊이 / 너비 순회 방식에 구애되지는 않는다.

8-1) 분기 한정 가지치기 최고 우선 알고리즘 구현 소스 코드

```
#include <stdio.h>
#include <stdlib.h>

struct Jewel {
  char name[20];
  int weight;
  int value;
  int valuePerWeight;
};

struct JewelArray {
  int value;
  int weight;
  int bound;
};

struct Node {
  struct JewelArray data;
  struct Node* next;
};

struct Queue {
  struct Node* front;
  struct Node* rear;
  int count;
};

void init(struct Queue* queue) {
  queue->front = NULL;
  queue->count = 0;
}

int isEmpty(struct Queue* queue) {
  return queue->count == 0;
}

void enqueue(struct Queue* queue, struct JewelArray data) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = NULL;

  if (isEmpty(queue) || data.bound > queue->front->data.bound) {
    newNode->next = queue->front;
    queue->front = newNode;
  }
  else {
    struct Node* current = queue->front;
    while (current->next != NULL && current->next->data.bound > data.bound) {
      current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
  }
  queue->count++;
}

struct JewelArray dequeue(struct Queue* queue) {
  struct JewelArray data;
  struct Node* ptr;
  if (isEmpty(queue)) {
    return JewelArray{ 0, 0, 0 };
```

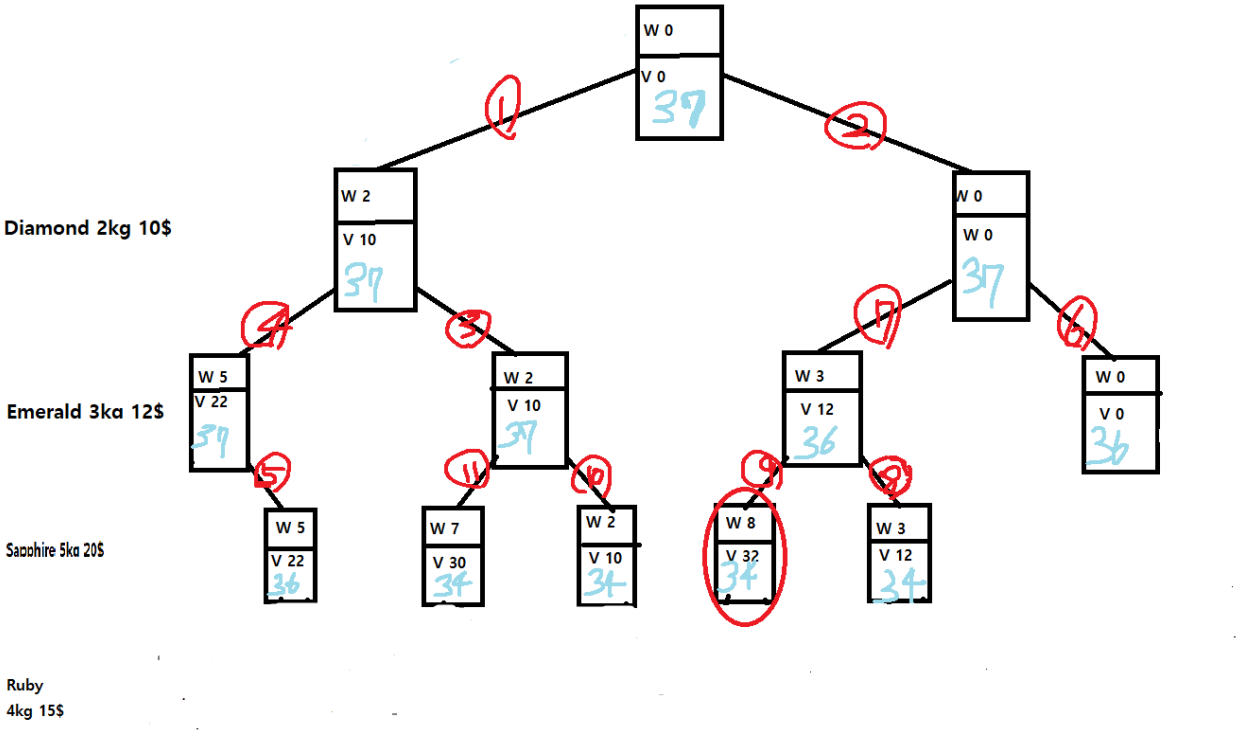
```
  }
  ptr = queue->front;
  data = ptr->data;
  queue->front = ptr->next;
  free(ptr);
  queue->count--;
  return data;
}

int compareNode(const void* a, const void* b) {
  return ((struct JewelArray*)b)->bound - ((struct JewelArray*)a)->bound;
}

struct JewelArray showQueue(struct Queue* queue, int i) {
  int maxweight = 8;
  struct JewelArray data;
  struct Node* ptr;
  if (isEmpty(queue)) {
    return JewelArray{ 0, 0, 0 };
  }
  ptr = queue->front;

  if (ptr->data.weight > maxweight) {
    printf("₩n<%d> 무게 초과이므로 되추적하여 탐색합니다.₩n", i);
  }
  else {
    printf("₩n%d번째 연산 후 큐 상태₩n", i + 1);
    data = ptr->data;
    printf("[Value : %d, Weight : %d, Bound : %d] ₩n", data.value, data.weight, data.bound);
  }

  while (ptr != NULL) {
    ptr = ptr->next;
  }
  printf("₩n");
  return data;
}

int calculateBound(struct JewelArray* currentNode, struct JewelArray* JewelArray, int maxWeight, int
JewelArraySize, int index) {
  int totweight = currentNode->weight;
  int bound = currentNode->value;
  int j = index + 1;

  while (j < JewelArraySize && totweight + JewelArray[j].weight <= maxWeight) {
    totweight += JewelArray[j].weight;
    bound += JewelArray[j].value;
    j++;
  }

  if (j < JewelArraySize) {
    bound += (maxWeight - totweight) * (JewelArray[j].value / (double)JewelArray[j].weight);
  }

  return bound;
}

int ValuePerWeight(struct Jewel* jewel) {
  jewel->valuePerWeight = jewel->value / jewel->weight;
  return jewel->valuePerWeight;
}

bool isOverMaxWeight(struct Jewel jewel) {
```

```
  return jewel.weight <= 8;
}

void sortByValuePerWeight(struct Jewel Jewels[], int size) {
  for (int i = 0; i < size - 1; i++) {
    int idx = i, max = (Jewels[i]).valuePerWeight;
    for (int j = i; j < size; j++) {
      if (max < (Jewels[j]).valuePerWeight) {
        max = (Jewels[j]).valuePerWeight;
        idx = j;
      }
    }
    struct Jewel* temp;
    temp = &Jewels[i];
    Jewels[i] = Jewels[idx];
    Jewels[idx] = *temp;
  }
}

void main() {
  Queue queue;
  init(&queue);

  struct Jewel Diamond = { "Diamond ", 2, 10 };
  struct Jewel Emerald = { "Emerald ", 3, 12 };
  struct Jewel Ruby = { "Ruby    ", 4, 15 };
  struct Jewel Sapphire = { "Sapphire", 5, 20 };
  struct Jewel Jewels[] = { Diamond, Emerald, Sapphire, Ruby };
  int size = sizeof(Jewels) / sizeof(Jewels[0]);
  struct JewelArray JewelArray[31] = { 0 };
  int JewelArraySize = sizeof(JewelArray) / sizeof(JewelArray[0]);

  int maxWeight = 8;
  int currentWeight = 0;
  int maxValue = 0;
  int index = 0;
  int LEVEL = 0;
  int j = 0;

  for (int i = 0; i < 15; i++) {
    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 1].value = JewelArray[j].value + Jewels[LEVEL].value;
    JewelArray[i * 2 + 1].weight = JewelArray[j].weight + Jewels[LEVEL].weight;

    j = (i > 0) ? (i) : 0;
    JewelArray[i * 2 + 2].value = JewelArray[j].value;
    JewelArray[i * 2 + 2].weight = JewelArray[j].weight;

    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
    }
  }

  LEVEL = 0;
  int bound = 0;
  enqueue(&queue, JewelArray[0]);
  int maxprofit = 0;

  for (int i = 0; i < 31; i++) {
    printf("%d) [value:%d, weight:%d] {%d, %d}₩n", i, JewelArray[i].value, JewelArray[i].weight,
LEVEL, i);
    if (i == 0 || i == 2 || i == 6 || i == 14) {
      LEVEL++;
```

```
    }
  }
  printf("₩n이진 트리 구성 완료 !₩n₩n");

  enqueue(&queue, JewelArray[0]);

  while (!isEmpty(&queue)) {
    struct JewelArray currentNode = dequeue(&queue);
    int bound = calculateBound(&currentNode, JewelArray, maxWeight, JewelArraySize, index);

    for (int i = 0; i < 31; i++) {
      JewelArray[i].bound = bound;
    }

    if (currentNode.value > maxprofit) {
      maxprofit = currentNode.value;
    }

    if (2 * index + 1 < JewelArraySize && JewelArray[index * 2 + 1].bound >= maxprofit) {
      enqueue(&queue, JewelArray[2 * index + 1]);
    }
    else (printf("₩n한계치 미달로 자식 노드는 탐색하지 않습니다.₩n"));

    if (2 * index + 2 < JewelArraySize && JewelArray[index * 2 + 2].bound >= maxprofit) {
      enqueue(&queue, JewelArray[2 * index + 2]);
    }
    else (printf("₩n한계치 미달로 자식 노드는 탐색하지 않습니다.₩n"));

    if ((currentNode.value > maxValue) && (currentNode.weight <= maxWeight)) {
      maxValue = currentNode.value;
      currentWeight = currentNode.weight;
    }
    showQueue(&queue, index);
    index++;
  }

  index = 0;

  printf("최대 가치) Value : %d, Weight : %d₩n", maxValue, currentWeight);
}
```

8-2) 예상 수행 결과

분기 한정 가지치기 최고 우선 기법의 실행 과정 예상도입니다.



이 예상도를 기반으로 소스 코드를 작성하게 되었습니다.

결과적으로, 9번째 과정을 거치며 얻은 최대값인 {W:8,V:32}을 저장해 두었다가 탐색을 마친 후 출력하게 될 것입니다.

8-3) 분기 한정 가지치기 최고 우선 실행 결과

```
0) [value:0, weight:0] {0, 0}
1) [value:10, weight:2] {1, 1}
2) [value:0, weight:0] {1, 2}
3) [value:22, weight:5] {2, 3}
4) [value:10, weight:2] {2, 4}
5) [value:12, weight:3] {2, 5}
6) [value:0, weight:0] {2, 6}
7) [value:42, weight:10] {3, 7}
8) [value:22, weight:5] {3, 8}
9) [value:30, weight:7] {3, 9}
10) [value:10, weight:2] {3, 10}
11) [value:32, weight:8] {3, 11}
12) [value:12, weight:3] {3, 12}
13) [value:20, weight:5] {3, 13}
14) [value:0, weight:0] {3, 14}
15) [value:57, weight:14] {4, 15}
16) [value:42, weight:10] {4, 16}
17) [value:37, weight:9] {4, 17}
18) [value:22, weight:5] {4, 18}
19) [value:45, weight:11] {4, 19}
20) [value:30, weight:7] {4, 20}
21) [value:25, weight:6] {4, 21}
22) [value:10, weight:2] {4, 22}
23) [value:47, weight:12] {4, 23}
24) [value:32, weight:8] {4, 24}
25) [value:27, weight:7] {4, 25}
26) [value:12, weight:3] {4, 26}
27) [value:35, weight:9] {4, 27}
28) [value:20, weight:5] {4, 28}
29) [value:15, weight:4] {4, 29}
30) [value:0, weight:0] {4, 30}

이진 트리 구성 완료 !

1번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 37]

2번째 연산 후 큐 상태
[Value : 0, Weight : 0, Bound : 37]

3번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 37]

4번째 연산 후 큐 상태
[Value : 22, Weight : 5, Bound : 37]

5번째 연산 후 큐 상태
[Value : 22, Weight : 5, Bound : 36]

<5> 무게 초과이므로 되추적하여 탐색합니다.

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

8번째 연산 후 큐 상태
[Value : 12, Weight : 3, Bound : 36]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

9번째 연산 후 큐 상태
[Value : 12, Weight : 3, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

10번째 연산 후 큐 상태
[Value : 32, Weight : 8, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

11번째 연산 후 큐 상태
[Value : 10, Weight : 2, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

12번째 연산 후 큐 상태
[Value : 30, Weight : 7, Bound : 34]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.

13번째 연산 후 큐 상태
[Value : 0, Weight : 0, Bound : 0]

한계치 미달로 자식 노드는 탐색하지 않습니다.
한계치 미달로 자식 노드는 탐색하지 않습니다.
최대 가치) Value : 32, Weight : 8
```

알고리즘 비교 평가 및 분석

수행 속도를 비교했을 때,
욕심쟁이 알고리즘 : 평균 21ms
동적 프로그래밍 : 평균 1ms
너비 우선 탐색 : 평균 53ms
깊이 우선 탐색 : 평균 24ms
되추적 기법 : 평균 21ms
분기 한정 가지치기 : 평균 43ms
분기 한정 가지치기 최고 우선 : 평균 34ms
라는 결과를 얻을 수 있었습니다.

이 중 제가 작성한 욕심쟁이 알고리즘, 동적 프로그래밍 기법을 제외한 모든 소스코드들은 트리를 구성하는 과정을 한 번 거친 이후 해당 트리를 대상으로 탐색을 시작하도록 구현되었습니다. 자료구조를 구성하는 능력이 미흡했기 때문입니다. 시정하겠습니다.

그러나 다행히도 소스코드들에서 트리를 구성하는 과정은 평균 2ms로, 현재 트리의 크기가 크지 않아 소요 시간이 적은 것으로 판단됩니다. 따라서 본 보고서 한정으로, 알고리즘의 성능을 평가하는 데에는 크게 지장이 없을 것입니다.

문제에 해결하는 데에 동적 프로그래밍이 압도적 1위로, 그 다음이 되추적 기법, 욕심쟁이 알고리즘, 깊이 우선 탐색 등의 순서로 빠른 수행 시간을 보이고 있습니다.

1, **욕심쟁이 알고리즘**의 시간복잡도는 O(size^2) 정도로 추정됩니다. 주요 함수인 SortByValuePerWeight 함수에서 선택 정렬을 사용하고 있기 때문입니다. 이외 ValuePerWeight 함수는 O(size), Greedy 함수는 O(size)의 시간복잡도를 가짐으로서 전체적인 시간복잡도는 O(size^2)라고 할 수 있겠습니다.

2. **동적 프로그래밍**의 시간복잡도는 O(size * maxWeight)입니다. 외부 반복문이 size번, 내부 반복문이 maxWeight 번 실행되기 때문입니다. 즉, 상수 시간 복잡도이므로 알고리즘 들 중 압도적으로 빠를 수 밖에 없었습니다.

3. **너비 우선 탐색**의 시간복잡도는 O(2^(size/2)) 정도가 될 것입니다. 지수 시간 복잡도이므로 입력의 크기가 커질수록 계산에 많은 시간이 소요될 것으로 추정됩니다. 따라서 알고리즘의 효율성을 개선하기 위하여 다른 방법을 고안해 봐야 할 것 같습니다.

4. **깊이 우선 탐색**의 시간 복잡도 또한 O(2^(size/2)) 정도로 추정됩니다. 너비 우선 탐색과 마찬가지로 지수 시간 복잡도이며, 입력의 크기가 커질수록 계산에 많은 시간이 걸릴 수 있을 것입니다. 큰 입력에 대해서는 다른 알고리즘을 고안해 보는 것이 좋을 것 같다고 판단됩니다.

5. **되추적 기법**은 DFS를 기반으로 만들었기에, 똑같이 O(2^(size/2))라는 지수 시간 복잡도를 가질 것입니다. 그러나 되추적 기법은 무게 초과시 서브 트리를 무시함으로써 서브 트리에 방문하는 시간을 절약하게 되어 같은 시간 복잡도를 가지는 다른 알고리즘보다 평균적으로 빠른 수행 속도를 보이고 있습니다.

6. **분기 한정 가지치기**는 일반적으로 가지치기를 통한 효율성 증가가 있지만, 최악의 경우에는 여전히 모든 경우의 수를 고려하게 됩니다. 노드의 개수를 n이라 하고, 각 노드에서의 결정이 2개(선택 혹은 미선택)이라고 가정했을 때, 트리의 높이가 최악의 경우 n이 될 수 있으므로 이 경우 시간 복잡도는 O(2^n)입니다. 이 또한 지수 시간 복잡도로, 만약 배낭의 크기나 보석의 개수가 증가하게 된다면 계산 시간이 기하급수적으로 늘어날 것으로 보입니다.

7. **분기 한정 가지치기 최고 우선 기법**도 일반적으로 가지치기를 통해 효율성을 증가시키나, 최악의 경우에는 분기 한정 가지치기와 같은 지수 시간 복잡도를 보일 것입니다. 따라서 보석의 개수가 증가하거나, 배낭의 크기가 증가한다면 다른 알고리즘을 고려하는 것이 좋을 것 같다고 판단됩니다.

결론적으로, 본 Knapsack 문제를 해결하는 데에 있어서 빠른 수행 속도와 최적해를 모두 보장한 알고리즘은 **동적 프로그래밍과 되추적 기법 두 개 뿐**이었습니다. 또한 이 알고리즘들은 최악의 경우에도, 다른 알고리즘의 최악의 경우보다 못한 시간 복잡도를 가지지는 않으니, 평균적으로 이 두 알고리즘을 사용하는 것이 원하는 결과를 빠른 시간 내에 얻기에 가장 좋을 것 같다고 생각합니다.

결과 및 검토 (과제 구성 과정 에피소드)

결과적으로 모든 알고리즘을 구성해내는 것은 성공했으나, 보고서 작성을 마치는 지금 한 번 더 확인하니 부족해 보이는 부분이 너무나 많습니다. 특히 이진 트리를 미리 구현해놓고 탐색하게 하는 부분이 가장 아쉽습니다. 초기에 판단을 잘못하여, 잘못된 방식을 채택한 채로 시작한 것이 원인인 듯 합니다. 그래도, 자료구조 강의가 아닌 알고리즘 강의이기에 알고리즘의 성능을 판단하는 데에는 현재로서 큰 지장이 없었으므로 다행이라고 생각됩니다. 이번 보고서를 작성하면서, 그림판을 매우 많이 활용하게 되었는데 가시적으로 조금 떨어지는 점 죄송합니다.

과제의 난이도에 대한 분석

저의 소스 코드 구현 실력이 매우 부족하다고 느끼게끔 하는 과제였습니다. 많은 알고리즘들을 직접 만들어 보고자 하니 알고리즘에 대해서 한번 더 공부하는 과정을 거쳐야 했고, 그것을 컴퓨터 상에서 구현하고자 하려니 정말 수도 없이 많은 벽을 만나게 되었습니다. 솔직하게, 정말 어려웠습니다. 어렵게 과제를 마치고 난 입장으로서, 다른 학생들은 어떻게 느꼈는지도 매우 궁금해집니다. 하지만 언제나 그래왔듯이, 정말 많은 실력의 상승이 있었습니다. 이렇게 과제로 부여받지 않았다면 하다가 포기하고, 또 조금 해보다가 포기하고, 결과적으로 포기했을 것 같은데, 이런 식으로 동기부여를 받아 힘들더라도 결국 해낸 자신이 자랑스럽습니다. 지식, 구현 실력 모두 향상시키기에 매우 좋은 과제였습니다.

알게 된 점

가장 신기했던 점은 동적 프로그래밍이었습니다. 교수님께서 강의 당시 피보나치 수열, 이항계수 등 여러 예시를 통해 동적 프로그래밍의 장점을 설명해 주셨으나, 실제로 그런 엄청난 차이가 있을 지 실감이 나지는 않았습니다. 그런데 모든 소스 코드의 작성을 마치고, 알고리즘들의 비교 분석을 위해 clock 함수를 사용하여 수행 시간을 측정하게 되었을 때, 한 자리 수의 시간이 나온 것은 동적 프로그래밍이 유일하였습니다. 솔직히, 정말 놀랐습니다. CPU나 메모리 등의 컴퓨터 자원은 유한하므로, 이를 최대한 적게 사용하며 효율적으로 가동되는 코드를 짜야 한다는 말은 많이 듣고 봐왔지만, 실감이 나지 않는 상태였는데 동적 프로그래밍을 보고 나니 그 중요성을 확 체감할 수 있게 되었습니다. 즉, 알고리즘을 사용해서 프로그램을 구성해야 하는 이유를 알게 되었습니다.

기타 하고 싶은 말

이번 학기도 고생 많으셨습니다 교수님. 언제나 감사드립니다.
1학년 때부터 교수님을 뵈어 왔습니다. 당시 저는 머리만 컸지 생각하는 것은 애나 다름없었고, 때문에 학과에서도, 교수님께도 많은 실수를 해 왔습니다. 군대를 제대하고, 2학년으로 복학하여, 2년이라는 시간이 지났음에도 저를 한눈에 알아봐 주신 교수님께 당시 너무나도 감사했으며, 다른 교수님들께도 현교가 많이 달라졌다, 좋아졌다 등 칭찬을 많이 해주셨다는 말을 들으며 언제나 감사한 마음을 가지고 있었습니다. 가능하면 교수님이 내년까지 남아 주셔서, 제 성장을 끝까지 봐 주시고, 어엿한 사회인으로 거듭나는 것을 봐주셨으면 하는 바램이 있습니다.
교수님께서 지나가는 말씀으로, 이번 학기가 마지막이 될 수도 있다고 하셨는데, 정말 이번 학기 시험이 끝나면 더 뵙지 못하게 될 까봐 한번 더 감사의 말씀 드립니다. 교수님, 사랑합니다, 감사했습니다. 언제나, 어디에서도 행복하시기를 진심으로 기도합니다.

제자 송현교 올림.