

[알고리즘 설계와 분석]

2023 학년도 1 학기 – HW3



학번	20190785
이름	박수빈
과목코드	CSE3081
분반	01
담당 교수님	임인성 교수님

1. Minimal triangulation problem

1) 구현 방법

- 이론

Dynamic programming 기법을 이용하여 다각형의 minimal triangulation 을 이루는 점의 index 를 찾고, triangulation 과정을 거친 이후 총 합도 구하는 것이 이 문제의 요지이다.

다각형을 이루는 점의 총 개수를 변수 s 로, 다각형의 시작점의 index 를 변수 i 로 설정하였다. S_{is} 는 index 가 i 인 점에서 시작하며 점의 총 개수가 s 인 다각형을 의미한다. 그리고 이 S_{is} 다각형에서 minimal triangulation 을 이루는 점의 index 를 변수 k 로 설정하였다. 그리고 이 S_{is} 의 삼각화의 최소 비용을 C_{is} 라고 설정하면, C_{is} 를 구하는 공식은 아래와 같다.

i) $0 \leq s < 4$ 인 경우, 0 (공식 a)

ii) $s \geq 4$ 인 경우, $\min_{1 \leq k \leq s-2} [C_{i,k+1} + C_{i+k, s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})]$ (공식 b)

이때, $D(p,q)$ 는 점 p 와 q 가 서로 인접한 경우에는 0 이고, 그렇지 않은 경우에는 두 점 사이의 거리를 나타낸다. 위의 공식을 기반으로 프로그램을 구현하였다. 또한, 혼란을 줄이기 위하여 앞서 언급한 변수들은 후술할 코드 내용에서도 동일한 의미를 지니도록 작성하였다. 이 문제에서 구해야 하는 것은 크게 3 가지이고, 각각 어떻게 구할 것인지에 대해 뒤의 1-1)번, 1-2)번, 1-3)번에 자세히 서술하였다.

- 구현을 위해 작성한 구조체

본격적으로 구현 방법에 대해 서술하기에 앞서, 구현을 위해 사용된 구조체에 대한 설명을 아래에 먼저 덧붙였다.

```
typedef struct {  
    double x; //x좌표  
    double y; //y좌표  
}Vertex;  
  
typedef struct {  
    int idx;  
    double value;  
}Minimum;  
  
typedef struct {  
    int from;  
    int to;  
}Vector;
```

- Vertex

input file 에 저장되어 있는 점들의 정보를 저장하는 구조체이다. 멤버 변수 x 에는 해당 점의 x 좌표를, 멤버 변수 y 에는 해당 점의 y 좌표를 나타낸다.

- Minimum

이후에 find_min 함수에서 double 형 배열을 인자로 받아서 해당 배열의 원소 중 최소값인 원소의 index 와 그 값을 반환할 때 쓰이는 구조체이다. 멤버 변수 idx 는 해당 원소의 idx 를, 멤버 변수 value 는 '배열[idx]'의 값을 나타낸다.

- Vector

다각형 S_{is} 의 삼각화의 비용을 최소화시키는 현들의 양 끝점을 저장하는 구조체이다. 예를 들어, Vector.from = 0 이고, Vector.to = 2 라면, 0 번째 점과 2 번째 점을 이은 현이 S_{is} 의 삼각화를 이루는 현들에 포함된다는 의미이다.

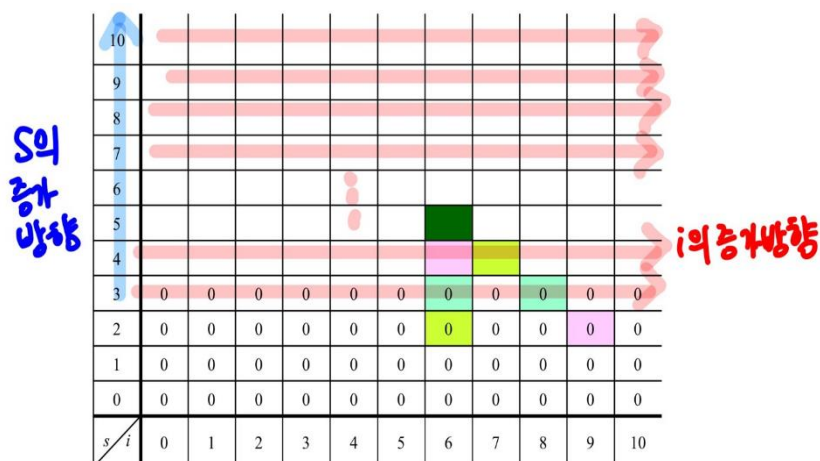
이제부터 이 문제에서 구해야 하는 3 가지를 구한 방법에 대한 본격적인 서술에 해당한다.

1-1) C_{is}

해당 값을 구한다면, value_table 이라는 20*20 크기의 double 형 2 차원 배열에 저장하였다. 해당 table 의 행을 s 로, 열을 i 로 설정하였다. 즉, C_{58} 의 정보는 value_table[8][5]에 저장하도록 구현하였다. 이 value_table 을 채울 때, '공식 a'와 '공식 b'에 의거하여 채우면 된다. 우선 공식 a 를 반영하여, value_table 에서 0 번째부터 3 번째 행까지 0 의 값을 갖도록 하였다.

```
//1. 기초값부터 채우기 (선언 부분에서 이미 초기화했지만 문맥적 이해를 위해 추가)
for (s = 0; s <= 3; s++) {
    for (i = 0; i < MAX_VERTEX_NUM; i++) {
        value_table[s][i] = 0.0;
        idx_table[s][i] = 0;
    }
}
```

그 다음부터는 '공식 b'를 따라 C_{is} 의 값을 구해야 하므로, value_table 에서 아래에 첨부한 그림과 같은 방향으로 변수 i와 s 를 증가시키면서 차례대로 value_table 의 데이터를 채우면



된다. 이러한 방향으로 value_table 을 채우면 모든 범위의 i 와 s 의 C_{is} 를, 즉 value_table[s][i]를 구할 수 있다. (해당 그림은 Minimal triangulation 보충자료로 나눠주신 강의자료 pdf 에

제가 아이패드로 덧그린 그림입니다.)

$\min_{1 \leq k \leq s-2} [C_{i,k+1} + C_{i+k, s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})]$ (공식 b)

value_table[s][i]에 넣을 값을 계산하는 방법은 위의 공식 b를 그대로 따르면 된다. 우선 k는 1 부터 s-2 까지 for loop 을 돌며 ' $C_{i,k+1} + C_{i+k, s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})$ ' 값을 계산한다. 그리고 이때 계산한 값을 double 형 1 차원 배열 compare 에 저장한다.

```
for (s = 4; s <= vertex_num; s++) {
    for (i = 0; i <= vertex_num; i++) {

        //배열 compare에는 가능한 cost 후보들이 모두 들어감
        double* compare = (double*)malloc(sizeof(double) * (s - 1));
        if (!compare) {
            fprintf(stdout, "The memory allocation is failed.\n");
            exit(1);
        }

        compare[0] = (double)(s - 2); //0번째 원소에는 비교할 원소의 개수 저장
        //배열 compare 모든 원소 -1.0으로 초기화
        for (int x = 1; x <= (s - 2); x++) {
            compare[x] = -1.0;
        }

        //배열 compare 값 채우는 for loop
        for (k = 1; k <= (s - 2); k++) {

            //D1 값 결정하기
            if (k == 1) D1 = 0;
            else D1 = cal_distance(vertex[i], vertex[i + k]);

            //D2 값 결정하기
            if ((k - s + 1) == 1 || (k - s + 1) == -1) D2 = 0;
            else D2 = cal_distance(vertex[i + k], vertex[i + s - 1]);

            //배열 compare 채우기
            compare[k] = value_table[k + 1][i] + value_table[s - k][i + k] + D1 + D2;
        }
    }
}
```

인접하지 않은 두 점 사이의 거리를 구하는 D(p,q)의 값은 아래와 같은 cal_distance 라는 함수를 추가적으로 작성하여 그를 통해 구하였다.

```
double cal_distance(Vertex v1, Vertex v2) {
    //두 점 사이의 거리를 구하는 함수이다.

    double x_square = (v1.x - v2.x) * (v1.x - v2.x);
    double y_square = (v1.y - v2.y) * (v1.y - v2.y);

    double result = sqrt(x_square + y_square);
    return result;
}
```

```

Minimum find_min(double* compare) {
    Minimum minimum;
    int total = (int)compare[0];

    minimum.idx = 1;
    minimum.value = compare[1];

    for (int i = 2; i <= (total); i++) {
        if (compare[i] <= minimum.value) {
            minimum.idx = i;
            minimum.value = compare[i];
        }
    }

    return minimum;
}

```

compare 에 저장한 원소들 중 최소값에 해당하는 것이 value_table[s][i]의 값이므로, double 형 1 차원 배열을 인자로 받아서 배열의 원소 중 최소값을 찾는 find_min 함수를 왼쪽에 첨부한 사진과 같이 새로 작성하였다.

이렇게 find_min 함수를 통해 compare 배열 중 최소값을 가지는 원소의 index 는 minimum.idx 에, 그 원소가 가지는 값은 minimum.value 에 저장하도록 하였다. 이때

```

//compare 배열 원소 중 최소값 구하기
Minimum minimum = find_min(compare);

//앞서 구한 최소값으로 table 값 채우기
value_table[s][i] = minimum.value; //C(i,s)
idx_table[s][i] = minimum.idx; //그때의 k값

```

minimum.value 의 값이 우리가 찾고자 하는 value_table[s][i]에 해당하는 것이고, 이는 곧 C_{is} 를 의미한다. 그리고 minimum.idx 는 그때의 k 값을 의미한다. k 또한

Minimal triangulation problem 을 해결할 때 구해야 하는 것들 중 하나에 해당한다. 이에 대한 더 자세한 서술은 아래 1-2)번 항목에 하였다.

```

double total = value_table[vertex_num][0];

```

최종적으로, C_{is} 는 value_table[vertex_num][0]에 저장되고, 이를 변수 total 에 저장하였다. 이 total 값을 파일과 stdout 에 fprintf 로 출력되도록 하면 'chord 들의 길이의 합'을 구하는 것은 해결된다.

1-2) k

k를 구해 놓아야 1-3)번에서 후술할 것도 구할 수 있다. 이 k는 1-1) 과정에서 변수 i와 s가 각각 for loop 을 돌며 C_{is} 를 구할 때 동시에 같이 구하면 된다. 이때 구한 k 의 값은 idx_table 이라는 20*20 크기의 int 형 2 차원 배열에 저장하면 된다.

```
//compare 배열 원소 중 최소값 구하기
Minimum minimum = find_min(compare);

//앞서 구한 최소값으로 table 값 채우기
value_table[s][i] = minimum.value; //C(i,s)
idx_table[s][i] = minimum.idx; //그때의 k값
```

find_min 함수는 배열 compare 에 저장되어 있던 원소들 중 가장 작은 값을 가지는 원소를 구하여 Minimum 형 구조체에 저장하여 반환한다.

따라서, 위의 코드에서도 볼 수 있듯이, minimum.idx 는 compare 의 원소들 중 최소값을 가지는 원소의 index 를 가지고 있으므로, 이는 k 값에 해당하고 이를 idx_table[s][i]에 저장하면 되는 것이다.

```
//2. find_k 함수를 통해 찾은 현들의 순서상 차례대로 파일에 저장하기
Vector* k_result = (Vector*)malloc(sizeof(Vector) * (vertex_num - 3));
if (!k_result) {
    fprintf(stdout, "The memory allocation is failed.\n");
    exit(1);
}

for (int m = 0; m <= (vertex_num - 4); m++) {
    k_result[m].from = 0;
    k_result[m].to = 0;
}

int t = 0;
find_k(0, vertex_num, idx_table, k_result, &t, vertex_num - 3);
```

앞서 찾은 idx_table 을 기반으로 minimal triangulation 을 이루는 각 현들의 양 꼭짓점을 찾기 위해 find_k 함수를 이용한다. find_k 함수에 대해 구현하기에 앞서, Vector 형 1 차원 배열 k_result 를 먼저 생성한다.

find_k 함수는 점의 시작 index, 점의 개수, idx_table 을 기반으로 C_{is} 의 k 값을 재귀적으로 계산하는 함수이다. 그리고 그 계산 결과를 앞서 생성해놓은 k_result 에 저장한다. t 는 k_result 의 인덱스 역할을 하는 변수로, max_t 에 도달하면 k_result 배열을 다 채운 것이므로 재귀함수 find_k 를 멈추도록 설계하였다. find_k 함수의 자세한 코드 설명은 1-3)번 항목에 있다.

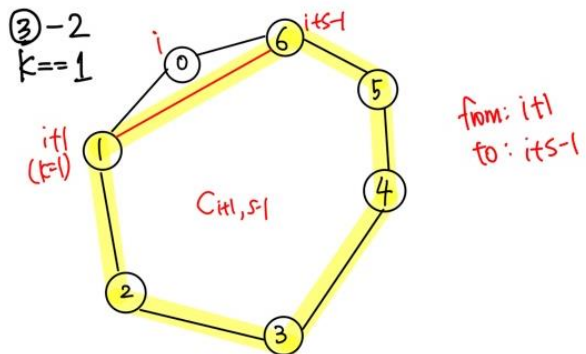
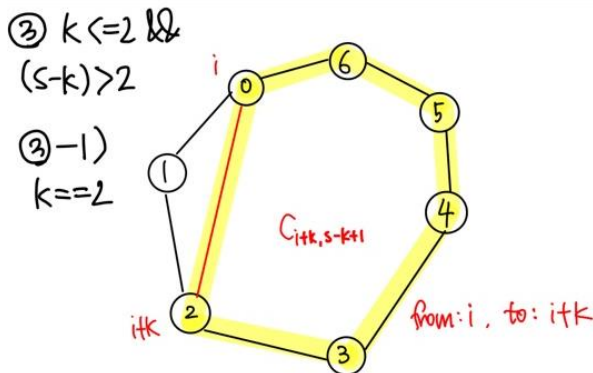
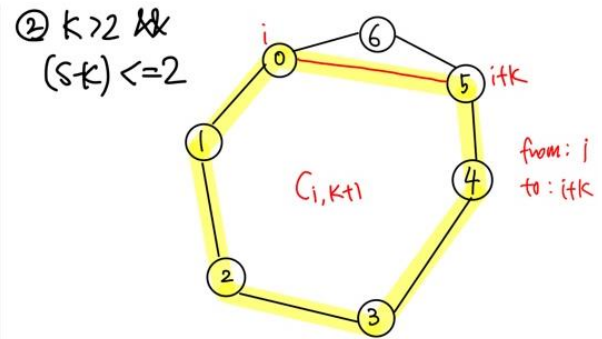
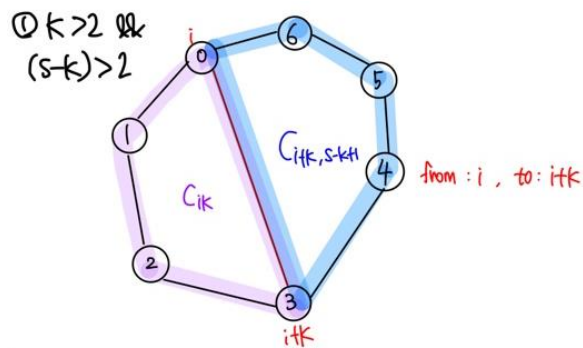
1-3) S_{is} 의 minimal triangulation 을 이루는 모든 현의 양 끝점

```
int k;  
  
if (*t >= max_t) return;  
  
k = idx_table[s][i];  
if (k <= 0) return;
```

앞 단락에서 언급하였듯이, t 는 k_result 의 인덱스 역할을 하는 정수형 포인터 변수이므로, max_t 에 도달하면 재귀함수가 종료되도록 하였다. 또한, k 가 0 이면 더 이상 다각형의 삼각화 과정이 이루어지지 않는다는 뜻이므로 k 가 0 보다 작거나 같은 경우에도

재귀함수가 종료되도록 설정하였다. s 가 5 이상인 경우, k 는 적어도 2 회는 뽑혀야 한다. 왜냐하면 오각형 이상의 다각형들은 k 를 한번 뽑아서 해당 다각형의 모든 내부를 삼각화할 수 없기 때문이다. 뽑힌 k 가 2 라면 삼각형이 만들어진 것이고, 남은 부분('s-k')에 대해 다시 모든 부분에 대해 k 가 2 가 되는 지점까지 $find_k$ 를 재귀적으로 호출하면 된다. 이 점을 이용하면 $find_k$ 재귀함수를 구현해낼 수 있다. 따라서 $find_k$ 는 크게는 총 4 가지, 자세히는 총 5 가지 경우로 나누어서 재귀함수의 인자를 달리 전달하고 k_result 에 저장하면 된다.

k 를 한번 뽑으면 다각형 내부는 $i \sim (i+k)$ 까지, 그리고 $(i+k) \sim (s-i-1)$ 까지 이렇게 두 부분으로 나뉜다. (참고로 $s-i-1$ 은 i 와 같은 말이다. 구별을 위해 $s-i-1$ 로 달리 표기한 것뿐이다.) 나뉜 두 부분 중 삼각형을 이룬 부분은 더 이상 재귀 호출을 하지 않고 끝나고, 삼각형을 이루지 못한 부분은 적절한 인자를 전달하여 재귀호출을 이어서 하면 된다. 이 재귀함수가 종료하는 시점은 앞 단락에서 이미 언급한 두 조건이 만족되는 시점이거나, k 선택 이후 나뉜 두 부분이 모두 2 보다 작거나 같아지는 시점이다. 이 점을 그림으로 표현하면 아래에 첨부한 사진과 같다. (참고로, 아래 그림은 제가 직접 아이패드로 그린 그림입니다.)



위의 그림에 없는 경우는 k 와 $s-k$ 둘 다 2 보다 작거나 같은 경우이다. 이 경우는 굳이 그림을 그리지 않더라도 이해가 어렵지 않은 경우라서 생략하였다. 위의 그림을 토대로, 코드를 작성하면 아래와 같다.

```

if (k > 2 && (s - k) > 2) {
    k_result[*t].from = i;
    k_result[*t].to = k + i;
    (*t)++;
    find_k(i, i + k, idx_table, k_result, t, max_t);
    find_k(i + k, s - k + 1, idx_table, k_result, t, max_t);
}

if (k <= 2 && (s - k) > 2) {
    if (k == 1) {
        k_result[*t].from = i + 1;
        k_result[*t].to = s - 1 + i;
        (*t)++;
        find_k(i + 1, s - 1, idx_table, k_result, t, max_t);
    }
    else {
        k_result[*t].from = i;
        k_result[*t].to = k + i;
        (*t)++;
        find_k(i + k, s - k + 1, idx_table, k_result, t, max_t);
    }
}

```

```

if (k > 2 && (s - k) <= 2) {
    k_result[*t].from = i;
    k_result[*t].to = i + k;
    (*t)++;
    find_k(i, k + 1, idx_table, k_result, t, max_t);
}

if (k <= 2 && (s - k) <= 2) {
    k_result[*t].from = i;
    k_result[*t].to = i + k;
    (*t)++;
    return;
}

```

이제 find_k 함수를 이용해 구한 k_result 배열을 정렬된 순서로 출력하는 단계만 남았다. 순서를 정렬하는 방법은 아래 코드와 같다.

```
Vector tmp;
for (int a = 0; a < vertex_num - 3; a++) {
    if (a < vertex_num - 4) {
        if (k_result[a].from > k_result[a + 1].from) {
            tmp = k_result[a];
            k_result[a] = k_result[a + 1];
            k_result[a + 1] = tmp;
        }

        if (k_result[a].from == k_result[a + 1].from) {
            if (k_result[a].to > k_result[a + 1].to) {
                tmp = k_result[a];
                k_result[a] = k_result[a + 1];
                k_result[a + 1] = tmp;
            }
        }
    }

    fprintf(stdout, "%d %d\n", k_result[a].from, k_result[a].to);
    fprintf(outFp, "%d %d\n", k_result[a].from, k_result[a].to);
}
```

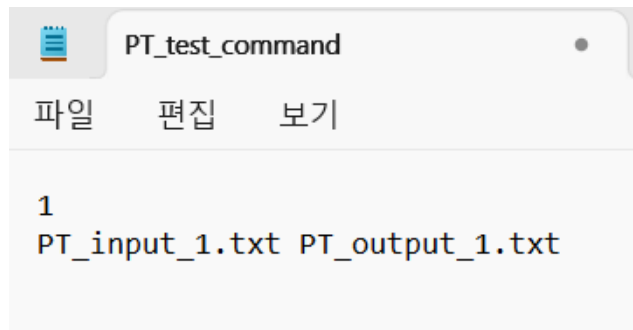
k_result[a]와 k_result[a+1] 중 어느 것을 먼저 출력할지는 from 멤버 변수를 보고 결정한다. from 변수의 값이 더 큰 것을 앞으로 당겨오고 작은 것을 뒤로 보내고, 만일 from의 값이 같다면 to의 값이 더 큰 것을 앞으로 당겨오고 작은 것을 뒤로 보낸다. 이렇게 k_result 배열을 정렬한 다음,

차례대로 0부터 vertex_num-2까지의 인덱스까지 k_result의 값을 출력하면 된다.

2) 구현에 실패한 부분

해당 문제에서는 구현에 실패한 부분은 없었다.

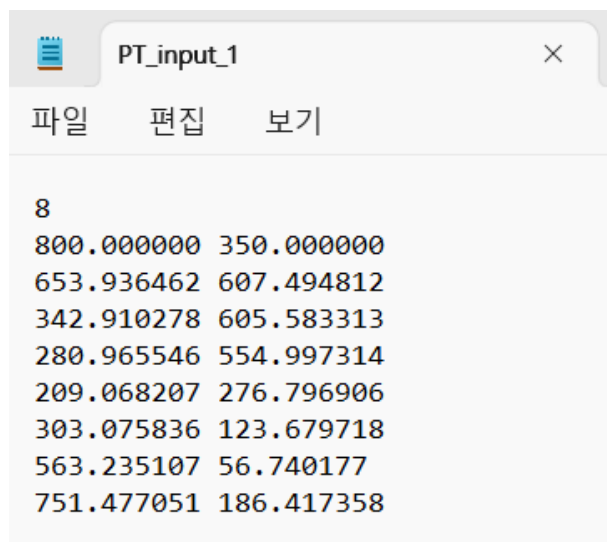
3) 실행 결과



A screenshot of a window titled "PT_test_command". It has a menu bar with "파일", "편집", and "보기". The main content area shows the number "1" followed by the text "PT_input_1.txt PT_output_1.txt".

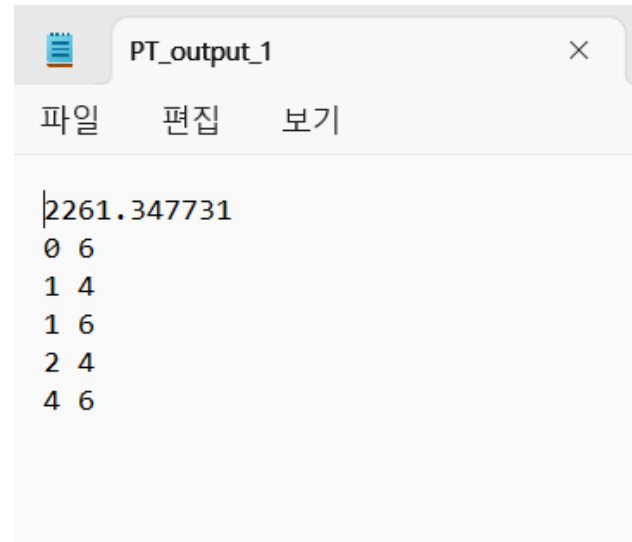
만일 PT_test_command.txt 파일의 내용이 왼쪽과 같다면, "PT_input_1.txt" 이름의 텍스트 파일을 열어서 점의 좌표 정보들을 저장하고, 그를 기반으로 계산한 'chord 들의 길이의 총 합'과 'minimal triangulation 을

이루는 모든 현의 양 끝점'을 "PT_output_1.txt" 이름의 텍스트 파일에 저장한다.



A screenshot of a window titled "PT_input_1". It has a menu bar with "파일", "편집", and "보기". The main content area shows the number "8" followed by two columns of numerical data:

800.000000	350.000000
653.936462	607.494812
342.910278	605.583313
280.965546	554.997314
209.068207	276.796906
303.075836	123.679718
563.235107	56.740177
751.477051	186.417358



A screenshot of a window titled "PT_output_1". It has a menu bar with "파일", "편집", and "보기". The main content area shows the result of the calculation:

```
|2261.347731
0 6
1 4
1 6
2 4
4 6
```

왼쪽과 같은 내용의 "PT_input_1.txt" 파일을 입력 값으로 넣는다면, 그 결과 값은 오른쪽과 같이 'chord 들의 길이의 총 합'이 첫 번째 줄에, 그 다음 정보들은 두 번째 줄부터 끝까지 "PT_output_1.txt" 파일에 저장된다.

2. Subset Sum problem

1) 구현 방법

-이론

이 문제 역시 Dynamic programming 기법을 이용하여 table 을 채운 다음 결과를 도출하는 방식으로 해결할 수 있다. 집합의 총 원소의 개수를 변수 num으로 설정하고 합이 L이 되는 subset 이 존재하는지 확인하는 과정을 거치는 식으로 진행하였다. 원소 값은 1 차원 정수형 배열 arr 에 입력되는 순서대로 arr[1]부터 arr[num]까지 저장하였다. 행의 개수가 (num+1), 열의 개수가 (L+1)인 2 차원 정수형 배열 table 을 채우고, table[num][L]의 값이 T(1)이면 subset 이 존재하고, F(0)이면 subset 이 존재하지 않는 것으로 판단하면 된다. 이때 T와 F는 True와 False의 약어로, 각각 1과 0의 값을 갖도록 매크로로 설정하였다. table 을 채우는 규칙은 아래와 같다.

i) $i=0$ and $j \neq 0 \rightarrow F$ (공식 a)

ii) $i=0$ and $j=0 \rightarrow T$ (공식 b)

iii) $i > 0 \rightarrow (table[i-1][j]) \text{ OR } (table[i-1][j-arr[i]] \text{ AND } arr[i] \leq j)$ (공식 c)

table[i][j]는 arr[1]부터 arr[i]까지 원소들을 조합하여 합이 j인 subset을 형성할 수 있는지를 의미한다. 만일 table[i][j]가 T의 값을 가진다면, arr[1]부터 arr[j]까지 원소들을 조합하여 그 합이 j가 되는 subset이 존재한다는 뜻이다. table의 행이 0일 경우, arr 배열의 어떠한 원소도 포함하지 않았을 때를 의미하므로 j가 0일 때는 제외하고는 모두 F의 값을 갖는다.

것은 자명하다. 이에 따라 공식 a 와 공식 b 가 도출된 것이다. 공식 c 는 그 외의 경우에서 $table[i][j]$ 값을 구하는 경우에 해당한다. $table[i-1][j]$ 가 참이라면 이미 $i-1$ 번째 원소까지의 원소들만으로도 합이 j 인 경우가 존재하는 것이므로 $table[i][j]$ 도 참이다. 하지만 $table[i-1][j]$ 가 거짓이더라도 $table[i][j]$ 가 참일 수 있다. 바로 i 번째 원소를 포함시켜야 비로소 합이 j 가 될 수 있는 경우가 그에 해당한다. 이는 곧 $i-1$ 번째 원소까지는 $j-arr[i]$ 만큼의 합을 이룰 수 있다는 것을 의미한다. $i-1$ 번째 원소까지 $j-arr[i]$ 만큼의 합을 이룰 수 있어야 i 번째 원소까지 포함시켰을 때 j 만큼의 합을 이룰 수 있기 때문이다. 따라서 $table[i-1][j-arr[i]]$ 가 참이면 $table[i][j]$ 도 참이다. 이를 정리한 것이 바로 공식 c 에 해당하는 것이다. Subset problem 에서 구해야 할 것은 크게 두 가지이다. 그 두 가지를 구하는 방법에 대한 자세한 설명은 다음 항목에 서술하였다.

1-1) table

table 은 앞서 서술한 이론에 따라 차례대로 값을 채우면 된다. 우선, 공식 a 와 b 에 의해 base 가 되는 값을 채워야 한다. 이를 코드로 표현하면 아래와 같다.

```
table[0][0] = T;
for (j = L; j >= 1; j--) table[0][j] = F;
```

공식 c 는 i 와 j 의 시작점과 끝점을 각각 잘 설정해야 한다. i 와 j 의 증가방향을 아래 사진처럼 설정하면 문제없이 table 의 값을 채울 수 있다. (참고로, 아래에 첨부한 사진은 강의자료 pdf 에 제가 아이패드로 덧그린 그림입니다.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F
3	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F
4	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F
5	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F
6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
7	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

*i*의 증가방향 (blue arrow pointing down)

*j*의 증가방향 (red arrow pointing right)

이를 코드로 구현하면 아래 사진과 같다.

```
for (i = 1; i <= num; i++) {
    for (j = 0; j <= L; j++) {
        table[i][j] = table[i - 1][j];
        if (j >= arr[i]) table[i][j] = table[i][j] || table[i - 1][j - arr[i]];
    }
}
```

이렇게 table 의 모든 값을 채운 후, table[num][L]의 값이 F 이면 이 문제에서 요구하는 subset 이 존재하지 않는 것을 의미하고, T 이면 존재한다는 것을 의미한다.

1-2) subset 의 원소의 개수&인덱스

1-1)번 항목에서 구한 table 을 토대로 문제에서 요구하는 subset 이 존재한다는 것을 확인한 뒤에는 해당 subset 의 원소의 개수와 각 원소의 인덱스 정보도 출력해야 한다. 이러한 subset 의 정보를 저장하기 위해 정수형 1 차원 배열 result 를 동적 할당한다. 그리고 result 의 원소로 subset 에 속하는 원소의 인덱스를 차례대로 저장하면 된다. 해당 정보는 table[i][j]를 역추적함으로써 얻을 수 있다. 앞서 1)번의 '이론' 항목에서 이미 언급했듯이, table[i][j]가 T 인 경우는 두 가지이다. table[i-1][j]가 참이거나 table[i-1][j-arr[i]]가 참이면 된다. 하지만 table[i-1][j]가 참이라서 table[i][j]가 참이 되는 경우는 arr[i] 원소가 굳이 없어도 이미 합이 j 인 경우에 도달하였으므로 그러한 것이므로 arr[i]가 합이 j 가 되도록 하는 부분집합의 필연적인 원소가 아닌 것이다. 하지만 table[i-1][j-arr[i]]가 참이라서 table[i][j]가 참인 경우는 arr[j]가 포함되어야 비로소 합이 j 인 경우에 도달할 수 있는 것이므로, 이 경우에는 arr[i]가 해당 부분집합의 필연적인 원소인 것이다. table[i-1][j]가 F 인 경우가 그에 해당하는 것이므로, table[i][j]부터 역추적을 시작하여 'table[i][j]==T && table[i-1][j]==F'인 경우라면 result 배열의 원소로 추가하면 된다. 이때, for loop 을 돌 때마다 i 는 1 씩 줄여나가고 j 는 arr[i]만큼씩 줄여나가는 식으로 진행하면 성공적인 역추적이 가능하다.

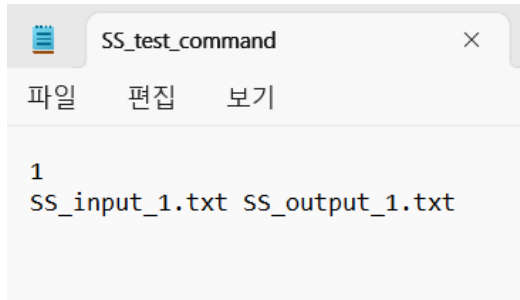
```
int* result = (int*)malloc(sizeof(int) * num);
int idx = 0;
j = L;
for (i = num; i >= 1; i--) {
    if (table[i][j] == T && table[i - 1][j] == F) {
        if (j > 0) {
            result[idx++] = i-1;
            j -= arr[i];
        }
    }
}
```

왼쪽의 그림은 이와 같은 내용을 코드로 구현한 것이다.

2) 구현에 실패한 부분

해당 문제에서는 구현에 실패한 부분은 없었다.

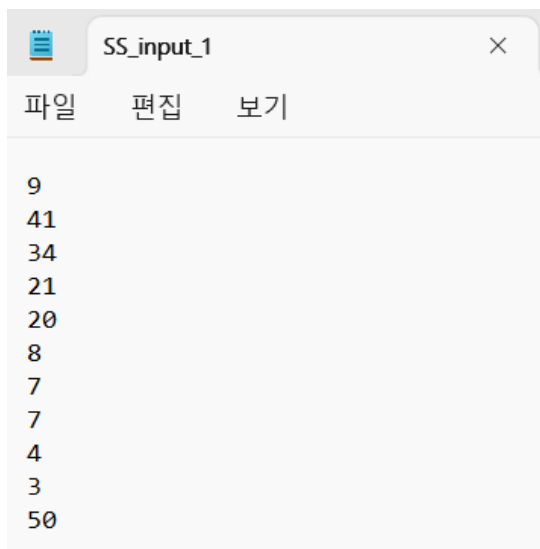
3) 실행 결과



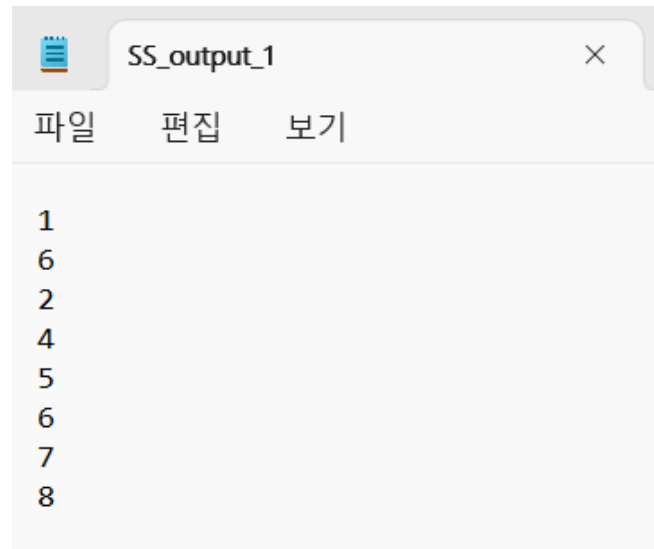
```
1
SS_input_1.txt SS_output_1.txt
```

만일 SS_test_command.txt 파일의 내용이 왼쪽과 같다면, "SS_input_1.txt" 이름의 텍스트 파일을 열어서 집합 원소의 개수 num, L, arr[1]~arr[num]을 저장하고, 그를 기반으로 계산한 '합이 L 이 되는 subset 의 존재 여부'와 'subset 원소의 개수'와

'subset 에 속하는 원소의 index'를 "SS_output_1.txt" 이름의 텍스트 파일에 저장한다.



```
9
41
34
21
20
8
7
7
4
3
50
```



```
1
6
2
4
5
6
7
8
```

왼쪽과 같은 내용의 "SS_input_1.txt" 파일을 입력 값으로 넣는다면, 그 결과 값은 오른쪽과 같이 '합이 L 이 되는 subset 의 존재 여부'가 첫 번째 줄에, 'subset 원소의 개수'는 두 번째 줄에, 그 다음 줄부터 끝까지 'subset 에 속하는 원소의 index' 정보가 "SS_output_1.txt" 파일에 저장된다.