

[컴퓨터공학설계 및 실험 I]

2023 학년도 1 학기 – 최종 프로젝트 보고서



학번	20190785
이름	박수빈
과목코드	CSE3013
분반	01(월)
프로젝트 주제	Disjoint set 알고리즘을 이용한 job scheduling

목차

1. 프로젝트 목표와 실험 환경
2. 핵심 변수 및 자료구조
3. 프로젝트 상세 설명
 - 3-1. 알고리즘과 함수
 - 3-2. 시간 및 공간 복잡도
4. 느낀 점 및 개선 사항

1. 프로젝트 목표와 실험 환경

오픈프레임웍스라는 툴을 이용하면 코딩으로 구현한 프로그램을 그래픽으로 화면에 표현하는 작업이 수월하다는 점에 주목하여, 다음과 같은 주제를 본 프로그램의 구현 목표로 설정하였다. disjoint set 알고리즘을 이용하여 입력 받은 '작업(job)'들을 각 작업의 '마감일(deadline)'이라는 제한 범위 내에서 최대의 '이익(profit)'을 창출하도록 '스케줄링(scheduling)'하여 그 결과를 확인하기 쉽도록 화면에 그려주는 프로그램을 구현하는 것이 이 프로젝트의 주제이다.

사용자는 본 프로그램을 실행함으로써 빠른 시간 내에 작업들을 최적화된 순서로 스케줄한 결과를 시각화된 자료로 볼 수 있다. 회사에서 직원들에게 업무를 배당할 때, 가게에서 주문을 받아 물건을 제작할 때 등 우리는 일상 생활에서 스케줄링을 해야 할 상황에 자주 마주친다. 빈발하는 스케줄링 문제를 알고리즘으로 접근하여 효율적인 작업 배당을 하는 것이 본 프로젝트의 핵심 operation이다. 입력 파일로부터 입력 받은 모든 job 정보들을 스케줄링 후보로 삼아 화면에 나타내고, 스케줄링 과정을 통해 선택된 job을 어떤 순서대로 작업하면 되는지 timeline 위에 그리는 것까지 하여 사용자 하여금 스케줄링 결과를 확인하기에 용이하게 만드는 것이 본 프로젝트의 최종 목표이다.

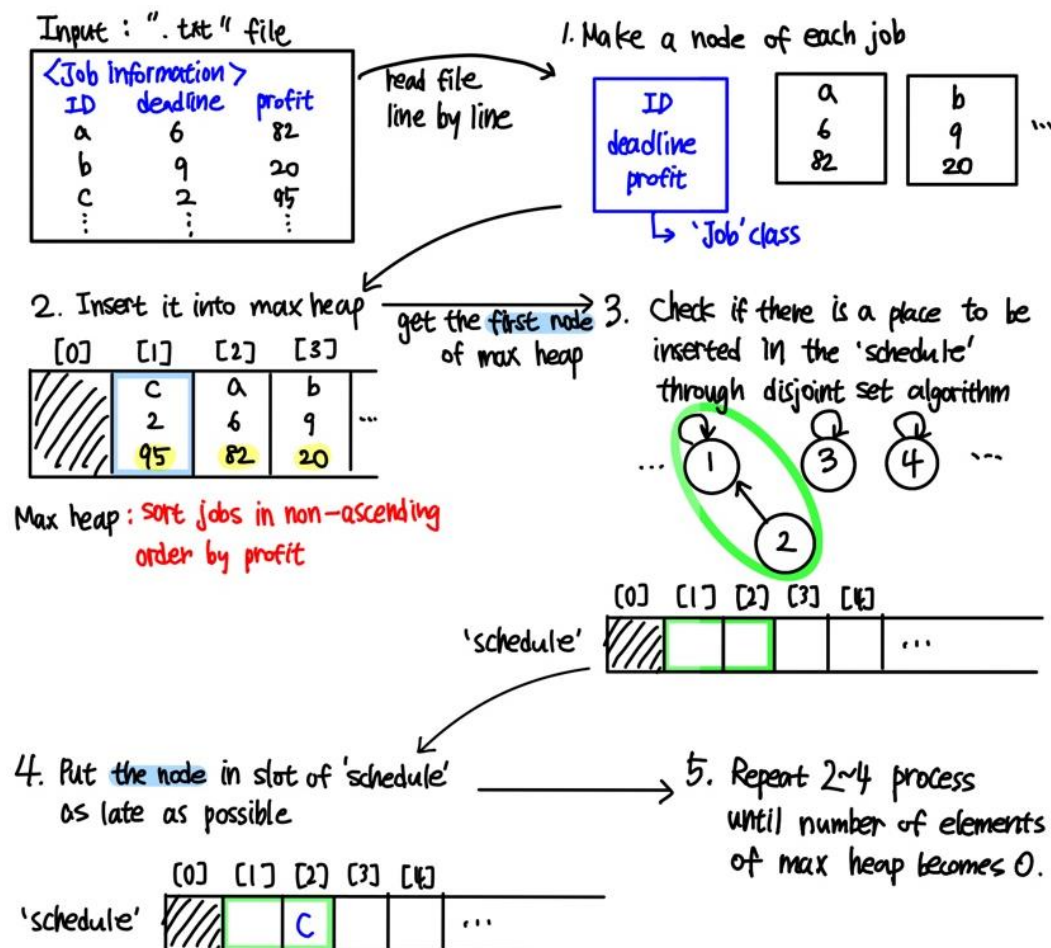
본 프로젝트는 Visual Studio 2022에서 오픈프레임웍스를 이용하여 개발되었다. 실험을 진행한 컴퓨터의 CPU 속도는 아래 사진과 같이, 기본 클럭 속도 1.60GHz로, 필요에 따라 최대

① Device specifications		1.80GHz까지 처리가 가능하다. 설치된 RAM의 총량은 8.00GB이고, 그 중 실제로 운영체제와 응용 프로그램에서 사용 가능한 메모리는 7.8GB로 실험을
Device name	DESKTOP-HUR7U9Q	
Processor	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz	
Installed RAM	8.00 GB (7.80 GB usable)	
Device ID	96F26FC7-5BA8-44F8-BFF1-7DCE5CDD67D1	
Product ID	00325-81573-07053-AAOEM	
System type	64-bit operating system, x64-based processor	
Pen and touch	No pen or touch input is available for this display	

진행하기에 적합한 용량이다. 컴퓨터 시스템 종류는 64bit 운영체제로, x64 기반 프로세서이다. 컴파일러는 개발 환경과 동일하게 Visual Studio 2022로 선택하였다.

2. 핵심 변수 및 자료구조

본 프로젝트에 사용된 변수 및 자료구조에 대해 본격적으로 설명하기에 앞서, 프로젝트의 전체적인 진행 흐름에 대해 핵심만 추려서 그림으로 나타내면 아래 사진과 같다. 그리고 이 프로그램의 전제는 모든 작업의 수행시간은 slot 1개로 모두 동일하고 ID는 다른 값을 가진다는 것이다. 또한, 화면에 timeline을 그려야 하기 때문에 마감일의 최대값을 20으로 정하였고, 이에 따라 입력 받을 수 있는 job의 최대 개수도 20개로 제한하였다. 마지막으로, 앞선 1번 항목에서도 서술했듯이, 본 프로젝트의 최종 목표는 효율적인 job scheduling을 통한 이익의 최대화이다.



위의 사진에서 확인할 수 있듯이, 본 프로젝트 구현을 위해서는 다음과 같은 자료구조들이 필요하다. 그리고 보다 편리한 이해를 위해 각 자료구조와 연관 있는 전역 변수가 있다면 함께 설명하는 식으로 서술했다.

1) 각 작업의 정보를 저장할 새로운 클래스 'Job' & 정수형 변수 'num_of_job'

```
/*Class Job is needed to store information of each job
which is the element of max heap*/
class Job {
public:
    char id; //ID of the job
    int dead; //Deadline of job
    int profit; //Profit of job
};
```

작업의 id는 멤버 변수 id에, 작업의 마감일은 멤버 변수 dead에, 이익은 멤버 변수 profit에 저장하는 식으로 클래스를 구성하였다. 정수형 전역

변수 num_of_job은 이러한 Job이 input file로부터 몇 개를 입력 받았는지를 나타낸다.

2) 'Job'을 저장하는 max heap (=> Job* job_heap) & 정수형 변수 'num_of_heap'

이 max heap은 앞서 생성한 클래스 'Job'을 원소로 갖는 1차원 배열로 형성하였다. 그리고 이 heap은 job_heap이라는 이름으로 ofApp.h 파일에 전역변수로 선언해 놓았다. 여기서 참고로, min heap이 아니라 max heap인 이유는 job을 profit이 증가하지 않는 순서로 정렬해줘야 하기 때문이다. 그리고 이 max heap의 원소의 개수를 num_of_heap으로 설정하였다. (처음에는 num_of_job을 사용하였지만, heap insertion/deletion 과정 중 원소의 개수가 계속 바뀌기 때문에 따로 num_of_heap을 선언하여 사용하였다.)

3) 결과를 저장하는 1차원 정수형 배열 'schedule' (=> char schedule[21])& 정수형 변수 'count'

이 schedule의 역할은 작업이 timeline에 배당된 결과를 저장하는 것이다. 이 schedule의 한 칸을 'slot'이라 명명하였다. 앞서 서술했듯이, 본 프로젝트에서는 마감일의 최댓값을 20으로 설정하였고, 0번째 slot은 dummy slot이기 때문에 schedule의 크기는 sizeof(int)*21이다. 이 배열은 처음에는 'W0'으로 초기화되어 있다. 이후에 작업 배당이 확실해지면 해당 자리에 배당된 작업의 id를 입력하면 된다. 예를 들어 schedule[3]의 값이 'a'라면, 'a'라는 id를 가진 작업이 timeline에서 slot 3에 위치하는 것이므로 작업 a는 timeline 상에서 2~3에 해당하는 시기에 수행되도록 배정받았다는 뜻이다. 그리고 이처럼 schedule에 작업이 할당이 완료되면 count의 값도 1증가시킴으로써, count는 스케줄링이 끝난 뒤 job 후보들 중에서 스케줄에 배당된 job이 몇 개인지 그 개수를 나타내는 역할을 한다.

4) disjoint set 작업을 위한 'Node' 클래스

```
class Node {
public:
    int parent; //Slot number of parent node
    int rank; //Height of the node
};
```

disjoint set을 구성하는 각 노드들의 클래스를 왼쪽의 사진과 같이 작성하였다. 각 노드는 자신의

부모의 번호를 멤버 변수 parent에, disjoint set에서 자신이 위치한 곳의 높이를 멤버 변수 height에 저장한다. 만일 부모가 없다면, 자기 자신의 번호를 parent에 저장한다.

5) 'Node'를 원소로 갖는 'disjoint_set' (=>Node* disjoint_set)

앞서 생성한 클래스 'Node'를 원소로 갖는 1차원 배열 disjoint set은 max heap에서 꺼낸 첫 번째 원소, 즉 profit이 가장 큰 원소가 schedule 배열에 들어갈 자리가 있는지 확인하기 위해 필요하다. 앞선 2페이지의 사진에서는 쉬운 이해를 위해 disjoint set을 트리와 같은 형태로 시각화했지만, 실제 코드에서는 각 Node에 parent 정보와 rank 정보가 저장되어 있고, 이 Node들이 배열의 원소로 구성되어 있는 형식으로 구현되어 있다. 그리고 disjoint_set은 schedule에 남아있는 자리가 있는지 확인하기 위해 사용되는 자료구조이기 때문에 이 배열 역시 21개의 원소를 갖는다. 만일 disjoint_set[2]와 disjoint_set[2]의 부모가 같다면, 즉 disjoint_set[2].parent == disjoint_set[3].parent 라면, schedule[2]와 schedule[3]이 같은 그룹으로 묶였다는 뜻이다. 여기서 같은 그룹으로 묶인다는 것의 의미에 대해서는 뒤의 항목에서 후술할 예정이다.

6) Job을 완벽히 정렬한 1차원 배열 'candidate' & 정수형 변수 num_of_can

앞서 소개한 자료구조들은 모두 job scheduling 작업을 위한 자료구조였다. 하지만 6)번 항목에 소개되는 candidate은 입력 파일로부터 입력 받은 모든 job을 화면에 그리기 위해 필요한 자료구조이다. 앞선 2)번 항목에서 소개한 max heap을 이용하려 했으나, job_heap을 그대로 1번 index부터 끝까지 출력하면 heap의 특성 상 완벽히 정렬된 상태로 출력되지 않는다. 그렇다고 max heap deletion을 통해 차례대로 추출해서 출력하려 하면, heap의 내용이 바뀌고 화면을 계속 일정 시간 간격으로 업데이트해주는 오픈프레임웍스의 특성으로 인해 그린 그림을 유지할 수 없다. 그래서 job_heap의 정보를 그대로 복사하여 candidate에 붙여넣기 한 다음, 이 candidate을 insertion sort를 사용하여 완벽히 정렬한 이후에 이를 1번 index부터 차례대로 화면에 그리는

식으로 해결하였다. (여기서 삽입 정렬을 사용하는 이유에 대해서는 후술할 예정이다.) 그리고 이러한 candidate 배열의 원소 개수 정보는 num_of_can이라는 전역 변수에 저장한다.

7) 오픈프레임웍스의 여러 툴을 이용하기 위한 flag 변수들

이제부터는 오픈프레임웍스의 여러 툴을 활용하여 파일을 로드하고 화면에 스케줄링 결과를 시각화하기 위해 필요한 전역변수로 선언된 여러 가지 flag 변수들에 대해 아래 표에서 서술하였다.

Key operation	Flag이름	1	0
'l' key	load_flag	입력 파일이 정상적으로 로드됨	정상적으로 로드되지 않음
't' key	draw_timeline_flag	화면에 timeline 주눈금을 그림	그리지 않음
's' key	draw_scale_flag	화면에 timeline의 수직선을 보조선으로 그림	그리지 않음
'c' key	draw_candidate_flag	화면에 입력 파일로부터 입력 받은 모든 job을 스케줄링의 후보로 그림	그리지 않음
'r' key	draw_result_flag	화면에 스케줄링 결과를 timeline 바로 위에 사각형으로 그림	그리지 않음
	write_id_flag	스케줄링 결과에서 각 job의 ID를 나타냄	나타내지 않음
'<-' or '->' key	draw_target_flag	스케줄링 결과에서 좌우 방향으로 targeting한 job을 연한 하늘색 사각형으로 그림	그리지 않음

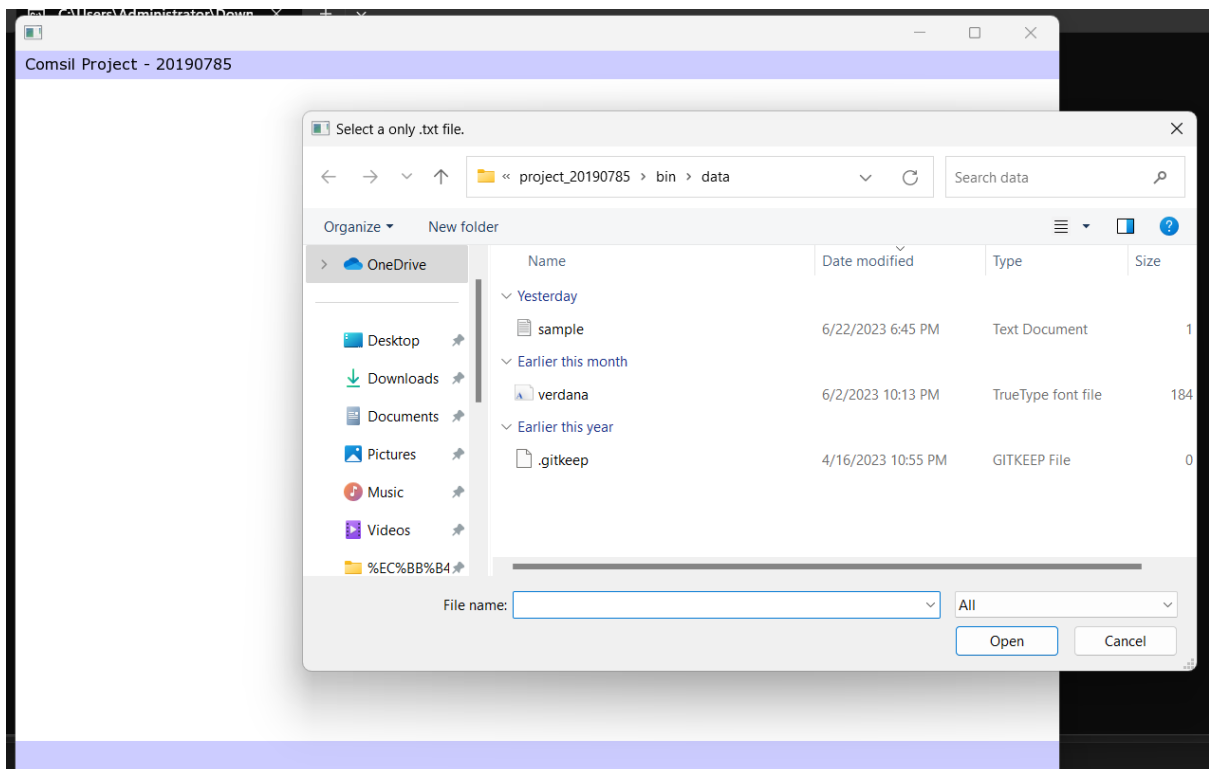
flag 변수는 아니라서 표에서 제외했지만, '<-' or '->' key operation과 관련이 깊은 전역 변수에 대해 이 항목에서 서술하고자 한다. 'selected_job'이라는 이름의 정수형 전역 변수는 스케줄링 결과에서 좌우 방향으로 targeting한 job을 연한 하늘색 사각형으로 그리기 위해 화면 상에서 옮겨다니는 x좌표의 역할을 한다.

3. 프로젝트 상세 설명

본격적으로 프로젝트에 대해 상세 설명하기에 앞서, 입력 파일의 형식은 다음과 같다고 가정한다. 첫 번째 줄에는 스케줄링해야 하는 작업의 총 개수, 두 번째 줄부터 끝까지는 각 작업의 정보에 대해 입력되어 있다. 공백을 기준으로 나뉘며 'job의 ID', 'deadline', 'profit' 순서대로 정보가 기재되어 있음을 전제로 입력 파일을 읽어들인다.

3-1. 알고리즘과 함수

본 프로젝트를 시작하려면, 우선 job들의 정보가 담긴 input file을 load 해야 한다. (이때, 입력 파일의 형식은 반드시 .txt 이어야 한다.) 'I' key를 누르면 아래와 같은 화면이 뜨면서 input file을 입력 받기 위한 창이 뜬다.



이 보고서에서는 sample.txt 라는 파일을 실험 진행에 사용하였다. sample.txt의 내용은 아래 사진과 같다.

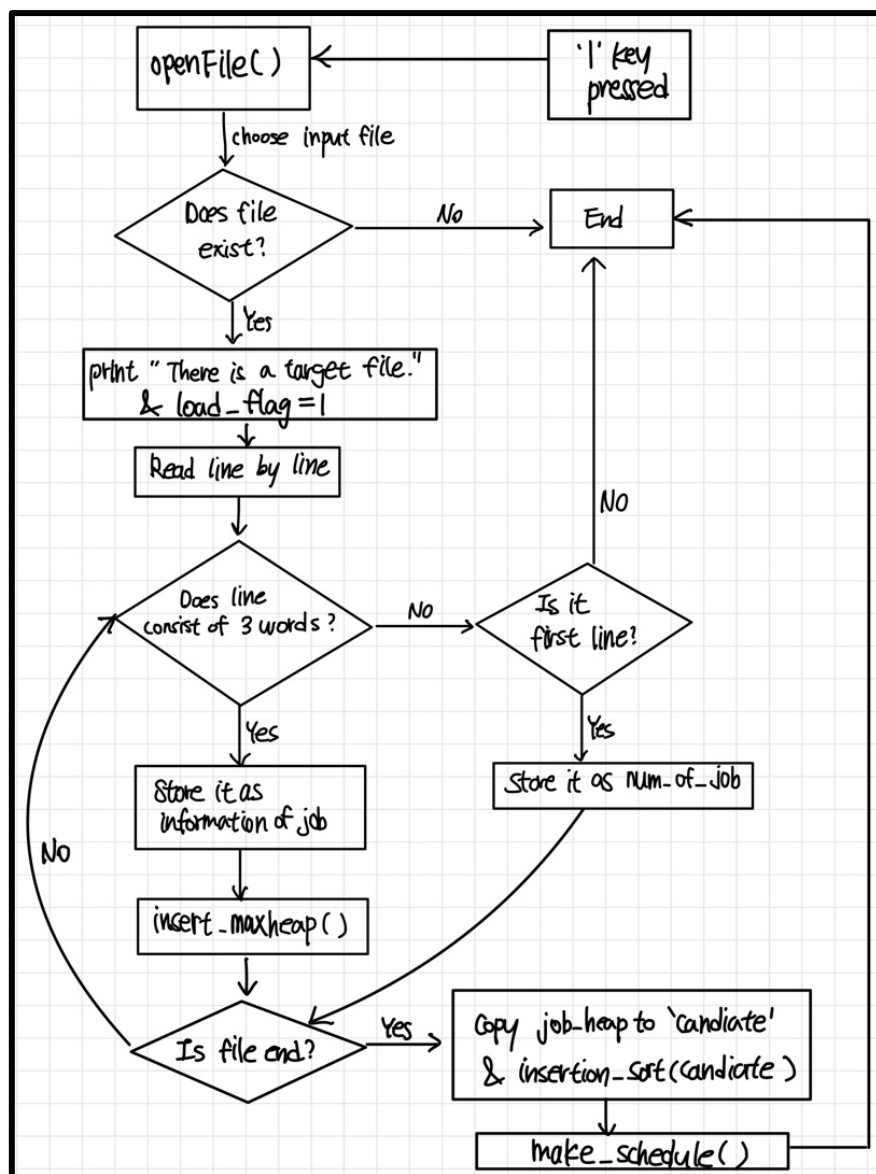

```

sample
File Edit View

18
a 1 100
b 6 80
c 3 90
d 3 120
e 5 40
f 4 105
g 1 115
h 2 85
i 4 50
j 20 190
k 14 170
l 13 150
m 18 175
n 10 95
o 12 50
p 9 140
q 7 70
r 8 60

```

총 18개의 job을 input data로 받는다는 정보가 첫 번째 줄에 적혀 있다. 그 이후부터는 각 job의 id, deadline, profit이 순서대로 적혀있다. (여기서, 각 job의 deadline은 20을 초과해서는 안 된다. 만일 초과하더라도 openFile 함수에서 예외로 처리된다.) 이렇게 입력 받은 파일의 내용을 읽는 과정을 하는 함수가 바로 openFile 함수이다. keyReleased 함수에 의해 'l' key가 인식된 이후에는 openFile 함수가 호출되는데, 이 함수의 매커니즘을 아래의 플로우 차트로 나타내었다.



이제 openFile 함수에서 쓰인 여러 함수들에 대해 서술하겠다.

1) void insert_maxheap(Job* job_heap, Job data, int* num_of_job)

```
void ofApp::insert_maxheap(Job* job_heap, Job data, int* num_of_heap) {
    /**Purpose of the function: insert the data which we get from input txt file into max heap schedule***/
    //This is for sorting the profits of jobs in non-increasing order

    if ((*num_of_heap) == 20) {
        fprintf(stderr, "The max heap is full.\n");
        std::exit(1);
    }
    (*num_of_heap)++;

    int i = (*num_of_heap);
    while (1) {
        if (i == 1) break; //If number of elements is 1, there's nothing to compare
        //i keep going up while it find right place to insert the data
        if (data.profit <= job_heap[i / 2].profit) break;
        job_heap[i] = job_heap[i / 2];
        i /= 2;
    }
    //Finally, i find right place and insert the data into the place
    job_heap[i] = data;
    return;
}
```

인자로 전달받은 data를 job_heap이라는 max heap에 삽입하는 함수이다. max heap의 가장 마지막 자리에서 시작하여 data의 profit 값보다 큰 값을 마주칠 때까지 계속해서 heap의 root 쪽으로 올라가는 식으로 자리를 찾는다. 그렇게 부모 자리로 계속 올라가는 과정에서 부모에 있던 노드를 자식 자리로 옮겨놓는 작업도 동시에 진행한다. 마침내 적합한 자리를 찾으면 그 곳에 data를 삽입하고 함수를 종료한다.

이렇게 insert_maxheap 함수를 통해 job_heap을 형성한 다음, input file로부터 모든 정보를 입력 받은 뒤에는 job_heap의 내용을 candidate으로 복사한다. 그러고는 candidate을 insertion_sort 함수를 사용하여 완벽히 정렬한다. 여기서 여러 가지 정렬 방법 중 삽입 정렬을 선택한 이유는 job_heap은 이미 heap에 데이터들을 삽입하는 과정에서 어느 정도 정렬을 해놓은 상태이기 때문이다. 삽입 정렬은 정렬의 방식으로 인해 완벽히 거꾸로 정렬된 경우는 $O(n^2)$ 의 시간 복잡도를 가지지만, 이처럼 어느 정도 정렬이 되어 있는 경우는 각 loop을 돌 때마다 많은 비교 작업 없이 원소가 삽입될 위치를 찾을 수 있기 때문에 시간 복잡도가 $O(n)$ 까지도 개선될 수 있기 때문에 candidate의 데이터 특성을 고려하여 삽입 정렬을 정렬의 방식으로 선택하였다.

2) `int insertion_sort(Job* data, int left, int right)`

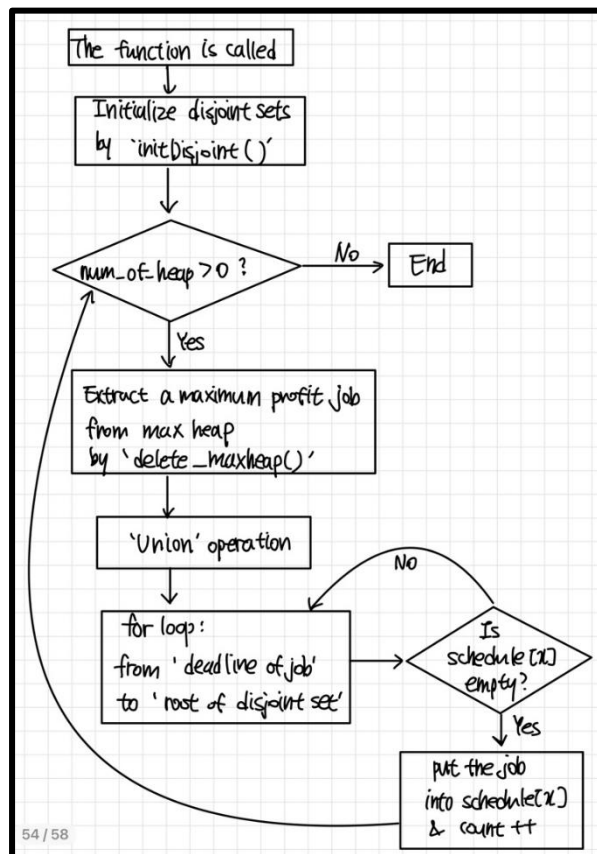
```
int ofApp::insertion_sort(Job* data, int left, int right) {
    if (left < 0 || right < 0 || left > right || data == NULL) {
        return -1;
    } //return error code when execution is not available

    int i, j;
    Job temp;
    int size = right - left + 1; //size of data
    for (i = 1; i <= size; i++) {
        temp = data[i]; //Store information of the object into 'tmp'
        j = i; //Set the last point of comparison
        //Keep comparison until it becomes smaller than other
        while ((temp.profit > data[j - 1].profit) && (j > 1)) {
            data[j] = data[j - 1];
            j--;
        }
        data[j] = temp;
    }
    return 0;
}
```

이 삽입 정렬의 메커니즘은 이미 너무나도 잘 알려진 방식이고, 핵심적인 사항은 왼쪽의 사진에서 주석으로 서술하였으므로 함수 설명을 생략하도록 하겠다.

위와 같은 `insertion_sort` 함수로 candidate 정렬을 마친 뒤에는, `make_schedule` 함수를 통해 본격적인 job scheduling을 진행한다.

3) `void make_schedule()`



이 함수는 job scheduling 작업을 진행하는 데 있어서 핵심적인 함수 중 하나이므로, 플로우 차트로 전체적인 흐름에 대해 먼저 설명하고 난 뒤, `make_schedule` 함수에서 쓰인 여러 다른 함수들에 대해서도 설명하도록 하겠다.

우선 함수가 호출되면 `initDisjoint` 함수를 호출하여 `disjoint_set`을 초기화해준다. 그런 다음, `job_heap`에서 첫 번째 원소(즉, profit이 가장 큰 Job)를 뽑은 다음, 그 첫 번째 원소의 deadline과 그 보다 1만큼 작은 값을 Union 함수를 통해 결합한다. 이런 과정을 통해 해당 Job의 deadline이 결합되어 있는 구간을 알 수

있고, 해당 구간에서 schedule slot이 비어있는 곳이 있는지를 찾는다. 만일 해당 구간에서 빈 slot이 있다면 job을 그 자리에 넣어주고 count를 1 증가시켜준다. 이 과정을 job_heap의 원소 개수가 0이 될 때까지 반복해준다.

아래는 이 make_schedule 함수에서 사용된 함수들에 대한 설명이다.

3-1) Node* initDisjoint(Node* disjoint_set, int n)

```
Node* ofApp::initDisjoint(Node* disjoint_set, int n) {
    /**Purpose of the function: Initialize all elements of disjoint set***/

    disjoint_set = (Node*)malloc(sizeof(Node) * n);
    for (int i = 0; i < n; i++) {
        disjoint_set[i].parent = i;
        disjoint_set[i].rank = 0;
    }
    return disjoint_set;
}
```

n개 Node만큼 disjoint_set을 동적 할당을 해준다. 그런 다음, disjoint_set의 각 노드의 부모는 자기 자신으로 입력하고, rank도 0으로 입력한다. (rank가 0인 이유는 아직 자신을 부모로 삼는 자식 노드와 같은 disjoint_set의 원소가 없기 때문이다.)

3-2) int Find(Node* disjoint_set, int i)

```
int ofApp::Find(Node* disjoint_set, int i) {
    /**Purpose of the function: find the ancestor of i element***/

    //Repeat the while loop until it reach the node whose value is same with job num itself
    while (disjoint_set[i].parent != i) {
        i = disjoint_set[i].parent;
    }
    return i;
}
```

이 Find 함수를 구현하는 데는 path compression, recursion 등 여러 방법이 있지만, 본 프로젝트에서는 iteration을 사용하여 구현하였다. 자기 자신을 부모로 갖는, 즉 자신이 root node인 원소를 찾을 때까지 while loop을 반복하는 식으로 i 원소의 조상을 찾는 함수이다.

3-3) void Union(Node* disjoint_set, int i, int j)

```
void ofApp::Union(Node* disjoint_set, int i, int j) {
    /**Purpose of the function: unite two sets who include i and j respectively***/

    //find the root node of i and j
    int i_root = Find(disjoint_set, i);
    int j_root = Find(disjoint_set, j);

    //Absorb the smaller rank of set into a tree whose root node value is greater or equal
    if (disjoint_set[i_root].rank < disjoint_set[j_root].rank) {
        disjoint_set[i_root].parent = j_root;
    }
    else if (disjoint_set[i_root].rank > disjoint_set[j_root].rank) {
        disjoint_set[j_root].parent = i_root;
    }
    else {
        if (i_root < j_root) {
            disjoint_set[j_root].parent = i_root;
            disjoint_set[i_root].rank++;
        }
        else {
            disjoint_set[i_root].parent = j_root;
            disjoint_set[j_root].rank++;
        }
    }
    return;
}
```

i를 포함하는 set와 j를 포함하는 set를 합치는 역할을 하는 함수이다. 우선 i와 j의 root node를 찾은 다음, 이 두 root node 중 rank가 큰 set에 작은 set를 포함시키는 방식으로 set을 합친다. rank가 큰 set에 작은 set를 합치는 이유는, 그 반대로 하면 나중에 Find 함수로 어떤 원소의 root 노드를 찾을 때 한참 올라가야 하는, 즉 while loop을 많이 돌려야 해서 시간 복잡도가 많이 증가할 수 있기 때문이다.

3-4) Job delete_maxheap(Job* job_heap, int* num_of_job)

```
Job ofApp::delete_maxheap(Job* job_heap, int* num_of_heap) {
    /**Purpose of the function: delete the maximum element from max heap**/
    //It means that we select the job whose profit is the biggest in the schedule

    if ((*num_of_heap) == 0) {
        fprintf(stderr, "The heap is empty.\n");
        std::exit(1);
    }

    //Save first element in 'first'
    Job first = job_heap[1];

    //Save the last element in 'last'
    Job last = job_heap[(*num_of_heap)];
    (*num_of_heap)--;

    //Starting point of parent and child
    int parent = 1, child = 2;

    while (1) {
        if (child > (*num_of_heap)) break;

        //Check which one is bigger among left child and right child
        if (child <= (*num_of_heap) - 1) {
            if (job_heap[child + 1].profit > job_heap[child].profit) ++child;
        }

        //Find the place where 'last' should be inputed
        if (last.profit >= job_heap[child].profit) break;

        //Move the 'child' up to 'parent'
        job_heap[parent] = job_heap[child];

        //Move to down in the heap
        parent = child;
        child = child * 2;
    }

    //Input the 'last' into the place where we had found in previous process
    job_heap[parent] = last;

    //Return the element what we have removed from the heap
    return first;
}
```

max heap에서 가장 첫 번째 원소를 뽑아낸 다음, 남은 원소들을 다시 max heap의 특성을 유지하도록 정렬해주는 함수이다. 가장 첫 번째 원소를 뽑아서 first에 저장해 놓은 다음 나중에 함수가 끝나면 이를 반환하도록 한다. 첫 번째 원소를 뽑은 다음에는, heap의 가장 마지막 원소를 가장 첫 번째 원소의 자리로 집어넣은 다음, 자식들과 계속 비교하며 자식들보다 본인이 작아지기 직전까지 내려가서 본인의 자리를

찾는 형식으로 max heap의 특성을 유지한다. 이와 같은 delete_maxheap 함수는 insert_maxheap 함수와 같이 쓰이면 heap sort라는 정렬 방법으로 이처럼 쓰일 수 있는데, 이 정렬 방법은 원소를 추가하거나 삭제할 때 모두 $O(n \log n)$ 이라는 시간 복잡도를 지닌다. disjoint set을 만들 때와 같이 원소의 추가와 삭제 모두가 빈번하게 일어나는 경우 heap sort가 매우 적절하다. (왜냐하면 다른 정렬 방법들은 추가 시 시간 복잡도가 매우 낮으면 삭제 시 시간 복잡도가 매우 높은 식으로, 한 쪽으로 치우친 경우가 많기 때문이다.)

여기까지가 make_schedule 함수에 쓰인 함수들에 대한 설명이었다.

```

void ofApp::make_schedule() {
    Job data;
    int root;
    disjoint_set = initDisjoint(disjoint_set, 21);

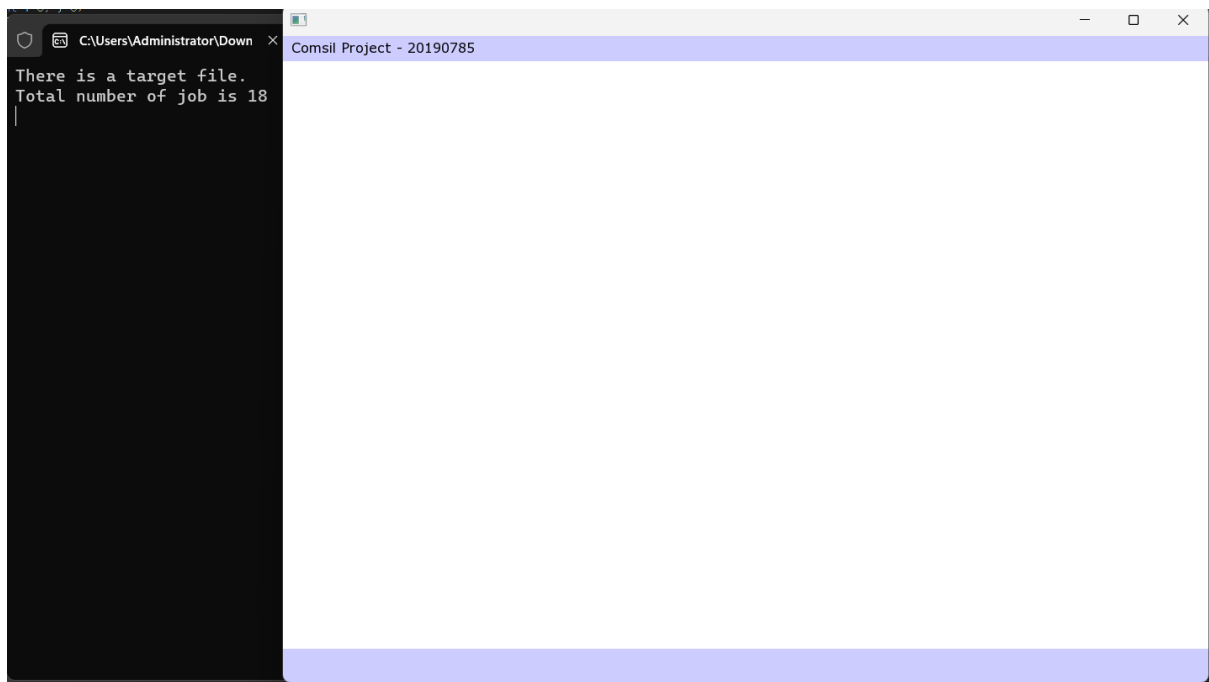
    while (num_of_heap > 0) {
        //Get the job which has biggest profit
        data = delete_maxheap(job_heap, &num_of_heap);

        Union(disjoint_set, data.dead, data.dead - 1);
        root = disjoint_set[data.dead].parent;
        for (int x = data.dead; x > 0; x--) {
            if (disjoint_set[x].parent != root) break;
            if (!schedule[x]) {
                schedule[x] = data.id;
                count++;
                break;
            }
        }
    }
}

```

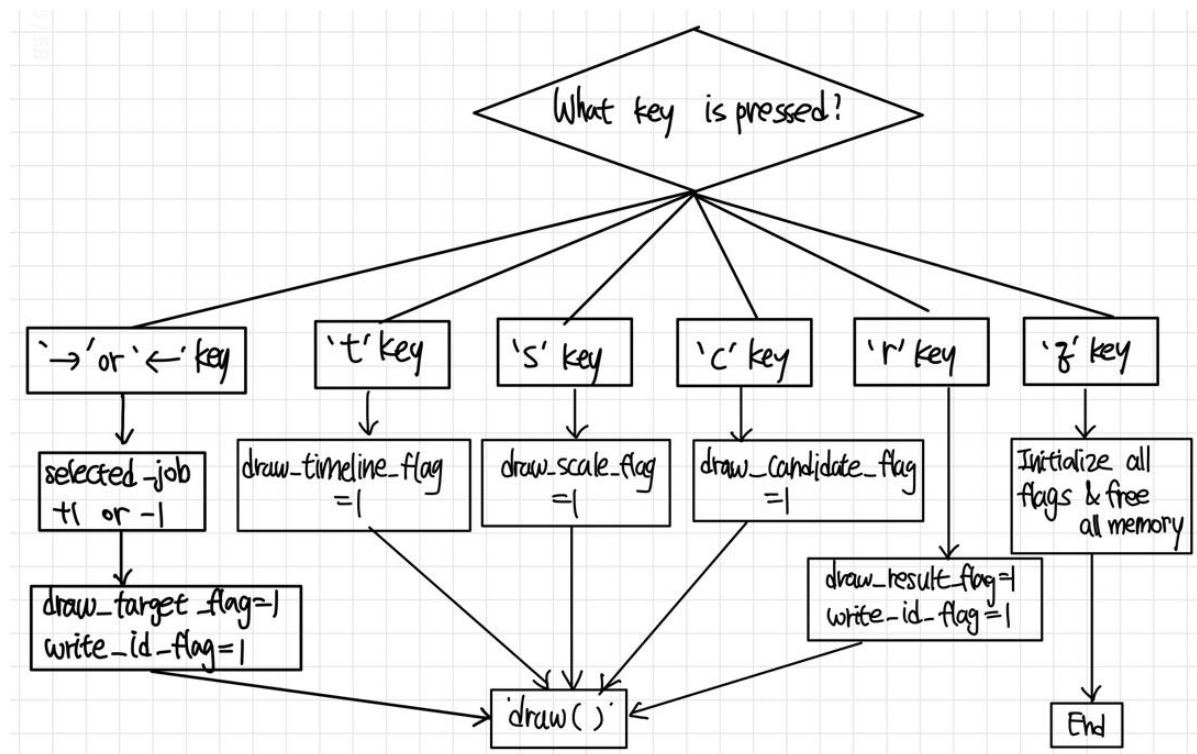
make_schedule 함수는 앞서 설명한 함수들을 이용하여 왼쪽의 사진과 같이 구현될 수 있다.

위와 같은 흐름으로 openFile 함수가 진행되며 입력 받은 정보들을 알맞은 자료구조에 저장한다. 성공적으로 파일을 load 완료했으면, 아래 사진과 같이 콘솔 창에 메시지를 띄우면서 'l' key operation은 끝난다.



이제껏 'l' key가 인식되면 어떤 알고리즘과 함수들이 작동하는지에 대해 설명하였다. 이제부터는 앞선 과정들을 통해 얻은 job scheduling 결과를 어떻게 화면에 창의적으로 구현해낼 수 있는지에 중점을 둔 부분에 대해 서술하고자 한다.

어떤 key를 입력 받는지에 따라 전체적인 플로우 차트를 그려보면 아래 그림과 같다. ('l' key operation은 이미 앞서 설명했으므로, 아래 플로우 차트에서는 제외하였다.)



각 key가 눌리는 대로 위의 플로우 차트와 같이 각각에 대응되는 flag의 값들이 1로 변한다. (참고로, setup 함수에서 모든 flag들의 값을 0으로 초기화해놓은 상태에서 시작한다.) flag 값들이 1로 변하면 그에 따라 draw 함수에서도 아래에 첨부한 코드와 같이 작동하게 된다.

1) 't' key

```
/****Draw shapes for timeline****/
if (draw_timeline_flag) {
    //Set the line width as 4
    ofSetLineWidth(4);
    ofSetColor(102, 102, 102);

    //1. Draw horizontal line
    ofDrawLine(20, 700, 1020, 700);

    //2. Draw vertical line & write scale
    for (i = 0; i <= 1020; i += 50) {
        ofDrawLine(20 + i, 690, 20 + i, 710);
        sprintf(str, "%d", i / 50);
        ofDrawBitmapString(str, 15 + i, 720);
    }
}
```

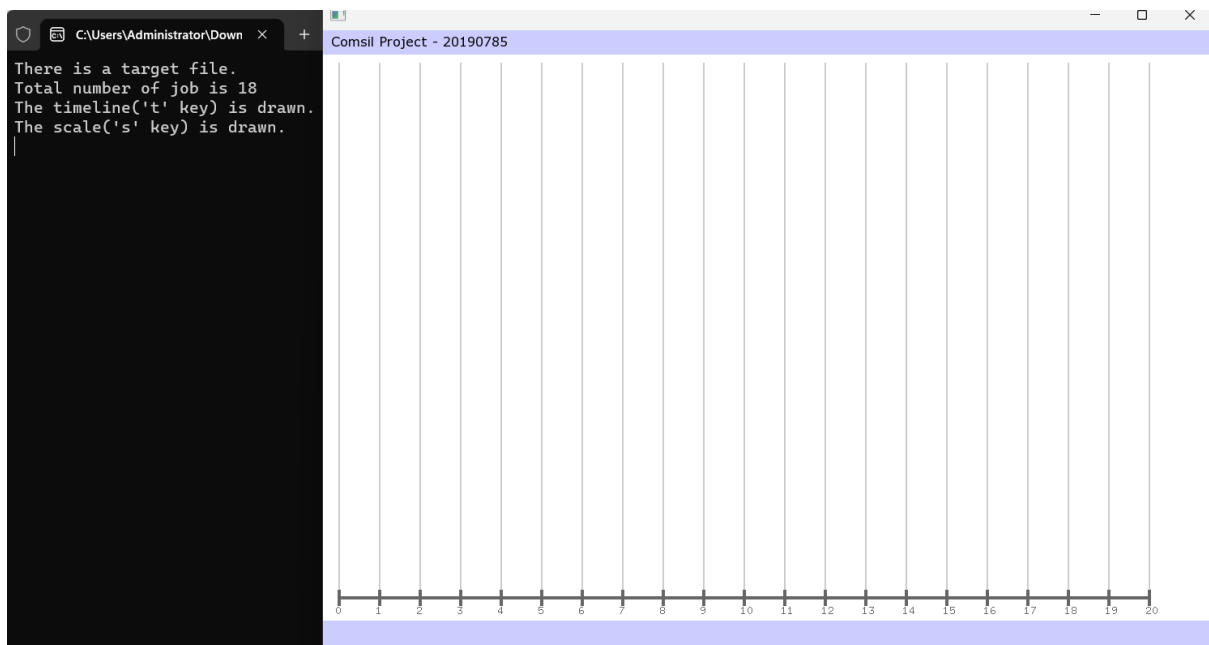
선의 굵기를 4로 정하고, 색상을 진한 회색으로 설정한다. 그 다음 (20, 700)부터 (1020, 700)까지 선을 그려준다. 이 선은 timeline의 수평선이 된다. 수평선을 그린 다음에는, 짧은 수직선을 일정 간격으로 그려서 수평선을 20등분하는 주눈금을 그린다. 그리고 각 주눈금의 밑에는 작은 글씨로 그 숫자를 써준다. 실행하면 아래 화면과 같다.



2) 's' key

```
/**Draw shapes for scale of timeline**/  
if (draw_scale_flag) {  
  
    //Draw thin vertical line  
    ofSetColor(204, 204, 204); //Light gray color  
    ofSetLineWidth(2);  
    for (i = 0; i <= 1020; i += 50) {  
        ofDrawLine(20 + i, 40, 20 + i, 690);  
    }  
}
```

draw_scale_flag의 값이 1로 바뀌면, 색깔을 연한 회색으로 설정하고 선의 굵기를 2로 줄인 다음, 앞서 't' key가 눌렸을 때 그려진 timeline 윗 부분에 수직 보조선을 그려준다. 실행 결과는 아래에 첨부한 사진과 같다.



3) 'c' key

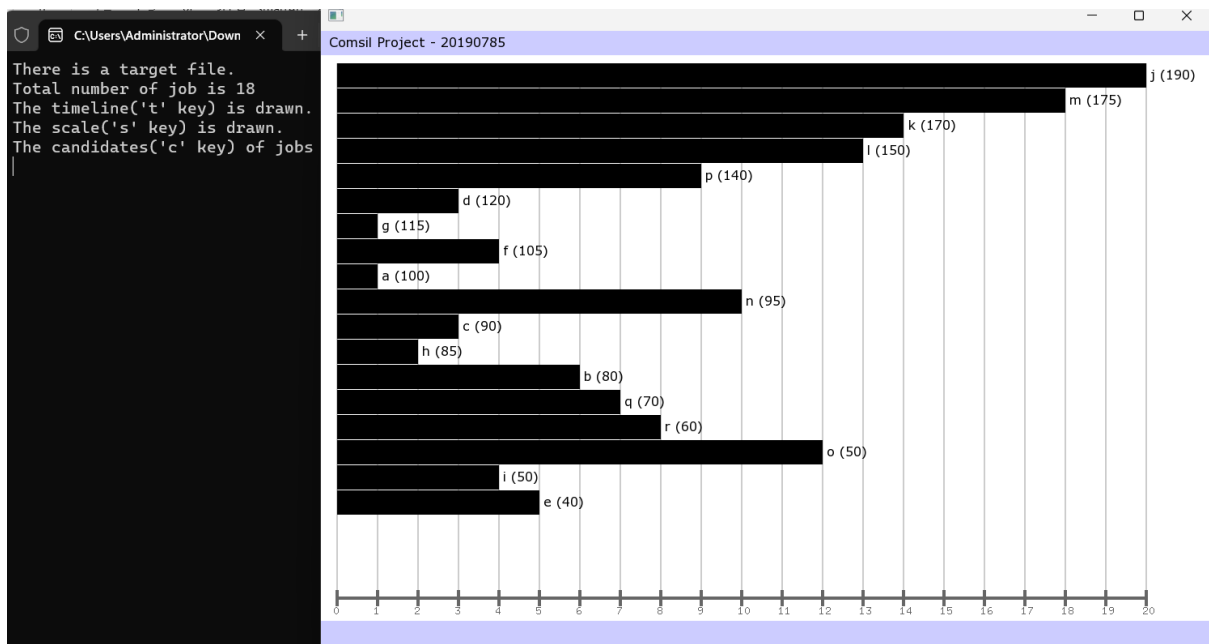
```

/**Draw candidates on the screen***/
if (draw_candidate_flag) {
    char job_id[30];
    ofSetColor(0, 0, 0); //Black color

    for (i = 1; i <= num_of_job; i++) {
        ofDrawRectangle(20, 40 + (i - 1) * 31, (candidate[i].dead) * 50, 30);
        sprintf(job_id, "%c (%d)", candidate[i].id, candidate[i].profit);
        font.drawString(job_id, 25 + (candidate[i].dead) * 50, 60 + (i - 1) * 31);
    }
}

```

이 'c' key 가 눌렸을 때 그릴 것을 위해, 앞선 과정에서 candidate 배열을 정렬해 놓은 것이다. input file 로부터 입력 받은 모든 job 들을 shceduling 의 후보로 저장하여 삽입 정렬을 통해 profit 이 증가하지 않는 순서로 정렬한 candidate 배열을 가장 위에서부터 차례대로 화면에 그려준다. 실행 결과는 아래의 사진과 같다. 이때 막대의 길이는 해당 job 이 수행될 수 있는 기간을 의미한다. 예를 들어, job 'm'는 timeline 에서 0 부터 18 까지의 기간 동안 실행될 수 있고, 19 부터는 이미 마감일을 넘어서서 해당 기간에는 실행될 수 없음을 의미하는 것이다. 막대의 옆에는 job ID 와 함께 괄호 안에는 profit 이 적혀있다.



4) 'r' key

```

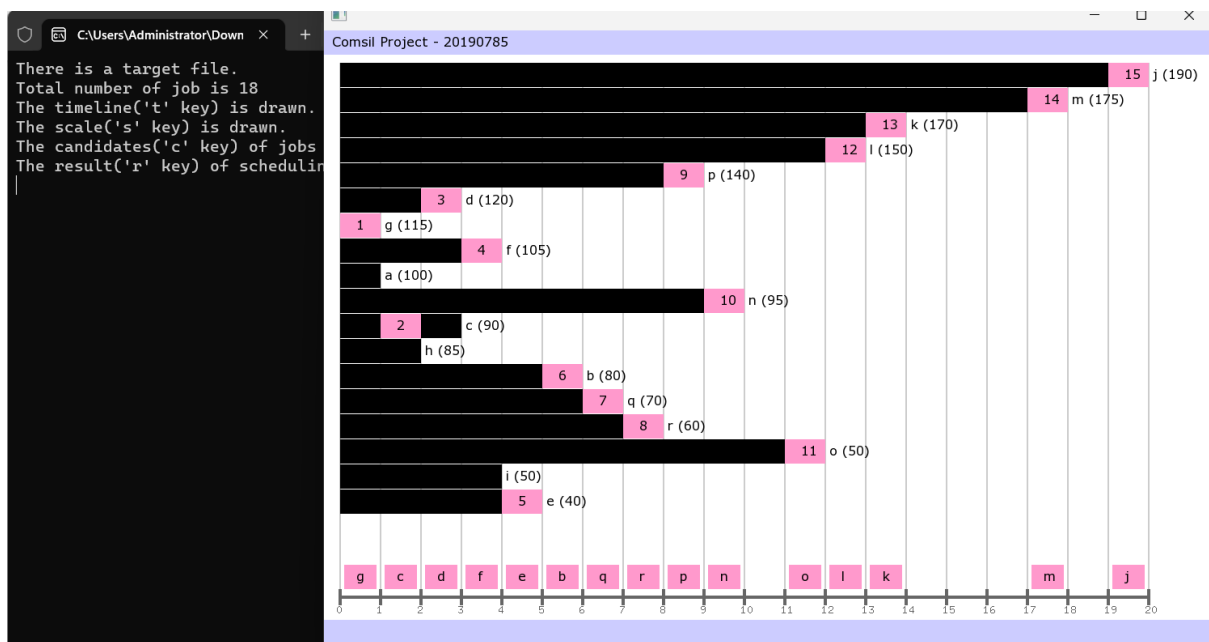
/**Draw shapes for result of scheduling***/
if (draw_result_flag) {

    ofSetColor(255, 153, 204); //Light pink color

    for (i = 1; i <= 21; i++) {
        if (schedule[i] != '\0') { //If the slot of schedule is not empty
            //Indicate there is a job which is assigned through drawing rectangle
            ofDrawRectangle(25 + (i - 1) * 50, 660, 40, 30);
        }
        for (j = 1; j <= num_of_can; j++) {
            if (candidate[j].id == schedule[i]) {
                ofDrawRectangle(20 + (i - 1) * 50, 40 + (j - 1) * 31, 50, 30);
            }
        }
    }
}

```

'r' key가 눌리면, job scheduling의 결과, 즉 배열 schedule의 내용을 기반으로 선택된 job들을 대응되는 slot에 사각형으로 그린다. timeline 바로 위에는 각 slot 별로 실행되어야 하는 job들이 id와 함께 나타나 있다. 그리고 앞서 그렸던 막대 그래프 위에도 각 작업 별로 기한 내 정확히 언제 시행되어야 하는지 사각형으로 표시하고, 실행되어야 하는 순서도 그 사각형 위에 적어주었다. 예를 들어, 아래 실행 결과에서는 job 'c'는 두 번째로 실행되어야 하며, 이는 원래 마감일인 3보다 빨리 실행되어야 한다는 것을 확인할 수 있다.



5) '<-' or '->' key

```
/**Draw shapes for target in the result of scheduling***/
if (draw_target_flag) {

    ofSetColor(102, 255, 255); //Light blue color
    i = selected_job;
    ofDrawRectangle(25 + (i - 1) * 50, 660, 40, 30);

    for (j = 1; j <= num_of_can; j++) {
        if (candidate[j].id == schedule[i]) {
            ofDrawRectangle(20 + (i - 1) * 50, 40 + (j - 1) * 31, 50, 30);
        }
    }
}
```

'->' key를 눌렀다가 떼면 timeline 위에서 오른쪽으로 한 칸 씩 움직이면서, timeline 위에 놓인 사각형들이 하늘색 사각형으로 대체되면서 찾고자 하는 job을 targeting할 수 있게 해준다. 반대로 '<-' key를 눌렀다가 떼면 timeline 위에서 왼쪽으로 한 칸 씩 움직이면서 targeting한다. 만일 slot이 비어 있는 곳에 도달하면 아래에 첨부한 사진처럼 timeline 바로 위에만 표시되고 윗 부분의 막대 그래프에는 대응되는 것이 없기 때문에 아무런 변화가 일어나지 않는다.



6) 'q' key

```
if (key == 'q') {
    if (!load_flag) return;

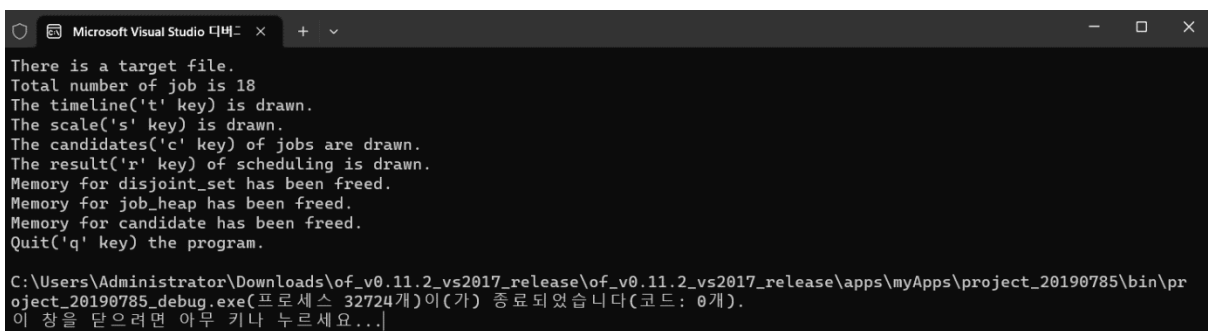
    //Reset the flags
    load_flag = 0;
    draw_timeline_flag = 0;
    draw_scale_flag = 0;
    draw_candidate_flag = 0;
    draw_result_flag = 0;
    num_of_job = 0;
    write_id_flag = 0;
    draw_target_flag = 0;

    //Free the dynamically allocated memory
    if (disjoint_set != NULL) {
        free(disjoint_set);
        disjoint_set = NULL;
    }
    cout << "Memory for disjoint_set has been freed." << endl;
    if (job_heap != NULL) {
        free(job_heap);
        job_heap = NULL;
    }
    cout << "Memory for job_heap has been freed." << endl;

    if (candidate != NULL) {
        free(candidate);
        candidate = NULL;
    }
    cout << "Memory for candidate has been freed." << endl;

    cout << "Quit('q' key) the program." << endl;
    _Exit(0);
}
```

'q' key가 눌리면 모든 flag들을 0으로 초기화시키고, 앞선 과정에서 동적 할당 받았던 모든 메모리들을 free해주면서 종료한다. 종료 이후에는 아래 사진과 같이 콘솔 창에 메모리가 정상적으로 해제되었음을 알리는 문구와 함께 창이 닫힌다.



```
Microsoft Visual Studio 디버그 콘솔
There is a target file.
Total number of job is 18
The timeline('t' key) is drawn.
The scale('s' key) is drawn.
The candidates('c' key) of jobs are drawn.
The result('r' key) of scheduling is drawn.
Memory for disjoint_set has been freed.
Memory for job_heap has been freed.
Memory for candidate has been freed.
Quit('q' key) the program.

C:\Users\Administrator\Downloads\of_v0.11.2_vs2017_release\of_v0.11.2_vs2017_release\apps\myApps\project_20190785\bin\project_20190785_debug.exe(프로세스 32724개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3-2. 시간 및 공간 복잡도

프로그램의 공간 복잡도는 사용된 자료구조의 크기를 기반으로 분석함으로써 알아낼 수 있다. 본 프로그램에서 사용한 자료구조로는 `Node* disjoint_set`, `char schedule[21]`, `Job* job_heap`, `Job* candidate`이 있다. `disjoint_set`은 `sizeof(Node)*21`만큼, `schedule`은 `sizeof(char)*21`만큼으로 고정된 값이다. 하지만 `job_heap`과 `candidate`은 file에서 입력 받은 job의 개수에 따라 달라진다. `job_heap`과 `candidate` 모두 `sizeof(Job)*(num_of_job)`만큼의 공간을 필요로 한다. 이 job의 개수를 n 이라고 하면, 본 프로그램의 공간 복잡도는 $O(n)$ 이라고 볼 수 있는 것이다.

본 프로그램에서 사용된 주된 알고리즘은 disjoint set 알고리즘이다. 이 알고리즘은 우선 heap sort로 정렬하는 데 $O(n\log n)$ 만큼의 시간 복잡도를 필요로 하고, 이후에 Find와 Union operation을 하는 데 $O(n\log m)$ 만큼의 시간 복잡도를 필요로 하게 된다. 여기서 m 은 n 과 job들의 deadline 중 최댓값 중 작은 값을 의미한다. 하지만, 본 프로그램에서는 deadline은 이미 20으로 제한을 둔 상태를 전제로 진행하기 때문에 $O(n\log m)$ 이 아니라 $O(n\log n)$ 으로 봐도 무방하다. 그리고 candidate 배열을 정렬할 때 사용한 삽입 정렬은 평균적으로는 $O(n^2)$ 의 시간 복잡도를 지니지만, 앞서 서술했듯이 candidate의 데이터가 어느 정도 정렬이 되어 있기 때문에 $O(n)$ 의 시간 복잡도를 지닌다고 봐도 무방하다. 따라서, 최종적인 본 프로그램의 시간 복잡도는 $O(n\log n)$ 이다.

4. 느낀 점 및 개선 사항

이론적으로만 알고 있던 disjoint set 알고리즘을 실제로 코드로 구현해보니 생각했던 점과 달랐던 점들이 많았다. 대표적으로 disjoint set의 각 노드들이 이론적으로는 편리한 이해를 위해 각 노드가 트리의 노드와 같은 형태라고 간주하고 그림을 그려가며 이해했기 때문에 tree의 자료구조를 지녀야 할 것 같았지만 실제로는 아니었다. 각 노드의 부모와 랭크 정보만 알면 되었기 때문에 굳이 트리로 구현할 필요는 없었다. 오히려 트리로 구현할 때보다 배열로 구현하면 자식 노드 포인터에 대한 관리를 안 해도 되기 때문에 더욱 간편하게 Union과 Find 함수를 구현할 수 있었다.

'find' 함수와 'union' 함수를 이용하는 disjoint set 알고리즘은 최소 신장 트리(MST)를 구하는 데에도 사용될 수 있고, 동치류(equivalence class)를 구하는 데에도 사용될 수 있는 만큼 중요도가 높다. 이처럼 중요도가 높은 disjoint set 알고리즘을 job scheduling 문제에 적용시켜 해결함으로써, 해당 알고리즘에 대한 이해도를 높일 수 있었다. 또한, disjoint set을 정렬하는 데 heap sort를 사용하고, 선택할 job 후보를 정렬하는 데 insertion sort도 사용하였기에 sorting algorithm에 대한 이해도 향상시킬 수 있었다. 또한, 코드로 구현한 결과물을 사람이 이해하기 쉬운 형태로 시각화하는 과정을 배울 수 있었다. 확실히 코드에 대한 출력 결과물로 숫자와 글자만 있을 때보다는, 도형으로 시각화하여 나타낸 결과물이 훨씬 이해하기에도 분석하기에도 용이하다는 것도 다시 한번 깨달을 수 있었다.