가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA Computer SCIENCE

자료구조 - 201921725 안성현

버스 탑승 스케줄링

<2021/04/05>

문제 소개 및 접근법

1-1] 버스 문제

① 문제

사람들에게 서울역에서 목장까지 이동하는 버스편을 제공하기로 한다. 사람들은 총 n명, 버스는 모두 m대, 그리고 버스 한 대에는 최대 c명까지 탑승할 수 있다.
각 버스는 최대 c명까지 탑승한 후 목장으로 출발한다. 버스에 탑승한 참가자는 버스가 출발할 때까지 오랜 시간을 기다려야 할 수도 있다.
이 때, 참가자들이 버스에 탑승한 후 버스가 출발할 때까지 기다리는 시간의 최대값을 최소화하라. (단 n<=mc, c<=n은 보장한다)

ex) 6 3 2 / 1 1 10 14 4 30이 입력이면 4가 출력이 된다. 버스A[1,1] 버스B[3,4] 버스C[10,14]로 탑승 시켰을 때보다 적은 최대 대기 시간(최대값)은 존재하지 않는다.

② 접근법

▶ 사람들이 서울역에 오는 시각은 정해져있다. 버스 관계자는 m개의 버스를 가지고 모든 사람들이 최대한 적게 대기하도록 만들고 싶다. 그래서 '최대 대기 시간'을 임의로 정하고 그 시간 안에 사람들이 버스를 나눠 탈 수 있는지 조사한다. 여기서 '최대 대기 시간'은 (마지막에 오는 사람의 시각)-(처음 오는 사람의 시각)으로, 0~(가장 늦은 시각) 사이에 존재한다. 만약 만족한다면 '최대 대기 시간'을 낮춰서 다시 나눠 탈 수 있는지 조사한다. 이 작업을 반복해서 더이상 사람들이 나눠 탈 수 없을 때까지 조사한다. 그럼 나눠 탈 수 없기 바로 전의 '최대 대기 시간'이 최적의 대기 시간이 될 것이다.
'최대 대기 시간' 개념을 사용하니까 가정이 필요하다. '최대 대기 시간'을 초과해서 사람이 오면 앞 버스는 가버린다는 점이다. 즉 m대의 버스로 사람들을 모두 탑승시키에는 충분하지만, 누군가 너무 늦게 온다면 m대를 다 놓쳐버릴 수 있다. 위 사항을 고려해서 코드를 작

성해야 한다.

ex) 사람2명,버스1대,좌석2개일 때 최대 대기 시간이 5이고 사람들이 (10,20)에 각각 버스 타러 온다. -> 10분기다릴 수 없으니 사람(10)만 버스타고 출발한다.

위의 경우, 2대의 버스가 와야지 최대 대기 시간 5를 만족한다. -> 1대의 버스로 처리하기 위해서는 최대 대기 시간을 늘려서 처리해야 될 것이다. -> 10이 최적의 대기시간이 됨

소스 코드와 알고리즘 설명

2-1] 서울역에 온 사람들의 시각 정렬

① 코드

```
void swap_elements(int* list, int index1, int index2) {
        int temp = list[index1];
        list[index1] = list[index2];
        list[index2] = temp;
}

int partition(int* list, int start, int end) {
        int i, b;
        i = b = start;
        int p = end;

        while (i < p) {
                if (list[i] <= list[p]) {
                        swap_elements(list, i, b);
                        b += 1;
                }
                i += 1;
        }
        swap_elements(list, b, p);
        p = b;

        return p;
}

int quick_sort(int* list, int start, int end) {
        if (end - start < 1)
                return 0;

        int p = partition(list, start, end);
        quick_sort(list, start, p - 1);
        quick_sort(list, p + 1, end);
}
```

② 알고리즘 설명

▶ 대기 시간은 (늦게 온 사람-빨리 온 사람)이므로 음수가 발생하면 안 된다. 따라서 정렬이 필요하다.
정렬하지 않으면 빨리 온 사람과 늦게 온 사람을 쉽게 구분하기가 힘들다.
또한 이후에 사용할 탐색 알고리즘의 성능에도 영향을 끼친다.
필자는 평균 시간복잡도가 O(nlogn)을 만족하는 '퀵 정렬'을 사용하였다.

2-2] 첫 번째 '최대 대기 시간' 구하기

① 코드

```
if ((n % c) == 0) { // n=m*c, n<m*c (n = k*c)
                if ((m * c - n) >= c) // ----- (m-α) buses can go
                        M = n / c;
                for (i = 0; i < M; i++) {
                        tmp = time_list[idx + (c - 1)] - time_list[idx];
                        idx += c;
                        //find max
                        if (tmp >= fmd)
                                fmd = tmp;
                }
        }
        else { // n<m*c (n != k*c)
                if ((m * c - n) >= c) // ----- (m-α) buses can go
                        M = (n / c) + 1;
                for (i = 0; i < M; i++) {
                        if (i == M - 1) {
                                tmp = time_list[idx + ((n % c) - 1)] - time_list[idx];
                        }
                        else {
                                tmp = time_list[idx + (c - 1)] - time_list[idx];
                                idx += c;
                        }
                        //find max
                        if (tmp >= fmd)
                                fmd = tmp;
                }
        }
```

② 알고리즘 설명

▶ 최대 대기 시간을 처음 정할 때는 버스를 최대한 낭비하지 않고 사람들을 태웠을 때를 기준으로 한다.
[사람들을 버스에 정원으로 채워도 남는 자리 >= 버스의 좌석 자리] [(m * c - n) >= c]는 버스가 낭비되는 경우이다. 이럴 경우 m을 일시적으로 변경한다.

ex) 8 /2 /4 이고 [1,2,3,4,5,6,7,8]이면 무조건 [1234/5678]로 태움 -> 최대 대기 시간 3

ex) 8 /3 /4 이고 [1,2,3,4,5,6,7,8]이면 무조건 [1234/5678]로 태움 -> 최대 대기 시간 3
<나머지 한 버스는 무시하기 위해 8/2/4로 바꿔서 풀었음>
ex) 8 /3 /3 이고 [1,2,3,4,5,6,7,8]이면 무조건 [123/456/78]로 태움 -> 최대 대기 시간 2
ex) 8 /5 /3 이고 [1,2,3,4,5,6,7,8]이면 무조건 [123/456/78]로 태움 -> 최대 대기 시간 2
<나머지 두 버스는 무시하기 위해 8/3/3으로 바꿔서 풀었음>

위 예시에 의하면 2를 '첫 번째 최대 대기 시간'으로 정하게 된다. 이제 (0~2)의 범위에서 최적의 '최대 대기 시간'을 찾으면 된다. 이렇게 미리 정해놓지 않으면 '최대 대기 시간'을 만족하는지 비교하는 횟수가 훨씬 늘어날 것이다.

2-3] 최적의 '최대 대기 시간' 탐색하기

① 코드

```
int binary_search_compare(int* list, int first, int last, int fmd) {

	if ((last == 0) || (first == fmd)) // 처음 값 혹은  마지막 값 처리 (최종의 경우), 조건문에서 생략했지만 fisrt=last임
		return first; // 최종의 경우까지 오면 나올 수밖에 없음

	int mid = (first + last) / 2;

	int bus_mid = bus_num(list, mid); // 최대 대기시간(max_diff)이 mid일 때, bus값

	if (mid == 0) { // first:0, last:1이어서 mid가 0이면 'Main Part'의 연산이 불가능 -> 따로 처리
		if (bus_mid <= m) // max_diff가 0일 때 조건 만족 -> 0이 최적의 최대 대기시간
			return 0;
		else
			return 1;
	}

	// Main Part
	if (bus_mid <= m) { // max_diff가 0이 아닌 mid일 때, bus<=m [현재 조건 만족(적거나 같음)]
		if (bus_num(list, mid - 1) > m) //  max_diff가 mid일 때, bus>m [바로 전일 때 조건 불만족]
			return mid; // best_time은 mid

		else //  [바로 전일 때 조건 만족]
			binary_search_compare(list, first, mid - 1, fmd); // '이전 값들 중 하나'에서 재조사
	}
	else // max_diff가 0이 아닌 mid일 때, bus>m [현재 조건 불만족]
		binary_search_compare(list, mid + 1, last, fmd); // '다음 값들 중 하나'에서 재조사

}
```

② 알고리즘 설명

▶ 어떠한 것이 최적의 '최대 대기 시간'인지 알아보기 위해 간단한 로직을 알아 보겠다.

범위 [0 1 2 3 4 5 6 7 8 9 10] 중 [최대 대기 시간:0]일 때 만족한다면 1은 조사할 필요도 없이 만족한다.
ex) 사람 두 명이 같은 시각에 도착해서 정원이 두 명인 버스를 탔다. 사실상 그냥 출발해도 상관이 없다. 즉 (최대)대기 시간이 0일때도 아무 상관이 없다. 그런데 기사님이 1분뒤에 출발한다고 한다. (최대)대기 시간이 1이 된 것이다. 그냥 출발하나 1~10분을 기다리고 출발하나 이 사람들은 항상 버스를 탑승하는 조건에 만족한다.

즉 k일 때 만족한다면 k+1일 때도 만족한다.
이를 바꿔서 생각하면 p일 때 만족하고 p-1일 때 만족하지 않는다면 p가 '최적의 최대 대기 시간'이 된다.
ex) [0:불만족, 1:불만족, 2:불만족, 3:불만족, 4:만족, 5:만족, ...]라면 4가 최적의 '최대 대기 시간'

최적의 '최대 대기 시간'은 (0~10)에서 찾고 3이 정답이라고 가정해보겠다. 만약 10에서 1씩 줄어서 만족하는지 찾는다면 8번을 찾게 된다. 하지만 이분 탐색을 이용하면 3번만에 찾을 수 있다. 실제로 순차 탐색과 이분 탐색의 시간 복잡도를 비교하면 O(n)과 O(logn)의 차이가 발생한다. 만일 범위가 (0~1,000,000,000)이였다면 999,999,998번과 29번의 차이라는 의미이다. 따라서 필자는 **이분 탐색 알고리즘을 응용**해서 문제를 풀었다.

0. 이분 탐색 알고리즘 중 mid를 최대 대기 시간으로 임의 설정한다.

1. mid가 최대 대기 시간일 때, 조건을 만족하면 mid-1이 최대 대기 시간일 때도 만족하는지 조사한다.
둘 다 만족한다면 mid는 최적의 최대 대기 시간이 아니다.
이 때는 last를 (mid-1)로 만들어서 mid보다 적은 범위의 수를 살펴본다.
<'불만족-만족'의 경우는 앞쪽에 위치하므로>
ex) [0:?, 1:?, 2:?, 3:?, 4:만족, 5:만족, ...]일 때 mid가 5일 경우

2. mid가 최대 대기 시간일 때, 조건을 만족하고 mid-1이 최대 대기 시간일 때는 불만족한다면 mid를 반환한다. <정답 찾음>
ex) [0:?, 1:?, 2:?, 3:불만족, 4:만족, 5:?, ...]일 때 mid가 4일 경우

3. mid가 최대 대기 시간일 때, 조건을 불만족한다면 first를 (mid+1)로 만들어서 mid보다 큰 범위의 수를 살펴본다.
<일단 mid가 만족하는 경우를 찾아야 함, '만족'은 항상 '불만족' 뒤에 있음>
ex) [0:?, 1:?, 2:?, 3:?, 4:?, 5:?,6:불만족 ....]일 때 mid가 6일 경우

예외 처리) mid가 0이 오면 mid-1이 -1이 돼서 stack over flow가 발생한다. 이때는 0일 때 조건을 만족하면 0리턴, 아니면 1을 반환한다.
<first가 0,last가 1이므로 둘 중 하나가 답>

종료 조건) last=0일 때, first=fmd(첫 번째 최대 대기 시간)일 때는 최종의 경우이다. 최종의 경우까지 온다면 답이 0 or fmd일 수밖에 없다.
ex) [0:만족, 1:만족, 2:만족, 3:만족, 4:만족, 5:만족, ...]일 때 last가 0일 경우
ex) [0:불만족, 1:불만족, 2:불만족,....9:불만족, 10:만족]일 때 first가 10일 경우

2-4] 조건 만족 확인하기

① 코드

```
int bus_num(int* list, int max_diff) {
	int a = 0; // list[a]~list[b]는 현재 버스에 탈 수 있을지 조사할 때 필요한
사람들(시각), list[a]~list[b-1]은 현재 버스에 탄 사람들
	int b = 1;
	int bus = 1;

	while (b < n) { // list[n-1]까지 존재함
		if (list[b] - list[a] <= max_diff) { // 대기시간이 최대 대기시간
이하이다.
			if ((b - a + 1) > c) { // 사람 수가 좌석 수에 불만족
				bus++; // 버스 늘림
				a = b;	// 다음 버스 처리 시작
			}
		}
		else { // 대기시간이 최대 대기시간 초과이다.
			bus++; // 버스 늘림
			if (bus > m) // 최대 버스 수 초과
				break; // 버스 조사 끝냄
			a = b; // 다음 버스 처리 시작
		}
		b++; // 사람 태움, 다음 경우 조사를 위해 b를 1증가
	}
	return bus;
}
```

② 알고리즘 설명

▶ 이제 조건을 만족하다는 것이 무슨 의미인지 알아 보겠다.

max_diff(최대 대기 시간)가 1이고 사람들이 서울역에 오는 시각은 [1,2,2,6]이고 버스의 수와 정원은 (3,2)라고 가정하겠다.
(1,2)는 1차이가 나니까 같은 버스에 탑승시킬 수 있다. 고로 (1,2)는 버스1에 탑승시킨다. 그리고 다음 사람도 2에 도착한다.
그러면 (1,2)는 다시 1차이가 나지만 정원이 2명이니까 사람(2)는 다음 버스(버스2)에 탑승시킨다.
마지막 사람은 6에 도착하는데 (2,6)은 4차이가 나니까 max_diff를 만족하지 못한다. 따라서 사람(6)을 다음 버스(버스3)에 탑승시킨다.
결론은 [1,2/2/6]으로 사람들을 세 버스에 탑승시키는 것에 성공했다. 이런 경우 조건을 만족한다고 말한다.

조건을 불만족하는 경우도 알아 보겠다.

max_diff(최대 대기 시간)가 3이고 사람들이 서울역에 오는 시각은 [1,7,11]이고 버스의 수와 정원은 (2,2)라고 가정하겠다.
(1,7)은 6차이가 나니까 max_diff를 만족하지 못한다. 따라서 사람(1)은 버스1에, 사람

(7)은 다음 버스(버스2)에 탑승시킨다.
(7,11)은 4차이가 나니까 역시 max_diff를 만족하지 못한다. 따라서 사람(11)은 다음 버스(버스3)에 탑승시킨다.
결론은 [1/7/11]으로 사람들을 두 버스에 탑승시키는 것에 실패했다. 이런 경우 조건을 불만족한다고 말한다.

이러한 사례들을 의사코드로 구성하면 다음과 같다.

```
while (모든 인원 조사)
        if (최대 대기시간 이하)
                if (탑승 시 좌석 수 초과)
                        버스 늘리고 다음 버스에 탑승 시킴
                else
                        현재 버스에 탑승 시킴
        else
                버스 늘리고 다음 버스에 탑승 시킴
                if (늘렸더니 버스 초과했다고 함)
                        < 조건 불만족 > 더이상 조사할 필요가 x
        다음 사람 조사
```

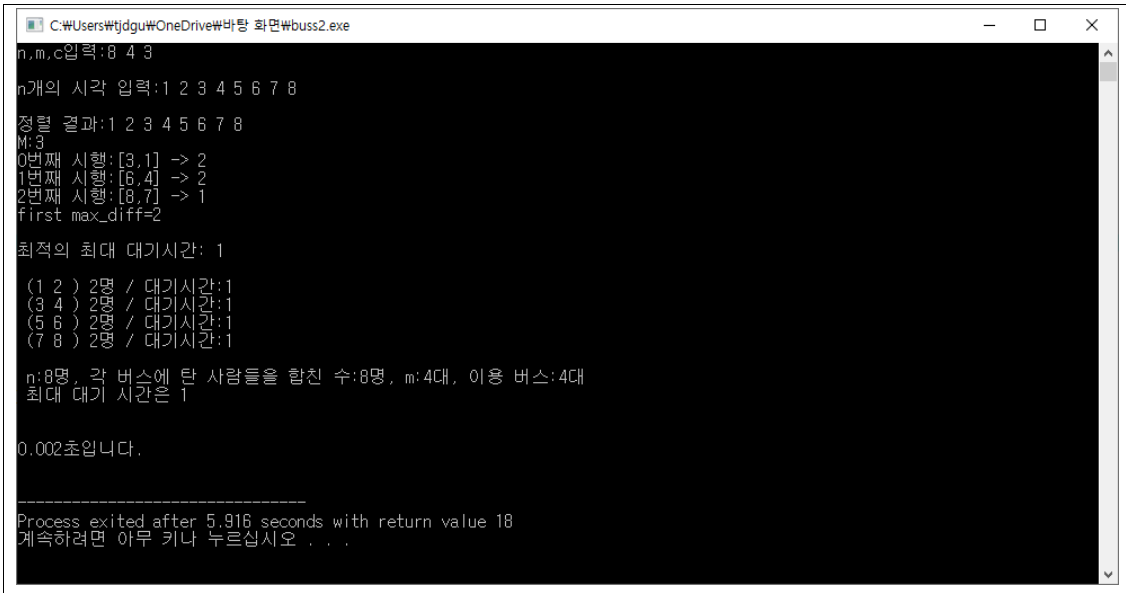
```
// n<=mc가 보장되기 때문에, if-if문에서 버스 늘려서 초과할 사건은 발생하지 않는다.
// ex) 3 1 2, [1,3,4] max_diff가 3이면 [1,3/4]형태가 된다. 이럴 경우 버스를 2대써서 불만족하는 것처럼 보이지만 애초에 n<=mc가 성립하지 않았다.

// n<=mc가 보장되어도 else문에서 버스를 늘려서 초과할 사건은 발생한다.
// ex) 3 1 3, [1,3,5] max_diff가 3이면 [1,3/5] 형태가 된다. 마지막 사람이 5가 아니라 4였다면 만족했겠지만 너무 늦게 와서 m대를 모두 놓쳐버린 경우이다.
```

실제 코드는 위 알고리즘을 기반으로 구성하였다.이분 탐색 알고리즘에서 위 알고리즘을 호출하다 보니 bus를 반환해서 <만족/불만족>을 판정한다.

실행 결과와 소요 시간

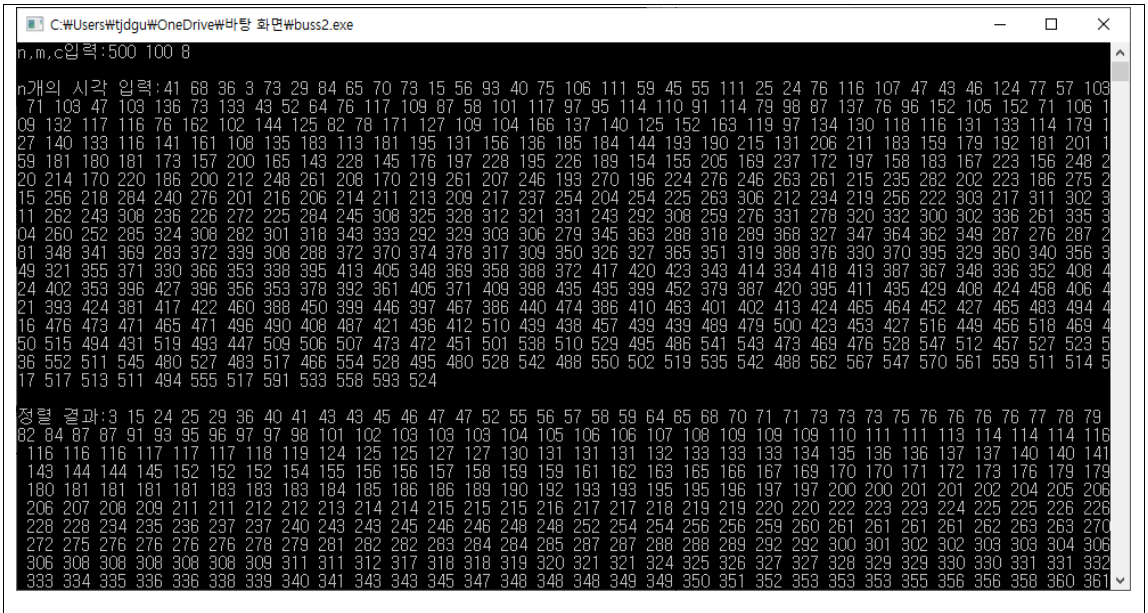
3-1] 실행 결과와 소요 시간

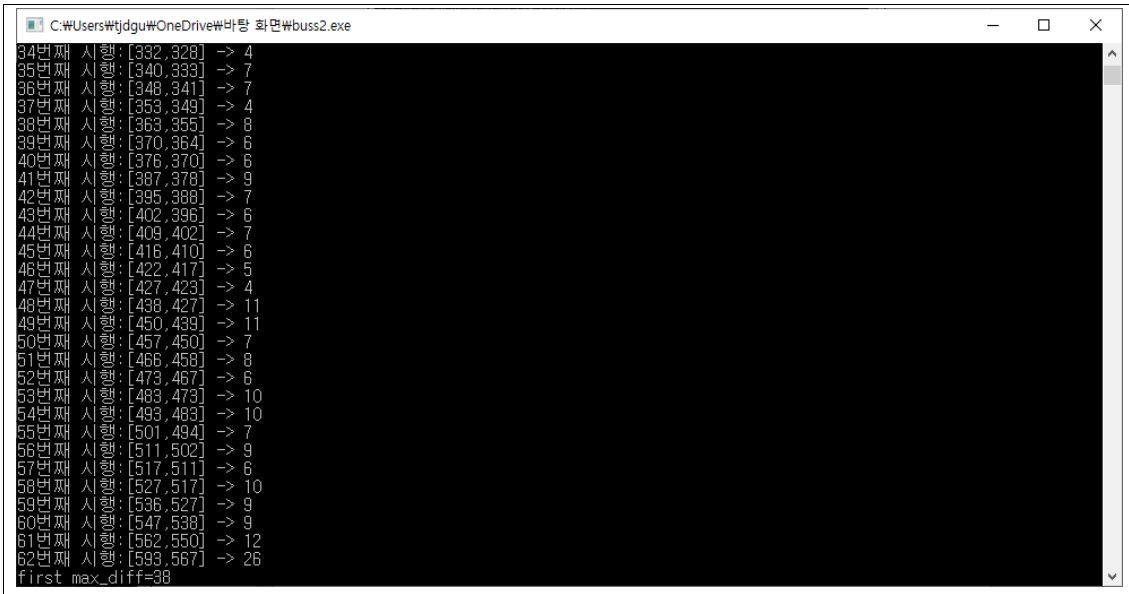
① DEV-C++ 컴파일러 이용 실행 결과 1

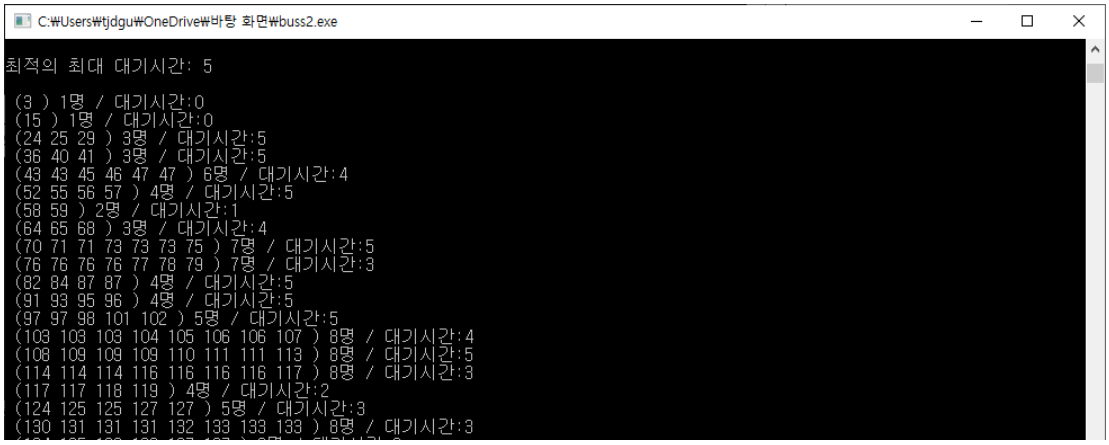


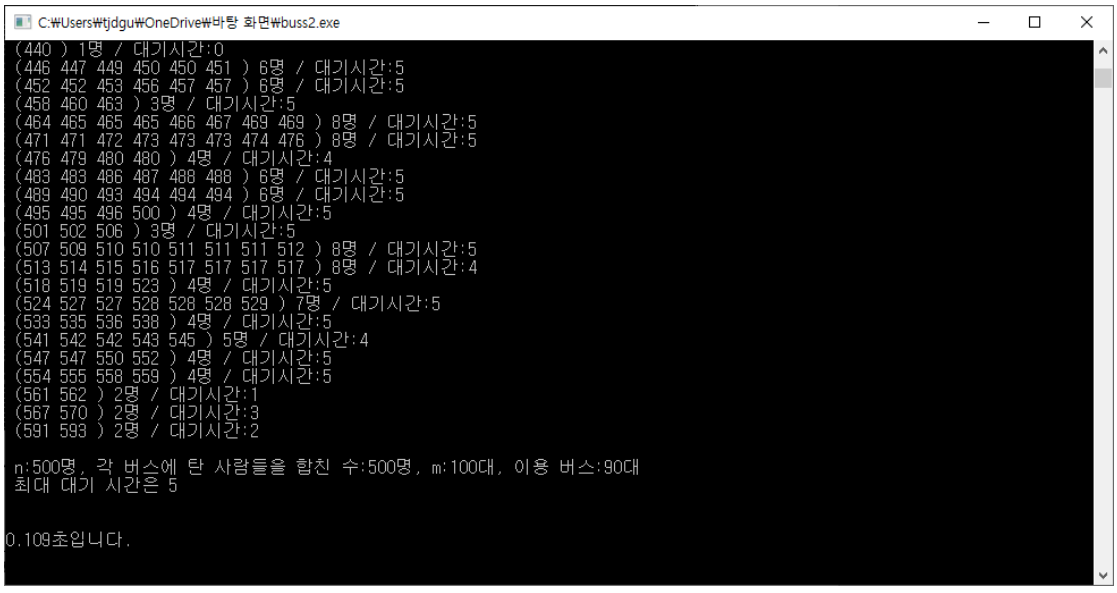
0.002초 (정렬~'최대 대기시간=1' 출력까지)

② DEV-C++ 컴파일러 이용 실행 결과 2



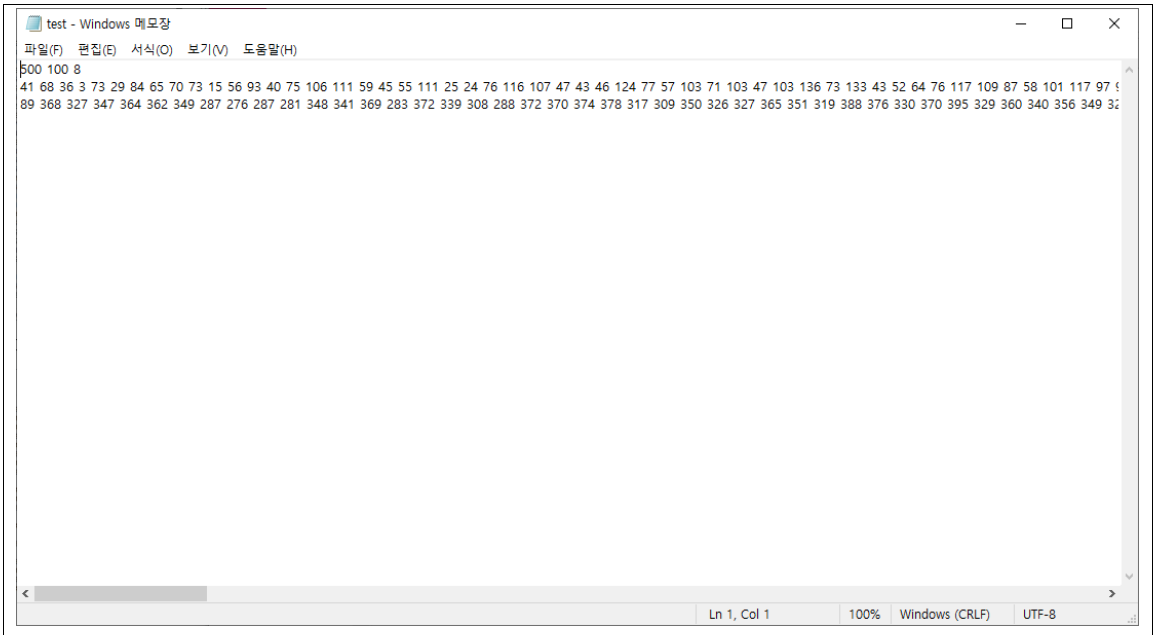




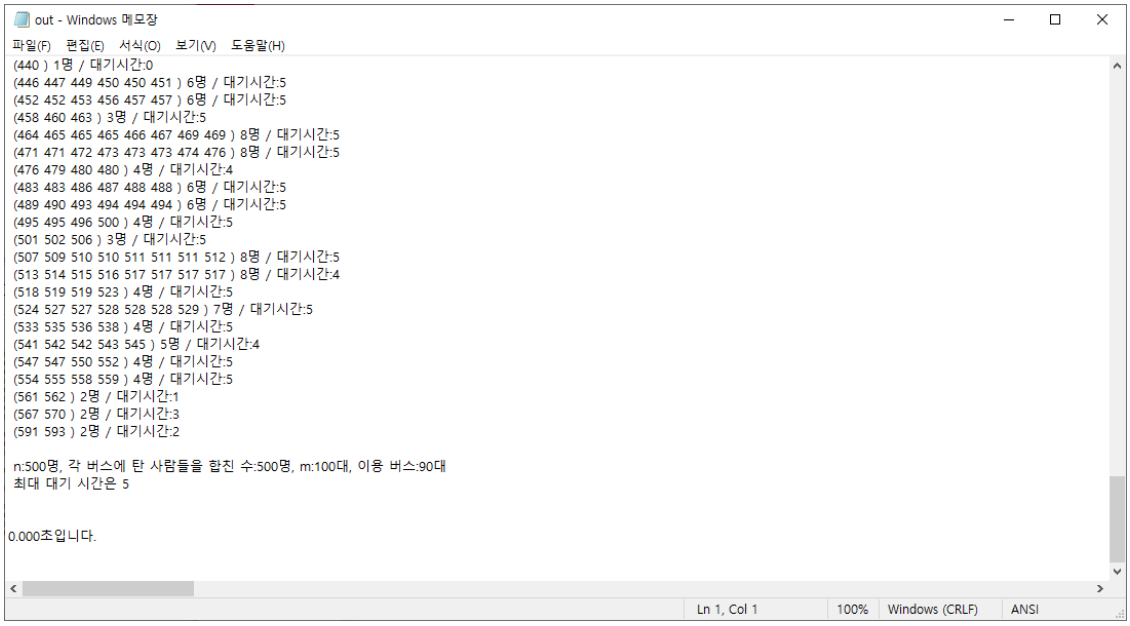


0.109초 (정렬~'최대 대기시간=5' 출력까지)

③ CMD 명령어 이용 TXT파일 출력 결과 1 (② 와 같은 입력)

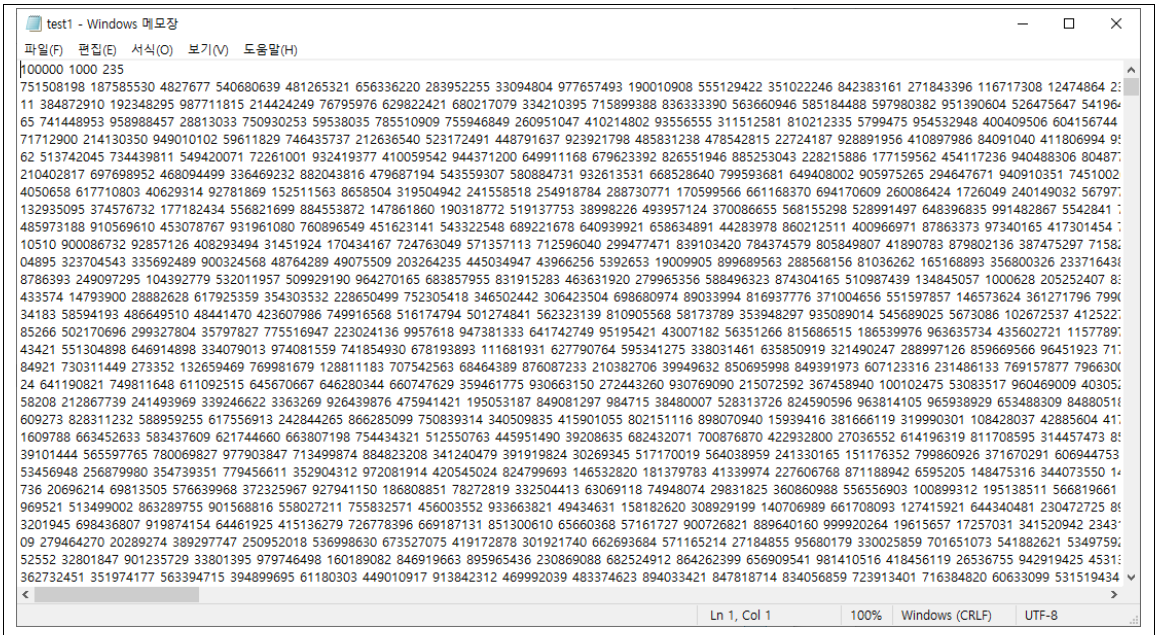


< test.txt > 입력용 텍스트 파일 (500개 데이터, 범위:3~593)



< out.txt > 출력 결과 / 0.000초(입력~'최대 대기 시간=5' 출력까지)

④ CMD 명령어 이용 TXT파일 출력 결과 2 (큰 수에 적용)



< test1.txt > 입력용 텍스트 파일 (100,000개 데이터, 범위:1만~9억9천)

< out.txt > 출력 결과 / 0.082초(입력~'최대 대기 시간=989499' 출력까지)

전체 코드

4-1] 버스 전체 코드

① 버스 분석 코드 제외한 코드

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

int n, m, c;

void swap_elements(int* list, int index1, int index2) {
        int temp = list[index1];
        list[index1] = list[index2];
        list[index2] = temp;
}

int partition(int* list, int start, int end) {
        int i, b;
        i = b = start;
        int p = end;

        while (i < p) {
                if (list[i] <= list[p]) {
                        swap_elements(list, i, b);
                        b += 1;
                }
                i += 1;
        }
        swap_elements(list, b, p);
        p = b;

        return p;
}

int quick_sort(int* list, int start, int end) {
        if (end - start < 1)
                return 0;

        int p = partition(list, start, end);
        quick_sort(list, start, p - 1);
        quick_sort(list, p + 1, end);
}

int bus_num(int* list, int max_diff) {
        int a = 0; // list[a]~list[b]는 현재 버스에 탈 수 있을지 조사할 때 필요한
사람들(시각), list[a]~list[b-1]은 현재 버스에 탄 사람들
        int b = 1;
        int bus = 1;
```

```
        while (b < n) { // list[n-1]까지 존재함
                if (list[b] - list[a] <= max_diff) { // 대기시간이 최대 대기시간
이하이다.
                        if ((b - a + 1) > c) { // 사람 수가 좌석 수에 불만족
                                bus++; // 버스 늘림
                                a = b;    // 다음 버스 처리 시작
                        }
                }
                else { // 대기시간이 최대 대기시간 초과이다.
                        bus++; // 버스 늘림
                        if (bus > m) // 최대 버스 수 초과
                                break; // 버스 조사 끝냄
                        a = b; // 다음 버스 처리 시작
                }
                b++; // 사람 태움, 다음 경우 조사를 위해 b를 1증가
        }
        return bus;
}

int binary_search_compare(int* list, int first, int last, int fmd) {

        if ((last == 0) || (first == fmd)) // 처음 값 혹은  마지막 값 처리 (최종의 경우),
조건문에서 생략했지만 fisrt=last임
                return first; // 최종의 경우까지 오면 나올 수밖에 없음

        int mid = (first + last) / 2;

        int bus_mid = bus_num(list, mid); // 최대 대기시간(max_diff)이 mid일 때, bus값

        if (mid == 0) { // first:0, last:1이어서 mid가 0이면 'Main Part'의 연산이 불가능
-> 따로 처리
                if (bus_mid <= m) // max_diff가 0일 때 조건 만족 -> 0이 최적의 최대
대기시간
                        return 0;
                else
                        return 1;
        }

        // Main Part
        if (bus_mid <= m) { // max_diff가 0이 아닌 mid일 때, bus<=m [현재 조건
만족(적거나 같음)]
                if (bus_num(list, mid - 1) > m) //  max_diff가 mid일 때, bus>m [바로 전일
때 조건 불만족]
                        return mid; // best_time은 mid

                else //  [바로 전일 때 조건 만족]
                        binary_search_compare(list, first, mid - 1, fmd); // '이전 값들
중 하나'에서 재조사
        }
        else // max_diff가 0이 아닌 mid일 때, bus>m [현재 조건 불만족]
                binary_search_compare(list, mid + 1, last, fmd); // '다음 값들 중
하나'에서 재조사
```

```
}


int main() {
	int i; // for iteration

	// input processing
	scanf("%d %d %d", &n, &m, &c);

	// find error
	if (n < c) {
		printf("[n<c] error");
		return 0;
	}
	if (m * c < n) {
		printf("[mc<n] error");
		return 0;
	}
	// end
	if (n <= 1 || c == 1) {
		printf("0");
		return 0;
	}

	int* time_list = (int*)malloc(sizeof(int) * n);

	for (i = 0; i < n; i++)
		scanf("%d", &time_list[i]);

	// sorting
	quick_sort(time_list, 0, n - 1);

	// make first max_diff (첫 번째 최대 대기시간(최대 대기차))
	int fmd = 0;
	int tmp = 0;
	int idx = 0; // index of time_list
	int M = m; // temp_m

	if ((n % c) == 0) { // n=m*c, n<m*c (n = k*c)
		if ((m * c - n) >= c) // ----- (m-α) buses can go
			M = n / c;
		for (i = 0; i < M; i++) {
			tmp = time_list[idx + (c - 1)] - time_list[idx];
			idx += c;
			//find max
			if (tmp >= fmd)
				fmd = tmp;
		}
	}
	else { // n<m*c (n != k*c)
		if ((m * c - n) >= c) // ----- (m-α) buses can go
			M = (n / c) + 1;
```

```
                for (i = 0; i < M; i++) {
                        if (i == M - 1) {
                                tmp = time_list[idx + ((n % c) - 1)] - time_list[idx];
                        }
                        else {
                                tmp = time_list[idx + (c - 1)] - time_list[idx];
                                idx += c;
                        }
                        //find max
                        if (tmp >= fmd)
                                fmd = tmp;
                }
        }

        // find best max_diff
        int best_wait = binary_search_compare(time_list, 0, fmd, fmd); //
(리스트,first,last,fmd)
        printf("%d", best_wait);

        return 0;
}
```