**알고리즘 HW.5**

2017114559 박선우

1. **문제 설명**

제한된 경로의 길이로 최대한 많은 점 방문하기

**Input**

* (고정값) 100 x 100 지도
* (고정값) 연료의 양은 6가지 200, 400, 600, 800, 1000, 1500
* (고정값) 시작 도시 좌표 (0, 0), 마지막 도시 좌표 (100, 100)
* (변수) 98개의 도시의 (x, y) 좌표, 1 <= x <= 99, 1 <= y <= 99

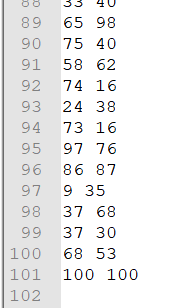
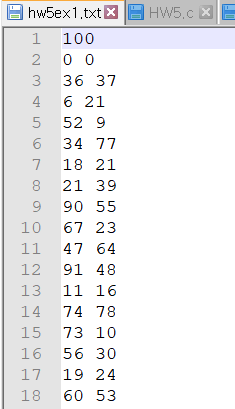
**Output**

* 200, 400, 600, 800, 1000, 1500 모두 출력

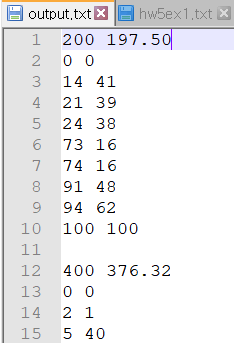
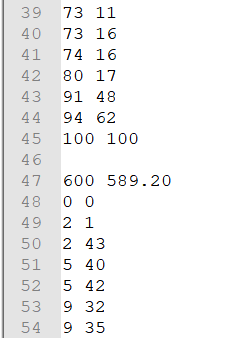
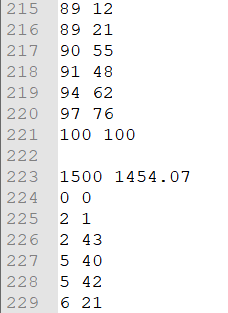
**Constraints**

* x축으로는 전진만이 가능
* x축 좌표가 같은 도시가 있을 수 있다. (방문순서 바꾸기 가능)
* (x, y) 좌표가 모두 같은 도시가 있을 수 있다. (복수로 센다)

**(입출력 예시)** Input



Output

1. **내가 구상한 알고리즘**
2. input data (x, y) 저장
3. (x, y) 데이터 x축 기준으로 정렬 (퀵소트 알고리즘 사용)
4. 각 점에서의 거리 이득 계산

점 n번째의 이득 (점 n을 뺄 경우 줄어드는 경로의 크기)

= 점(n-1 ~ n)까지의 거리 + 점(n ~ n+1)까지의 거리 – 점(n-1 ~ n+1)까지의 거리

1. 거리 이득이 큰 순으로 정렬 (퀵소트 알고리즘 사용)
2. 각 점들을 지나는 경로의 합이 주어진 연료보다 작아질 때까지

거리 이득이 큰 점부터 하나씩 제외하면서 경로의 합 계산

* + 방법 1) 점을 하나씩 뺄 때마다 남은 점들의 거리이득을 전부 다시 계산하기
  + 방법 2) 처음 계산한 거리이득을 가지고 계속 점을 제외하기

1. 방법 1)과 방법 2)의 결과를 비교하여 주어진 연료 내에서 더 많은 점을 방문한 결과를 출력.
2. **주요 함수 설명**

* **void quicksort ( int left, int right, int\* data1, int\* data2 )**

: 퀵소트 알고리즘을 사용하여 인덱스 범위 left ~ right 내에서 데이터1의 값을 비교한 뒤 오름차순으로 데이터 1과 2 모두 정렬한다, (x, y) 정렬에 사용

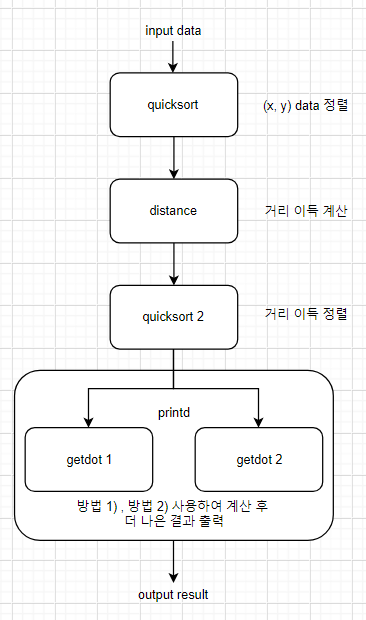
* **void quicksort2 ( int left, int right, float\* data1, int\* data2)**

: 위의 퀵소트 함수와 동일하지만 데이터 1의 자료형을 실수로 받기 위해 변경했다.

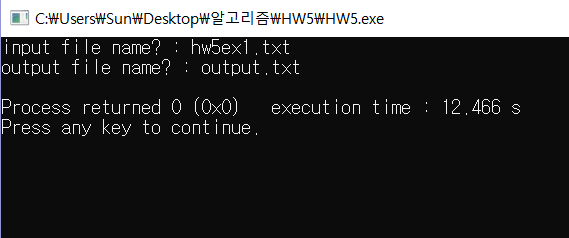
거리 이득 데이터 ( d )와 인덱스 배열을 정렬하기 위해 사용

* **float distance ( int i, int j )** : 전역변수 x, y 의 i와 j번째 인덱스에 위치한 두 점 사이 거리를 계산하여 반환한다
* **float distance2 ( int i, int j )** : 위와 동일하나 전역변수 x2, y2 에 저장된 점의 거리 계산
* **void getdot1 ( float dsum )** : 점들을 연결한 경로의 합이 주어진 연료 dsum보다 큰 경우 거리 이득이 가장 큰 점을 제외한다. 점을 제외할 때 마다 점들의 인덱스와 거리 이득을 다시 계산하여 업데이트 한다.
* **void getdot2 ( float dsum )** : 위의 getdot1 함수와 비슷하지만 초기에 구한 거리 이득만을­­­­­ 가지고 업데이트 없이 점을 제외한다.
* **void printd ( float dsum )** : getdot1과 getdot2함수를 실행한 후 두 결과를 비교하여 더 나은 결과를 아웃풋 파일에 출력한다.

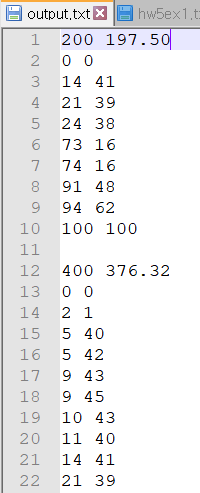
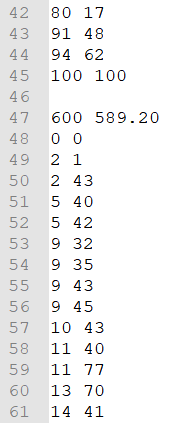
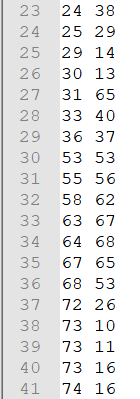
**(주요 함수의 흐름도)**



1. **실행 결과**

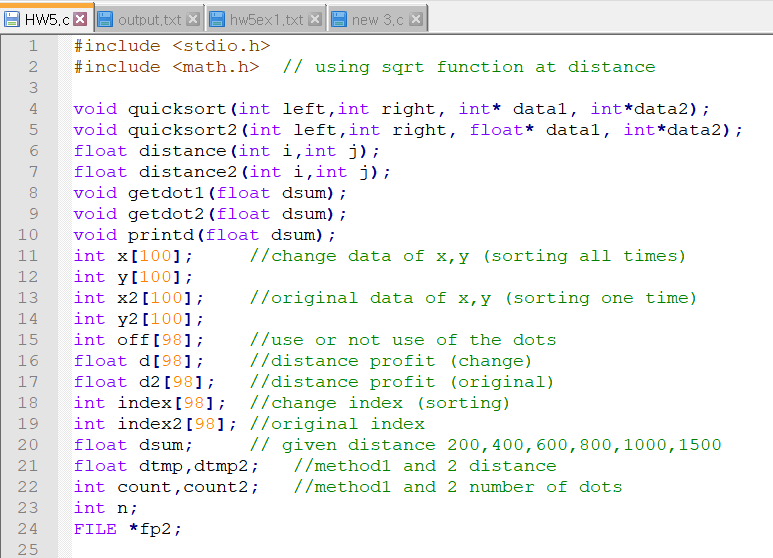


**(Output 파일)**

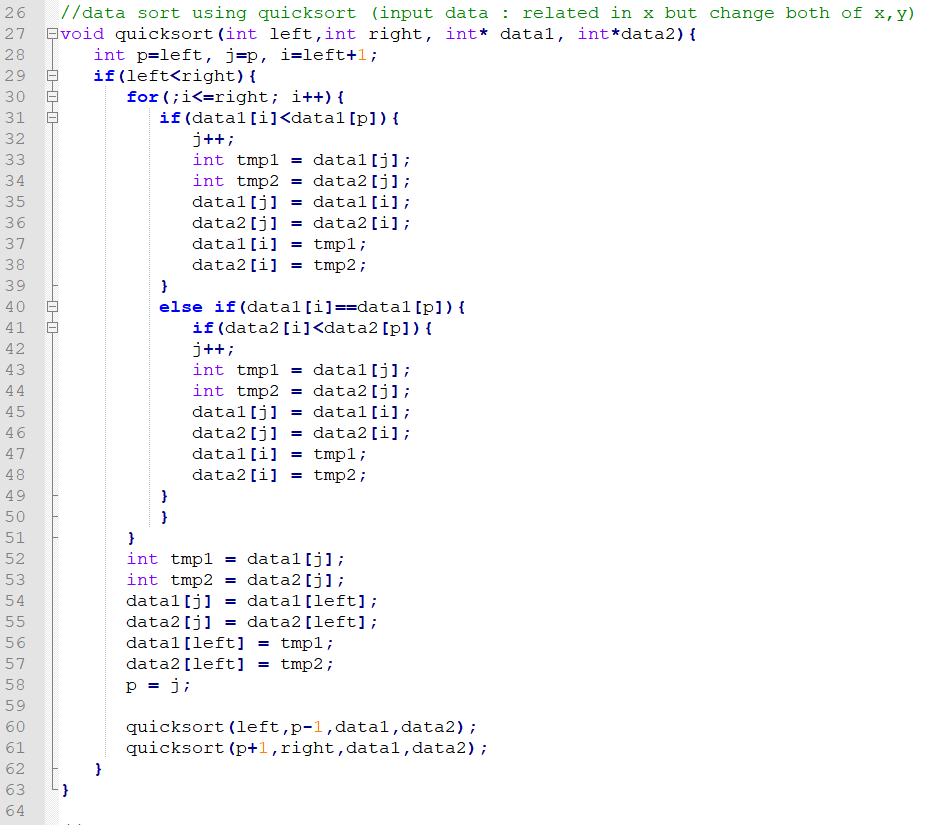
 

1. **프로그램 소스 코드**

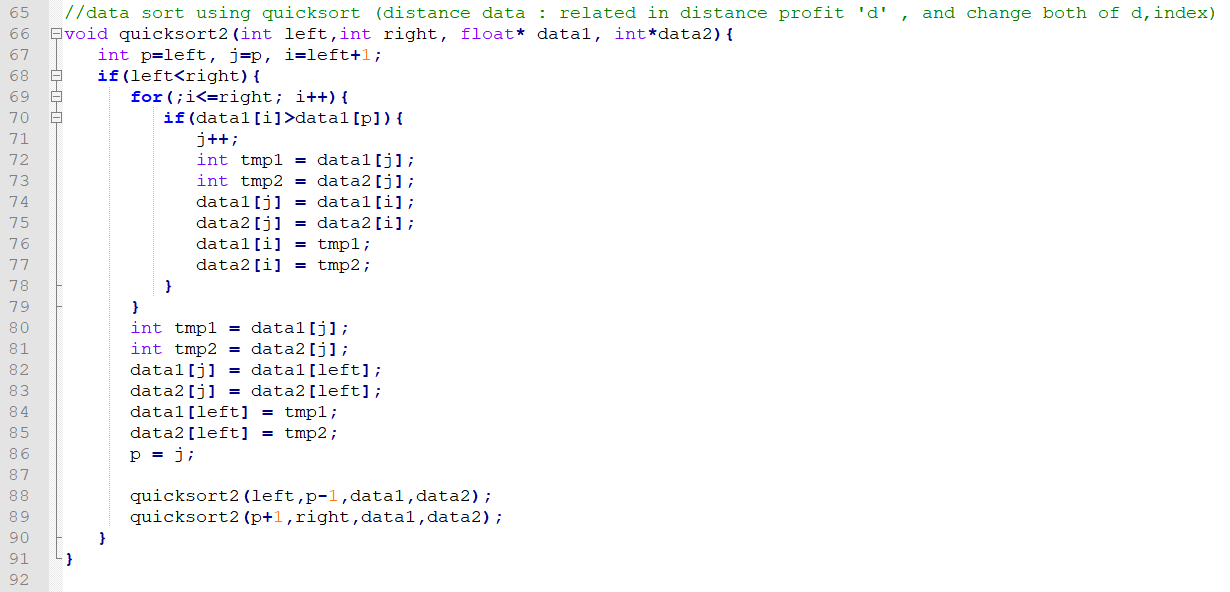
* **함수,변수 선언**



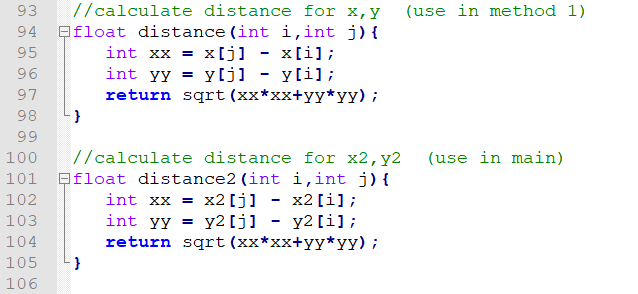
* **Quicksort 함수 (stable sorting이 되도록 코드를 수정하였다.**



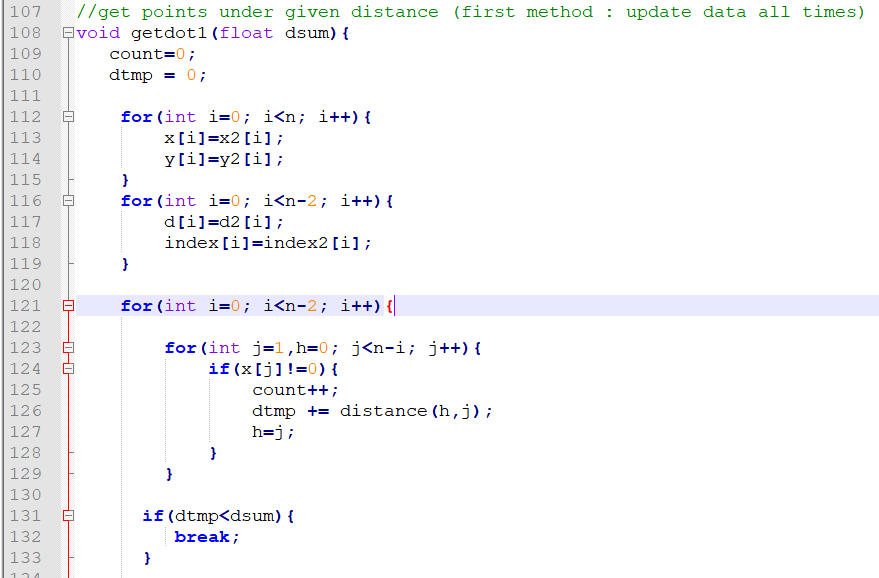
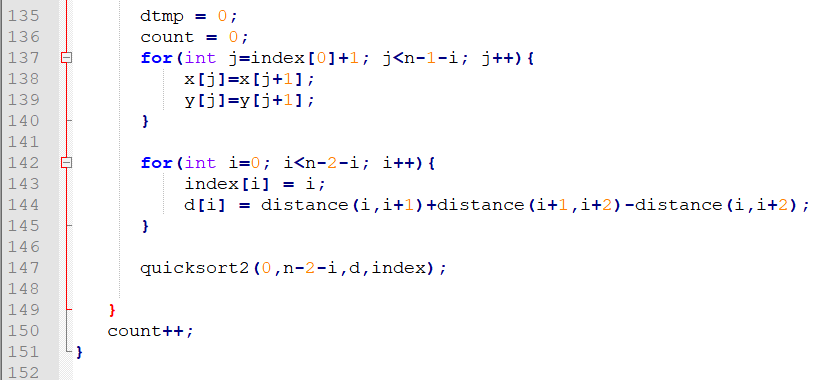
* **Quicksort 함수 2**



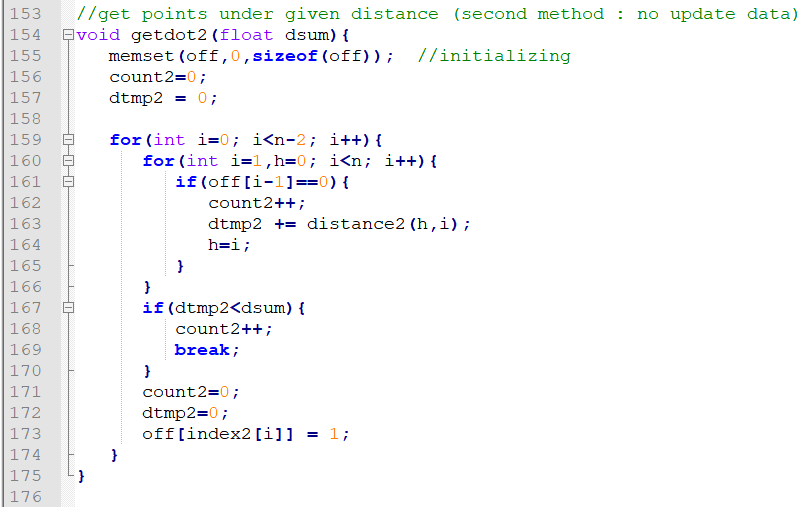
* **Distance 함수**



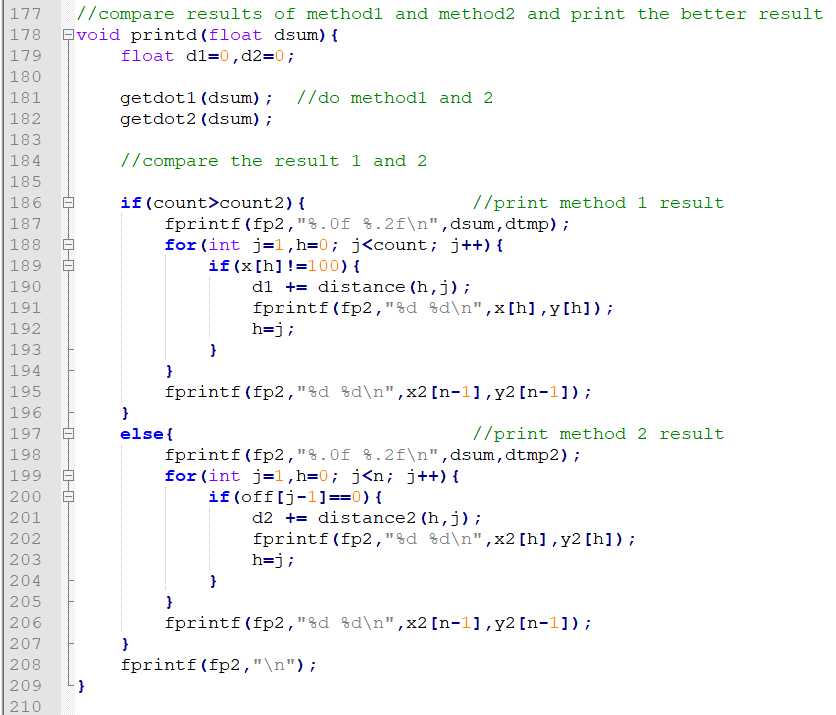
* **Getdot1 함수 (거리이득을 계속 업데이트 하는 방법)**

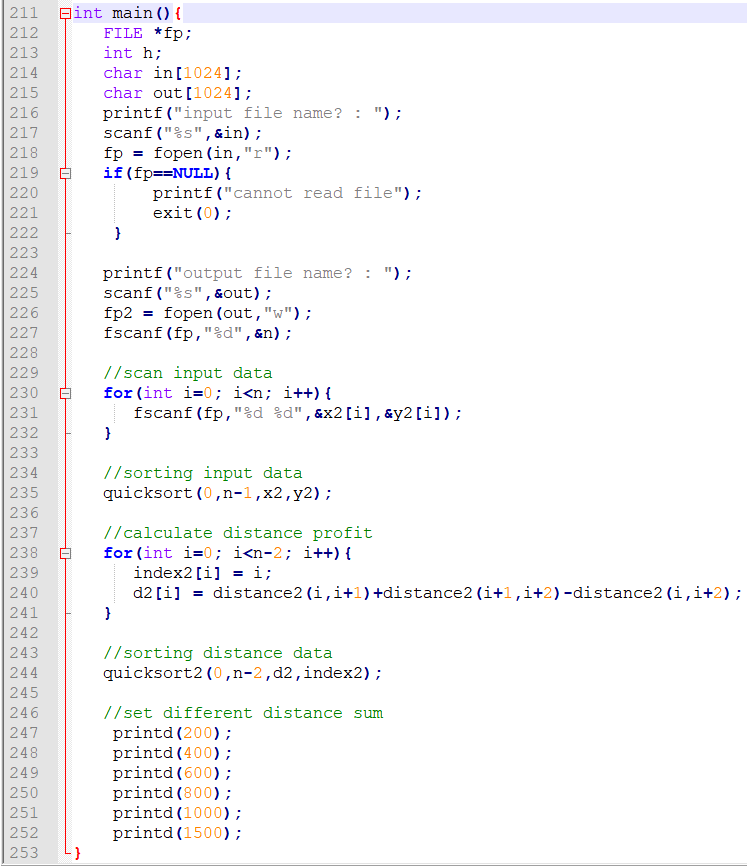
* **Getdot2 함수 (거리이득 초기값 계속 사용)**



* **Printd 함수**



* **Main 함수**



1. **고찰**

이번 과제를 진행하면서 최적의 결과를 찾는 방법을 많이 고민해 보았지만 굉장한 방법을 생각해 내기는 힘들었다. 그래도 내가 해볼 수 있는 방법부터 도전해 보기로 마음먹고 교수님께서 언급하신 그리디 알고리즘부터 생각해 보고 코드를 짜 보았다.

가장 먼저 생각한 방법이 거리 이득을 계산하여 이득이 가장 큰 순서대로 점을 하나씩 제외시키는 **2번 방법 (getdot2 함수)**이었는데 실행시켜 보니 주어진 연료보다 작은 값의 경로를 가지는 점들을 결과로 잘 나타내긴 하였지만, 이 방법은 처음 계산한 거리 이득만을 기준으로 삼기 때문에 점을 제외함으로써 바뀌는 거리 이득을 고려하지 못하는 문제가 있었다.

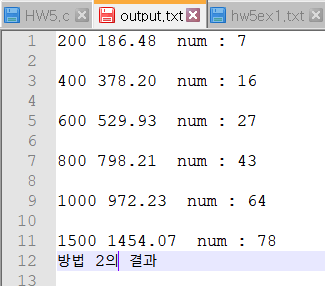
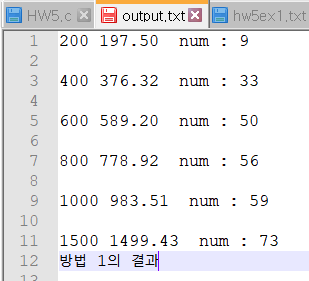
그래서 다음으로 생각한 것이 점을 제외할 때마다 다시 거리 이득을 계산하여 업데이트하는 **1번 방법 (getdot1 함수)**이다. 이 방법은 실시간으로 바뀌는 거리 이득을 반영하기 때문에 2번 방법보다 더 나은 결과를 보일 것으로 예상했고 실제로 낮은 연료구간 (200,400,600…)에서 더 좋은 결과를 보였다.

하지만 이 방법에도 문제가 발견되었는데 특정 입력 예제에서 높은 연료구간(1000,1500)의 결과값이 방법2보다 나쁘게 나왔다.

왜 그런지 생각해 보았는데 방법1은 현재의 최대 거리이득에만 집중하기 때문에 지금 당장에서는 최선의 방법이지만 전체적으로 보았을 때는 최선이 아닌 경우를 선택할 가능성이 있다. 그리고 이는 그리디 알고리즘의 치명적인 단점이기도 하다.

그래서 나는 방법 1과 방법 2를 합쳐서 둘 다 실행시켜 본 뒤 더 나은 결과값을 선택하도록 알고리즘을 설계하였다. 이 방식은 모든 경우를 다 따져볼 수는 없기 때문에 최선의 값을 선택하지 못할 가능성이 존재한다. 하지만 여러 방법들을 추가하여 더 많은 경우의 수(가능하다면 모든 경우의 수)를 따져볼 수 있다면, 최선의 해답에 가까워질 수 있을 것이라 생각된다.

**(방법 1,2의 출력 결과 비교) : 연료가 1000,1500인 경우 방법2의 결과가 더 크다.**



1. **참고 자료**

**퀵소트 알고리즘**

<https://coderkoo.tistory.com/7>

**다이어그램 그리는 사이트**

<https://www.draw.io/>