숭실대학교 Soongsil University

알고리즘 2021 보고서

보고서 제출서약서

나는 숭실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.

나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
   1.1. 나는 동료의 보고서를 베끼지 않았습니다.
   1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히 인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)
3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.
4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

과목	알고리즘 2021
과제명	다양한 정렬 알고리즘 구현 및 시간 측정
담당교수	최 재 영 교 수
제출인	컴퓨터학부 20162449 김상현 [327]
제출일	2021년 09월 22일

차 례

1. 프로젝트 동기 및 목적

선택 정렬, 버블 정렬, 삽입 정렬, 병합 정렬, 퀵 정렬, 힙 정렬을 각각 구현하고 알고리즘을 이해한다. 또 방대한 데이터를 정렬한다는 가정하에 걸리는 시간을 측정해 본다.

2. 설계 구현 아이디어

(0) 설계 개요

(a)번 문제 : 12 30 21 55 25 72 45 50를 입력 받고 정렬 알고리즘 별로 처음 다섯 개 step 의 결과를 출력하시오.

(b),(c) 번 문제 : -1 ~ 1 사이의 난수를 이용하여 정렬 알고리즘 별로 결과를 출력하고 수행시간을 측정하시오

* 힙 정렬의 경우 시작 5개를 뽑아내는 것이 어려워 그부분은 제거하였음을 미리 서술한다.

(1) 프로그램 구조

1. 선택 정렬 함수 - selection_sort(float input[] , int n) 이하 모두 배열,과 크기
2. 버블 정렬 함수 - bubble_sort(float input[] , int n)
3. 삽입 정렬 함수 - insertion_sort(float input[], int n)
4. 병합 정렬 알고리즘 - merge_sort(float input[], int left, int mid, int right)
   <-divide(분할)
   merge(float input[], int n)
   <-combine(합성)
5. 힙 정렬 함수 - bulid_heap(float input[], int n)
   <- 초기 배열을 힙으로 만듬
   heapify(float input[], int n, int start)
   <- 노드를 자식노드와 비교하여 힙상태로 만드는 함수
   heap_sort(input[],int n)
6. 퀵 정렬 함수 - quick_sort(float input[], int start, int end)
7. main 함수 - (a) 번과 (b)(c) 번을 각각 나누어서 선택할 수 있게 한다.
   rand() 함수를 이용한 -1~1까지 난수 발생 함수
   실행 시간 체크 , 결과 출력

3. 수행결과

(a) 선택 정렬

버블정렬

```
 숫자로 입력하세요 :1
1 회전 : 12.000  30.000  21.000  55.000  25.000  50.000  45.000  72.000
2 회전 : 12.000  30.000  21.000  45.000  25.000  50.000  55.000  72.000
3 회전 : 12.000  30.000  21.000  45.000  25.000  50.000  55.000  72.000
4 회전 : 12.000  30.000  21.000  25.000  45.000  50.000  55.000  72.000
5 회전 : 12.000  25.000  21.000  30.000  45.000  50.000  55.000  72.000
12  21  25  30  45  50  55  72
 숫자로 입력하세요 :2
1 회전 : 12.000  21.000  30.000  25.000  55.000  45.000  50.000  72.000
2 회전 : 12.000  21.000  25.000  30.000  45.000  50.000  55.000  72.000
3 회전 : 12.000  21.000  25.000  30.000  45.000  50.000  55.000  72.000
4 회전 : 12.000  21.000  25.000  30.000  45.000  50.000  55.000  72.000
5 회전 : 12.000  21.000  25.000  30.000  45.000  50.000  55.000  72.000
12  21  25  30  45  50  55  72
```

삽입정렬

```
 숫자로 입력하세요 :3
1 회전 key = 25.000000 : 12.000  25.000  21.000  30.000  55.000  45.000  50.000  72.000
2 회전 key = 21.000000 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
3 회전 key = 30.000000 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
4 회전 key = 55.000000 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
5 회전 key = 45.000000 : 12.000  21.000  25.000  30.000  45.000  55.000  50.000  72.000
12  21  25  30  45  50  55  72
```

병합 정렬

```
 숫자로 입력하세요 :4
1 회전 : 12.000  21.000  30.000  25.000  55.000  45.000  50.000  72.000
2 회전 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
3 회전 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
4 회전 : 12.000  21.000  25.000  30.000  45.000  55.000  50.000  72.000
5 회전 : 12.000  21.000  25.000  30.000  45.000  55.000  50.000  72.000
12  21  25  30  45  50  55  72
```

퀵 정렬

```
 숫자로 입력하세요 :5
1 회전 : 12.000  30.000  21.000  25.000  55.000  45.000  50.000  72.000
2 회전 : 12.000  25.000  21.000  30.000  55.000  45.000  50.000  72.000
3 회전 : 12.000  21.000  25.000  30.000  55.000  45.000  50.000  72.000
4 회전 : 12.000  21.000  25.000  30.000  50.000  45.000  55.000  72.000
5 회전 : 12.000  21.000  25.000  30.000  45.000  50.000  55.000  72.000
12  21  25  30  45  50  55  72
```

힙 정렬

```
1. 선택정렬
2. 버블정렬
3. 삽입 정렬
4. 병합 정렬
5. 퀵  정렬
6. 힙  정렬
 숫자로 입력하세요 :6
12  20  21  25  30  45  55  72
```

(b) 선택 정렬

```
 숫자로 입력하세요 :1
1 회전 : -0.562  -0.615  0.105  -0.546  0.076  -0.685  0.197  0.289
2 회전 : -0.562  -0.615  0.105  -0.546  0.076  -0.685  0.197  0.289
3 회전 : -0.562  -0.615  -0.685  -0.546  0.076  0.105  0.197  0.289
4 회전 : -0.562  -0.615  -0.685  -0.546  0.076  0.105  0.197  0.289
5 회전 : -0.562  -0.615  -0.685  -0.546  0.076  0.105  0.197  0.289
 걸린 시간 : 0.003000
-0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
```

버블 정렬

```
 숫자로 입력하세요 :2
1 회전 : -0.615  -0.562  -0.546  0.076  -0.685  0.105  0.197  0.289
2 회전 : -0.615  -0.562  -0.546  -0.685  0.076  0.105  0.197  0.289
3 회전 : -0.615  -0.562  -0.685  -0.546  0.076  0.105  0.197  0.289
4 회전 : -0.615  -0.685  -0.562  -0.546  0.076  0.105  0.197  0.289
5 회전 : -0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
 걸린 시간 : 0.002000
-0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
```

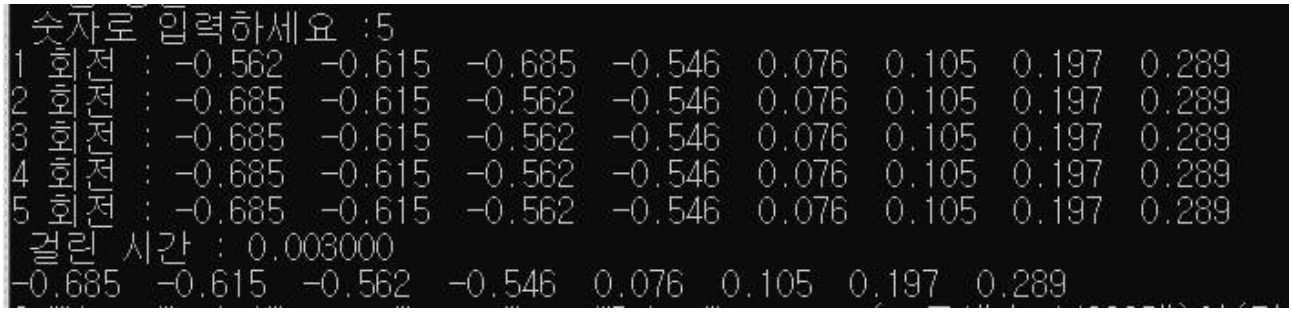
삽입정렬

```
 숫자로 입력하세요 :3
1 회전 key = -0.615436 : -0.615  -0.562  0.105  -0.546  0.076  -0.685  0.197  0.289
2 회전 key = 0.105136 : -0.615  -0.562  0.105  -0.546  0.076  -0.685  0.197  0.289
3 회전 key = -0.545671 : -0.615  -0.562  -0.546  0.105  0.076  -0.685  0.197  0.289
4 회전 key = 0.076235 : -0.615  -0.562  -0.546  0.076  0.105  -0.685  0.197  0.289
5 회전 key = -0.684835 : -0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
 걸린 시간 : 0.003000
-0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
```

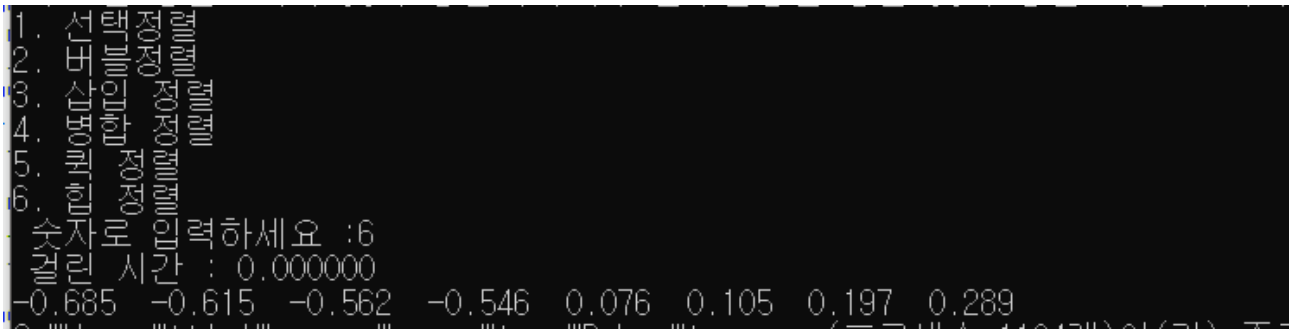
병합 정렬

```
 숫자로 입력하세요 :4
1 회전 : -0.615  -0.562  0.105  -0.546  0.076  -0.685  0.197  0.289
2 회전 : -0.615  -0.562  -0.546  0.105  0.076  -0.685  0.197  0.289
3 회전 : -0.615  -0.562  -0.546  0.105  0.076  -0.685  0.197  0.289
4 회전 : -0.615  -0.562  -0.546  0.105  -0.685  0.076  0.197  0.289
5 회전 : -0.615  -0.562  -0.546  0.105  -0.685  0.076  0.197  0.289
 걸린 시간 : 0.003000
-0.685  -0.615  -0.562  -0.546  0.076  0.105  0.197  0.289
```

퀵 정렬



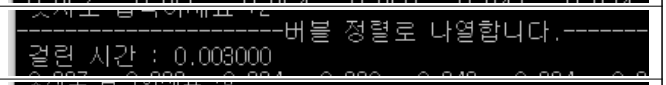
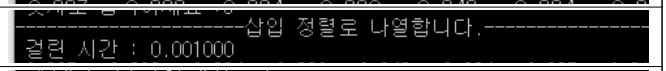
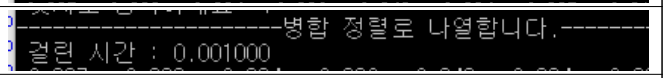
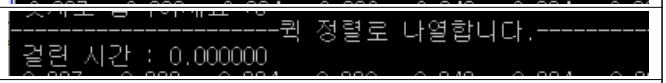
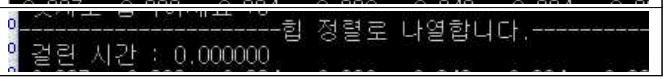
힙 정렬



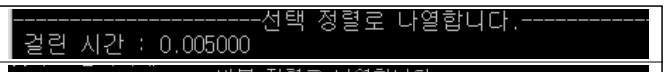
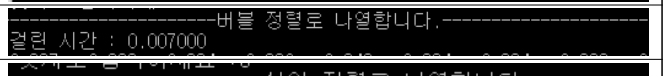
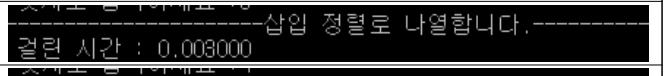
(c)

(3) 실행 결과 (sec 기준. 0.001 sec 는 1msec)

n= 1000

선택 정렬	0.001	
버블 정렬	0.003	
삽입 정렬	0.001	
병합 정렬	0.001	
퀵 정렬	0.0000	
힙 정렬	0.0000	

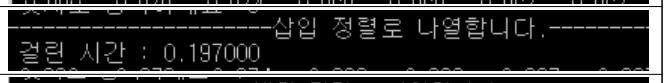
n=2000

선택 정렬	0.005	
버블 정렬	0.007	
삽입 정렬	0.002	
병합 정렬	0.001000	
퀵 정렬	0.001000	
힙 정렬	0.001	

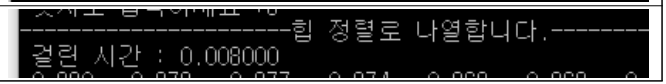
n= 3000...

선택 정렬	0.012	
버블 정렬	0.017	
삽입 정렬	0.006	
병합 정렬	0.001	
퀵 정렬	0.001	
힙 정렬	0.001	

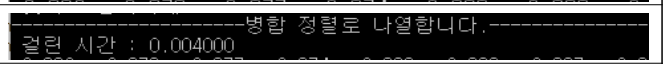
n= 18000

선택 정렬	0.41	
버블 정렬	0.87	
삽입 정렬	0.211	
병합 정렬	0.004	
퀵 정렬	0.002	
힙 정렬	0.008	

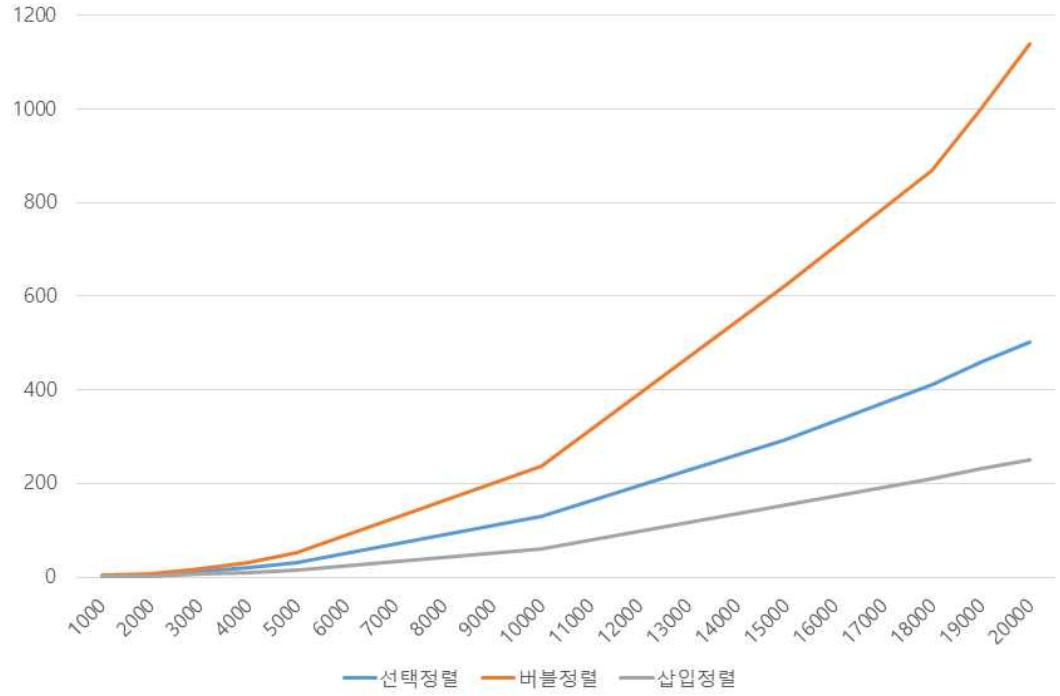
n=19000

선택 정렬	0.46	
버블 정렬	1.001	
삽입 정렬	0.233	
병합 정렬	0.004	
퀵 정렬	0.002	
힙 정렬	0.008	

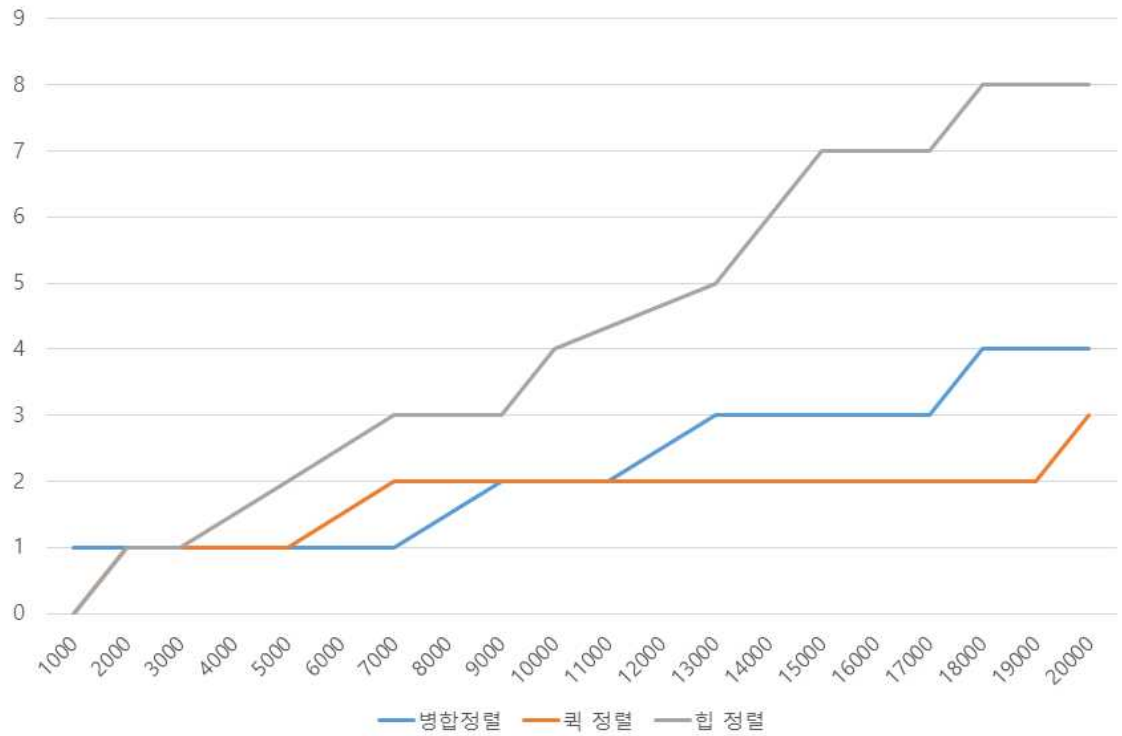
n=20000

선택 정렬	0.502	
버블 정렬	1.139	
삽입 정렬	0.251	
병합 정렬	0.004	
퀵 정렬	0.003	
힙 정렬	0.08	

$O(n^2)$의 시간 복잡도를 가진 선택 정렬, 버블 정렬, 삽입 정렬의 그래프



병합 정렬, 퀵 정렬, 힙 정렬



4. 결론 및 보충할 점

알고리즘의 성능과 시간복잡도에 대해 고찰해 보겠다.

높은 실행시간이 나온 선택 정렬과 버블 정렬 삽입 정렬은 값이 증가 됨에 따라 가장 많은 변화를 보여주었다. 그 예로 삽입 정렬을 보았다.

삽입 정렬은 key 값과 비교하여 킷값보다 작은 값이 나올 경우 이미 좌측 값들은 정렬이 되어있다는 뜻이므로 key 값을 우측으로 변경하여 위의 과정을 반복한다. 이러한 과정에서 정렬대상이 완전히 정렬된 상태면 비교대상이 계속 작거나 큰 값이므로 데이터의 변화가 일어나지 않는다. 결국 key 는 1~ n까지 n-1번 설정되게된다. 이는 삽입 정렬의 최선의 경우 시간복잡도로 $O(n)$ 이 된다. 하지만 모든 값에 대하여 데이터의 변화가 일어난다면 이는 key 설정 횟수 n-1 번 과 좌측 값들과 모두 비교하여 변화하기 때문에(n-1 번) 이는 $O(n^2)$ 가 최악의 경우 시간복잡도가 된다. n의 값이 커지면 커질수록 이 차이는 심해질 것이고 위와 같은 결과가 나왔다.

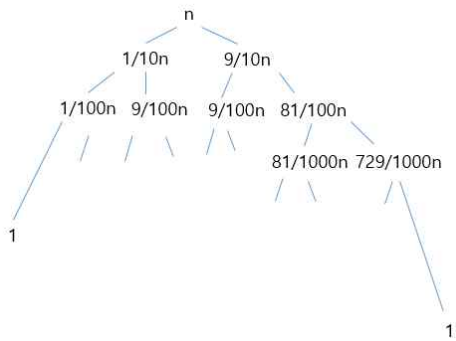
낮은 실행시간이 나온 병합, 퀵, 힙 정렬을 차례로 보자. 먼저

병합정렬은 전체 데이터가 분할되고, 분할된 좌측값을 분할한다. 이과정을 반복하여 더 이상 분할할수 없다면 분할된 값을 정렬하여 합친다. 좌측 값 합병이 완료 되었을 때 오른쪽 값을 같은 방법으로 반복한다. 분할하는 데 걸리는 시간은 왼쪽$t(\frac{n}{2})$ + 오른쪽$t(\frac{n}{2})$ 이다. 분할 할때마다 각 부분의 크기는 $\frac{n}{2}$이 되는데 반복되면 $\frac{n}{2^k}$이된다. k는 log n으로 표현할수 있고 곧 병합 정렬의 시간 복잡도는 $O(nlogn)$이 된다.

퀵 정렬은 피벗을 기준으로 피벗 보다 작은 값은 왼쪽, 큰 값은 오른쪽 값에 위치하게 시킴으로써 하나의 리스트를 분할한다. 그리고 왼쪽 오른쪽에 각각 또다른 피벗을 설정하여 이를 반복한다. 퀵 정렬은 완벽하게 리스트가 정렬되어있는 상태라면 피벗을 왼쪽에 잡던 오른쪽에 잡던 모든 값을 비교해야하기 때문에 n-1번 확인하고 피벗의 위치는 그대로가 된다. 이는 최악의 경우 시간복잡도로 $O(n^2)$ 가 된다. 하지만 리스트가 이미 모두 정렬되어있을 경우는 거의 없고 분할 자체가 한쪽으로치우치거나 하더라도 같은 비율로 분할 된다면 $O(nlogn)$이 된다. 예를 들어 1:9 로 분할되면

분할 후 좌측 크기 : $\frac{1}{10^k}n$

분할 후 우측 크기 : $\frac{9}{10^k}n$



$\frac{1}{10^k}n$ =1 k=log n

$\frac{9}{10^k}n$ = 1 k=log$\frac{9}{10}n$

시간 T(n)은

log n < T(n) < log$\frac{9}{10}n$ 이고

처음 n번 비교하는 횟수와 합치게 된다면 시간복잡도는 $O(nlogn)$이 된다.

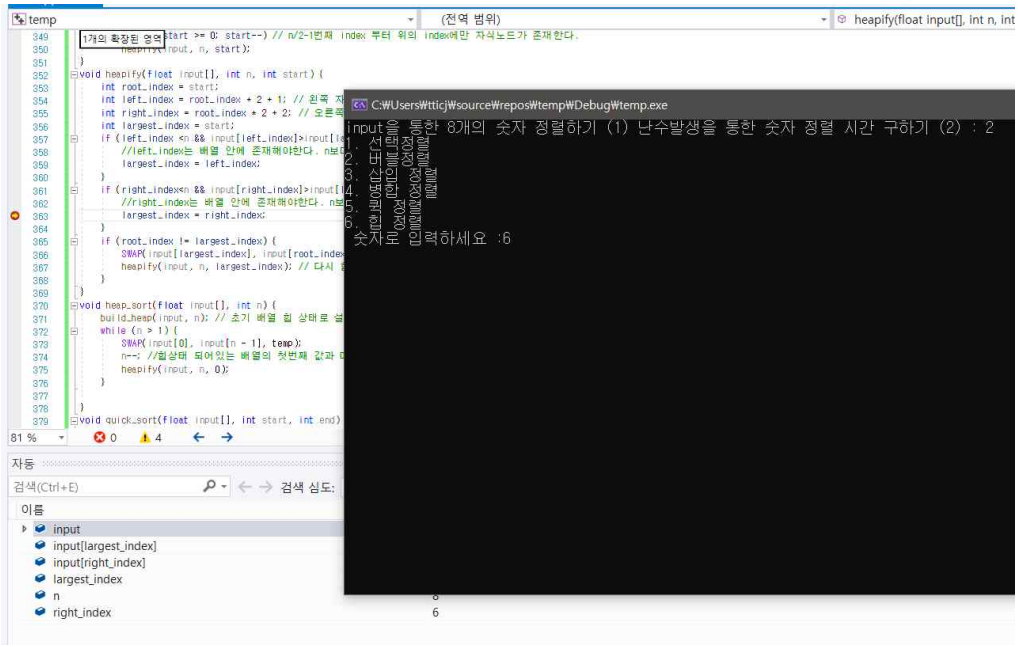
마지막으로 **힙 정렬**은 부모노드가 자식모드보다 커야하는 힙 상태로 초기 트리(배열)을 만들고 맨 마지막값과 첫 값을 바꾸고 마지막 값을 제외하고 다시 힙 상태로 만든다. 이를 반복하여 정렬한다. 시간복잡도로 생각해본다면 가장먼저 초기배열을 힙상태로 만드는 시간 복잡도는 $O(n)$이 된다. 또 힙을 재구성하는 시간은 트리의 높이 만큼 걸린다. 즉 log n번 시행되게 되고 자식노드의 부모노드는 n/2이므로 n/2번 실행되게 된다. 따라서 n/2*logn = $O(nlogn)$, $O(n)$ + $O(nlogn)$은 $O(nlogn)$의 시간복잡도를 가지게 된다.

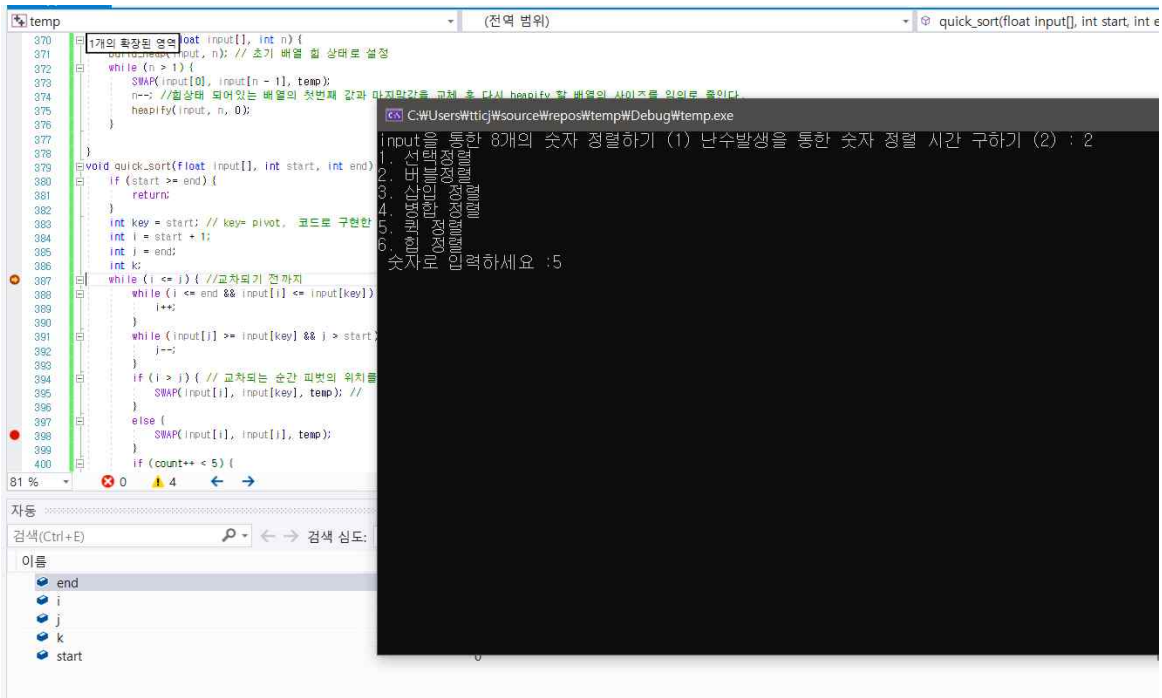
위 그래프로 성능 비교를 해보았을 때 같은 시간복잡도를 가지는 알고리즘끼리 비교해본다면

$O(n^2)$ 시간복잡도를 가지는 그래프 중에서는 삽입 정렬이, $O(nlogn)$시간복잡도를 가지는 그래프 중에서는 퀵 정렬이 가장 빨랐다. 버블 정렬이 제일 느렸던 이유를 추측한다면 버블정렬은 무조건 모든 수와 비교를 하면서 정렬을 해가야하기 때문에 제일 느렸고, 힙정렬의 경우에는 배열 사이즈가 줄면서 계속 heapify가 일어나므로 이과정에서 시간이 걸렸다고 생각한다.

전반적으로 제일 빨랐던 정렬 알고리즘은 합병 정렬 알고리즘과 퀵 알고리즘 이였다. 간단하게 이 둘의 단점에 대해 얘기하자면 합병 정렬 알고리즘은 기존의 데이터를 담을 추가 적인 배열 공간이 필요하다는 점에서 메모리 활용이 비효율적 일 수 있다는 문제가 있다. 퀵 알고리즘은 위에서 적었다시피 최악의 경우 $O(n^2)$시간 복잡도를 가지기 때문에 단순히 최악만 놓고 본다면 제일 느려질 수도 있는 가능성이 존재한다.

5. 디버깅


temp
(전역 범위)
heapify(float input[], int n, int
C:₩Users₩tticj₩source₩repos₩temp₩Debug₩temp.exe
input을 통한 8개의 숫자 정렬하기 (1) 난수발생을 통한 숫자 정렬 시간 구하기 (2) : 2
1. 선택정렬
2. 버블정렬
3. 삽입 정렬
4. 병합 정렬
5. 퀵 정렬
6. 힙 정렬
숫자로 입력하세요 :6
자동
검색(Ctrl+E)
검색 심도:
이름
input
input[largest_index]
input[right_index]
largest_index
n
right_index
6


temp
(전역 범위)
quick_sort(float input[], int start, int e
C:₩Users₩tticj₩source₩repos₩temp₩Debug₩temp.exe
input을 통한 8개의 숫자 정렬하기 (1) 난수발생을 통한 숫자 정렬 시간 구하기 (2) : 2
1. 선택정렬
2. 버블정렬
3. 삽입 정렬
4. 병합 정렬
5. 퀵 정렬
6. 힙 정렬
숫자로 입력하세요 :5
자동
검색(Ctrl+E)
검색 심도:
이름
end
i
j
k
start

6. 소스 코드

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h >
#include <stdlib.h >
#include <time.h >
#define SWAP(x,y,temp) ((temp)=(x),(x)=(y),(y)=(temp)) // 위치를 바꾸는 함수
#define MAXSIZE 8 // 정렬하는 입력값의 개수를 변경 하는 상수.
float sorted[MAXSIZE];// 병합정렬을 위한 임시 배열
void selection_sort(float input[], int n);
void bubble_sort(float input[], int n);
void insertion_sort(float input[], int n);
void merge_sort(float input[], int left, int right);
void merge(float input[], int left, int mid, int right);
void build_heap(float input[], int n);
void heapify(float input[], int n, int start);
void heap_sort(float input[], int n);
void quick_sort(float input[], int start, int end);
int count =0;
float temp;
int main(void) {
	int n = MAXSIZE;
	int a;
	int num;
	clock_t finish, start;
	double duration;
	float input[MAXSIZE];
	float array[8];
	printf("input을 통한 8개의 숫자 정렬하기 (1) 난수발생을 통한 숫자 정렬 시간 구하기 (2) : ");
	scanf("%d", &num);
	switch (num)
	{
	case 1:
		printf("input 8 numbers : ");
		for (int i =0; i <8; i ++) {
			scanf("%f", &array[i]);
		}
		printf("1. 선택정렬 ₩n2. 버블정렬 ₩n3. 삽입 정렬 ₩n4. 병합 정렬 ₩n5. 퀵정렬₩n6. 힙 정렬₩n 숫자로 입력하세요 :");
		scanf_s("%d", &a);
		if (a ==1) {
			selection_sort(array, 8);
		}
		else if (a ==2) {
			bubble_sort(array, 8);
		}
		else if (a ==3) {
			insertion_sort(array, 8);
		}
		else if (a ==4) {
			merge_sort(array, 0, 7);
		}
		else if (a ==5) {
			quick_sort(array, 0, 7); // 첫번째 값을 key로 가질수 있게 설정한다.
		}
		else if (a ==6) {
			heap_sort(array, 8);
		}
		else {
			printf("1~6까지의 수만 입력해주세요");
			break;
		}
		for (int i =0; i <8; i ++)
			printf("%d ",(int)array[i]);
		break;
	case 2:
		for (int i =0; i < n; i ++) { //난수 생성 (seed)값이 같기 때문에 항상 같은 값이 나올 것이다.
			input[i] = rand() / (float)RAND_MAX - rand() / (float)RAND_MAX;//0~1까지의 실수를 뽑아내고 그걸 다시 0~1까지 실수로 빼준다면 -1~1까지의 실수의 랜덤 값이 될 것이다.
		}
		printf("1. 선택정렬 ₩n2. 버블정렬 ₩n3. 삽입 정렬 ₩n4. 병합 정렬 ₩n5. 퀵정렬₩n6. 힙 정렬₩n 숫자로 입력하세요 :");
		scanf_s("%d", &a);
		if (a ==1) {
			start = clock(); // 함수 시작 시간 저장
			selection_sort(input, n);
			finish = clock();// 함수 종료 시간 저장

//printf("----------------------선택 정렬로 나열합니다.----------------------₩n");
		}
		else if (a ==2) {
			start = clock();
			bubble_sort(input, n);
			finish = clock();

//printf("----------------------버블 정렬로 나열합니다.----------------------₩n");
		}
		else if (a ==3) {
			start = clock();
```

```
                        insertion_sort(input, n);
                        finish = clock();

//printf("----------------------삽입 정렬로 나열합니다.----------------------₩n");
                }
                else if (a ==4) {
                        start = clock();
                        merge_sort(input, 0, n-1);
                        finish = clock();

//printf("----------------------병합 정렬로 나열합니다.----------------------₩n");
                }
                else if (a ==5) {
                        start = clock();
                        quick_sort(input, 0, n-1); // 첫번째 값을 key로 가질수 있게 설정한다.
                        finish = clock();

//printf("----------------------퀵 정렬로 나열합니다.----------------------₩n");
                }
                else if (a ==6) {
                        start = clock();
                        heap_sort(input, n);
                        finish = clock();

//printf("----------------------힙 정렬로 나열합니다.----------------------₩n");
                }
                else
                        printf("1~6까지의 수만 입력해주세요");
                duration = (float)(finish - start) / CLOCKS_PER_SEC; // 종료시간에서 시작시간을 뺀후 초단위로 바꿔준다.
                // msec 기준으로 하려면 뒤에 CLOCKS_PER_SEC를 나누지 않으면 된다.
                printf(" 걸린 시간 : %lf₩n", duration);
                for (int i =0; i < n; i ++)
                        printf("%.3f ", input[i]);
                break;
        default:
                break;
        }

        return 0;
}
void selection_sort(float input[], int n) {
        int i, j,k, maxindex =0;
        int count =0;
        float max =-2;
        // -1~1 까지의 난수이기때문에 최소값보다 작아야함
        for (i = n -1; i >=0; i --) {
                for (j =0; j <= i; j ++) {
                        if (input[j] > max) {
                                max = input[j];
                                maxindex = j;
                        }
                }
                max =-2;
                if (i != maxindex)
                        SWAP(input[i], input[maxindex], temp);
                if (count ++<5) {
                        printf("%d 회전 : ", count);
                        for (k =0; k < n; k ++) {
                                printf("%.3f ", input[k]);
                        }
                        printf("₩n");
                }
        }
}
void bubble_sort(float input[], int n) {
        int i, j,k;
        int count =0;
        for (i = n -1; i >=0; i --) {
                for (j =0; j <= i -1; j ++) {
                        if (input[j] > input[j+1])

SWAP(input[j], input[j +1], temp);
                }
                if (count ++<5) {
                        printf("%d 회전 : ", count);
                        for (k =0; k < n; k ++) {
                                printf("%.3f ", input[k]);
                        }
                        printf("₩n");
                }
        }
}
void insertion_sort(float input[], int n) {
        int i, j,k;
        float key;
        int count =0;
        for (i =1; i < n; i ++) {
                key = input[i];//key 설정
                for (j = i -1; j >=0 && input[j] > key; j --) {
```

```
                        input[j +1] = input[j];
                }
                input[j +1] = key;
                if (count ++<5) {
                        printf("%d 회전 key =
%f : ", count,key);
                        for (k =0; k < n; k
++) {
                                printf("%.3f
", input[k]);
                        }
                        printf("₩n");
                }
        }
}
void merge_sort(float input[], int left, int right)
{
        int mid;
        if (left < right) {
                mid = (left + right) /2; //중앙
값을 잡는다.
                merge_sort(input, left, mid); //
왼쪽기준으로 재귀
                merge_sort(input, mid +1,
right); //종료후 오른쪽기준으로 재귀
                merge(input, left, mid, right);
        }
}
void merge(float input[], int left, int mid, int
right) {
        int i, j, k, l;
        i = left;
        j = mid +1;
        k = left;
        while (i <= mid && j <= right) { // 분
할 정렬된 input 배열의 합병
                if (input[i] <= input[j])
                        sorted[k++] = input[i
++];
                else
                        sorted[k++] = input[j
++];
        }
        if (i > mid) {
                for (l = j; l <= right; l ++) //
남아 있는 값들을 일괄 복사
                        sorted[k++]
input[l];
        }
        else {
                for (l = i; l <= mid; l ++)
        // 남아 있는 값들을 일괄 복사
                        sorted[k++]
input[l];
        }
        for (l = left; l <= right; l ++) { //
sorted[](임시 배열)의 리스트를 input 배열로
재복사
                input[l] = sorted[l];
        }
        if (count ++<5) {
                printf("%d 회전 : ", count);
                for (k =0; k <8; k ++) {
                        printf("%.3f
", input[k]);
                }
                printf("₩n");
        }
}
void build_heap(float input[], int n) {
        int start = n /2 -1;// 자식노드가 존재
하는 노드는 전체 수 n/2번째 노드에 존재할 수
있다. -1은 배열의 인덱스가 0으로 시작함을 감
안하였다.
        for (start; start >=0; start --) //
n/2-1번째 index 부터 위의 index에만 자식노드
가 존재한다.
                heapify(input, n, start);
}
void heapify(float input[], int n, int start) {
        int root_index = start;
        int left_index = root_index *2 +1; //
왼쪽 자식 노드
        int right_index = root_index *2 +2; //
오른쪽 자식 노드
        int largest_index = start;
        if (left_index <n &&
input[left_index]>input[largest_index]) {
                //left_index는 배열 안에 존재해
야한다. n보다 작아야함
                largest_index = left_index;
        }
        if (right_index <n &&
input[right_index]>input[largest_index]) {
                //right_index는 배열 안에 존재
해야한다. n보다 작아야함
                largest_index = right_index;
        }
        if (root_index != largest_index) {
                SWAP(input[largest_index],
input[root_index], temp); // 위치를 바꾸고
                heapify(input, n,
largest_index); // 다시 힙 화 해준다.
        }
}
void heap_sort(float input[], int n) {
        build_heap(input, n); // 초기 배열 힙
상태로 설정
        while (n >1) {
                SWAP(input[0], input[n -1],
temp);
                n--; //힙상태 되어있는 배열의
첫번째 값과 마지막값을 교체 후 다시 heapify 할
배열의 사이즈를 임의로 줄인다.
                heapify(input, n, 0);
        }
```

```
}
void quick_sort(float input[], int start, int end)
{
	if (start >=end) {
		return;
	}
	int key = start; // key= pivot,  코드로 구현한 피벗은 제일 첫번째 값으로 한다
	int i = start +1;
	int j =end;
	int k;
	while (i <= j) { //교차되기 전까지
		while (i <=end && input[i] <= input[key]) {
			i++;
		}
		while (input[j] >= input[key] && j > start) {
			j--;
		}
		if (i > j) { // 교차되는 순간 피벗의 위치를 고정 한다. 위의 반복문도 탈출
			SWAP(input[j], input[key], temp); //
		}
		else {
			SWAP(input[i], input[j], temp);
		}
		if (count ++<5) {
			printf("%d 회전 : ", count);
			for (k =0; k <8; k ++) {
				printf("%.3f ", input[k]);
			}
			printf("₩n");
		}
	}
	quick_sort(input, start, j -1); // 피벗 기준 왼쪽
	quick_sort(input, j +1, end); // 피벗 기준 오른쪽
}
```