자료구조와 알고리즘 보고서

20193281 송형준

1-1. 피보나치 수열 <재귀적용법>

```
#include <iostream>
#include <ctime>

using namespace std;

int Fibo_reflex(int);

int main(){

	cout << "20193281 송형준 " << endl;
	cout << '₩n';

	int num;
	cout << "몇 번째 수열까지 출력할까요? : " ;
	cin >> num;

	clock_t start = clock();
	for(int i=1;i<=num;i++){
		cout << Fibo_reflex(i) << "  ";
		if(i%5==0) cout << '₩n';
	}
	clock_t end = clock();

	cout << num <<"번째 피보나치 수열 계산 시간 : " << (double)(end-start)/CLOCKS_PER_SEC;

	return 0;
}

int Fibo_reflex(int num){
	if(num==1 || num==2) return 1;
	else return Fibo_reflex(num-1)+Fibo_reflex(num-2);
}
```

연구조사

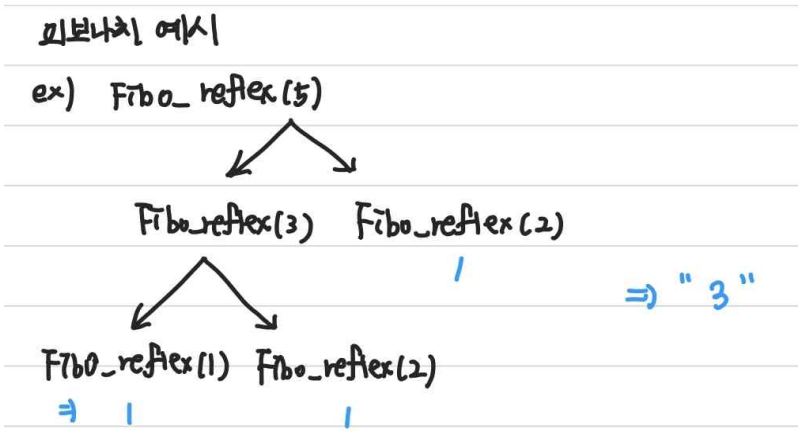
피보나치 수열 : n번째 항이 n-1번째와 n-2번째 항의 합인 수열

f(n)=f(n-2)+f(n-1)

f1, f2 = 1

첫 번째 항, 두 번째 항일 땐 1, 다른 숫자일 경우엔 재귀함수 호출

Fibo_reflex는 계속 재귀를 반복하여 n=1,2 에 도달, 1을 반환하고 이를 합한 최종 값을 얻는다. (예시참고)



실행결과

```
20193281 송형준

몇 번째 수열까지 출력할까요? : 35
1  1  2  3  5
8  13  21  34  55
89  144  233  377  610
987  1597  2584  4181  6765
10946  17711  28657  46368  75025
121393  196418  317811  514229  832040
1346269  2178309  3524578  5702887  9227465
35번째 피보나치 수열 계산 시간 : 0.154
PS C:\Users\dhn2\Desktop\Algorithm_class_code\Week5_assignment>
```

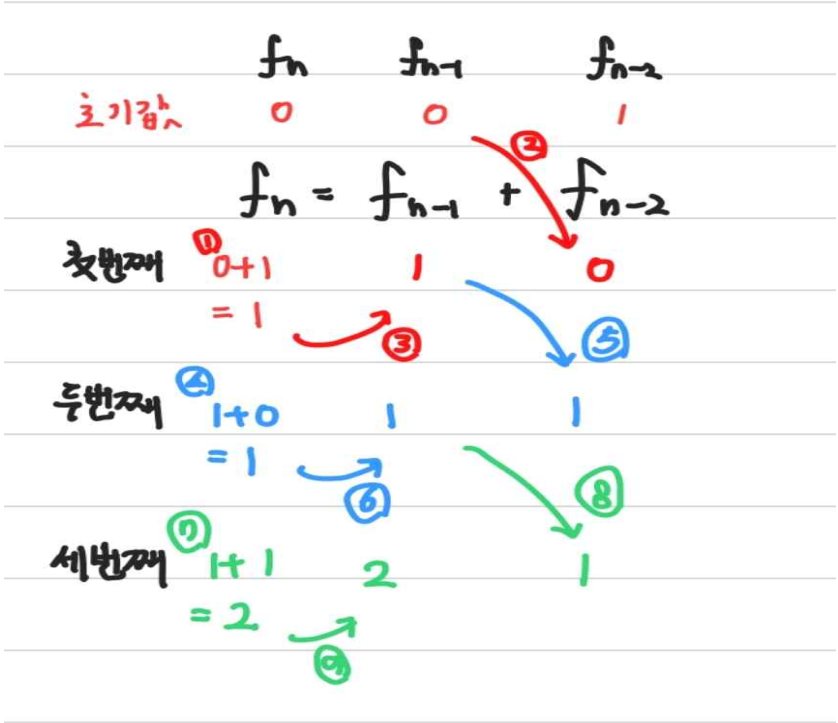
<재귀적 용법>

1-2. 피보나치 수열 <비재귀적 용법>

```
#include <iostream>
#include <ctime>

using namespace std;

void Fibo_repeat(int);

int main(){

	cout << "20193281 송형준 " << endl;
	cout << '₩n';

	int num;
	cout << "몇 번째 수열까지 출력할까요? : " ;
	cin >> num;

	Fibo_repeat(num);

	return 0;
}

void Fibo_repeat(int num){

    int fn=0, fn_1=0, fn_2=1;

    clock_t start = clock();
    for(int i=0;i<num;i++){
		fn = fn_1 + fn_2;

		cout << fn << "   ";

		fn_2 = fn_1;
		fn_1 = fn;

		if((i+1)%5==0) cout << '₩n';
    }
     clock_t end = clock();

     cout << num <<"번째 피보나치 수열 계산 시간 : " << (double)(end-start)/CLOCKS_PER_SEC;
     return;
}
```

연구조사

비재귀의 경우, 반복문을 이용하여 계산한다. 이 때, n=1, n=2를 위해 초기값을 fn=0, fn-1=0, fn-2=1로 둔다. (예시 참고)



실행결과

```
20193281 송형준

몇 번째 수열까지 출력할까요? : 35
1  1  2  3  5
8  13  21  34  55
89  144  233  377  610
987  1597  2584  4181  6765
10946  17711  28657  46368  75025
121393  196418  317811  514229  832040
1346269  2178309  3524578  5702887  9227465
35번째 피보나치 수열 계산 시간 : 0.011
PS C:\Users\dhn2\Desktop\Algorithm_class_code\Week5_assignment>
```

두 가지 방법으로 구현한 피보나치 수열을 비교했을 때, 비재귀적 용법의 속도가 더 빠른 것을 확인할 수 있다. 이는 재귀적 용법이 함수 호출에 시간이 걸리기 때문이다.

2. 이진 탐색 알고리즘

```
#include <iostream>

using namespace std;

int* binarySearch(int* pArr,int* pFirst,int* pLast, int key);
void PRINT(int* pArr,int num);

int main(){

    cout << "20193281 송형준 " << endl;
    cout << '₩n';

    int arr[10]={5,9,13,17,21,28,37,46,55,86};

    PRINT(arr,10);
    cout << '₩n';

    while(true){
        int num;
        cout << "검색 데이터 입력(검색 종료 : 0) : ";
        cin >> num;

        if(num ==0) break;

        int* answer = binarySearch(arr,arr,arr+9,num);

        if(answer==NULL) cout << "없다고 !!!!! " << endl;
        else {
            cout << "검색 데이터의 위치 : " << (int)(answer-arr)+1 << endl;
        }
    }

    return 0;
}
```

```
int* binarySearch(int* pArr,int* pFirst,int* pLast, int key){ // key : 찾아야 하는 값

    if(*pFirst > *pLast) {
        return NULL; //first가 더 크면 아래과정 할 필요 x
    }
    int* mid = pFirst+(pLast-pFirst)/2;

    if(key==*mid) return mid;
    else if(key < *mid) binarySearch(pArr,pFirst,mid-1,key);
    else if(key > *mid) binarySearch(pArr,mid+1,pLast,key);
}

void PRINT(int* pArr,int num){

    cout << " 원시 데이터 : ";

    for(int i=0;i<10;i++){
        cout << pArr[i] << "   ";
    }
}
```

연구조사

이진 탐색 : 중간값을 이용하여 찾고자하는 값인 key값을 찾아내는 알고리즘

전제조건

1) 배열이 정렬 상태여야 한다.
2) 각 데이터는 유일한 값을 갖는다.

과정

1) 탐색할 범위 경계값의 중간 값을 구한다.
2) 중간값과 key값을 비교한다
3)

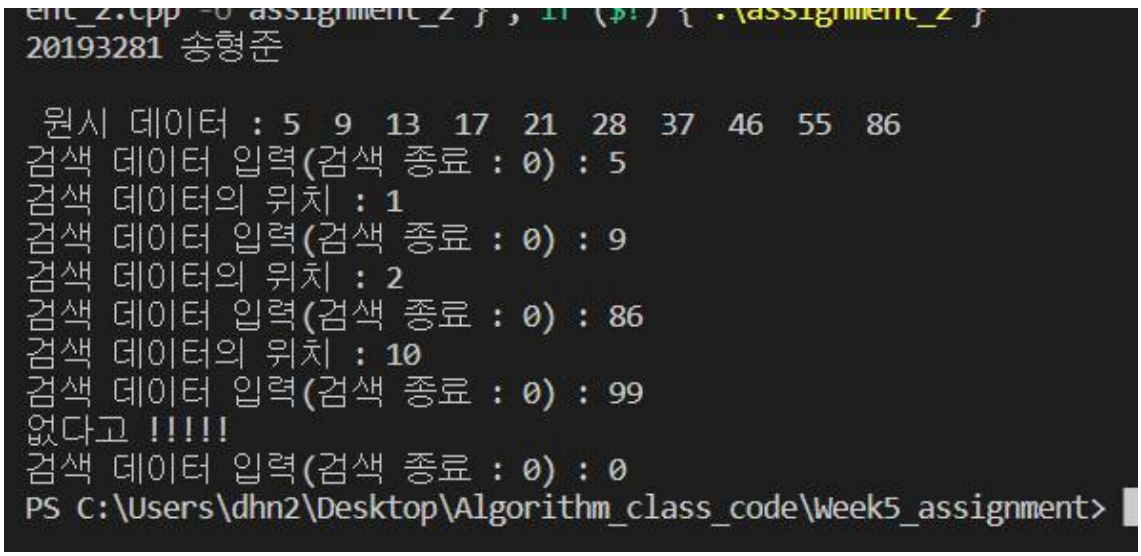
중간값 > key값 : 탐색 범위를 첫번째값 ~ 중간값-1 로 바꾼다
중간값 < key값 : 탐색 범위를 중간값+1 ~ 마지막값 으로 바꾼다.

Q. mid 찾을 때, (pFirst+pLast)/2 를 못하는 이유?

포인터 연산
포인터끼리의 연산은 뺄셈만 의미를 가진다. 다른 연산은 임의의 위치를 나타낼 뿐이므로 의미를 갖지 않는다.
포인터끼리의 뺄셈은 두 포인터 간의 상대적 길이이다. 이 때 길이는 포인터의 자료형에 따라 다르게 나온다.
ex) int* 의 경우, 포인터 한칸 당 4byte 를 의미
ex) 만약 int* a와 b의 차 b-a 의 실제 값이 12이더라도, 이는 12/4바이트인 3으로 나타난다.

주소값을 16진수로 나타낸 수, 둘을 더하면 이를 표현할 수 있는 방법이 없음 + 의미도 없음
pFirst+(pLast-pFirst)/2 -> 주소값의 빼기 연산은 주소값의 차이를 바이트 수로 나눈 것을 반환한다.

실행결과



3. 계수 정렬

```
#include <iostream>
#include <ctime>
#include <random>

#define arrMAXSIZE 15

using namespace std;

void countingSort(int* pArr, int num);

int main(){

	int arr[arrMAXSIZE+1]={0};

	random_device rd;
		mt19937	gen(rd());
		uniform_int_distribution<int> dis(0, 99); //0~99 를 생성하는 dis 객체 선언



	for(int i=1;i<arrMAXSIZE+1;i++){
		*(arr+i)=dis(gen); //dis안에 난수 생성 엔진을 넣어 숫자 생성
	}

	cout << " 정렬 전 : ";
	for(int i=1;i<arrMAXSIZE+1;i++){
		cout << arr[i] << " ";
	}
	cout << "₩n";

	countingSort(arr,arrMAXSIZE);
}
```

```
void countingSort(int* pArr, int num){
	int counting[100]={0};
	int ans[arrMAXSIZE+1]={0};

	for(int i=1;i<num+1;i++){
		counting[pArr[i]]++;
	}//갯수 세기

	for(int i=1;i<100;i++){ //1부터 시작
		counting[i]+=counting[i-1];
	}//누적합 배열

	for(int i=num;i>=1;i--){
//counting[pArr[i]의 의미 : (정렬하고자 하는 원소)(pArr)가 들어갈 (정답배열)(ans)의 인덱스
		ans[counting[pArr[i]]]=pArr[i];
		counting[pArr[i]]--;
	}

	cout << " 정렬 후 : ";
	for(int i=1;i<=num;i++){
		cout << ans[i] << " ";
	}
}
```

연구조사

계수 정렬 : 각 숫자의 갯수를 센 수, 작은 수부터 갯수만큼 출력

전제조건

1) 정수로 표현 가능한 배열만 정렬할 수 있다.
2) arr의 data값 중 최댓값을 알아야한다.

과정

1) 각 data의 갯수를 센다
2) 카운트 배열에 갯수를 저장한다.
   * 카운트 배열(count_arr)이란 : count_arr의 인덱스 = data값, count_arr의 데이터 = data의 갯수
3) 누적합 배열 생성
   -> count_arr의 data값을 왼쪽부터 계속 누적하여 자기 자신 자리에 대입
4) for문으로 배열을 정리하여 출력

stable 과 unstable
배열 내에 동일한 값이 두개 이상 존재할 때 그 값의 순서가 보장되는 것을 stable하다고 표현한다.

ex) {1,3,2,1} 을 {1ㄱ,3,2,1ㄴ} 이라 표현할 때, 정렬 후의 모습이 {1ㄱ,1ㄴ,2,3}이면 stable하다
만약 {1ㄴ,1ㄱ,2,3} 일 경우 unstable 하다.

stable은 매번 동일한 값을 출력하고 싶은 경우에 사용한다.

stable : mergeSort, insertSort, countSort, bubbleSort
unstable : quickSort,heapSort, selectionSort, shellSort

rand함수

C 스타일 난수 생성(srand와 rand 함수)

```
srand(static_cast<unsigned int>(time(nullptr)));
static_cast<int>(rand() % 100);
```

문제점

1. 수학적 알고리즘에 의한 난수 생성이기에, 실제 난수가 아니다.
2. 시드가 너무 느리게 변하기 때문에, 여러 난수 프로그램을 동시에 돌릴 경우 같은 난수가 생성된다.
3. 0~99까지의 숫자 발생 확률이 균등하지 않다.

따라서 c++ 스타일의 난수 생성을 이용한다
<random> 내의 random_device 클래스 사용

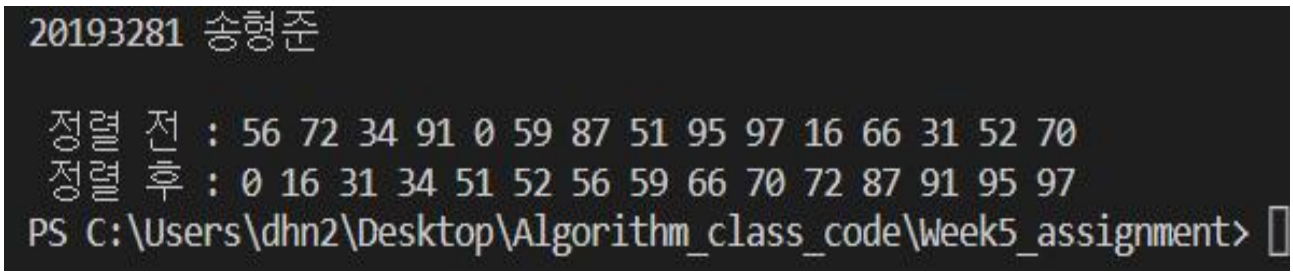
실제 컴퓨터의 무작위적인 요소를 이용해 진정한 난수를 생성 가능하다.
허나 난수 생성 속도가 느리기 때문에, 시드값 초기화에만 사용하고, 이후의 난수는 난수 엔진을 사용하여 생성한다.

mt19937 gen(시드번호)

mt19937은 엔진의 한 종류로, <random> 내에는 다른 엔진들 역시 정의되어있다
ex) 객체의 크기가 클 경우 minstd_rand

이후엔 난수 생성의 범위를 지정해준다.
std::uniform_int_distribution<int> : 균등분포, 범위의 숫자를 동일한 확률로 생성한다.

실행결과



4. 성능평가 - 6개의 정렬성능 비교

```
#include <iostream>
#include <ctime>
#include <random>

#define arrSize 30000
using namespace std;

void RANDOM(int* pArr,int num);
void SWAP(int* pa,int* pb);
void PRINT(int* pArr,int num);

//기본정렬
void selectSort(int* pArr,int num);
void bubbleSort(int* pArr,int num);
void insertSort(int* pArr,int num);

//고급정렬
void mergeSort(int* pArr,int* pFirst,int* pLast);
void merge(int* pArr,int* pFirst,int* pMid, int* pLast);
void quickSort(int* pArr,int* pfirst,int* plast);
int* Partition(int* pArr, int* pfirst, int* plast);
void shellSort(int* pArr, int num);
void intervalSort(int* pArr, int start, int num, int interval);

int main(){

	int arr[arrSize]={0};
	clock_t start;
	clock_t end;

	cout << "20193281 송형준 " << endl;
	cout << '₩n';

//선택정렬
	RANDOM(arr,arrSize);
	cout << "정렬 전 : " ;
	PRINT(arr,arrSize);

	start = clock();
	selectSort(arr,arrSize);
	end = clock();
```

```
    cout << "선택정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);

//버블정렬
    cout << '₩n';

    RANDOM(arr,arrSize);
    cout << "정렬 전 : " ;
    PRINT(arr,arrSize);

    start = clock();
    bubbleSort(arr,arrSize);
    end = clock();
    cout << "버블정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);

//삽입정렬
    cout << '₩n';

    RANDOM(arr,arrSize);
    cout << "정렬 전 : " ;
    PRINT(arr,arrSize);

    start = clock();
    insertSort(arr,arrSize);
    end = clock();
    cout << "삽입정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);

//병합정렬
    cout << '₩n';

    RANDOM(arr,arrSize);
    cout << "정렬 전 : " ;
    PRINT(arr,arrSize);

    start = clock();
```

```
    mergeSort(arr,arr,arr+arrSize-1);
    end = clock();
    cout << "병합정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);

//퀵정렬
    cout << '₩n';

    RANDOM(arr,arrSize);
    cout << "정렬 전 : " ;
    PRINT(arr,arrSize);

    start = clock();
    quickSort(arr,arr,arr+arrSize-1);
    end = clock();
    cout << "퀵정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);

//쉘정렬
    cout << '₩n';

    RANDOM(arr,arrSize);
    cout << "정렬 전 : " ;
    PRINT(arr,arrSize);

    start = clock();
    shellSort(arr,arrSize);
    end = clock();
    cout << "퀵정렬 : " << (double)(end-start)/CLOCKS_PER_SEC << endl;

    cout << "정렬 후 : " ;
    PRINT(arr,arrSize);
    return 0;
}

void RANDOM(int* pArr,int num){
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> dis(0,99);
```

```
    for(int i=0;i<arrSize;i++){
        *(pArr+i)=dis(gen);
    }
}

void SWAP(int* pa,int* pb){
    int temp = *pa;
    *pa=*pb;
    *pb=temp;
}

void PRINT(int* pArr,int num){
        for (int i = 0; i < num; i++) {
            cout.width(3);
            cout << *(pArr + i);
        }
        cout << endl;
        return;
}

void selectSort(int* pArr,int num){
    for(int i=0;i<(num-1);i++){
        int smallest = i;
        for(int j=i+1;j<num;j++){
            if(pArr[j] < pArr[smallest]) SWAP(pArr+j,pArr+smallest);
        }
    }
}

void bubbleSort(int* pArr,int num){
    for(int i=0;i<(num-1);i++){
        bool state = true;
        for(int j=num;j>i;j--){
            if(pArr[j] < pArr[j-1]){
                SWAP(pArr+j,pArr+(j-1));
                state=false;
            }
        }
        if(state) break;
    }
}
```

```
void insertSort(int* pArr,int num){
	for(int i=1;i<num;i++){
		int j = i-1;
		int temp = pArr[i];

		while(j>=0 && temp<pArr[j]){
			pArr[j+1]=pArr[j];
			j--;
		}
		pArr[j+1]=temp;
	}
}

void mergeSort(int* pArr,int* pFirst,int* pLast){

	int* mid = NULL;

	if((pFirst-pArr) < (pLast-pArr)){
		mid = pFirst+(pLast-pFirst)/2;
		mergeSort(pArr,pFirst,mid); //처음~중간 정렬
		mergeSort(pArr,mid+1,pLast); //중간~끝 정렬
		merge(pArr,pFirst,mid,pLast);
	}
}

void merge(int* pArr,int* pFirst,int* pMid, int* pLast){
	int i = pFirst-pArr;
	int j = (pMid-pArr)+1;
	int k = i;

	int* tempArr = new int[arrSize];

	while(i <= (pMid-pArr) && j <= (pLast-pArr)){
		if(pArr[i] <= pArr[j] ){
			tempArr[k++] = pArr[i++];
		}
		else{
			tempArr[k++] = pArr[j++];
		}
	}

	while(i <= (pMid-pArr)) tempArr[k++] = pArr[i++];
	while(j <= (pLast-pArr)) tempArr[k++] = pArr[j++];
```

```
    i=pFirst-pArr;
    k=i;

    while(i <= (pLast-pArr)) pArr[i++] = tempArr[k++];

    delete[] tempArr;
}

void quickSort(int* pArr,int* pfirst,int* plast){

    int* pmid=NULL;
    if(pfirst-pArr < plast-pArr){ //기준값보다 작으면
        pmid = Partition(pArr,pfirst,plast);
        quickSort(pArr,pfirst,pmid-1);
        quickSort(pArr, pmid+1, plast);
    }
}

int* Partition(int* pArr, int* pfirst, int* plast){
    int pivot = *plast; //마지막 원소 기준

    int i = (pfirst-pArr)-1; //first-1,

    for(int j=(pfirst-pArr);j<(plast-pArr);j++){
        if(pArr[j]<=pivot) SWAP(pArr+(++i),pArr+j);
    }
    SWAP(pArr+i+1,plast);

    return pArr+i+1;
}

// 쉘 정렬
void    shellSort(int* pArr, int num) {
    int interval = num;
    double result;

    while (interval >= 1) {
        interval = interval / 2;
        for (int i = 0; i < interval; i++) {
            intervalSort(pArr, i, num, interval);
        }
    }
```

```
    return;
}

void    intervalSort(int* pArr, int start, int num, int interval) {
    int temp, i, j;

    for (i = start+interval; i < num; i = i+interval) { // 부분리스트 원소 수만큼 반복 + 첫 원소는 정렬
됐다 가정
        temp = *(pArr + i); //정렬 안된 부분리스트의 첫 원소

        // temp가 들어갈 공간을 확보
        for (j = i-interval; 0 <= j && *(pArr + j) > temp; j = j-interval)
            *(pArr + j + interval) = *(pArr + j);

        *(pArr + j + interval) = temp; //원소 삽입
    }

    return;
}
```

연구조사

1. 선택정렬 : 가장 작은 원소를 선택하여 정렬
제자리 정렬의 하나로서, 입력 배열(정렬되지 않은 값들) 이외에 다른 추가 메모리를 요구하지 않는 정렬 방법이다. 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘
첫 번째 순서에는 첫 번째 위치에 가장 최솟값을 넣는다.
두 번째 순서에는 두 번째 위치에 남은 값 중에서의 최솟값을 넣는다.

선택 정렬 수행 과정
1) 비정렬구역 중에서 최솟값을 찾는다.
2) 그 값을 비정렬구역의 맨앞과 교체한다.
3) 하나의 원소만 남을 때까지 위의 1~3 과정을 반복한다

메모리 사용 공간
n개의 원소에 대하여 n개의 메모리 사용

연산 시간

1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2
2) Worst Case : 정렬된 자료에서 최솟값만 맨 뒤에 위치한 경우, 총 소요시간: n(n-1)/2 + (n-1)

2. 버블정렬 : 인접한 두 원소의 크기를 비교하고, 작은 원소를 왼쪽으로 옮긴다.
거품이 수면 위로 올라오는 듯한 모습을 닮아 붙여진 이름, 1회전을 수행하고 나면 가장 큰 자료가 맨 뒤로 이동하므로 2회전에서는 맨 끝에 있는 자료는 정렬에서 제외되고, 2회전을 수행하고 나면 끝에서 두 번째 자료까지는 정렬에서 제외된다. 이렇게 정렬을 1회전 수행할 때마다 비정렬구역에서 제외되는 데이터가 하나씩 늘어난다.

버블 정렬 수행 과정
1) 비 정렬구역의 맨 칸 원소와 한칸 앞의 원소를 비교한다.
2) 맨 뒤 칸의 원소가 더 작을 경우 둘의 위치를 변경한다.
3) 비 정렬구역의 맨 앞 경계를 한 칸 뒤로 미룬다.
4) 위 과정을 반복한다.

메모리 사용 공간
n개의 원소에 대하여 n개의 메모리 사용

연산 시간
1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2
2) Worst Case : 자료가 역순으로 정렬되어 있는 경우, 총 소요시간: n(n-1)

구현이 매우 간단하나, 구조 자체의 비효율성, 그리고 일반적으로 자료의 SWAP이 자료를 이동시키는 것보다 더 복잡하기 때문에 단순성에도 불구하고 거의 쓰이지 않는다.

3. 삽입정렬 : 정렬되지 않은 원소를 정렬된 영역의 적당한 위치에 넣어주는 정렬

삽입 정렬 수행 과정
1) 각 단계에서 비 정렬 구역의 첫 번째 원소를 선택한다.
2) 첫 번째 원소를 정렬 구역의 원소와 비교하여 적당한 자리에 삽입한다.

메모리 사용 공간
n개의 원소에 대하여 n개의 메모리 사용

연산 시간
1) Best Case : 자료가 이미 정렬되어 있는 경우
2) Worst Case : 자료가 역순으로 정렬되어 있는 경우

4. 병합정렬 : 정렬된 집합을 합하여 하나의 집합을 만든다.
하나의 배열을 두 개의 균등한 크기로 분할하고 분할된 부분 집합을 정렬한 다음, 두 개의 정렬된 부분 집합를 합하여 전체가 정렬되게 하는 방법이다

분할 : 입력 배열을 같은 크기의 2개의 부분 배열로 분할한다.
정복 : 부분 배열을 정렬한다. 부분 배열의 크기가 충분히 작지 않으면 순환 호출 을 이용하여 다시 분할 정복 방법을 적용한다.
결합 : 정렬된 부분 배열들을 하나의 배열에 합병한다

병합 정렬 수행 과정
1) 정렬되지 않은 리스트를 원소 1개의 부분리스트로 분할한다.
2) 부분 리스트가 하나만 남을 때까지 반복 병합하면서 정렬된 부분 리스트를 생성한다.
3) 마지막 하나 남은 부분 리스트가 정렬된 리스트가 된다.

메모리 사용 공간
n개의 원소에 대하여 (2*n)개의 메모리 공간이 사용된다.

5. 퀵정렬 : 기준 값을 이용하여 부분집합을 두 개로 나누고, 각각을 정렬한다.
기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로, 큰 원소들은 오른쪽 부분 집합으로 분할하여 정렬하는 방식의 정렬 알고리즘

퀵 정렬 수행 과정
1) Left index를 리스트 가장 왼쪽에, Right index를 리스트 가장 오른쪽에 위치시킨다.
2) Left 에 위치한 원소가 pivot보다 큰 값이 나올 때까지 Left를 리스트의 오른쪽 방향으로 이동시킨다. Right에 위치한 원소가 pivot보다 작거나 같은 값이 나올 때까지 리스트의 왼쪽 방향으로 이동시킨다.
3) Left와 Right가 교차하지 않았다면, 두 인덱스에 위치한 원소를 교환한 후, 2번 과정으로 돌아간다.
4) Left 와 Right가 교차했다면, pivot과 Right에 위치한 원소를 교환한다. 교환한 후의 pivot의 위치를 기준으로 왼쪽과 오른쪽의 부분집합을 구분하고, 두 부분 집합에 대하여 퀵 정렬 알고리즘을 재귀 호출한다.

메모리 사용 공간
n개의 원소에 대하여 n개의 메모리 사용

연산 시간
1) Best Case : 왼쪽 부분집합과 오른쪽 부분집합이 정확히 이등분 되는 경우
2) Worst Case : 부분집합이 1개와 n-1개로 치우쳐 분할되는 경우가 반복되는 경우

6. 쉘 정렬 : 일정한 간격으로 부분집합을 나누고, 각 부분집합의 정렬을 수행한다.
일정 간격 떨어져 있는 원소들끼리 부분집합을 구성한 후, 각 부분집합의 원소들에 대해서 삽입 정렬을 수행하되, 간격을 줄여가며 삽입 정렬을 반복하여 전체 원소들을 정렬하는 방식의 정렬 알고리즘

쉘 정렬 수행 과정
1) 첫 번째 원소와 첫 번째 원소로부터 interval만큼 간격이 있는 원소들을 부분 집합에 포함
2) 부분 집합에 포함된 원소들을 삽입 정렬 수행
3) 두 번째 원소와 두 번째 원소로부터 h의 배수만큼 간격이 있는 원소들을 부분 집합에 포함
4) 부분 집합에 포함된 원소들을 삽입 정렬 수행
5) 위 과정을 반복

메모리 사용 공간
n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용

연산 시간 : 원소의 상태와 상관없이 간격에 의해 결정

실행결과

각 정렬의 결과를 보면, 고급 정렬에 해당하는 병합, 퀵, 쉘 정렬의 속도가 더 큰 것을 확인할 수 있다. 이는 각 알고리즘의 구조에 따라 반복문 및 재귀 호출 횟수에 차이가 나기 때문이다.

파이썬 ver.

```
# -*- coding: euc-kr -*-

from Sort_Class import Sort_Class    #정렬 클래스 import

def constant(func):         #constant 사용을 통해 상수 변경 시도시 에러 호출
    def func_set(self, value):
        raise TypeError

    def func_get(self):
        return func()
    return property(func_get, func_set)

class Const(object):      #상수 사용을 위한 클래스 선언
    @constant
    def LENGTH():
        return 30000

    @constant
    def LOW():
        return 0

    @constant
    def HIGH():
        return 100

    @constant
    def TIME_LENGTH():
        return 6

if __name__ == '__main__':
    CONST = Const()
    s_c = Sort_Class(CONST.LENGTH, CONST.TIME_LENGTH, CONST.LOW, CONST.HIGH)
    s_c.Time_Check()      #정렬 시간 체크
    s_c.Print_Time()      #정렬 시간 출력
```

```
# -*- coding: euc-kr -*-

import random, time
import os

class Sort_Class:
```

```
#사용할 변수 및 배열 초기화
def __init__(self, _max, _time_max, _low, _high):
    self.merge_temp = [0 for i in range(_max)]
    self.time_count = 0

    self.arr = [random.randrange(_low, _high) for i in range(_max)]
    self.temp_arr = [0 for i in range(_max)]
    self.time_arr = [0 for i in range(_time_max)]
    self.start = 0
    self.end = 0

def Time_Check(self):

    # 선택정렬 시간 체크
    self.temp_arr = self.arr.copy()
    self.start = time.time()
    self.Selection_Sort(self.temp_arr)
    self.end = time.time()
    self.time_arr[self.time_count] = self.end - self.start
    self.time_count += 1

    # 버블정렬 시간 체크
    self.temp_arr = self.arr.copy()
    self.start = time.time()
    self.Bubble_Sort(self.temp_arr)
    self.end = time.time()
    self.time_arr[self.time_count] = self.end - self.start
    self.time_count += 1

    # 삽입정렬 시간 체크
    self.temp_arr = self.arr.copy()
    self.start = time.time()
    self.Insert_Sort(self.temp_arr)
    self.end = time.time()
    self.time_arr[self.time_count] = self.end - self.start
    self.time_count += 1

    # 쉘 정렬 시간 체크
    self.temp_arr = self.arr.copy()
    self.start = time.time()
    self.Shell_Sort(self.temp_arr)
    self.end = time.time()
    self.time_arr[self.time_count] = self.end - self.start
```

```
        self.time_count += 1

        # 퀵 정렬 시간 체크
        self.temp_arr = self.arr.copy()
        self.start = time.time()
        self.Quick_Sort(self.temp_arr, 0, len(self.temp_arr) - 1)
        self.end = time.time()
        self.time_arr[self.time_count] = self.end - self.start
        self.time_count += 1

        # 병합 정렬 시간 체크
        self.temp_arr = self.arr.copy()
        self.start = time.time()
        self.Merge_Sort(self.temp_arr, 0, len(self.temp_arr) - 1)
        self.end = time.time()
        self.time_arr[self.time_count] = self.end - self.start
        self.time_count = 0


    def Print_Time(self):          # 정렬마다 걸린 시간 출력
        for i in self.arr:
            print(i, end = ' ')
        print('₩n')

        print('선택 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n')
        self.time_count += 1
        print('버블 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n')
        self.time_count += 1
        print('삽입 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n')
        self.time_count += 1
        print('  쉘 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n')
        self.time_count += 1
        print('  퀵 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n')
        self.time_count += 1
        print('병합 정렬 : ' + str(self.time_arr[self.time_count]) + '초₩n₩n')

        print(os.getcwd())

    def Selection_Sort(self, _arr):        # 선택정렬
        _min = 0
        for i in range(len(_arr) - 1):
            _min = i
            for j in range(i + 1, len(_arr)):
```

```
                if _arr[j] < _arr[_min]:
                    _min = j
            _arr[_min], _arr[i],  = _arr[i], _arr[_min]


    def Bubble_Sort(self, _arr):            # 버블정렬
        for i in range(len(_arr) - 1):
            for j in range(len(_arr) - 1):
                if _arr[j + 1] < _arr[j]:
                    _arr[j + 1] ^= _arr[j]
                    _arr[j] ^= _arr[j + 1]
                    _arr[j + 1] ^= _arr[j]


    def Insert_Sort(self, _arr):            # 삽입정렬
        temp = 0
        j = 0
        for i in range(len(_arr) - 1):
            temp = _arr[i + 1]
            j = i
            while j >= 0 and temp < _arr[j]:
                _arr[j + 1] = _arr[j]
                j -= 1
            _arr[j + 1] = temp


    def Shell_Sort(self, _arr):             # 쉘 정렬
        interval = len(_arr)
        j = 0
        temp = 0
        while interval > 0:
            interval //= 2
            for i in range(interval, len(_arr)):
                j = i - interval
                temp = _arr[i]
                while j >= 0 and temp < _arr[j]:
                    _arr[j + interval] = _arr[j]
                    j -= interval
                _arr[j + interval] = temp

    def Partition(self, _arr, _fir, _last):       # 퀵정렬에서 파티션 부분 탐색 및 정렬
        j = _fir - 1
        for i in range(_fir, _last):
```

```
            if _arr[i] <= _arr[_last]:
                j += 1
                _arr[i], _arr[j] = _arr[j], _arr[i]
        _arr[_last], _arr[j + 1] = _arr[j + 1], _arr[_last]
        return j + 1

    def Quick_Sort(self, _arr, _fir, _last):      # 퀵 정렬
        if _fir < _last:
            mid = self.Partition(_arr, _fir, _last)
            self.Quick_Sort(_arr, _fir, mid - 1)
            self.Quick_Sort(_arr, mid + 1, _last)

    # 병합 정렬에서 나누었던 배열 병합 및 정렬 과정
    def Merge(self, _arr, _fir, _mid, _last):
        i = _fir
        j = _mid + 1
        t = _fir
        while i <= _mid and j <= _last:
            if _arr[i] <= _arr[j]:
                self.merge_temp[t] = _arr[i]
                t += 1
                i += 1
            else:
                self.merge_temp[t] = _arr[j]
                t += 1
                j += 1
        while i <= _mid:
            self.merge_temp[t] = _arr[i]
            t += 1
            i += 1
        while j <= _last:
            self.merge_temp[t] = _arr[j]
            t += 1
            j += 1

        i = _fir
        t = _fir
        while i <= _last:
            _arr[i] = self.merge_temp[t]
            t += 1
            i += 1

    def Merge_Sort(self, _arr, _fir, _last):      # 병합 정렬
```

```
if _fir < _last:
        mid = (_fir + _last) // 2
        self.Merge_Sort(_arr, _fir, mid)
        self.Merge_Sort(_arr, mid + 1, _last)
        self.Merge(_arr, _fir, mid, _last)
```