typedef struct \_Student

{

    char id[10];

    char name[512];

} Student;

원래 Student 자료형의 id는 int형이였지만 입력을 int로 받는다면 001과 같은 id의 자릿수를 체크할 수 없기에 문자열의 형태로 입력받았습니다.

// Scenario 1: Number sorting

int compare\_numbers(const void\* lhs, const void\* rhs)

{

    int\* lhs\_int = (int\*)lhs;

    int\* rhs\_int = (int\*)rhs;

    if (\*lhs\_int > \*rhs\_int)return 0;  //오른이 작으면 0

    if (\*lhs\_int == \*rhs\_int)return 1;  //      같으면 1

    if (\*lhs\_int < \*rhs\_int)return 2;  //왼쪽이 작으면 2

}

Void 포인터로 넘겨 받은 lhs와 rhs를 int로 바꿔 대소 비교를 진행했습니다.

// Scenario 2: Single digit string sorting - using ASCII

int compare\_strings(const void\* lhs, const void\* rhs)

{

    char\* lhs\_char = (char\*)lhs;

    char\* rhs\_char = (char\*)rhs;

    char lhs\_charr;

    char rhs\_charr;

    if(\*lhs\_char>='a'&&\*lhs\_char<='z')lhs\_charr= \*lhs\_char-'a'+'A';

    else lhs\_charr=\*lhs\_char;

    if(\*rhs\_char>='a'&&\*rhs\_char<='z')rhs\_charr= \*rhs\_char-'a'+'A';

    else rhs\_charr=\*rhs\_char;

    if (lhs\_charr > rhs\_charr)return 0;  //오른이 작으면 0

    if (lhs\_charr== rhs\_charr){

        if (\*lhs\_char < \*rhs\_char)return 0;  //오른이 작으면 0

        if (\*lhs\_char == \*rhs\_char)return 1; //       같으면 1

        if (\*lhs\_char > \*rhs\_char)return 2;  //왼쪽이 작으면 2

    }

    if (lhs\_charr < rhs\_charr)return 2;  //왼쪽이 작으면 2

}

문자열 비교에는 2개의 규칙이 있었습니다.

1. 대소문자 구별없이 알파벳 순서로 정렬하라
2. 만약 대소문자가 같을 경우 소문자가 왼쪽에 오도록 해라.

이 두가지 규칙을 만족시키기 위해 먼저 lhs,rhs를 char형태인 lhs\_char,rhs\_charr로 바꾸고, 이 수들을 대문자면 그대로 소문자면 대문자로 바꿔서 ,lhs\_charr,rhs\_charr를 만듭니다. 일단 \_charr을 통해 대소문자 구별없이 비교하여 알파벳 순으로 정렬했고, 만약 \_charr가 동일 하다면 lhs\_char,rhs\_char 를 비교해 대소문자를 구별했습니다.

// Scenario 3: Struct sorting. Sort by ID not name.

int compare\_students(const void\* lhs, const void\* rhs)

{

    Student\* lhs\_Stu = (Student\*)lhs;

    Student\* rhs\_Stu = (Student\*)rhs;

    int i=0;

    while(1){

        if (((\*lhs\_Stu).id)[i] == 0||((\*rhs\_Stu).id)[i] == 0)return 1;

        if (((\*lhs\_Stu).id)[i] > ((\*rhs\_Stu).id)[i])return 0;  //오른이 작으면 0

        if (((\*lhs\_Stu).id)[i] < ((\*rhs\_Stu).id)[i])return 2;  //왼쪽이 작으면 2

        if (((\*lhs\_Stu).id)[i] == ((\*rhs\_Stu).id)[i]){

            if(i<9)i++;

            else if(i>=9){

                return 1;//같으면 1

            }

        }

    }

}

Student.id를 char의 형태로 받았기 때문에 단순 숫자비교를 할 수 없었습니다. 어차피 id간 자릿수는 같을 것이기에 높은 자릿수부터 서로 비교하여 1의자리 수까지 같으면 그제서야 같다고 return하고 그게 아니면 값의 차이가 있다는 것이기에 비교할 수 있었습니다.

int compare\_students2(const void\* lhs, const void\* rhs)

{

    Student\* lhs\_Stu = (Student\*)lhs;

    Student\* rhs\_Stu = (Student\*)rhs;

    int i=0;

    int lhs\_id=0,rhs\_id=0;

    for(int i=0;i<10 && (\*lhs\_Stu).id[i]!=0;i++){

        lhs\_id=10\*lhs\_id + (\*lhs\_Stu).id[i];

    }

    for(int i=0;i<10 && (\*rhs\_Stu).id[i]==0;i++){

        rhs\_id=10\*rhs\_id + (\*rhs\_Stu).id[i];

    }

    if (lhs\_id > rhs\_id)return 0;  //오른이 작으면 0

    if (lhs\_id == rhs\_id)return 1; //       같으면 1

    if (lhs\_id < rhs\_id)return 2;  //왼쪽이 작으면 2

}

위에서는 Student.id 정렬을 char형태 그대로 비교를 했는데 여기에선 char로 입력받은 id를 숫자의 형태로 바꾸어 대소비교를 진행했습니다. 하지만 여기서는 id가 int이지만 일반적인 id는 문자로 표현할 수 있기에 위의 방식이 더 범용성 있다고 판단했습니다.

void insertion\_sort(void \*arr, size\_t nmemb, size\_t size, compare\_t compare)

{

    if(size==sizeof(int)){

        int i, j;

        int key;

        int\* arr\_int = (int\*)arr;

        for (i = 1; i < nmemb; i++) {

            key = arr\_int[i];

            for (j = i - 1; j >= 0 && compare(&arr\_int[j], &key) == 0; j--) {

                arr\_int[j + 1] = arr\_int[j];

            }

            arr\_int[j + 1] = key;

        }

    }

    if(size==sizeof(char)){

        int i, j;

        char key;

        char\* arr\_char = (char\*)arr;

        for (i = 1; i < nmemb; i++) {

            key = arr\_char[i];

            for (j = i - 1; j >= 0 && compare(&arr\_char[j], &key) == 0; j--) {

                arr\_char[j + 1] = arr\_char[j];

            }

            arr\_char[j + 1] = key;

        }

    }

    if(size==sizeof(Student)){

        int i, j;

        Student key;

        Student\* arr\_Student = (Student\*)arr;

        for (i = 1; i < nmemb; i++) {

            key = arr\_Student[i];

            for (j = i - 1; j >= 0 && compare(&arr\_Student[j], &key) == 0; j--) {

                arr\_Student[j + 1] = arr\_Student[j];

            }

            arr\_Student[j + 1] = key;

        }

    }

}

처음엔 insertion,selelction,merge sort를 짤 때 입력받은 자료형에 따라 구별을 지어 각 자료형 별로 sorting알고리즘을 짰다가 이후에 포인터 주소로 어찌저찌하면 자료형 상관없이 코드를 짤 수 있다고 하여 아래와 같이 바꿨습니다.

void insertion\_sort(void\* arr, size\_t nmemb, size\_t size, compare\_t compare){

char\* arr\_1= (char\*) arr;

   for (int i = 1; i < nmemb; i++) {

      for(int j = i - 1; j >= 0 && compare(arr\_1 + size \* j, arr\_1 + size \* (j + 1)) == 0; j--){

         void\* temp = malloc(size);

         memcpy(temp, (arr\_1 + size \* (j + 1)), size);

         memcpy((arr\_1 + size \* (j + 1)), (arr\_1 + size \* j), size);

         memcpy((arr\_1 + size \* j), temp, size);

         free(temp);

      }

   }

}

void selection\_sort(void \*arr, size\_t nmemb, size\_t size, compare\_t compare)

{

    int i, j, min\_num = 0;

    char\* arr\_1= (char\*) arr;

    for (i = 0;i < nmemb-1;i++) {

        min\_num = i;

        for (j = i+1;j < nmemb;j++) {

            if (compare(arr\_1+size\*min\_num, arr\_1+size\*j) != 2) {

                min\_num = j;

            }

        }

        void\* temp = malloc(size);

        memcpy(temp, (arr\_1 + size \* (i)), size);

        memcpy((arr\_1 + size \* (i)), (arr\_1 + size \* min\_num), size);

        memcpy((arr\_1 + size \* min\_num), temp, size);

        free(temp);

    }

}

void conqure(void\* arr, int start, int mid, int end,size\_t size,compare\_t compare){

    int i=0, j=0, k=0;

    int arr\_size= end-start;

    char\* arr\_1 = (char\*)arr;

    void\* new\_arr\_void = malloc(size \* (arr\_size+1));

    char\* new\_arr=(char\*)new\_arr\_void;

    while(start+i<mid&&mid+j<end){

        if(compare(arr\_1+size\*(start+i),arr\_1+size\*(mid+j))==0){

            memcpy(new\_arr+size\*k,arr\_1+size\*(mid+j),size);

            j++;

        }

        else{

            memcpy(new\_arr+size\*k,arr\_1+size\*(start+i),size);

            i++;

        }

        k++;

    }

    if(mid+j<end){

        memcpy(new\_arr+size\*k,arr\_1+size\*(mid+j),size\*(arr\_size-k));

    }

    if(start+i<mid){

        memcpy(new\_arr+size\*k,arr\_1+size\*(start+i),size\*(arr\_size-k));

    }

    memcpy(arr\_1+size\*start,new\_arr,size\*arr\_size);

    free(new\_arr);

}

void merge(void \*arr, int start, int arr\_size,size\_t size,compare\_t compare){

    if(arr\_size!=1){

        merge(arr, start, arr\_size/2,size,compare);

        merge(arr, start+arr\_size/2,arr\_size-arr\_size/2,size,compare);

        conqure(arr,start,start+arr\_size/2,start+arr\_size,size,compare);

    }

}

void merge\_sort(void \*arr, size\_t nmemb, size\_t size, compare\_t compare)

{

    merge(arr, 0,nmemb,size,compare);

}

void포인터의 주소를 복사하는게 안되어서 고민을 많이 했는데, memcpy라는 함수를 발견하여 사용하니 문제없이 잘 작동했습니다. 자잘한 오류는 많았으나 디버깅을 통해서 몇 번 수정을 거치니 잘 작동하게 됐습니다.

과제하면서 한 가지 오류를 가지고 2~3시간 잡아 먹은 오류들이 여럿 있었는데 몇가지 말하자면, if문 안에서 대소비교할 때 ==연산자 대신 =를 쓴다던지, 다중포문 돌릴 때 변수 초기화를 안했다던지,, 그리고 작성해둔 코드 백업을 안했는데 블루스크린이 떴다던지… 등이 있습니다. 이외에도 포인터의 개념이 헷갈려서 이렇게 적어보고 저렇게 적어보고 한 것들도 많아서 코드 작성하는데 꽤나 많은 시간이 걸렸습니다. 알고리즘 자체가 이해하기 힘든 알고리즘은 아니라 수도코드로 짜는 것 까지는 금방했는데 역시나 포인터 사용 미숙+ 구조체 사용 미숙 등 실력 부족으로 여러번의 시행 착오를 통해 과제를 해결했습니다.

int/char/student 자료형의 데이터를 insertion/selection/merge sort로 10/100/1000/10000개의 자료를 정렬을 10번씩 시행하고 걸린 시간들의 평균값을 구해 그래프로 나타내봤습니다. Insertion sort는 자료가 10배씩 많아질수록 소요 시간이 약 100배씩 증가합니다. 또한 selection sort의 big-oh 표기법은 insertion sort와 같은 n의 제곱이기에 자료가 10배 많아질수록 소요시간이 100배가 됐습니다. 그러나 알고리즘 특성상 selection이 더 짧은 시간이 걸리는걸 볼 수 있다. 데이터 개수가 적을 때는 merge sort가 가장 느리게 작동했는데 데이터의 개수가 많으면 많아질수록 다른 종류의 정렬보다 훨씐 빨라진다는 것을 확인할 수 있었다.

컴퓨터를 재부팅하고나서 연산을 돌려보니 4초대에서 머무르던 num4 insertion sort가 1초대로 나온다. 파일 시행시간에는 컴퓨터의 상태가 중요하게 작용되긴 한가보다.