Computer Algorithms

제 출 일	2024.12.04	전 공	컴퓨터소프트웨어공학과
과 목	알고리즘	학 번	20233523
담당교수	홍 민	이 름	최윤호

목 차

1. 문제 제시

• data.txt 파일에 있는 정수 데이터를 읽어
들여 지금까지 배운 정렬 방법들(삽입, 선택,
버블, 쉘, 퀵, 히프, 합병)을 적용하여
결과값을 출력하는 프로그램을 작성 하고
각 정렬별 수행 시간 결과를 토대로
분석하시오
- 파일에 정수는 랜덤하게 최대한 많이 생성해서 저장할 것
- 동적 할당을 이용하여 메모리를 할당 할 것

2. 문제 분석

2.1 삽입 정렬

삽입 정렬에 대해 알아보자 삽입 정렬은 정렬되어 있는 부분에 새로운 레코드를 적절한 위칭 삽입하는 과정을 반복하며 정렬을 진행한다. 보통 두번째 레코드부터 왼쪽 레코드와 차례로 비교하는 과정을 통해 정렬한다.

2.2 선택 정렬

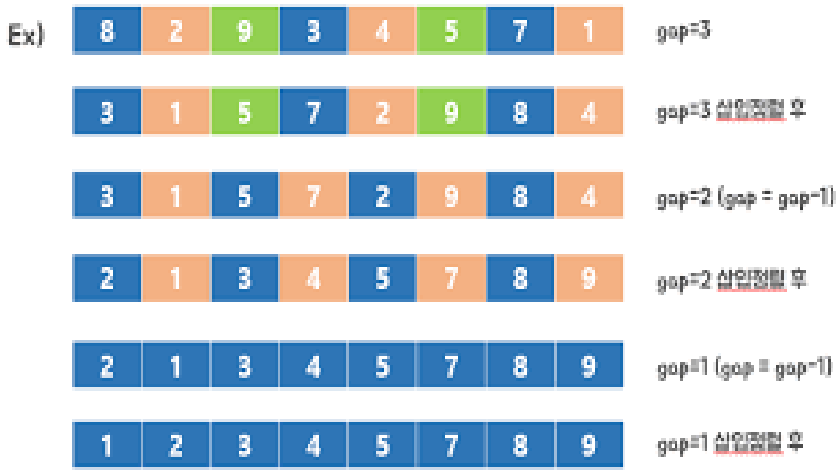
선택 정렬의 경우 오름차순 정렬인 경우 가장 작은 값을 가진 레코드를 선택해 저장해야할 위치인 가장 첫번째 위치의 레코드와 교환한다. 그 후 그 다음 최소값을 찾아 2번째 레코드와 교환한다. 이러한 방식으로 왼쪽에는 정렬된 레코드를 점차 쌓아나가는 방식으로 정렬을 진행한다.

2.3 버블 정렬

버블 정렬의 경우 인접한 두 레코드를 비교해 정렬하는 알고리즘이다. 레코드의 개수만큼 반복하며 반복을 진행함에 따라 끝에서 부터 정렬이 완성된다.

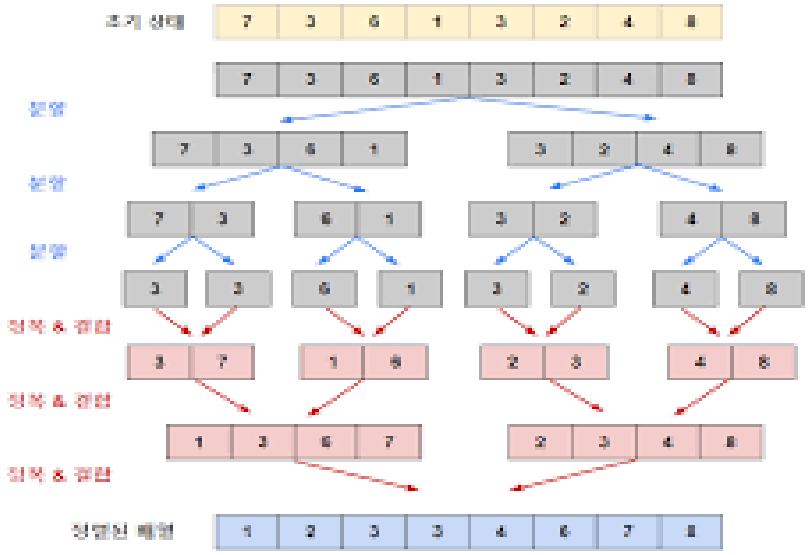
2.4 쉘 정렬

쉘정렬이란 일반적으로 레코드가 저장된 리스트(배열)을 일정 간격의 gap으로 나누고 나눠진 부분리스트(배열)을 삽입정렬을 통해 정렬 후 합치는 과정을 gap을 줄여나감을 통해 정렬하는 방법이다. 어느정도 정렬이 완성된 경우 유리하다.



2.5 합병 렬정

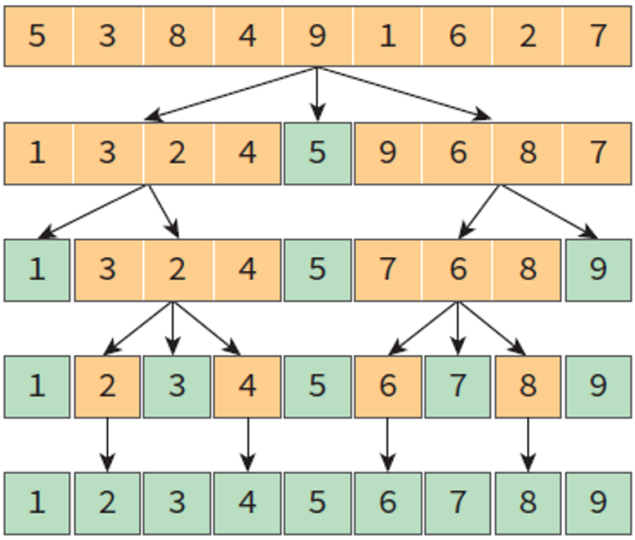
합병 정렬의 경우 분할 정복법을 통해 정렬을 진행한다. 레코드를 저장하는 리스트(배열)을 균등한 크기로 분할하고 분할된 부분 리스트(배열)을 정렬 후 합해 전체 리스트(배열)을 정렬한다. 이 과정에서 더 작은 부분 리스트(배열)로 분할하기 위해 재귀호출을 이용하며, 이 방법에서 실질적인 정렬은 결합과정에서 발생한다.



2.6 퀵 정렬

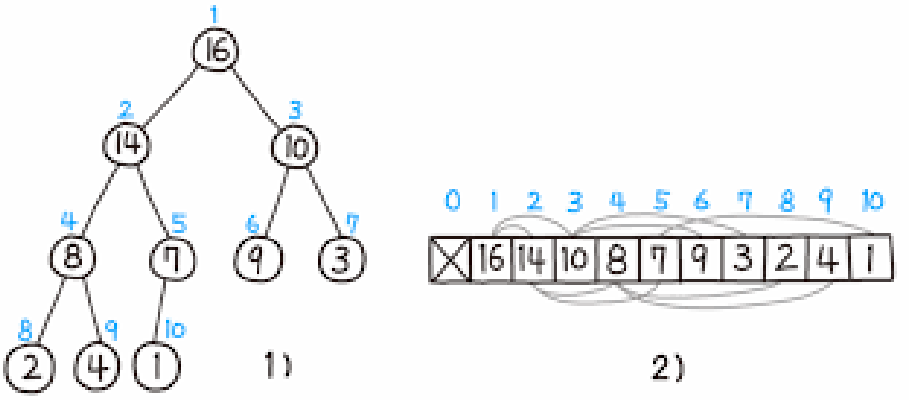
퀵 정렬은 평균적으로 가장 빠른 정렬 방법으로 분할 정복법을 이용한다. 합병 정렬과 비슷하지만 단순히 리스트(배열)의 크기를 기준으로 분할하는 것이 아닌 pivot이라는 데이터를 저장해 이용하며 레코드를 교환하는 형식이다.

※초록색의 값이 pivot



2.7 히프 정렬

히프 정렬의 경우 히프의 특성을 활용한다. 오름차순의 정렬을 예로 들면 레코드를 최대 힙에 차례로 삽입한 뒤 한번에 하나씩 최대 히프에서 삭제 연산을 진행해 저장함으로써 정렬을 진행한다.



3.7 정렬 데이터

정렬 할 데이터를 어떻게 구상할것인지에 대해 말해보고자 한다. 먼저 우리는 각 정렬 방법에 대한 시간을 비교해야 하기 때문에 모든 정렬 과정에 똑같은 데이터를 활용해야한다. 또한 어느정도 정렬이 완성된 데이터에 대한 시간 비교도 필요하기 때문에 간단히 txt파일에 어느 범위의 난수를 입력하는 방식이 아닌 0~n까지의 데이터를 배열에 저장 후. 원하는 퍼센트에 따라 랜덤으로 인덱스 번호를 교환함으로써 무작위로 섞인 데이터를 만들어 txt파일에 저장할 것이다. 이러한 방식이 데이터의 중복을 제거하고, 균일한 테이터를 사용할 수 있는 장점이 있다.

3. 전체 코드

뒷장에 첨부. (파일이 여러개임으로 상단 파일명 주의)

```h
#pragma once
typedef struct {
    int key;
} element;
typedef struct {
    element* heap;
    int heap_size;
} HeapType;

void swap(int arr[], int i, int j);
void print(int arr[], int n);
void selection_sort(int arr[], int n);
void insert_sort(int arr[], int n);
void inc_insert_sort(int arr[], int fir, int last, int gap);
void shell_sort(int arr[], int n);
void bubble_sort(int arr[], int n);
void merge(int sorted[],int arr[], int left, int mid, int right, int n);
void merge_sort(int sorted[],int arr[], int left, int right, int n);
int partition(int arr[], int left, int right);
void quick_sort(int arr[], int left, int right);
HeapType* create();
void init(HeapType* h, int n);
void insert_max_heap(HeapType* h, element item);
element delete_max_heap(HeapType* h);
void heap_sort(element arr[], int n);

```

```c
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include "헤더.h"


//인덱스 번호를 통해 스왑하는 함수
void swap(int arr[], int i, int j)
{
    int temp;
    temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}


//선택  정렬
void selection_sort(int arr[], int n)
{
    int least; //가장 작은 값의 인덱스
    for (int i = 0; i < n - 1; i++) //마지막 요소는 자동으로 정렬되므로 n-1까지만 반복
    {
        least = i; //가장 작은 값의 인덱스를 i로 초기화
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[least])least = j; //가장 작은 값의 인덱스를 찾음
        }
        swap(arr, i, least); //가장 작은 값과 i번째 값을 교환
    }
}

//삽입 정렬
void insert_sort(int arr[], int n)
{
    int key, j;

    for (int i = 1; i < n; i++) { //첫번째 요소는 자동으로 정렬되므로 1부터 시작
        key = arr[i]; //key에 i번째 값을 저장
        for (j = i - 1; j >= 0 && arr[j] > key; j--) //key보다 큰 값을 찾을 때까지 반복
        {
            arr[j + 1] = arr[j];    //key보다 큰 값을 한 칸씩 뒤로 이동
        }
        arr[j + 1] = key; //key값을 삽입

    }
}

//간격을 이용한 삽입 정렬
void inc_insert_sort(int arr[], int fir, int last, int gap)
{
    int key; int j;

    for (int i = fir + gap; i <= last; i += gap)  //간격을 통래 삽입 정렬
```

```
55      {
56          key = arr[i]; //key에 i번째 값을 저장
57          for (j = i - gap; j >= fir && arr[j] > key; j -= gap)  //key보다 큰 값을 찾을
               때까지 반복
58          {
59              arr[j + gap] = arr[j];//key보다 큰 값을 한 칸씩 뒤로 이동
60          }
61          arr[j + gap] = key;//key값을 삽입
62      }
63  }
64
65  //쉘 정렬
66  void shell_sort(int arr[], int n)
67  {
68      int gap; //간격
69      for (gap = n / 2; gap > 0; gap /= 2) //간격을 절반씩 줄여가며 반복
70      {
71          if (gap % 2 == 0)gap++;//간격이 짝수일 경우 홀수로 만들어줌
72
73          for (int i = 0; i < gap; i++)//간격만큼 반복
74          {
75              inc_insert_sort(arr, i, n - 1, gap);//간격을 이용한 삽입 정렬
76          }
77      }
78  }
79
80  //버블 정렬
81  void bubble_sort(int arr[], int n)
82  {
83      for (int i = n - 1; i > 0; i--)//마지막 요소는 자동으로 정렬되므로 n-1까지만 반복
84      {
85          for (int j = 0; j < i; j++)//마지막 요소는 자동으로 정렬되므로 i까지만 반복
86          {
87              if (arr[j] > arr[j + 1])//j번째 값이 j+1번째 값보다 크면
88              {
89                  swap(arr, j, j + 1);//두 값을 스왑
90              }
91          }
92      }
93  }
94
95  //합병 정렬
96  void merge(int sorted[],int arr[], int left, int mid, int right)
97  {
98      int i = left, j = mid + 1, k = left; //i는 왼쪽 배열의 시작, j는 오른쪽 배열의 시
          작, k는 정렬된 배열의 시작
99      int l; //반복문을 위한 변수
100
101     while (i <= mid && j <= right) { //왼쪽 배열과 오른쪽 배열을 비교하면서 정렬
102         if (arr[i] <= arr[j]) //왼쪽 배열의 값이 오른쪽 배열의 값보다 작거나 같으면
103             sorted[k++] = arr[i++];
104         else //오른쪽 배열의 값이 왼쪽 배열의 값보다 작으면
105             sorted[k++] = arr[j++];
106     }
107
108     if (i > mid) { //왼쪽 배열이 먼저 끝난 경우
```

```
109         for (l = j; l <= right; l++)//오른쪽 배열의 남은 값들을 일괄 복사
110             sorted[k++] = arr[l];
111     }
112     else { //오른쪽 배열이 먼저 끝난 경우
113         for (l = i; l <= mid; l++)//왼쪽 배열의 남은 값들을 일괄 복사
114             sorted[k++] = arr[l];
115     }
116
117     //정렬된 배열을 원래 배열에 복사
118     for (int l = left; l <= right; l++)
119         arr[l] = sorted[l];
120
121 }
122
123 //합병 정렬
124 void merge_sort(int sorted[],int arr[], int left, int right)
125 {
126
127     if (left < right) {
128         int mid = (left + right) / 2; //중간값을 구함
129         merge_sort(sorted, arr, left, mid);//왼쪽 배열을 정렬
130         merge_sort(sorted, arr, mid + 1, right);//오른쪽 배열을 정렬
131         merge(sorted, arr, left, mid, right);//정렬된 두 배열을 합병
132     }
133 }
134
135 //퀵 정렬
136 int partition(int arr[], int left, int right)
137 {
138     int pivot, low, high; //피벗, low, high
139     low = left; //low는 왼쪽 끝
140     high = right + 1; //high는 오른쪽 끝
141     pivot = arr[left];//피벗은 가장 왼쪽 값
142
143     do {//low와 high가 교차할 때까지 반복
144         do {//low가 피벗보다 작은 값을 찾을 때까지 반복
145             low++;
146
147         } while (low <= right && arr[low] < pivot);
148         do//high가 피벗보다 큰 값을 찾을 때까지 반복
149         {
150             high--;
151         } while (high >= left && arr[high] > pivot);
152         if (low < high)swap(arr, low, high);//low와 high가 교차하지 않았다면 low와 ↩
              high를 스왑
153     } while (low < high);
154     swap(arr, left, high);//low와 high가 교차하면 피벗과 high를 스왑
155
156     return high;
157 }
158
159 //퀵 정렬
160 void quick_sort(int  arr[], int left, int right)
161 {
162     if (left < right)
163     {
```

```
164         int q = partition(arr, left, right);//피벗을 기준으로 나누어 q에 저장
165         quick_sort(arr, left, q - 1);//왼쪽 부분집합을 퀵정렬
166         quick_sort(arr, q + 1, right);//오른쪽 부분집합을 퀵정렬
167     }
168 }
169
170 // 생성 함수
171 HeapType* create()
172 {
173     return (HeapType*)malloc(sizeof(HeapType));
174 }
175
176 // 초기화 함수
177 void init(HeapType* h, int n)
178 {
179     h->heap_size = 0;
180     h->heap = (HeapType*)malloc(sizeof(HeapType) * n);
181 }
182
183 // 삽입 함수
184 void insert_max_heap(HeapType* h, element item)
185 {
186     int i;
187     i = ++(h->heap_size);
188
189     //  트리를 거슬러 올라가면서 부모 노드와 비교하는 과정
190     while ((i != 1) && (item.key > h->heap[i / 2].key)) {
191         h->heap[i] = h->heap[i / 2];
192         i /= 2;
193     }
194     h->heap[i] = item;     // 새로운 노드를 삽입
195 }
196
197 // 삭제 함수
198 element delete_max_heap(HeapType* h)
199 {
200     int parent, child;
201     element item, temp;
202
203     item = h->heap[1];
204     temp = h->heap[(h->heap_size)--];
205     parent = 1;
206     child = 2;
207     while (child <= h->heap_size) {
208         // 현재 노드의 자식노드 중 더 작은 자식노드를 찾는다.
209         if ((child < h->heap_size) &&
210             (h->heap[child].key) < h->heap[child + 1].key)
211             child++;
212         if (temp.key >= h->heap[child].key) break;
213         // 한 단계 아래로 이동
214         h->heap[parent] = h->heap[child];
215         parent = child;
216         child *= 2;
217     }
218     h->heap[parent] = temp;// 삭제된 노드의 위치에 마지막 노드를 삽입
219     return item;
```

```
220 }
221
222 // 힙 정렬
223 void heap_sort(element arr[], int n)
224 {
225     int i;
226     HeapType* h;
227
228     h = create();
229     init(h, n);
230     for (i = 0; i < n; i++) {//힙에 요소들을 삽입
231         insert_max_heap(h, arr[i]);
232     }
233     for (i = (n - 1); i >= 0; i--) {//힙에서 요소들을 삭제
234         arr[i] = delete_max_heap(h);//삭제된 요소들을 배열에 저장
235     }
236     free(h);
237 }
```

```c
#define _CRT_SECURE_NO_WARNINGS
#include <Windows.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
#include "헤더.h"

//배열 출력 함수
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("₩n-------------------------₩n");
}

//난수 추출 함수
long long random()
{
    return (long long)rand() << 32 | rand();//64비트 난수 생성
}

//데이터 생성 함수
void create_dataset(float ratio, int n)
{
    int* arr = (int*)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)arr[i] = i;//순차적으로 데이터 생성

    //부분적으로 정렬 진행
    srand(time(NULL));
    for (int i = 0; i < n * ratio; i++) {//데이터의 개수에 비례한 횟수만큼 반복
        //두 인덱스를 랜덤하게 선택하여 스왑
        int idx1 = random() % n;
        int idx2 = random() % n;
        int temp = arr[idx1];
        arr[idx1] = arr[idx2];
        arr[idx2] = temp;
    }

    FILE* fp = fopen("data.txt", "w");
    for (int i = 0; i < n; i++)
    {
        fprintf(fp, "%d ", arr[i]);

    }
    free(arr);
    fclose(fp);

}

//각 정렬 시간 비교 함수
void sort(int n)
{   int* original = (int*)malloc(sizeof(int) * n);//원본 데이터
    FILE* fp = fopen("data.txt", "r");
```

```
57      for (int i = 0; i < n; i++)
58      {
59          fscanf(fp, "%d", &original[i]);
60      }
61      fclose(fp);
62
63      int* copy = (int*)malloc(sizeof(int) * n); //복사본 데이터
64
65      //선택정렬
66      memcpy(copy, original, sizeof(int) * n);
67      clock_t start = clock();
68      selection_sort(copy, n);
69      clock_t end = clock();
70      printf("선택 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
71
72      //삽입정렬
73      memcpy(copy, original, sizeof(int) * n);
74      start = clock();
75      insert_sort(copy, n);
76      end = clock();
77      printf("삽입 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
78
79      //버블정렬
80      memcpy(copy, original, sizeof(int) * n);
81      start = clock();
82      bubble_sort(copy, n);
83      end = clock();
84      printf("버블 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
85
86
87      //쉘 정렬
88      memcpy(copy, original, sizeof(int) * n);
89      start = clock();
90      shell_sort(copy, n);
91      end = clock();
92      printf("쉘 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
93
94      //합병 정렬
95      memcpy(copy, original, sizeof(int) * n);
96      int* sorted = (int*)malloc(sizeof(int) * n);
97      start = clock();
98      merge_sort(sorted,copy, 0, n - 1, n);
99      end = clock();
100     free(sorted);
101     printf("합병 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
102
103     //퀵 정렬
104     memcpy(copy, original, sizeof(int) * n);
105     start = clock();
106     quick_sort(copy, 0, n-1);
107     end = clock();
108     printf("퀵 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
109
110     //힙정렬
111     memcpy(copy, original, sizeof(int) * n);
112     start = clock();
```

```
113     heap_sort(copy, n);
114     end = clock();
115     printf("힙 정렬 소요 시간: %lf초₩n", (double)(end - start) / CLOCKS_PER_SEC);
116
117     //파일에 정렬된 데이터   저장
118     fp = fopen("data.txt", "w");
119     for (int i = 0; i < n; i++)
120     {
121         fprintf(fp, "%d ", copy[i]);
122     }
123
124     fclose(fp);
125     free(original);
126     free(copy);
127
128 }
129
130
131
132 int main()
133 {
134
135     int n = 30;//데이터의 개수
136     create_dataset(1, n); //데이터 생성
137
138     int* original = (int*)malloc(sizeof(int) * n);
139     int* copy = (int*)malloc(sizeof(int) * n);
140
141     FILE* fp = fopen("data.txt", "r");
142     for (int i = 0; i < n; i++)
143     {
144         fscanf(fp, "%d", &original[i]);
145     }
146     fclose(fp);
147
148     printf("정렬 전 데이터₩n");
149     print(original, n);
150 ////////////////////실제 데이터 정렬 확인////////////////////////////////
151     //선택정렬
152     memcpy(copy, original,sizeof(int)* n);
153     printf("선택 정렬₩n");
154     selection_sort(copy, n);
155     print(copy,n);
156
157     //삽입정렬
158     memcpy(copy, original, sizeof(int) * n);
159     printf("삽입 정렬₩n");
160     insert_sort(copy, n);
161     print(copy, n);
162
163     //버블정렬
164     memcpy(copy, original, sizeof(int) * n);
165     printf("버블 정렬₩n");
166     bubble_sort(copy, n);
167     print(copy, n);
168
```

```
169     //쉘정렬
170     memcpy(copy, original, sizeof(int) * n);
171     printf("쉘 정렬₩n");
172     shell_sort(copy, n);
173     print(copy, n);
174
175     //합병 정렬
176     memcpy(copy, original, sizeof(int) * n);
177     int* sorted = (int*)malloc(sizeof(int) * n); //정렬된 데이터 저장
178     printf("합병 정렬₩n");
179     merge_sort(sorted,copy,0,n-1,n);
180     free(sorted);
181     print(copy, n);
182
183     //퀵 정렬
184     memcpy(copy, original, sizeof(int) * n);
185     printf("퀵 정렬₩n");
186     quick_sort(copy ,0, n-1);
187     print(copy, n);
188
189     //힙 정렬
190     memcpy(copy, original, sizeof(int) * n);
191     printf("힙 정렬₩n");
192     heap_sort(copy,n);
193     print(copy, n);
194
195     free(original);
196     free(copy);
197
198 ////////////////////////////정렬 상태별 수행 시간 비교////////////////////////////
199
200
201     n = 2000000;//데이터의 개수
202
203     printf("데이터 개수 : %d₩n", n);
204     printf("=========================================================₩n");
205     printf("0%% 정렬된 데이터에 대한 시간 비교₩n");
206     printf("=========================================================₩n");
207     create_dataset(1, n);
208     sort(n);
209
210     printf("=========================================================₩n");
211     printf("50%% 정렬된 데이터에 대한 시간 비교₩n");
212     printf("=========================================================₩n");
213     create_dataset(0.5, n);
214     sort(n);
215
216     printf("=========================================================₩n");
217     printf("80%% 정렬된 데이터에 대한 시간 비교₩n");
218     printf("=========================================================₩n");
219     create_dataset(0.2, n);
220     sort(n);
221
222
223     printf("=========================================================₩n");
```

```
224         printf("95%% 정렬된 데이터에 대한 시간 비교₩n");
225         printf("=================================================₩n");
226         create_dataset(0.05, n);
227         sort(n);
228 
229         return 0;
230 
231     }
```

4. 코드 분석

코드의 구조를 보다 명확하게 파악할 수 있도록 여러 개의 소스파일, 헤더 파일로 작성하였기 때문에 소스 파일별로 코드를 분석하도록 하겠다.

4.1 sort.c

4.1.1 selection_sort

```
24 void selection_sort(int arr[], int n)
25 {
26     int least;
27     for (int i = 0; i < n - 1; i++)
28     {
29         least = i;
30         for (int j = i + 1; j < n; j++)
31         {
32             if (arr[j] < arr[least])least = j;
33         }
34         swap(arr, i, least);
35     }
36 }
```

선택 정렬을 진행하는 함수이다. 가장 최소 값을 찾아 저장할 least 변수를 생성해 두번째 for문을 통해 최소값을 찾은 뒤 첫번째 for의 i를 통해 배열의 i번째와 교환한다.

4.1.2 insert_sort

```
38 void insert_sort(int arr[], int n)
39 {
40     int key, j;
41
42     for (int i = 1; i < n; i++) {
43         key = arr[i];
44         for (j = i - 1; j >= 0 && arr[j] > key; j--)
45         {
46             arr[j + 1] = arr[j];
47         }
48         arr[j + 1] = key;
49
50     }
51 }
```

삽입 정렬을 진행하는 함수이다 첫번째 for문을 통해 두번째 index부터 접근해 두번째 for문을 통해 왼쪽 index의 값을 차례로 비교해 정렬에 따라 한칸씩 오른쪽으로 값을 밀어 넣는다. 그 후 마지막에 감소된 j에 1을 더한 key이 삽입될 위치에 key값을 대입한다.

4.1.3 inc_insert_sort

```
53 void inc_insert_sort(int arr[], int fir, int last, int gap)
54 {
55     int key; int j;
56
57     for (int i = fir + gap; i <= last; i += gap)
58     {
59         key = arr[i];
60         for (j = i - gap; j >= fir && arr[j] > key; j -= gap)
61         {
62             arr[j + gap] = arr[j];
63         }
64         arr[j + gap] = key;
65     }
66 }
```

쉘 정렬시 분할한 부분리스트에 대해 삽입 정렬을 진행해주는 함수이다. 삽입 정렬은 차례로 비교하는것과 다르게 gap을 기준으로 삽입 정렬을 진행한다.

4.1.4 shell_sort

```
68  void shell_sort(int arr[], int n)
69  {
70      int gap;
71      for (gap = n / 2; gap > 0; gap /= 2)
72      {
73          if (gap % 2 == 0)gap++;
74
75          for (int i = 0; i < gap; i++)
76          {
77              inc_insert_sort(arr, i, n - 1, gap);
78          }
79      }
80  }
```

쉘 정렬을 진행한다. 배열을 부분으로 나누기 위해 gap을 데이터 개수인 n을 2로 나누는 과정을 반복한다 if문은 만약 gap이 짝수인 경우 홀수를 만들기 위해 1을 증가시켜 준다. 나눠준 배열의 수 는 gap의 수와 동일하기 때문에 gap만큼 반복하는 for문을 작성 뒤 inc_inser_sort함수를 호출한다.

4.1.5 bubble_sort

```
83  void bubble_sort(int arr[], int n)
84  {
85      for (int i = n - 1; i > 0; i--)
86      {
87          for (int j = 0; j < i; j++)
88          {
89              if (arr[j] > arr[j + 1])
90              {
91                  swap(arr, j, j + 1);
92              }
93          }
94      }
95  }
```

버블 정렬을 진행하는 함수이다. 첫번째 for문을 진행함에 따라 배열에 끝에서 부터 차례로 정렬이 완료되며 두번째 for문을 통해 비교를 진행해 스왑을 진행한다.

4.1.6 merge

```
 97  void merge(int sorted[],int arr[], int left, int mid, int right)
 98  {
 99      int i = left, j = mid + 1, k = left;
100      int l;
101      while (i <= mid && j <= right) {
102          if (arr[i] <= arr[j])
103              sorted[k++] = arr[i++];
104          else
105              sorted[k++] = arr[j++];
106      }
107
108      if (i > mid) {
109          for (l = j; l <= right; l++)
110              sorted[k++] = arr[l];
111      }
113      else {
114          for (l = i; l <= mid; l++)
115              sorted[k++] = arr[l];
116      }
117
118      for (int l = left; l <= right; l++)
119          arr[l] = sorted[l];
120
121  }
```

합병 정렬을 진행하는 과정에서 실질적인 정렬이 이루어 지는 merge를 수앵하는 함수이다. while문을 통해 i, j 를 제어하며 정렬과정에서 나눠서 관리하는 arr배열을 i, j를 통해 mid기준으로 접근해 비교해 정렬이 완료된 값이 들어가는 sorted배열에 저장한다. 그 후 if문을 통해 남은 값들을 sorted 배열에 넣어준다.
그 후 arr배열에 정렬된 값을 넣어준다.

4.1.7 merge_sort

```
123  void merge_sort(int sorted[],int arr[], int left, int right)
124  {
125
126      if (left < right) {
127          int mid = (left + right) / 2;
128          merge_sort(sorted, arr, left, mid);
129          merge_sort(sorted, arr, mid + 1, right);
130          merge(sorted, arr, left, mid, right);
131      }
132  }
```

합병 정렬을 진행하는 함수이다. 데이터의 개수를 나누어 배열을 나누기 위한 기준점인 mid를 저장 후 나눠진 두 부분을 merge_sort를 재귀로 호출해 계속해 나누며 merge함수 호출을 통한 정렬을 진행한다.

4.1.8 partition

```
134  int partition(int arr[], int left, int right)
135  {
136      int pivot, low, high;
137      low = left;
138      high = right + 1;
139      pivot = arr[left];
140
141      do {
142          do {
143              low++;
144
145          } while (low <= right && arr[low] < pivot);
146          do
147          {
148              high--;
149          } while (high >= left && arr[high] > pivot);
150          if (low < high)swap(arr, low, high);
151      } while (low < high);
152      swap(arr, left, high);
153
154      return high;
155  }
```

퀵 정렬을 진행하기 위해 pivot을 통해 정렬시키는 함수이다. low는 나눠진 배열의 처음인 left를 high는 크기보다 하나 더 크게 저장한다. 그 후 pivot도 가장 첫 데이터로 저장후 do-while문을 통해 low가 high보다 크지 않을 때 까지 반복한다. 반복 과정에서 pivot보다 low인덱스의 값이 커질때 까지 low인덱스를 증가시키고 high도 마찬가지로 pivot보다 high인덱스의 값이 커질때 까지 증가시킨다. 그 후 low값이 high보다 작다면 해당 인덱스 값을 서로 교환한다, 그 후 pivo t인덱스의 값을 high 인덱스 값으로 교환 후 바뀐 pivot의 high를 return한다.

4.1.9 quick_sort

```
157  void quick_sort(int  arr[], int left, int right)
158  {
159      if (left < right)
160      {
161          int q = partition(arr, left, right);
162          quick_sort(arr, left, q - 1);
163          quick_sort(arr, q + 1, right);
164      }
165  }
```

퀵 정렬을 진행하는 함수이다. 배열을 나눠서 관리할 기준인 pivot값을 저장한 변수 q를 생성하고 partition함수를 호출해 정렬과 pivot값을 리턴받는다. 그 후 리턴받은 pivot값을 기준으로 배열을 나누어 재귀 호출을 진행해 정렬을 진행한다.

4.1.10 heap_sort

```
218  void heap_sort(element arr[], int n)
219  {
220      int i;
221      HeapType* h;
222
223      h = create();
224      init(h, n);
225      for (i = 0; i < n; i++) {
226          insert_max_heap(h, arr[i]);
227      }
228      for (i = (n - 1); i >= 0; i--) {
229          arr[i] = delete_max_heap(h);
230      }
231      free(h);
232  }
```

히프 정렬을 진행하는 함수이다. 먼저 최대 힙을 생성해 정렬할 데이터를 모두 삽입한 후, 힙 삭제를 통해 가장 큰 값을 배열의 끝에서 부터 처음까지 차례로 저장해 정렬을 완료한다.

4.2 main.c

4.2.1 random

```
 9  long long random()
10  {
11      return (long long)rand() << 32 | rand();
12  }

54  //반복을 이용한 순회
55  int top = -1;//스택의 top을 -1로 초기화
56
```

난수를 생성하기 위한 함수이다. rand함수의 경우 32767이 최대 값이기 때문에 이를 증가시키키위해 rand의 값을 32비트만큼 왼쪽 시프트 후 rand를 다시 호출해 이어 붙히는 방식으로 생성 가능한 값을 늘려준다.

4.2.2 memcpy

string.h 에서 제공하는 함수로 배열의 복사를 진행한다. 하나의 배열을 가지고 정렬을 진행하면 배열의 값이 정렬되어 나오기 때문에 다른 정렬 방법을 진행 할때 같은 데이터를 가지고 진행하지 못해 비교가 불가능하기 때문에 이러한 경우를 ㅂ장지하기 위해 각 정렬 시작전 다른 배열에 데이터 값이 저장된 배열을 복사해 복사된 배열을 통해 정렬을 진행한다.

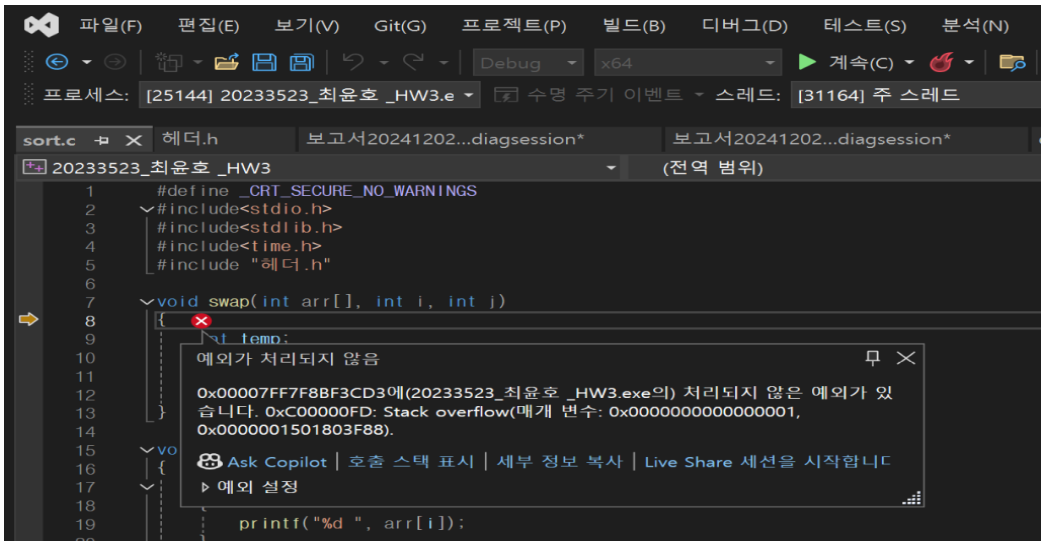
4.2.3 create_dataset

```
26  void create_dataset(float ratio, int n)
27  {
28      int* arr = (int*)malloc(sizeof(int) * n);
29      for (int i = 0; i < n; i++)arr[i] = i;//순차적으로 데이터 생성
30
31      //부분적으로 정렬 진행
32      srand(time(NULL));
33      for (int i = 0; i < n * ratio; i++) {//데이터의 개수에 비례한 횟수만큼 반복
34          //두 인덱스를 랜덤하게 선택하여 스왑
35          int idx1 = random() % n;
36          int idx2 = random() % n;
37          int temp = arr[idx1];
38          arr[idx1] = arr[idx2];
39          arr[idx2] = temp;
40      }
41
42      FILE* fp = fopen("data.txt", "w");
43      for (int i = 0; i < n; i++)
44      {
45          fprintf(fp, "%d ", arr[i]);
46
47      }
48      free(arr);
49      fclose(fp);
50
51  }
```

정렬을 진행할 데이터를 생성해 txt 파일로 생성하는 함수이다. 생성할 데이터의 수인 n을 통해 arr을 동적할당한 후 반복문을 통해 0~n까지 정렬된 데이터를 저장한다. 그 후 데이터를 섞기위해 어느 비율로 섞을지 결정하는 ratio를 통해 비율을 계산하고 그 비율을 n과 곱해 for문의 반복 횟수를 결정한다. 그 후 두개의 인덱스 번호를 랜덤으로 저장해 해당 인덱스 값을 교환해준다. 그 후 섞인 arr의 데이터를 txt파일에 저장한다. 만약 ratio가 1이라면 n번 반복해 0%정렬된 데이터를 0.2라면 n*2번 데이터를 섞기때문에 80% 정렬된 데이터를 가진다. 만약 0이라면 반복하지 않기 때문에 arr은 정렬된 데이터이다.

4.6 그 외

최대한 많은 데이터를 정렬하기 위해서는 스택오버플로우를 방지해야해한다. 데이터를 읽어 배열에 저장하는 과정은 동적할당으로 이루어지기 때문에 스택오버플로우에 영향을 주지 않는다. 하지만 스택오버플로우가 아래와같이 발생하는데



그 이유는 재귀 호출에있다. 합병 정렬과 퀵 정렬은 재귀 호출을 이용하는데 재귀 호출의 경우 호출 시 실행 컨텍스트가 생성되어 스택메모리에 쌓이기 때문에 스택 오버플로우가 발생한다. 이러한 문제를 해결하기 위해 ide상에서 스택 메모리를 아래와 같이 변경 한다.

스택 예약 크기	**104857600**
스택 커밋 크기	

Visual Studio의 경우 1Mb 를 기본으로 설정하지만 위와 같이 100MB로 크기를 늘려줌에 따라 스택 오버플로우를 해결할 수 있다.

5. 실행창

```
정렬 전 데이터
1 0 22 21 23 18 6 27 19 9 3 11 20 29 2 8 26 14 12 5 4 15 28 13 25 16 17 7 10 24
-------------------------
선택 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
삽입 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
버블 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
쉘 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
합병 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
퀵 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
힙 정렬
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
-------------------------
데이터 개수 : 2000000
=========================================================
0% 정렬된 데이터에 대한 시간 비교
=========================================================
선택 정렬 소요 시간: 1686.027000초
삽입 정렬 소요 시간: 937.173000초
버블 정렬 소요 시간: 8704.097000초
쉘 정렬 소요 시간: 0.662000초
합병 정렬 소요 시간: 0.283000초
퀵 정렬 소요 시간: 0.225000초
힙 정렬 소요 시간: 0.396000초
=========================================================
50% 정렬된 데이터에 대한 시간 비교
=========================================================
선택 정렬 소요 시간: 1662.016000초
삽입 정렬 소요 시간: 733.096000초
버블 정렬 소요 시간: 7501.943000초
쉘 정렬 소요 시간: 0.640000초
합병 정렬 소요 시간: 0.242000초
퀵 정렬 소요 시간: 0.203000초
힙 정렬 소요 시간: 0.376000초
=========================================================
80% 정렬된 데이터에 대한 시간 비교
=========================================================
선택 정렬 소요 시간: 1680.150000초
삽입 정렬 소요 시간: 410.859000초
버블 정렬 소요 시간: 4798.225000초
쉘 정렬 소요 시간: 0.580000초
합병 정렬 소요 시간: 0.187000초
퀵 정렬 소요 시간: 0.161000초
힙 정렬 소요 시간: 0.316000초
=========================================================
95% 정렬된 데이터에 대한 시간 비교
=========================================================
선택 정렬 소요 시간: 2053.850000초
삽입 정렬 소요 시간: 128.477000초
버블 정렬 소요 시간: 2531.628000초
쉘 정렬 소요 시간: 0.501000초
합병 정렬 소요 시간: 0.142000초
퀵 정렬 소요 시간: 0.252000초
힙 정렬 소요 시간: 0.275000초
```

6. 느낀 점

먼저 이론으로만 배운 정렬에 대해 직접 코드를 작성하며 line by line 으로 알아갈 수 있었다. 특히 어느정도 정렬된 데이터에 따라 시간을 비교해 보며 각 정렬 방법에 따른 장단점을 체감할 수 있었는데 예를 들어 퀵 정렬의 경우 데이터가 어느정도 정렬된 경우 효율성이 떨어지는 것을 직접 확인할 수 있었다. 또한 최대한 많은 데이터를 정렬하기 위해 스택오버플로우를 해결하는 과정이 필수적인데 이 과정을 통해 메모리의 구조, 그리고 스택에 어떤 데이터가 저장되는지 알 수 있어 뜻깊은 시간이였다. 또한 데이터 자체를 어떻게 구성할 것인지에 대해 깊은 고민을 하였는데. 가장 간단한 범위를 지정해 난수를 받아 구성하는것 외에도 이번 과제에서 사용한 인덱스번호를 랜덤으로 값을 섞는 방식에 대해 이러한 방법도 존재한다는 것을 배울 수 있었다. 결론적으로 구현이 가장 쉬운 버블 정렬을 평소 자주 사용하였는데 이번 출력물을 보고 상황에 따라 적합한 정렬 알고리즘을 사용한다는 점이 실질적으로 와닿았다.