# 자료구조 정렬 보고서
## 퀵 정렬, 삽입 정렬, 합병 정렬

Siwon Yun

November 13, 2023

# Contents

# 1 퀵 정렬(Quick Sort)

## 1.1 개념

퀵 정렬은 Hoare가 제안한 정렬 알고리즘으로, (성능) 최악의 경우 $O(n^2)$의 시간 복잡도를 가지며 평균적으로 $O(n \log n)$의 시간 복잡도를 가진다. (특징) 퀵 정렬은 모든 loop에 대하여 하나의 원소가 배열에 위치되는데, 이 때문에 현재(CPU에서 동작하는) 가장 빠른 정렬 방식으로 알려져 있다. 퀵 정렬의 방법은 다음과 같다.

1. **pivot 설정**
   정렬할 배열에서 난수적으로 pivot을 설정한다. pivot은 배열의 최소이거나 최대인 것을 허용한다.

2. **배열 분할**
   pivot을 기준으로 pivot보다 작은 원소만을 모아둔 배열과 이에 해당하지 않은 원소를 모다운 배열으로 배열을 분할한다.

3. **재귀적 반복**
   분할 된 각 배열에 대하여 1번 과정부터 다시 재귀적으로 반복한다. 배열의 크기가 1이 되면 재귀적 반복을 멈춘다.

## 1.2 성능 분석

quick sort에서 최악의 경우 $O(n^2)$의 시간 복잡도를 가진다. quick sort에서 최악의 경우는 분할 될 때, pivot값이 배열의 최소이거나 최대여서 각각 $n-1$, 0개로 분할되는 경우이다. 이 경우 모든 원소를 선택하여야 하므로 $n$번, 각 선택에서 재귀적으로 선택되므로 $n$번, 즉 $n \times n = n^2$번 반복된다. 따라서 big-O 표기법으로 표기하면 $O(n^2)$이다.

## 1.3 Code

본 code는 구름 및 교과서의 개념만을 참고하여 제작하였다. 전체 code는 부록 A에 첨부하였다.

### 1.3.1 struct

struct data_t(alias: Data)와 result_t(alias: Result)는 각각 배열 및 정렬된 결과를 표현한다. 이는 inin_data 및 init_result 함수를 통해 초기화 가능하다. data_t는 배열의 각 원소 및 크기를 저장하며, result_t는 value를 가지는 binary tree며 left, right는 result_t의 값을 가르키는 pointer이다.

### 1.3.2 quick sort function

재귀적인 함수 quick_sort의 경우 data_t의 data를 argument로 받으며 result_t의 값을 return한다. 해당 함수의 각 호출에서 새로운 result_t의 값을 생성하여 노드를 이어가는 작업이 발생하는데, 새로운 result_t의 값은 다음 code에서 생성된다.

```
Result* new_result = init_result();
```

만약 argument인 배열 data의 원소가 하나 뿐이라면, result_t 값의 value 값 만을 선택 후(left, right는 null로 설정) return한다.

```
if (data->size == 1)
{
  free(data);
  return new_result;
}
```

위 code에서 data는 data_t struct로 quick_sort 함수에 argument로 할당 된 배열이다. quick sort의 1번째 순서인 pivot 설정은 data-¿value의 첫번째 값으로 설정하였다. 입력 값에 대한 정보가 존재하지 아니하여, 임의의 index로 첫번째 index를 설정하여도 무방하다. 다음은 pivot을 설정하는 code이다.

```
int pivot = data->value[0];
```

quick sort의 2번째 순서는 배열을 분할하는 것이다. argument로 할당 받은 배열 data를 기반으로 새로운 배열 next_data_l과 next_data_r을 생성한다. 두 배열 또한 data와 같이 data_t struct이다. 배열 next_data_l에는 pivot 미만의 값으로, 배열 next_data_r에는 pivot 이상의 값으로 구성된 배열이다. 다음은 배열 next_data_l과 next_data_r을 생성하는 code이다.

```
for (int i = 1; i < data->size; i ++)
  {
    if (data->value[i] >= pivot)
    {
      next_data_l->size ++;
      next_data_l->value = (int*) realloc(next_data_l->value, next_data_l->size *
    sizeof(int));
      next_data_l->value[next_data_l->size - 1] = data->value[i];
    }
    else
    {
      next_data_r->size ++;
      next_data_r->value = (int*) realloc(next_data_r->value, next_data_r->size *
    sizeof(int));
      next_data_r->value[next_data_r->size - 1] = data->value[i];
    }
  }
  free(data);
```

배열 data에 존재하는 모든 원소에 대하여 pivot을 기준으로 배열 next_data_l과 next_data_r(이하 묶어서 next_data)로 나누었다. 각각의 value는 이전 malloc으로 초기화하여 개수 증가에 따라 추가로 메모리를 할당할 필요가 있다. 또한, 배열 data의 경우 앞으로 사용되지 않아 메모리를 해제하여도 무방하였다.
quick sort의 3번째 순서는 재귀적으로 반복하는 것이다. 2번째 순서로 생성 된 next_data를 argument 값으로 함수 quick_sort를 재귀한다. 각각의 값은 binary tree인 new_result의 left, right 값으로 할당하였다. 아래는 본 과정의 code이다.

```
  if (next_data_l->size)
  {
    new_result->left = (struct result_t*) malloc(sizeof(struct result_t));
    new_result->left = quick_sort(next_data_l);
  }
  if (next_data_r->size)
  {
    new_result->right = (struct result_t*) malloc(sizeof(struct result_t)
    new_result->right = quick_sort(next_data_r);
  }
```

재귀된 함수에서 배열 new_data의 메모리는 해제되므로 본 함수에서 메모리를 해제할 필요는 없다.

### 1.3.3 result 출력

함수 quick_sort는 argument인 data를 정렬하여 binary tree 형식을 가진 result_t struct의 값을 return 한다. 해당 값은 tree 순회를 통해 정렬된 결과를 출력할 수 있다. 순회 방법에 따라 역순(큰 수부터 작은 수로)으로 출력이 가능하나, 본 code에서는 정순만을 다루었다. 다음은 result를 출력하는 함수이다.

```
void inorder(Result* p) {
  if(p) {
    inorder(p->left);
    printf("%d\n", p->value);
    inorder(p->right);
  }
}
```

수업 시간에 학습한 code와 동일하다.

## 1.4 문제

본 節에서는 quick sort를 통해 배열을 정렬한다.

#### 1.4.1 문제 1

**Problem 1.1: 원소 12개**

**다음을 정렬하라:**

0번째 원소: 5423 1번째 원소: 5324 2번째 원소: 23 3번째 원소: 654
4번째 원소: 23465 5번째 원소: 3425 6번째 원소: 4321 7번째 원소: 454231
8번째 원소: 3452 9번째 원소: 23475 10번째 원소: 876 11번째 원소: 765

**결과:** 454231 23475 23465 5423 5324 4321 3452 3425 876 765 654 23

#### 1.4.2 문제 2

**Problem 1.2: 원소 32개**

**다음을 정렬하라:**

0번째 원소: 1 1번째 원소: 1 2번째 원소: 2 3번째 원소: 3
4번째 원소: 4 5번째 원소: 5 6번째 원소: 6 7번째 원소: 7
8번째 원소: 8 9번째 원소: 9 10번째 원소: 0 11번째 원소: 2
12번째 원소: 2 13번째 원소: 1 14번째 원소: 32 15번째 원소: 43
16번째 원소: 65 17번째 원소: 76 18번째 원소: 32 19번째 원소: 765
20번째 원소: 4321 21번째 원소: 3241 22번째 원소: 3421 23번째 원소: 765
24번째 원소: 75 25번째 원소: 75 26번째 원소: 435 27번째 원소: 3452
28번째 원소: 7645 29번째 원소: 4325 30번째 원소: 643543 31번째 원소: 654

**결과:** 643543 7645 4325 4321 3452 3421 3241 765 765 654 435 76 75 75 65 43 32 32 9 8 7 6 5 4 3 2 2 2 1 1 1 0

# 2 삽입 정렬(Insert Sort)

## 2.1 개념

삽입 정렬(Insert Sort)은 비교적 쉽게 구현 가능한 정렬로 각 정렬을 순서대로 뽑아 앞 부분에 대하여 정렬한다. 입력된 data가 비교적 정렬된 배열과 유사할 때 유리하며, 방식이 간단한 만큼 안정적인 장점이 있다. insert sort는 비교적 매우 간단하여 단계에 대한 설명을 생략한다.

## 2.2 성능 분석

insert sort는 최대 $O(n^2)$의 시간 복잡도를 가진다. 모든 원소에 대하여 정렬하여야 하니 $n$번, 각 원소에 대하여 기존 정렬된 배열을 기준으로 정렬하여야 하니 평균 $\frac{1}{2}n$번이 필요하다. 즉, $n \times \frac{1}{2}n = \frac{1}{2}n^2$번 반복하며 big-O 표기법으로 표현하면 $O(n^2)$의 시간 복잡도를 가진다.

## 2.3 code

본 code 또한 구름 및 교과서의 개념만을 참조하여 제작하였다. 전체 code는 부록 C에 첨부하였다.

### 2.3.1 struct

본 code에서 정의된 struct data_t(alias: Data)의 경우 quick sort에서 정의한 바와 동일하다. 비단 struct result_t(alias: Result)는 binary tree를 위한 node가 아닌 linked list를 위한 node이기에 하나의 next pointer가 존재한다. 다음은 struct result_t의 code이다.

```
typedef struct result_t
{
  int value;
  struct result_t* next;
} Result;
```

### 2.3.2 insert sort function

함수 insert sort의 경우 data_t의 data를 argument로 받으며 result_t의 값을 return한다. result_t는 linked list를 위한 struct로 처음 값(output)은 value를 비우고 next 값만을, 즉 head처럼 사용하였다. return되는 result_t의 값은 다음 code에서 정의된다.

```
Result* output = init_result();
```

위 code에서 init_result는 struct result_t를 초기화하는 code로 quick sort에서 함수 init_result와 유사하다.

insert sort에서는 모든 원소에 대하여 정렬하여야 한다. 즉 다음과 같이 반복할 필요가 있다.

```
for (int i = 0; i < data_size; i++)
```

이후, 배열 된 부분(output)에서 각 원소에 위치를 찾는다. 이는 list 순회를 활용하여 구현하였다.

```
Result* iNode = output;
while (iNode->next)
{
  if (new_node->value <= iNode->next->value)
    break;
  iNode = iNode->next;
}
Result* sNode = iNode->next;
iNode->next = new_node;
new_node->next = sNode;
```

해당 함수의 return 값 또한 순회를 통해 출력하였다.

## 2.4 문제

본 節에서는 insert sort를 통해 배열을 정렬한다.

### 2.4.1 문제 1

**Problem 2.1: 원소 12개**

**다음을 정렬하라:**

element 0: 56 element 1: 43 element 2: 67 element 3: 43 element 4: 65 element 5: 98
element 6: 23 element 7: 87 element 8: 23 element 9: 97 element 10: 23 element 11: 90

**결과:** 23 23 23 43 43 56 65 67 87 90 97 98

### 2.4.2 문제 2

**Problem 2.2: 원소 32개**

**다음을 정렬하라:**

element 0: 435 element 1: 432 element 2: 65 element 3: 243 element 4: 674 element 5: 4532
element 6: 76 element 7: 342 element 8: 76 element 9: 345 element 10: 6 element 11: 345
element 12: 67 element 13: 43 element 14: 78 element 15: 45 element 16: 78 element 17: 45
element 18: 879 element 19: 453 element 20: 765 element 21: 43 element 22: 74
element 23: 78 element 24: 23 element 25: 65 element 26: 23 element 27: 56 element 28: 32
element 29: 65 element 30: 32 element 31: 45

**결과:** 6 23 23 32 32 43 43 45 45 45 56 65 65 65 67 74 76 76 78 78 78 243 342 345 345 432 435 453 674 765 879 4532

# 3 합병 정렬(Merge Sort)

## 3.1 개념

합병 정렬(Merge Sort)은 Von Neumann이 제안한 분할 정복 알고리듬이다. merge sort는 주기억 장치의 용량 한계를 이유로 사용된다. 주기억 장치는 보조기억 장치에 비해 상대적으로 빠르나 용량이 작다. 따라서 merge sort를 사용하여 정렬할 target data를 분할하고 각각에 대하여 주기억 장치에서 정렬을 진행한다. merge sort 또한 $O(n \log n)$의 시간 복잡도를 가진다. merge sort의 단계는 다음과 같다.

1. **분할**
   target data를 사용자가 지정한 크기에 따라 분할한다. 분할된 각각은 run 또는 block이라 부른다. 각각의 block은 quick sort(또는 다른 sort algorithm을 사용하여도 무방함)을 통해 정렬한다.
2. **합병**
   분할된 각각의 block을 정렬하며 merge한다.

### 3.1.1 성능 분석

merge sort는 $O(n \log n)$의 시간 복잡도를 가진다. 합병하는 과정서 $n$번 반복되며, 이는 각각 2개로 나뉜 것에서 반복되므로 총 $n \times \log n$번 반복된다. 따라서 big-O 표기법으로 표기하면 $O(n \log n)$의 시간 복잡도를 가진다.

## 3.2 Code

본 code 또한 구름 및 교과서의 개념만을 참조하여 제작하였다. 전체 code는 부록 B에 첨부하였다. 해당 code는 quick sort의 code를 포함한다.

### 3.2.1 struct

정의된 struct는 quick sort에서 사용한 struct data_t(alias: Data), result_t(alias: Result)를 활용하였다. 즉, 추가로 정의한 struct는 없다. block의 경우 data_t의 이중 pointer를 통해 각 block의 원소를 저장한다.

### 3.2.2 merge_sort function

함수 merge_sort는 target data인 data_t의 data와 block의 개수를 설정하는 int형의 run_size를 argument로 받으며 data_t의 값을 return한다. 함수 quick_sort에서는 result_t의 값을 return하여 binary tree 형식으로 정렬된 데이터를 표현하였으나, 본 함수에서는 정렬된 배열 자체를 data_t 타입으로 return한다. merge sort의 첫 단계는 target data를 분할하는 것이다. c언어에서 (int) / (int)의 경우 내림이 됨을 이용하여 다음과 같이 분할된 block을 생성하고 초기화할 수 있다.

```
1  size_t data_size = data->size;
2  int default_size = data_size / run_size;
3  Data** datas = (Data**) malloc(run_size * sizeof(struct data_t));
4  for (int i = 0; i < run_size - 1; i ++)
5  {
6    datas[i] = (Data*) malloc(sizeof(struct data_t));
7    datas[i]->size = default_size;
8    datas[i]->value = (int*) malloc(default_size * sizeof(int));
9  }
10 datas[run_size - 1] = (Data*) malloc(sizeof(struct data_t));
11 datas[run_size - 1]->size = data_size - (run_size - 1) * default_size;
12 datas[run_size - 1]->value = (int*) malloc(datas[run_size - 1]->size * sizeof(int
     ));
13
14 for (int i = 0; i < run_size - 1; i ++)
15   for (int j = 0; j < default_size; j ++)
16     datas[i]->value[j] = data->value[i * default_size + j];
17 for (int i = 0; i < datas[run_size - 1]->size; i ++)
```

```
    datas[run_size - 1]->value[i] = data->value[(run_size - 1) * default_size + i];
  free(data);
```

data_t의 data의 경우 모든 값(size의 경우 새로운 변수 data_size에 저장함)을 datas에 복사하였기 때문에 더이상 필요하지 않다. 따라서 free를 통해 메모리를 해제하는 것이 바람직하다. 분할된 각각의 block에 대하여 정렬해야 한다. 앞서 언급한 바와 같이 본 code의 경우 quick sort를 활용하여 각 block을 정렬하였다. 아래는 block별 정렬 code이다.

```
for (int i = 0; i < run_size; i ++)
{
  Result* result = quick_sort(datas[i]);
  Data* new_data = init_data();
  new_data->value = (int*) malloc(sizeof(int));
  inorder(result, new_data);
  free(datas[i]);
  datas[i] = new_data;
}
```

merge sort의 두번째 단계는 정렬된 각각의 block를 merge하는 것이다. 이는 각 block에서 가장 작은 값을 비교하며 정렬이 진행된다. 위 code의 경우 역순(큰 값부터 작은 값 순으로)으로 정렬하였기 때문에, 각 block의 마지막 값을 비교한다. block의 마지막 값은 data_t의 idx를 통해 구한다. 비교하여 가장 작은 값은 data_t의 output에 저장한다. code는 다음과 같다.

```
Data* output = init_data();
output->size = data_size;
output->value = (int*) malloc(data_size * sizeof(int));

Data* idx = init_data();
idx->size = run_size;
idx->value = (int*) malloc(run_size * sizeof(int));

for (int i = 0; i < run_size; i ++)
  idx->value[i] = datas[i]->size - 1;

int cnt = 0;
while (1)
{
  int min = -1;
  int min_idx;

  for (int i = 0; i < run_size; i ++)
  {
    if (idx->value[i] >= 0)
    {
      if (min == -1)
      {
        min = datas[i]->value[idx->value[i]];
        min_idx = i;
      }
      else if (datas[i]->value[idx->value[i]] < min)
      {
        min = datas[i]->value[idx->value[i]];
        min_idx = i;
      }
    }
  }

  output->value[cnt] = min;
  idx->value[min_idx] --;

  cnt ++;
  bool isBreak = true;
  for (int i = 0; i < run_size; i ++)
  {
    if (idx->value[i] >= 0)
    {
      isBreak = false;
```

```
      continue;
    }
  }
  if (isBreak)
    break;
}

free(datas);
```

output은 정렬된 값이기 때문에 return한다.

## 3.3 문제

본 節에서는 merge sort를 통해 배열을 정렬한다.

### 3.3.1 문제 1

**Problem 3.1: 원소 17개, run_size: 4**

**다음을 정렬하라:**

element 0: 12 element 1: 34 element 2: 12 element 3: 67 element 4: 78 element 5: 43 element 6: 76 element 7: 32 element 8: 76 element 9: 3 element 10: 87 element 11: 23 element 12: 76 element 13: 2 element 14: 76 element 15: 12 element 16: 65

**결과:** 2 3 12 12 12 23 32 34 43 65 67 76 76 76 76 78 87

### 3.3.2 문제 2

**Problem 3.2: 원소 31개, run_size: 8**

**다음을 정렬하라:**

element 0: 21 element 1: 45 element 2: 78 element 3: 34 element 4: 76 element 5: 21 element 6: 76 element 7: 21 element 8: 56 element 9: 90 element 10: 67 element 11: 43 element 12: 12 element 13: 54 element 14: 78 element 15: 23 element 16: 90 element 17: 56 element 18: 32 element 19: 21 element 20: 87 element 21: 56 element 22: 98 element 23: 34 element 24: 76 element 25: 43 element 26: 87 element 27: 43 element 28: 78 element 29: 34 element 30: 78

**결과:** 12 21 21 21 21 23 32 34 34 34 43 43 43 45 54 56 56 56 67 76 76 76 78 78 78 78 87 87 90 90 98

# A quick sort code

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct data_t
{
  size_t size;
  int* value;
} Data;

typedef struct result_t
{
  int value;
  struct result_t* left;
  struct result_t* right;
} Result;

Data* init_data()
{
  Data* data = (Data*) malloc(sizeof(struct data_t));
  data->size = 0;
  data->value = (void*) 0;

  return data;
}

Result* init_result()
{
  Result* result = (Result*) malloc(sizeof(struct result_t));
  result->value = 0;
  result->left = (void*) 0;
  result->right = (void*) 0;

  return result;
}

Result* quick_sort(Data* data)
{
  Result* new_result = init_result();
  int pivot = data->value[0];

  new_result->value = pivot;
  if (data->size == 1)
  {
    free(data);
    return new_result;
  }

  Data* next_data_l = (Data*) malloc(sizeof(struct data_t));
  next_data_l->value = (int*) malloc(sizeof(int));
  Data* next_data_r = (Data*) malloc(sizeof(struct data_t));
  next_data_r->value = (int*) malloc(sizeof(int));

  for (int i = 1; i < data->size; i ++)
  {
    if (data->value[i] >= pivot)
    {
      next_data_l->size ++;
      next_data_l->value = (int*) realloc(next_data_l->value, next_data_l->size *
    sizeof(int));
      next_data_l->value[next_data_l->size - 1] = data->value[i];
    }
    else
    {
      next_data_r->size ++;
      next_data_r->value = (int*) realloc(next_data_r->value, next_data_r->size *
```

```
    sizeof(int));
      next_data_r->value[next_data_r->size - 1] = data->value[i];
    }
  }
  free(data);

  if (next_data_l->size)
  {
    new_result->left = (struct result_t*) malloc(sizeof(struct result_t));
    new_result->left = quick_sort(next_data_l);
  }
  if (next_data_r->size)
  {
    new_result->right = (struct result_t*) malloc(sizeof(struct result_t));
    new_result->right = quick_sort(next_data_r);
  }

  return new_result;
}

void inorder(Result* p) {
  if(p) {
    inorder(p->left);
    printf("%d\n", p->value);
    inorder(p->right);
  }
}

int main()
{
  size_t root_size = 0;
  printf("number of element: ");
  scanf("%ld", &root_size);

  Data* root = init_data();
  root->value = (int*) malloc(sizeof(int));
  root->size = root_size;
  for (int i = 0; i < root_size; i ++)
  {
    int value;
    printf("element %d: ", i);
    scanf("%d", &value);
    root->value = (int*) realloc(root->value, (i + 1) * sizeof(struct data_t));
    root->value[i] = value;
  }

  Result* result = quick_sort(root);
  inorder(result);

  return 0;
}
```

# B merge sort code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct data_t
{
  size_t size;
  int* value;
} Data;

typedef struct result_t
{
  int value;
```

```
14    struct result_t* left;
15    struct result_t* right;
16  } Result;
17
18  Data* init_data()
19  {
20    Data* data = (Data*) malloc(sizeof(struct data_t));
21    data->size = 0;
22    data->value = (void*) 0;
23
24    return data;
25  }
26
27  Result* init_result()
28  {
29    Result* result = (Result*) malloc(sizeof(struct result_t));
30    result->value = 0;
31    result->left = (void*) 0;
32    result->right = (void*) 0;
33
34    return result;
35  }
36
37  Result* quick_sort(Data* data)
38  {
39    Result* new_result = init_result();
40    int pivot = data->value[0];
41
42    new_result->value = pivot;
43    if (data->size == 1)
44    {
45      free(data);
46      return new_result;
47    }
48
49    Data* next_data_l = (Data*) malloc(sizeof(struct data_t));
50    next_data_l->value = (int*) malloc(sizeof(int));
51    Data* next_data_r = (Data*) malloc(sizeof(struct data_t));
52    next_data_r->value = (int*) malloc(sizeof(int));
53
54    for (int i = 1; i < data->size; i ++)
55    {
56      if (data->value[i] >= pivot)
57      {
58        next_data_l->size ++;
59        next_data_l->value = (int*) realloc(next_data_l->value, next_data_l->size *
      sizeof(int));
60        next_data_l->value[next_data_l->size - 1] = data->value[i];
61      }
62      else
63      {
64        next_data_r->size ++;
65        next_data_r->value = (int*) realloc(next_data_r->value, next_data_r->size *
      sizeof(int));
66        next_data_r->value[next_data_r->size - 1] = data->value[i];
67      }
68    }
69    free(data);
70
71    if (next_data_l->size)
72    {
73      new_result->left = (struct result_t*) malloc(sizeof(struct result_t));
74      new_result->left = quick_sort(next_data_l);
75    }
76    if (next_data_r->size)
77    {
78      new_result->right = (struct result_t*) malloc(sizeof(struct result_t));
```

```
79      new_result->right = quick_sort(next_data_r);
80    }
81
82    return new_result;
83  }
84
85  void inorder(Result* p, Data* result) {
86    if(p) {
87      inorder(p->left, result);
88      result->value = (int*) realloc(result->value, (result->size + 1) * sizeof(int))
        ;
89      result->value[result->size] = p->value;
90      result->size ++;
91      inorder(p->right, result);
92  //      free(p);
93    }
94  }
95
96  Data* merge_sort(Data* data, int run_size)
97  {
98    size_t data_size = data->size;
99    int default_size = data_size / run_size;
100   Data** datas = (Data**) malloc(run_size * sizeof(struct data_t));
101   for (int i = 0; i < run_size - 1; i ++)
102   {
103     datas[i] = (Data*) malloc(sizeof(struct data_t));
104     datas[i]->size = default_size;
105     datas[i]->value = (int*) malloc(default_size * sizeof(int));
106   }
107   datas[run_size - 1] = (Data*) malloc(sizeof(struct data_t));
108   datas[run_size - 1]->size = data_size - (run_size - 1) * default_size;
109   datas[run_size - 1]->value = (int*) malloc(datas[run_size - 1]->size * sizeof(int
        ));
110
111   for (int i = 0; i < run_size - 1; i ++)
112     for (int j = 0; j < default_size; j ++)
113       datas[i]->value[j] = data->value[i * default_size + j];
114   for (int i = 0; i < datas[run_size - 1]->size; i ++)
115     datas[run_size - 1]->value[i] = data->value[(run_size - 1) * default_size + i];
116   free(data);
117
118   for (int i = 0; i < run_size; i ++)
119   {
120     Result* result = quick_sort(datas[i]);
121     Data* new_data = init_data();
122     new_data->value = (int*) malloc(sizeof(int));
123     inorder(result, new_data);
124     free(datas[i]);
125     datas[i] = new_data;
126   }
127
128   Data* output = init_data();
129   output->size = data_size;
130   output->value = (int*) malloc(data_size * sizeof(int));
131
132   Data* idx = init_data();
133   idx->size = run_size;
134   idx->value = (int*) malloc(run_size * sizeof(int));
135
136   for (int i = 0; i < run_size; i ++)
137     idx->value[i] = datas[i]->size - 1;
138
139   int cnt = 0;
140   while (1)
141   {
142     int min = -1;
143     int min_idx;
```

```
144
145     for (int i = 0; i < run_size; i ++)
146     {
147       if (idx->value[i] >= 0)
148       {
149         if (min == -1)
150         {
151           min = datas[i]->value[idx->value[i]];
152           min_idx = i;
153         }
154         else if (datas[i]->value[idx->value[i]] < min)
155         {
156           min = datas[i]->value[idx->value[i]];
157           min_idx = i;
158         }
159       }
160     }
161
162     output->value[cnt] = min;
163     idx->value[min_idx] --;
164
165     cnt ++;
166     bool isBreak = true;
167     for (int i = 0; i < run_size; i ++)
168     {
169       if (idx->value[i] >= 0)
170       {
171         isBreak = false;
172         continue;
173       }
174     }
175     if (isBreak)
176       break;
177   }
178
179   free(datas);
180
181   return output;
182 }
183
184 int main()
185 {
186   size_t root_size;
187   printf("number of element: ");
188   scanf("%ld", &root_size);
189
190   int run_size;
191   printf("run size: ");
192   scanf("%d", &run_size);
193
194   Data* root = init_data();
195   root->value = (int*) malloc(sizeof(int));
196   root->size = root_size;
197   for (int i = 0; i < root_size; i ++)
198   {
199     int value;
200     printf("element %d: ", i);
201     scanf("%d", &value);
202     root->value = (int*) realloc(root->value, (i + 1) * sizeof(struct data_t));
203     root->value[i] = value;
204   }
205
206   Data* output = merge_sort(root, run_size);
207
208   for (int i = 0; i < output->size; i ++)
209   {
210     printf("%d ", output->value[i]);
```

```
  }
  printf("\n");

  return 0;
}
```

# C insert sort code

```
#include <stdio.h>
#include <stdlib.h>

typedef struct data_t
{
  size_t size;
  int* value;
} Data;

typedef struct result_t
{
  int value;
  struct result_t* next;
} Result;

Data* init_data()
{
  Data* data = (Data*) malloc(sizeof(struct data_t));
  data->size = 0;
  data->value = (void*) 0;

  return data;
}

Result* init_result()
{
  Result* result = (Result*) malloc(sizeof(struct result_t));
  result->value = 0;
  result->next = (void*) 0;

  return result;
}

Result* inseart_sort(Data* data)
{
  Result* output = init_result();

  int output_size = 0;

  int data_size = data->size;
  for (int i = 0; i < data_size; i++)
  {
    Result* new_node = (Result*) malloc(sizeof(struct result_t));
    new_node->value = data->value[--data->size];
    data->value = (int*) realloc(data->value, data->size * sizeof(int));

    if (output_size == 0)
    {
      output->next = new_node;
      output_size ++;
      continue;
    }

    output_size ++;

    Result* iNode = output;
    while (iNode->next)
    {
      if (new_node->value <= iNode->next->value)
```

```
60          break;
61        iNode = iNode->next;
62      }
63      Result* sNode = iNode->next;
64      iNode->next = new_node;
65      new_node->next = sNode;
66    }
67    free(data);
68
69    return output;
70  }
71
72  int main()
73  {
74    size_t input_size = 0;
75    printf("number of element: ");
76    scanf("%ld", &input_size);
77
78    Data* input = init_data();
79    input->size = input_size;
80    input->value = (int*) malloc(sizeof(int));
81    for (int i = 0; i < input_size; i++)
82    {
83      int value;
84      printf("element %d: ", i);
85      scanf("%d", &value);
86      input->value = (int*) realloc(input->value, (i + 1) * sizeof(struct data_t));
87      input->value[i] = value;
88    }
89
90    Result* output = inseart_sort(input);
91
92    Result* iNode = output;
93    while (iNode->next)
94    {
95      printf("%d ", iNode->next->value);
96      iNode = iNode->next;
97    }
98    printf("\n");
99
100   return 0;
101 }
```