**———————————————————————————————————————————————————**

[1] 기초 문제

———————————————————————————————————————————————————

1. **정렬 용어와 관련된 문제다. 빈 칸에 들어가는 단어를 맞춰보도록 한다.**
2. ① 오름차순 ② 작은 ③ 큰
3. ① 안정성 ② 상대적인
4. **정렬을 사용하고 있는 곳을 세 가지씩만 나열해 보자.**
5. 영한사전, 1000대 기업, 판매 현황, 이자 정렬, 전국 석차
6. 작업 관리자, dir 명령, db 서버, 글꼴 설정, 아이콘 정렬, 경매 사이트

|  |
| --- |
| 해답  정답으로 간주할 수 있는 것들이 너무 많아서 이번 장에서 설명한 것만 실었습니다. |

1. **간접 정렬과 직접 정렬의 장단점을 비교해 보자.**

|  |  |  |
| --- | --- | --- |
|  | 직접 정렬 | 간접 정렬 |
| 구성 | 원본 배열 | 원본 배열 + 포인터 배열 |
| 다중 정렬 | 원본 배열 여러 개 | 원본 배열 한 개 + 포인터 배열 여러 개 |
| 정렬 성능 | 나쁘다 | 좋다 |
| 탐색 성능 | 좋다 | 조금 떨어진다 |
| 유연성 | 나쁘다 | 좋다 |

|  |
| --- |
| 해답  표에 대한 완전한 설명은 1장의 정렬의 종류에 있습니다. 설명이 너무 길어 기본적인 답안만 제공합니다. |

1. **여러 가지 정렬 방법들이 사용되는 상황에 대해서 말해보자.**

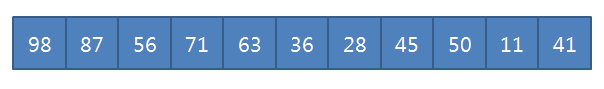
|  |  |
| --- | --- |
| 정렬 | 상황 |
| 거품 정렬 | 대부분 정렬된 배열 |
| 선택 정렬 | 크기가 작은 배열 + 크기가 큰 자료형(구조체) + 불안정성 |
| 삽입 정렬 | 크기가 작은 배열 + 크기가 작은 자료형(int) + 안정성 |
| 히프 정렬 | 균일하게 좋은 성능 + 크기가 큰 배열 + 불안정성 + 자체 메모리 |
| 합병 정렬 | 균일하게 좋은 성능 + 크기가 큰 배열 + 안정성 + 메모리 충분 |
| 빠른 정렬 | 최고의 성능 + 크기가 큰 배열 + 불안정성 + 자체 메모리 |

|  |
| --- |
| 해답  표에 대한 완전한 설명은 2장의 심화 학습에 있습니다. 설명이 너무 길어 기본적인 답안만 제공합니다. |

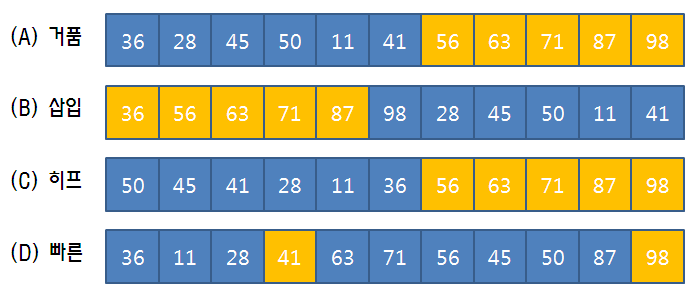
1. **본문에 나왔던 코드에 대해 설명을 하는 곳이다. 정렬을 배우려는 친구에게 설명한다는 생각으로 차분하게 기술해 보자.**

|  |
| --- |
| 해답  자세한 설명이 2장의 심화 학습에 있습니다. 설명이 너무 길어 답안은 생략했습니다. |

1. **11개의 요소를 갖는 정렬되지 않은 배열이 있다. 아래 그림을 보고 질문에 답하라.**



[그림 9-28] 정렬에 따른 배열 재배치 문제



**———————————————————————————————————————————————————**

[2] 기본 실습 문제

———————————————————————————————————————————————————

1. **정렬에 나왔던 코드에서 파생되거나 연관성을 갖는 단편적인 코드들이다. 직접 구현해 보도록 하자.**
   1. 빠른 정렬의 split 함수를 조금 수정했습니다. 음수와 양수라고 규정했기 때문에 피벗이 없어졌고, 반복문을 벗어나서 피벗과 교환하는 코드도 없어졌습니다. 0은 어느 쪽에 포함돼도 괜찮은 걸로 했습니다. 이 함수는 첫 번째 양수의 위치를 반환하므로, 양수밖에 없다면 0을 반환합니다. 만약 음수밖에 없다면 배열의 크기를 반환할 수도 있었지만, 에러를 분명하게 표현하는 것이 낫기 때문에 -1을 반환합니다.

int partition(int\* array, int size)

{

int left = 0, right = size-1;

while(1)

{

while(array[left] < 0)

left++;

while(array[right] > 0)

right--;

if(left >= right)

break;

swap(array+left, array+right);

left++, right--;

}

return (left >= size) ? -1 : left;

}

* 1. 매개 변수로 전달되는 배열은 굳이 정렬되지 않아도 됩니다. 원본의 배치와 다른 형태로 배치하는 것이 목적입니다. 일반적으로는 rand() 함수를 두 번 호출해서 배열 크기보다 월등하게 많이 반복합니다. 그러나, 이 방법은 ‘월등하게 많이’라는 대목이 너무 애매합니다. 또한 성능도 좋지 않습니다.

void shuffle(int\* array, int size)

{

int i;

for(i = 0; i < size\*10; i++)

swap(array+rand()%size, array+rand()%size);

}

이 방법은 rand 함수를 한번만 호출하고, 정확하게 크기만큼 반복합니다. 반복 변수인 i가 핵심인데, 이 변수는 배열 전체를 순회하므로 모든 요소는 최소 한번은 교환될 수 있는 기회를 얻습니다. 여기에 rand 함수로 다시 한번 교환을 하기 때문에, 모든 요소는 평균적으로 두 번 교환될 수 있습니다. 절반만 반복한다면, 기회를 얻지 못하고 제자리를 지키는 요소가 발생하기 때문에 바람직하지 않습니다.

void shuffle(int\* array, int size)

{

int i;

for(i = 0; i < size; i++)

swap(array+i, array+rand()%size);

}

* 1. 최소최대 정렬에서 코드를 가볍게 만드는 가장 중요한 요소는 start를 증가, end를 감소시키는 부분입니다. 한번에 두 개씩 정렬하기 때문에 양 끝에서 가운데로 첨자가 이동합니다. 그런데, 생각보다 다양한 상황이 많이 존재해서 헷갈리는 코드가 나옵니다.

void minMaxSort(int\* array, int size)

{

int i, min, max, start, end;

for(start = 0, end = size-1; start < end; start++, end--)

{

min = max = start;

for(i = start+1; i <= end; i++)

{

if(array[min] > array[i]) min = i;

else if(array[max] < array[i]) max = i;

}

if(max == start && min == end) // 두 번 교환하면 제자리가 되므로, 한번만 교환

{

swap(array+min, array+max);

}

else if(max == start) // 최대값이 첫 번째라면 최대값 먼저 교환

{

swap(array+max, array+end );

swap(array+min, array+start);

}

else // end에는 최소값이 있을 수 있으므로 start 먼저 교환

{

swap(array+min, array+start);

swap(array+max, array+end );

}

}

}

1. **저자는 여러분이 복잡한 코드까지 직접 구현하기를 바라지는 않는다. 대신 비어있는 곳의 코드를 채울 정도는 되어야 한다고 생각한다. 빈 칸에 들어가는 코드는 무엇인가?**
   1. ① reheapDown(array, size, i);

② swap(array, array+i);

③ reheapDown(array, i, 0);

* 1. ① int pivot = split(array, size);

② quickSort(array, pivot );

③ quickSort(array+pivot+1, size-pivot-1);

**———————————————————————————————————————————————————**

[3] 응용 실습 문제

———————————————————————————————————————————————————

1. 코드의 핵심은 메모리 할당을 전체 배열의 크기만큼 한번만 하는 데 있습니다. 그래서, 래퍼 함수를 사용해서 메모리를 할당하고, 매개 변수로 전달합니다. 출력 결과를 보면 2.5배 정도 빠릅니다. 합병 정렬은 정렬에 들어가는 비용보다 할당과 해제에 더 많은 비용이 들어가는 정렬임을 알 수 있습니다.

void mergeSortWrapper(int\* array, int size)

{

int\* temp = malloc(size\*sizeof(int));

mergeSortHi(array, size, temp);

free(temp);

}

void mergeSortHi(int\* array, int size, int\* temp)

{

if(size > 1)

{

mergeSortHi(array, size/2 , temp);

mergeSortHi(array+size/2, size-size/2, temp);

mergeHi(array, size/2, size, temp);

}

}

void mergeHi(int\* array, int middle, int size, int\* temp)

{

int i, left, right;

// int\* temp = malloc(size\*sizeof(int));

left = 0;

right = middle;

for(i = 0; left < middle && right < size; i++)

{

if(array[left] <= array[right]) temp[i] = array[left++];

else temp[i] = array[right++];

}

while(left < middle)

temp[i++] = array[left++];

while(right < size)

temp[i++] = array[right++];

for(i = 0; i < size; i++)

array[i] = temp[i];

// free(temp);

}

[output]

개수 : 65536

저속 : 0.078

고속 : 0.031

개수 : 131072

저속 : 0.140

고속 : 0.047

개수 : 262144

저속 : 0.297

고속 : 0.125

개수 : 524288

저속 : 0.594

고속 : 0.219

1. 고속 정렬 세 가지만 다룹니다. sortTest 함수는 정렬 함수를 매개 변수로 받아서 해당 함수의 성능을 측정합니다. 본문에 나왔던 heapTest 함수와의 차이점은 정렬 제목과 함수 포인터를 매개 변수로 사용한다는 것뿐입니다. 출력 결과는 본문에서 다루었던 것처럼 네 번에 걸쳐 비교합니다.

void main(void)

{

int\* array = NULL;

int i, size;

srand((unsigned) time(NULL));

size = 4096\*16;

array = malloc(size\*8\*sizeof(int));

for(i = 0; i < 4; i++)

{

fillArray(array, size);

printf("개수 : %d\n", size);

sortTest("히프", array, size, heapSort);

sortTest("합병", array, size, mergeSort);

sortTest("빠른", array, size, quickSort);

printf("\n");

size \*= 2;

}

free(array);

}

void sortTest(const char\* name, const int\* array, int size, void (\* sort)(int\*, int))

{

clock\_t start, end;

int\* temp;

temp = malloc(size\*sizeof(int)+4); // 빠른 정렬에서 사용하는 오른쪽 경계 값

temp[size] = INT\_MAX;

memcpy(temp, array, size\*sizeof(int));

start = clock(); sort(temp, size); // 정렬 함수 호출

end = clock();

printf("%s : %.3f\n", name, (end-start) / (double) CLOCKS\_PER\_SEC);

free(temp);

}

1. compare 함수가 핵심입니다. qsort 함수는 배열에 저장된 자료형의 주소를 전달하므로, compare 함수에 전달되는 자료형은 char\*\* 입니다. 주소를 받았으므로 \* 연산자를 붙여서 문자열로 만들고, strcmp 함수로 비교하면 끝입니다.

#pragma warning(disable:4996)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void printWords(const char\*\* words, int size);

int compare(const void\* p1, const void\* p2);

void main(void)

{

char\* words[] =

{

"naver", "page", "web", "net", "black", "hand", "rain", "meal", "hi", "wow",

"news", "movie", "screen", "car", "sky", "snow", "paper", "task", "air", "book",

"2008", "red", "fantasy", "sound", "mickey", "wizard", "voice", "face", "mob", "girl",

"miss", "hope", "sea", "apple", "hotdog", "man", "camera", "think", "freedom", "smile",

"hat", "door", "water", "inline", "soccor", "claim", "venus", "music", "banana", "stop",

"beauty", "green", "gas", "terminal", "wind", "house", "right", "no", "sorry", "mine",

"tv", "cry", "friend", "boys", "street", "scope", "dress", "coffee", "star", "farm",

"animal", "ancient", "future", "disney", "ever", "play", "piano", "listen", "gag", "child",

};

printWords(words, 80);

qsort(words, 80, sizeof(words[0]), compare);

printWords(words, 80);

}

int compare(const void\* p1, const void\* p2)

{

return strcmp(\*(char\*\*) p1, \*(char\*\*) p2);

}

void printWords(const char\*\* words, int size)

{

int i;

for(i = 0; i < size; i++)

{

printf("%8s ", words[i]);

if(i%8 == 7)

printf("\n");

}

printf("\n");

}

1. 이번 문제를 통해 qsort 함수의 구현 원리를 알려주고 싶었습니다. main 함수에서 세 가지 자료형에 대해 정렬하는 모습을 보여줍니다. swap 함수는 별도로 만들지 않고 자료형을 모르기 때문에 memcpy 함수로 직접 구현했습니다. 이런 이유로 성능이 떨어진다는 것도 얘기하고 싶었습니다.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void generalSort(void\* p, int arraySize, int elementSize, int (\* cmp)(const void\*, const void\*));

int cmp1(const void\* p1, const void\* p2);

int cmp2(const void\* p1, const void\* p2);

int cmp3(const void\* p1, const void\* p2);

void main(void)

{

int array1[5] = { 40, 10, 80, 60, 20 };

char array2[5][32] = { "naver", "page", "web", "net", "black" };

char\* array3[5] = { "naver", "page", "web", "net", "black" };

int i;

generalSort(array1, 5, sizeof(\*array1), cmp1);

generalSort(array2, 5, sizeof(\*array2), cmp2);

generalSort(array3, 5, sizeof(\*array3), cmp3);

for(i = 0; i < 5; i++)

printf("%2d : %-5s : %-5s\n", array1[i], array2[i], array3[i]);

}

int cmp1(const void\* p1, const void\* p2)

{

return \*(int\*) p1 - \*(int\*) p2;

}

int cmp2(const void\* p1, const void\* p2)

{

return strcmp(p1, p2);

}

int cmp3(const void\* p1, const void\* p2)

{

return strcmp(\*(char\*\*) p1, \*(char\*\*) p2);

}

void generalSort(void\* p, int arraySize, int elementSize, int (\* cmp)(const void\*, const void\*))

{

char\* array = p, \* temp = malloc(elementSize);

int i, j, max;

for(i = arraySize; i > 1; i--)

{

max = 0;

for(j = 1; j < i; j++)

{

if(cmp(array+j\*elementSize, array+max\*elementSize) > 0)

max = j;

}

memcpy( temp, array+max\*elementSize, elementSize);

memcpy( array+max\*elementSize, array+(i-1)\*elementSize, elementSize);

memcpy(array+(i-1)\*elementSize, temp, elementSize);

}

free(temp);

}