**Lab 06**

|  |
| --- |
| 그림입니다. 원본 그림의 이름: YU_UI_RGB-10.png 원본 그림의 크기: 가로 2256pixel, 세로 3047pixel 프로그램 이름 : Adobe ImageReady |

|  |  |
| --- | --- |
| 과목명 | 프로그래밍 언어 |
| 교수님 | 김영탁 교수님 |
| 이 름 | 김주환 |
| 학 번 | 21812158 |
| 일 자 | 2021.04.08.목 |

**6.1 다중 소스파일 프로그램 구성**

1) BigArray.cpp와 BigArray.h 파일을 준비

2) BigArray.cpp 파일에는 배열의 size가 50,000개 이상인 큰 규모의 배열 데이터를 처리하기 위한 함수들을 구현

3) BigArray.h 파일에는 BigArray.cpp 파일에 포함된 함수들의 함수 원형과 필요한 기호 상수, 전처리기 지시자 등을 포함

**6.2 RAND\_MAX (32767) 보다 더 큰 값의 Big Rand 난수 배열 생성 및 출력**

1) RAND\_MAX (32767) 보다 더 큰 값의 siz와 offset가 주어질 때 난수 (random number) 값의 크기가 (0 ~ size – 1) + offset인 중복되지 않는 난수들을 생성하여 배열에 저장하여 주는 함수 genBigRandArray(int \*bigArray, int size, int offset)를 작성하라.

// 30비트 난수 생성

void genBigRandArray(int\* array, int size, int offset) {

char\* flag; // 난수가 생성되었는지 확인

int count = 0;

unsigned int u\_int32 = 0;

unsigned int bigRand; // 생성된 난수값

flag = (char\*)calloc(size, sizeof(char)); // 동적 할당

if (flag == NULL) {

printf("Error in dynamic creation of big\_array (size = %d) !!!", size);

}

while (count < size) {

u\_int32 = ((long)rand() << 15) | rand();

bigRand = u\_int32 % size;

if (flag[bigRand] == 1) {

continue;

}

else { // 이미 생성된 난수인지 확인

flag[bigRand] = 1;

array[count++] = bigRand + offset;

}

}

array[count] = -1;

free(flag); // 할당 해제

}

2) genBigRandArray() 함수 호출에서 전달되는 bigArray는 동적으로 할당된 메모리 블록의 주소가 전달되며, 정수가 size개 저장될 수 있는 공간이며, 배열로 사용될 수 있다.

3) Big Rand 난수 배열을 출력하기 위해void printBigArraySample(int \*bigArray, int size, int items\_per\_line, int num\_sample\_lines) 함수를 작성하라. 이 함수는 주어진 bigArray[] 배열의 첫 부분에서 한 줄에 items\_per\_line개씩 num\_sample\_lines줄을 출력하며, 중간 부분은 “. . . . .”표시를 출력한다.

// 실행창에 배열 정보 출력

void printBigArraySample(int\* array, int size, int line\_size = 10, int num\_sample\_lines = 2) {

int count = 0; // 몇번째 수인지 확인

int brk\_temp = 0; // 센티널을 만나면 탈출

for (int i = 0; i < num\_sample\_lines; i++) { // 기본 배열 출력

for (int j = 0; j < line\_size; j++) { // line\_size개 만큼 num\_sample\_lines줄 출력

if (count > size) { // 지정한 크기 초과하면 탈출

printf("\n");

return;

}

if (array[count] == -1) { // 센티널 만나면 탈출

brk\_temp++;

break;

}

printf("%8d", array[count]);

count++;

}

printf("\n");

if (brk\_temp == 1)break;

}

if (20 < size && size < 41) { // 20 초과 40 이하인 경우의 배열 출력

count = num\_sample\_lines \* line\_size; // 첫번째 출력 이후로 초기값 재설정

brk\_temp = 0;

for (int i = 0; i < num\_sample\_lines; i++) {

for (int j = 0; j < line\_size; j++) {

if (count > size) {

printf("\n");

return;

}

if (array[count] == -1) {

brk\_temp++;

break;

}

printf("%8d", array[count]);

count++;

}

printf("\n");

if (brk\_temp == 1)break;

}

}

else if (40 < size) { // 40 초과인 경우의 배열 출력

brk\_temp = 0;

if (count < (size - line\_size \* num\_sample\_lines)) // 마지막 num\_sample\_lines줄 만큼 출력

count = size - (line\_size \* num\_sample\_lines); // 하도록 초기값 재설정

if (count >= size) return;

printf("\n . . . . .\n\n");

brk\_temp = 0;

for (int i = 0; i < num\_sample\_lines; i++) {

for (int j = 0; j < line\_size; j++) {

if (count > size) {

printf("\n");

return;

}

if (array[count] == -1) {

brk\_temp++;

break;

}

printf("%8d", array[count]);

count++;

}

printf("\n");

if (brk\_temp == 1)break;

}

printf("\n");

}

}

4) genBigRandArray(), printBigArraySample(), fprintBigArraySample() 함수들은 BigArray.cpp 파일에 구현하고, 그 함수 원형은 BigArray.h에 포함시킬 것.

**6.3 hybridQuickSelectionSort(int \*bigArray, int size)**

1) 주어진 정수 배열을 신속하게 정렬하기 위한 hybridQuickSelectionSort() 함수를 구현하라. 이 정렬함수는 정렬 대상 배열 구간의 원소 개수가 작을 때는 선택 정렬 방식을 사용하고, 정렬 대상 배열 구간의 원소 개수가 많을 때는 퀵정렬 방식을 사용하여야 한다. 선택 정렬과 퀵 정렬 방식의 선택은 BigArray.h에서 기호 상수로 설정된 QUICK\_SELECTION\_THRESHOLD값에 따라 결정하도록 하라.

2) hybridQuickSelectionSort() 함수의 정렬 시간을 최소화하기 위하여 partition 기능을 구현하기 위한 함수 호출을 사용하지 않고 재귀 함수로 실행되는 \_hybridQuickSelectionSort(int \*bigArray, int size, int left, int right, int level) 함수내에 직접 구현할 것.

3) \_hybridQuickSelectionSort(int \*bigArray, int size, int left, int right, int level) 함수 와 hybridQuickSelectionSort() 함수를 BigArray.cpp 파일에 포함시키고, 이 함수들의 함수 원형은 BigArray.h에 포함시킬 것.

**6.4 프로그램 모듈 실행 시간 측정**

1) 큰 규모의 시스템에 포함된 프로그램 모듈의 성능을 평가하기 위하여, 해당 모듈의 실행 시간을 측정하고, 분석할 수 있어야 한다. 이 때, 실행 시간을 측정하기 위한 방법으로 Windows 환경에서는 Performance Counter를 사용하여 microsecond 단위로 정밀하게 측정할 수 있다.

QueryPerformanceCounter (LARGE\_INTEGER & time)를 사용하여, performance counter의 값을 microsecond 단위로 읽을 수 있다. 이 기능을 사용하여, 지정된 프로그램 모듈이 실행되기 이전의 performance counter값과 해당 모듈의 실행이 끝난 후의 performance counter값을 각각 기록하여, 그 차이를 계산함으로써, 해당 프로그램 모듈의 실행시간을 측정할 수 있다.

2) Performance Counter를 사용하기 위해서는 <Windows.h> 헤더파일을 포함시켜야 하며, 미리 해당 변수들을 선언하여야 한다: LARGE\_INTEGER freq, t1, t2; LONGLONG t\_diff; double elapsed\_time;

3) Performance Counter에서 사용되는 클락 주파수를 읽기 위해서는 QueryPerformance Frequency(&freq); 함수를 사용한다.

4) Performance Counter값을 읽기 위해서는 QueryPerformanceCounter (LARGE\_INTEGER & time)를 사용한다.

**6.5 파일 입출력**

1) fopen() 함수를 사용하여 “output.txt” 파일을 출력 모드로 생성하라.

2) BigRand 난수 배열을 지정된 파일로 출력하기 위하여 void fprintBigArraySample() 함수를 작성하라. 이 함수는 printBigArraySample() 함수와 동일한 기능을 수행하며, 단지 지정된 출력파일 fout으로 출력한다.

3) 동적 배열의 생성, bigArray 난수 배열의 생성 및 정렬 등에서 처리되는 내용 (예를 들어 정렬하기 전의 샘플 데이터, 정렬한 후의 샘플 데이터 등)을 출력파일에 저장하라.

4) 프로그램 실행이 완료되면 출력파일을 fclose() 함수를 사용하여 닫을 것.

**6.5 main() 함수와 실행 결과 예시**

1) 메뉴 기반으로 다양한 기능 시험 구성

2) main() 함수

int main(void) {

FILE\* fout;

int menu;

fout = fopen("output.txt", "w");

if (fout == NULL) {

printf("Error in creation of array\_output.txt !!\n");

return -1;

}

while (1) {

fprintf(fout, "\nTest Array Algorithms :\n");

printf("\nTest Array Algorithms :\n");

fprintf(fout, " 1: Performance Comparison of Selection Sort and Quick Sort for Small Integer Array\n");

printf(" 1: Performance Comparison of Selection Sort and Quick Sort for Small Integer Array\n");

fprintf(fout, " 2: Test Big Rand Array (Array Size : 1,000,000 ~ 10,000,000)\n");

printf(" 2: Test Big Rand Array (Array Size : 1,000,000 ~ 10,000,000)\n");

fprintf(fout, " 3: Performance Measurements of hybrid\_QS\_SS for Integer Array\n");

printf(" 3: Performance Measurements of hybrid\_QS\_SS for Integer Array\n");

fprintf(fout, "Input menu (-1 to terminate) : ");

printf("Input menu (-1 to terminate) : ");

scanf("%d", &menu);

printf("\n");

fprintf(fout, "%d\n", menu);

if (menu == -1) break;

switch (menu) {

case 1:

Compare\_Sorting\_Algorithms\_SmallIntArray(fout);

break;

case 2:

testBigRandArray(fout);

break;

case 3:

PM\_Hybrid\_QS\_SS\_IntArray(fout);

break;

default:

break;

}

fflush(fout);

}

fclose(fout);

return 0;

}

3) 화면 출력 결과 1

텍스트이(가) 표시된 사진

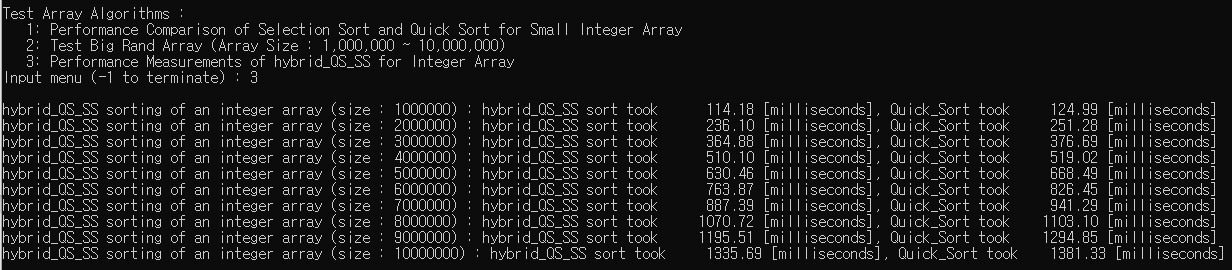
자동 생성된 설명

4) 화면 출력 결과 2

텍스트이(가) 표시된 사진

자동 생성된 설명

5) 화면 출력 결과 3



Oral Test

|  |
| --- |
| **(1) 동적 메모리 할당의 필요성에 대하여 설명하고, 동적 메모리 할당을 사용하여 동적 배열을 생성하는 방법에 대하여 예를 들어 설명하라.**  배열은 생성 크기에 제한이 있고 메모리 낭비가 크다. 하지만, 동적 메모리 할당 같은 경우에는 배열보다 더 크게 생성할 수 있고 필요할 때 필요한 만큼 배열을 생성하기 때문에 메모리 낭비가 없다. 따라서, 동적 메모리 할당이 필요하다.  ex.  array\_c = (int\*)calloc(size, sizeof(int));  free(array\_c);  array\_m = (int\*)malloc(sizeof(int) \* size);  free(array\_m); |

|  |
| --- |
| **(2) RAND\_MAX (32,767) 보다 큰 값인 배열 크기 size와 offset가 주어지면 (0 ~ size – 1) + offset 범위의 값을 가지며 중복되지 않는 난수들을 생성하여 지정된 동적 배열에 담아주는 void genBigRandArray() 함수의 동작 원리를 설명하라.**   * 변수 생성 flag : 난수가 생성되었는지 확인하는 역할 count : 생성한 배열의 크기만큼 반복할 수 있도록 카운트하는 역할 u\_int32 : 30비트 크기의 난수를 저장하는 역할 bigRand : 30비트 난수의 크기를 원하는 크기로 재구성하는 역할 * 동적 할당 calloc을 통해 flag를 array 와 같은 크기로 동적할당 해준다. * 난수 생성 15bit의 난수를 생성한 후 15bit 왼쪽으로 이동해서 빈 자리에 다시 15bit 난수를 생성하고 두 난수를 bitOR 해주면 30bit 크기의 난수를 생성할 수 있다. 이후, % size를 통해 원하는 크기로 재구성해준다. * 이미 생성된 난수인지 확인 flag[BigRand]에 1이 들어 있다면 난수 생성과정을 다시 거쳐서 다른 난수를 생성하고, 0이 들어 있다면 1을 넣고 생성된 난수를 array[count]에 넣고 count++를 실행해준다. * 할당 해제 free(flag)를 통해 동적할당을 해제해준다.   추가로 offset(초기값)을 설정할 수 있는데, 이는 배열을 0부터 시작하는 것이 아니라 원하는 값부터 시작할 수 있도록 설정해준다. |

|  |
| --- |
| **(3) Windows 운영체제에서 제공하는 Performance Counter를 사용하여 함수의 실행시간을 millisecond와 microsecond 단위로 정밀하게 측정하는 방법에 대하여 예를 들어 설명하라.**   * 변수 생성 LARGE\_INTEGER freq, t1, t2; LONGLONG t\_diff; double elapsed\_time; * 동작 QueryPerformanceFrequency(&freq); QueryPerformanceCounter(&t1); 측정하려는 동작 QueryPerformanceCounter(&t2); t\_diff = t2.QuadPart – t2.QuadPart; elapsed\_time = (double)t\_diff / freq.QuadPart;   추가로 elapsed\_time은 초단위로 저장되어있다. 밀리초 마이크로 초로 변환하려면 각각 1,000과 1,000,000을 곱해주면 된다. |

|  |
| --- |
| **(4) hybridQuickSelectionSort() 함수가 어떻게 selection sorting과 quick sorting의 장점을 활용하여 다양한 배열의 크기에 대하여 빠르게 정렬할 수 있는지에 대하여 상세하게 설명하라.**  반복문을 통해 동작하는 선택 정렬은 특정 임계값 이전에, 재귀 함수를 통해 동작하는 퀵정렬은 특정 임계값 이후에 더 빠른 속도를 보여준다. 이를 이용하여 두 정렬을 임계 값을 기준으로 더 빠른 정렬을 실행하여 다양한 배열의 크기에 대하여 빠르게 정렬할 수 있다.  속도 자체로 보면 반복문이 재귀함수보다 빠르기 때문에 배열의 크기가 작을 때는 선택정렬이 빠르게 동작한다. 하지만, 퀵정렬은 배열을 나눠가며 동시에 동작하기 때문에 크기가 커질수록 효율이 좋아진다.  임계값은 초기에 1번을 통해 140으로 설정했으나 속도가 전혀 빨라지지 않아서 조금씩 내려가며 다시 측정했고 50정도로 측정했다. |