프로그래밍언어 (실습)

# 실습 6 – (보충설명) 동적 배열 생성, Hybrid Quick-Selection Sorting


**교수 김 영 탁**

**영남대학교 정보통신공학과**

(Tel : +82-53-810-2497; Fax : +82-53-810-4742

http://antl.yu.ac.kr/; E-mail : ytkim@yu.ac.kr)

# Outline

- 동적 배열 생성
- genBigRandArray()
- Hybrid Quick-Selection-Sorting
  - Selection Sorting
  - Quick Sorting
  - 배열의 크기에 따라 정렬 알고리즘을 선택하여 실행

# 동적 메모리 할당 (Dynamic Memory Allocation)
# 동적 배열 (Dynamic Array)

# 동적 할당 메모리의 개념

## ◆ 프로그램이 메모리를 할당 받는 방법

### ● auto 지역 변수의 메모리 할당

- 프로그램 소스코드에 배열을 선언:
  #define SIZE 100
  int array[SIZE];
- 프로그램 실행 단계에서 필요한 크기가 변경되는 것에 상관없이 항상 일정한 크기를 유지
- 항상 예상되는 최대 크기의 배열을 선언해야 하며, 메모리 낭비가 발생할 수 있음

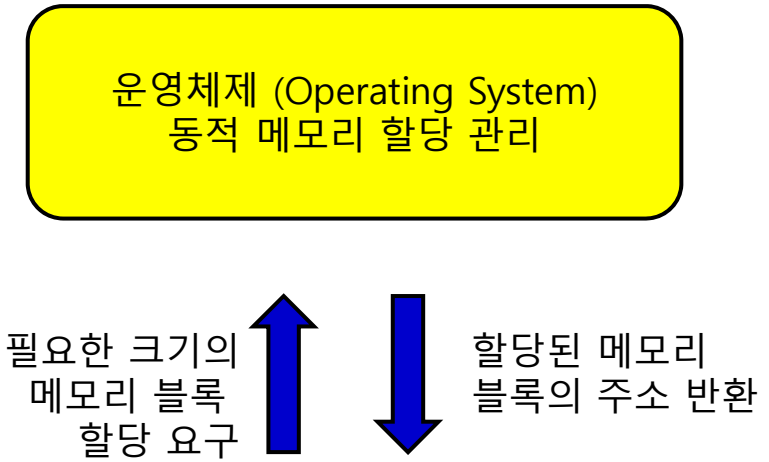
### ● 동적 할당(dynamic allocation)

- 프로그램 실행 단계에서 필요에 따라 동적으로 배열을 생성:
  int size;
  int *darray;
  scanf("%d", &size);
  darray = (int *)calloc(size, sizeof(int));
- 프로그램 실행 단계에서 필요한 크기가 변경된 것 만큼의 메모리 할당
- 항상 필요한 크기의 배열을 구성할 수 있어 메모리 사용에 낭비가 없음

# 동적 (dynamic) 메모리 할당

## ◆ 동적 메모리 할당

- 실행 도중에 동적으로 메모리를 할당 받는 것
- 사용이 끝나면 시스템에 메모리를 반납
- 필요한 만큼만 할당을 받고 메모리를 매우 효율적으로 사용
- calloc(), malloc(), realloc() 계열의 라이브러리 함수를 사용




```
int size;
double *darray;

printf("Input size : ");
scanf("%d", &size);
darray = (double *)calloc(size, sizeof(double));
if( darray == NULL )
{
        ... // 오류 처리
}
 darray[5] = 123.456;

 free(darray);
```

# 동적 메모리 블록 할당 및 반환 함수

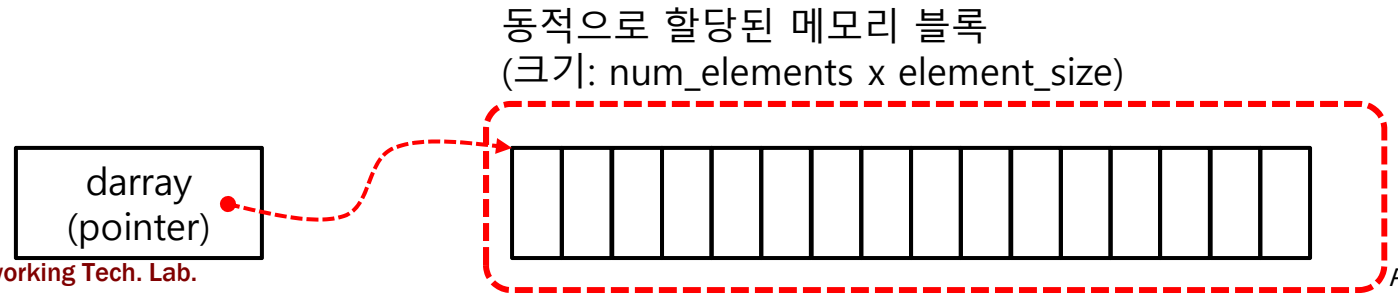
| 분류 | 함수 원형과 인수 | 기 능 |
|---|---|---|
| 동적 메모리 블록 할당 및 반환 <stdlib.h> | void* malloc(size_t size) | 지정된 size 크기의 메모리 블록을 할당하고, 그 시작 주소를 void pointer로 반환 |
| | void *calloc(size_t n, size_t size) | size 크기의 항목을 n개 할당하고, 0으로 초기화한 후, 그 시작 주소를 void pointer로 반환 |
| | void *realloc(void *p, size_t size) | 이전에 할당받아 사용하고 있는 메모리 블록의 크기를 변경<br>p는 현재 사용하고 있는 메모리 블록의 주소, size는 변경하고자 하는 크기; 기존의 데이터 값은 유지된다 |
| | void free(void *p) | 동적 메모리 블록을 시스템에 반환; p는 현재 사용하였던 메모리 블록 주소 |

# 동적 메모리 할당

## ◆ void *calloc(size_t num_elements, size_t element_size)

- size는 바이트의 수
- calloc()함수는 메모리 블록의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환

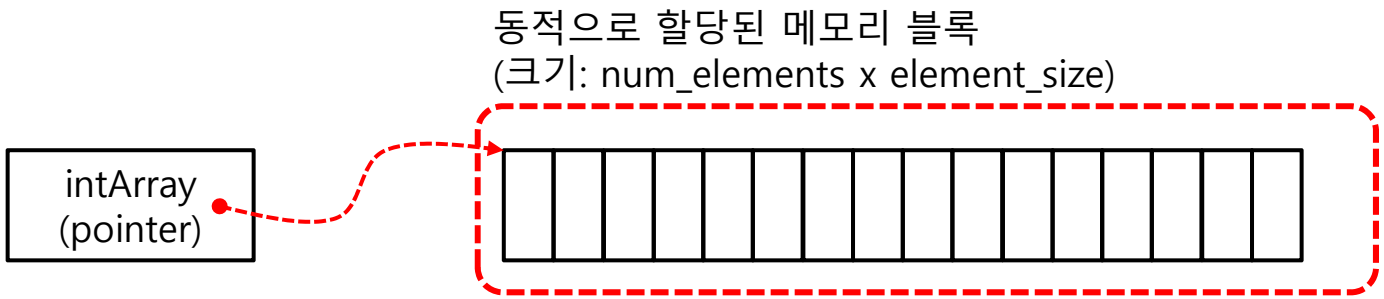
```
int size;
double *darray;
scanf("%d", &size);
darray = (int *)calloc(size, sizeof(double));
if( darray == NULL )
{
        ... // 오류 처리
}
```



# 동적 메모리 할당 블록을 배열로 사용

## ◆ 동적으로 할당된 메모리 블록을 배열과 같이 사용 가능

- darray[0] = 123.456;
- darray[1] = 256.789;
- darray[2] = 31.5;
- ...

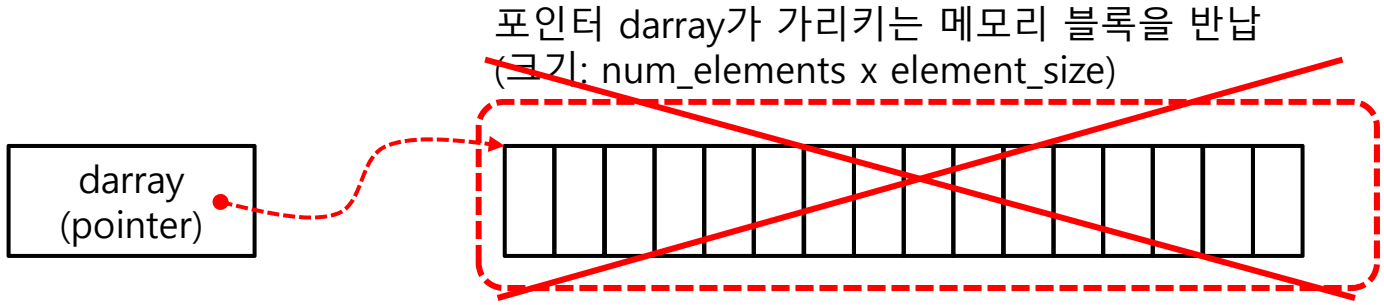


# 동적 메모리 반납

## ◆ void free(void *ptr)

- free()는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- ptr은 calloc()을 이용하여 동적 할당된 메모리를 가리키는 포인터

```
int size;
double *darray;
scanf("%d", &size);
darray = (double *)calloc(size, sizeof(double));
...
free(darray);
darray = NULL;  // 메모리 블록 반납 후에는 반드시  NULL로 설정
```



# Big Rand Array

# RAND_MAX(32,767)보다 더 큰 난수의 생성

## ◆ rand() 함수의 한계

- rand() randomly generates 0 ~ RAND_MAX (32,767) integer value
- if big random numbers (e.g., 0 ~ 500,000) are necessary, rand() cannot be used

## ◆ 32,767보다 더 큰 난수로 구성된 배열 생성

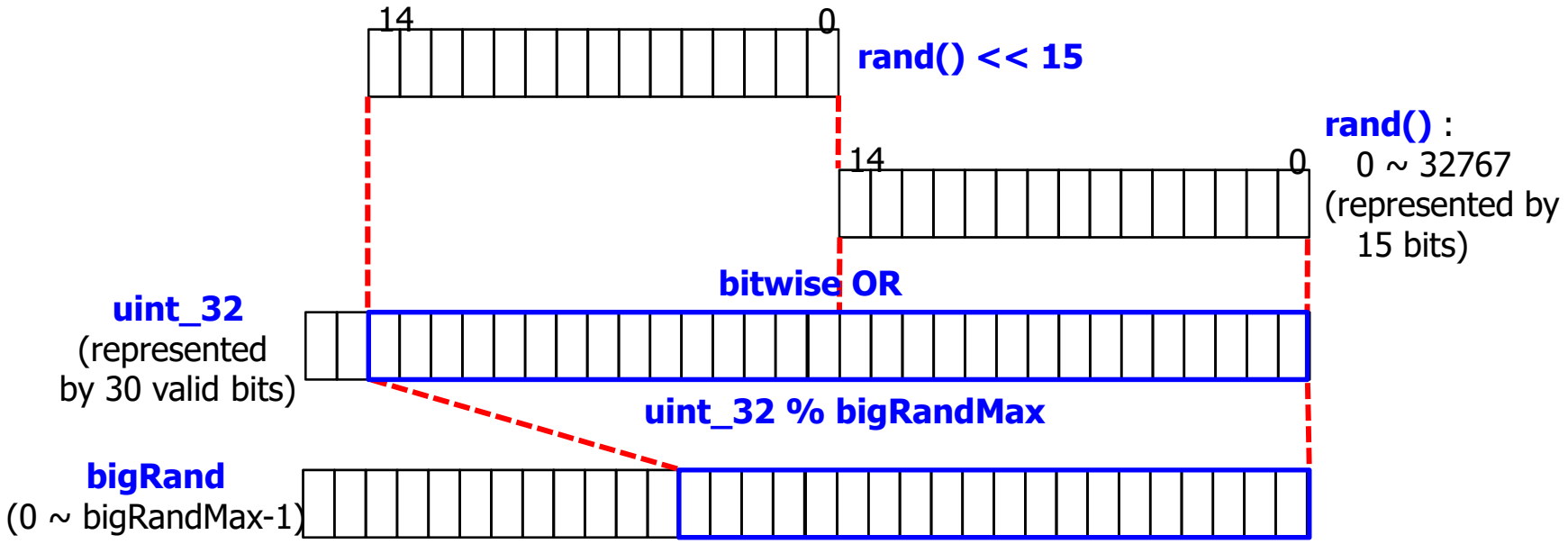
- **genBigRandArray(int *array, int size, offset)**
- generates *non-duplicated* big random numbers in the range of (0 ~ size-1) + offset, where size can be bigger than RAND_MAX (32,767)
- as result, the non-duplicated random numbers are contained in array[]

# genBigRandArray()

## ◆ Generation of random numbers with size (up to $2^{30}$)

```
unsigned int uint_32, bigRand;

uint_32 = ((unsigned int)rand() << 15) | rand();  // bitwise left shift, bitwise OR
bigRand = uint_32 % size;
```



```
void genBigRandArray(int *bigRandArray, int size, int offset)
{
    char *flag; // for checking duplicated rand_data
    int count = 0;
    unsigned int u_int32 = 0, bigRand;

    flag = (char *)calloc(size, sizeof(char));
    while (count < size)
    {
        u_int32 = ((long)rand() << 15);
        u_int32 = u_int32 + rand();
        bigRand = u_int32 % size;
        if (flag[bigRand] == 1)
        {
            continue;  // bigRand has been generated already !
                       // So, retry to get non-duplicated random number
        }
        else
        {
            flag[bigRand] = 1;
            bigRandArray[count] = bigRand + offset;
            count = count + 1;
        }
    }
    free(flag);
}
```

```
void printBigArraySample(int *array, int size, int items_per_line, int num_sample_lines)
{
	int last_block_start;
	int count = 0;

	for (int i=0; i<num_sample_lines; i++)
	{
		for (int j = 0; j < items_per_line; j++)
		{
			if (count >= size)
			{
				printf("\n");
				return;
			}
			printf("%7d ", array[count]);
			count++;
		}
		printf("\n");
	}  // end for

	if (count >= size)
		return;
	if (count < (size - items_per_line * num_sample_lines))
	   count = size - items_per_line * num_sample_lines;

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

	for (int i = 0; i < num_sample_lines; i++)
	{
		for (int j = 0; j< items_per_line; j++)
		{
			if (count >= size)
			{
				printf("\n");
				return;
			}
			printf("%7d ", array[count]);
			count++;
		}
		printf("\n");
	}  // end for
	printf("\n");
}
```

# testBigRandArray()

```
void testBigRandArray(FILE *fout)
{
	int *bigArray;
	fprintf(fout, "Testing Big Integer Random Arrays(size = 1,000,000 ~ 10,000,000): \n");
	printf("Testing Big Integer Random Arrays(size = 1,000,000 ~ 10,000,000): \n");
	for (int size = 5000000; size <= 10000000; size += 5000000)
	{
		bigArray = (int *)calloc(size, sizeof(int));
		if (bigArray == NULL)
		{
			printf("Error in memory allocation for big_int_array of size (%d) !!!\n",
			   big_size);
			return;
		}
		printf("Generating Big Integer array (size = %8d) . . . . ", size);
		genBigRandArray(bigArray, size, 0);
		fprintf(fout, "Generated Big Integer array (size = %8d): \n", size);
		fprintBigArraySample(fout, bigArray, size, 10, 2);
		printf("\nGenerated Big Integer array (size = %8d): \n", size);
		printBigArraySample(bigArray, size, 10, 2);
		quickSort(bigArray, size);
		printf("Sorted Big Integer array (size = %8d): \n", size);
		printBigArraySample(bigArray, size, 10, 2);
		fprintf(fout, "Sorted Big Integer array (size = %8d): \n", size);
		fprintBigArraySample(fout, bigArray, size, 10, 2);
		free(bigArray);
		fflush(fout);
	}
}
```

# 실행 결과

```
Test Array Algorithms  :
   1: Performance Comparison of Selection Sort and Quick Sort for Small Integer Array
   2: Test Big Rand Array (Array Size: 1,000,000 ~ 10,000,000)
   3: Performance Measurements of hybrid_QS_SS for Integer Array
Input menu (-1 to terminate) : 2
Testing Big Integer Random Arrays(size = 1,000,000 ~ 10,000,000):
Generating Big Integer array (size =  5000000) . . . .
Generated Big Integer array (size =  5000000):
 1068821  4397704  1504833  1072351   392668   777818  2618175  1794108   897519  4284614
 2682316   361224   506124   994166  3291752  4563006  2776661  3562393  4260214  2200929
  . . . . .
 2389259  1944816   658936  2401476  1682586  3614526  1245376  2138140   452630  4396186
 1513717  3901637  3773763  4140969  4440290  3372150    38864   318655  2416552  2783686
Sorted Big Integer array (size =  5000000):
       0        1        2        3        4        5        6        7        8        9
      10       11       12       13       14       15       16       17       18       19
  . . . . .
 4999980  4999981  4999982  4999983  4999984  4999985  4999986  4999987  4999988  4999989
 4999990  4999991  4999992  4999993  4999994  4999995  4999996  4999997  4999998  4999999
Generating Big Integer array (size = 10000000) . . . .
Generated Big Integer array (size = 10000000):
 6481909   713896  5948314  5634162  5521656   869982  3207439   100121  8144000  9776623
  263202  5584004  2252755  9807814  2325388  1738911  2512697  5486959  1330119  2506425
  . . . . .
 9632005  8906349  3877017  1687273  6344316  3286839  9920781  5906139  2271338   522492
 9514642  2726704   741333  6307980  4872755  7794861  8081166   551641  5922928  6484560
Sorted Big Integer array (size = 10000000):
       0        1        2        3        4        5        6        7        8        9
      10       11       12       13       14       15       16       17       18       19
  . . . . .
 9999980  9999981  9999982  9999983  9999984  9999985  9999986  9999987  9999988  9999989
 9999990  9999991  9999992  9999993  9999994  9999995  9999996  9999997  9999998  9999999
```

# Hybrid Quick Selection Sorting

# Hybrid_Quick_Selection_Sort()

## ◆ Hybrid Quick – Selection Sorting 개념

- 주어진 정수 배열을 신속하게 정렬하기 위한 hybridQuickSelectionSort(int *bigArray, int size)를 구현
- 정렬 대상 배열 구간의 원소 개수가 작을 때는 선택정렬 방식을 사용하고, 정렬 대상 배열 구간의 원소 개수가 많을 때는 퀵 정렬 방식을 사용
- 선택 정렬과 퀵 정렬 방식의 선택은 BigArray.h에서 기호 상수로 설정된 QUICK_SELECTION_THRESHOLD 값에 따라 결정
- hybridQuickSelectionSort(int *bigArray, int size) 함수의 정렬 시간을 최소화하기 위하여 partition 기능은 별도의 함수 호출을 사용하지 않고 재귀 함수로 실행되는 _hybridQuickSelectionSort(int int *bigArray, int size, int left, int right, int level) 함수 내에 직접 구현

# Compare Sorting Algorithms

```
void Compare_Sorting_Algorithms_SmallIntArray(FILE *fout)
{
	int *array;
	LARGE_INTEGER freq, t1, t2;
	LONGLONG t_diff;
	double elapsed_time;

	QueryPerformanceFrequency(&freq);
	srand(time(NULL));
	for (int size = 5; size <= 200; size += 5)
	{
		array = (int *)calloc(size, sizeof(int));
		if (array == NULL)
		{
			printf("Error in memory allocation for big_array of size (%d) !!!\n", size);
			return;
		}
		genBigRandArray(array, size, 0);
		fprintf(fout, "Big integer array before quick sorting : \n");
		fprintBigArraySample(fout, array, size, 20, 2);
		printf("Sorting of an integer array (size : %3d) : ", size);
		QueryPerformanceCounter(&t1);
		quickSort(array, size);
		QueryPerformanceCounter(&t2);
```

```
            fprintf(fout, "Big integer array after quick sorting : \n");
            fprintBigArraySample(fout, array, size, 20, 2);
            t_diff = t2.QuadPart - t1.QuadPart;
            elapsed_time = (double)t_diff / freq.QuadPart;
            fprintf(fout, "Quick_Sort took %10.2lf [milliseconds]\n", elapsed_time * 1000.0);
            printf(" Quick_Sort took %10.2lf [milliseconds], ", elapsed_time * 1000.0);

            fprintf(fout, "Shuffling word list . . . \n");
            suffleArray(array, size);
            fprintf(fout, "Big integer array before quick sorting : \n");
            fprintBigArraySample(fout, array, size, 20, 2);
            //printf("Quick sorting of an integer array (size : %7d) . . . . ", size);
            QueryPerformanceCounter(&t1);
            selectionSort(array, size);
            QueryPerformanceCounter(&t2);
            fprintf(fout, "Big integer array after quick sorting : \n");
            fprintBigArraySample(fout, array, size, 20, 2);
            t_diff = t2.QuadPart - t1.QuadPart;
            elapsed_time = (double)t_diff / freq.QuadPart;
            fprintf(fout, "Selection_Sort took %10.2lf [milliseconds]\n", elapsed_time * 1000.0);
            printf("  Selection_Sort took %10.2lf [milliseconds]\n", elapsed_time * 1000.0);
            free(array);
        } // end for
}
```

# 정렬 대상 배열의 크기가 작을 때의 Quick Sorting과 Selection Sorting의 비교 결과

```
Sorting of an integer array (size :   5) :  Quick_Sort took      0.51 [micro-seconds],   Selection_Sort took      0.26 [micro-seconds]
Sorting of an integer array (size :  10) :  Quick_Sort took      0.77 [micro-seconds],   Selection_Sort took      0.26 [micro-seconds]
Sorting of an integer array (size :  15) :  Quick_Sort took      1.03 [micro-seconds],   Selection_Sort took      0.77 [micro-seconds]
Sorting of an integer array (size :  20) :  Quick_Sort took      1.28 [micro-seconds],   Selection_Sort took      1.03 [micro-seconds]
Sorting of an integer array (size :  25) :  Quick_Sort took      1.80 [micro-seconds],   Selection_Sort took      1.28 [micro-seconds]
Sorting of an integer array (size :  30) :  Quick_Sort took      1.80 [micro-seconds],   Selection_Sort took      1.54 [micro-seconds]
Sorting of an integer array (size :  35) :  Quick_Sort took      2.31 [micro-seconds],   Selection_Sort took      2.05 [micro-seconds]
Sorting of an integer array (size :  40) :  Quick_Sort took      2.82 [micro-seconds],   Selection_Sort took      2.82 [micro-seconds]
Sorting of an integer array (size :  45) :  Quick_Sort took      3.08 [micro-seconds],   Selection_Sort took      3.08 [micro-seconds]
Sorting of an integer array (size :  50) :  Quick_Sort took      3.34 [micro-seconds],   Selection_Sort took      3.59 [micro-seconds]
Sorting of an integer array (size :  55) :  Quick_Sort took      3.85 [micro-seconds],   Selection_Sort took      4.36 [micro-seconds]
Sorting of an integer array (size :  60) :  Quick_Sort took      4.11 [micro-seconds],   Selection_Sort took      5.13 [micro-seconds]
Sorting of an integer array (size :  65) :  Quick_Sort took      4.36 [micro-seconds],   Selection_Sort took      5.64 [micro-seconds]
Sorting of an integer array (size :  70) :  Quick_Sort took      4.36 [micro-seconds],   Selection_Sort took      6.41 [micro-seconds]
Sorting of an integer array (size :  75) :  Quick_Sort took      4.62 [micro-seconds],   Selection_Sort took      7.44 [micro-seconds]
Sorting of an integer array (size :  80) :  Quick_Sort took      5.13 [micro-seconds],   Selection_Sort took      8.21 [micro-seconds]
Sorting of an integer array (size :  85) :  Quick_Sort took      5.64 [micro-seconds],   Selection_Sort took      9.24 [micro-seconds]
Sorting of an integer array (size :  90) :  Quick_Sort took      5.90 [micro-seconds],   Selection_Sort took     10.52 [micro-seconds]
Sorting of an integer array (size :  95) :  Quick_Sort took      6.41 [micro-seconds],   Selection_Sort took     11.29 [micro-seconds]
Sorting of an integer array (size : 100) :  Quick_Sort took      6.93 [micro-seconds],   Selection_Sort took     12.06 [micro-seconds]
Sorting of an integer array (size : 105) :  Quick_Sort took      6.93 [micro-seconds],   Selection_Sort took     13.34 [micro-seconds]
Sorting of an integer array (size : 110) :  Quick_Sort took      7.70 [micro-seconds],   Selection_Sort took     14.63 [micro-seconds]
Sorting of an integer array (size : 115) :  Quick_Sort took      7.70 [micro-seconds],   Selection_Sort took     15.65 [micro-seconds]
Sorting of an integer array (size : 120) :  Quick_Sort took      7.95 [micro-seconds],   Selection_Sort took     16.93 [micro-seconds]
Sorting of an integer array (size : 125) :  Quick_Sort took      8.21 [micro-seconds],   Selection_Sort took     18.73 [micro-seconds]
Sorting of an integer array (size : 130) :  Quick_Sort took      8.98 [micro-seconds],   Selection_Sort took     20.01 [micro-seconds]
Sorting of an integer array (size : 135) :  Quick_Sort took      8.98 [micro-seconds],   Selection_Sort took     21.30 [micro-seconds]
Sorting of an integer array (size : 140) :  Quick_Sort took      9.75 [micro-seconds],   Selection_Sort took     22.58 [micro-seconds]
Sorting of an integer array (size : 145) :  Quick_Sort took     10.26 [micro-seconds],   Selection_Sort took     24.63 [micro-seconds]
Sorting of an integer array (size : 150) :  Quick_Sort took     10.01 [micro-seconds],   Selection_Sort took     25.66 [micro-seconds]
Sorting of an integer array (size : 155) :  Quick_Sort took     11.03 [micro-seconds],   Selection_Sort took     27.71 [micro-seconds]
Sorting of an integer array (size : 160) :  Quick_Sort took     11.29 [micro-seconds],   Selection_Sort took     29.25 [micro-seconds]
Sorting of an integer array (size : 165) :  Quick_Sort took     11.29 [micro-seconds],   Selection_Sort took     31.05 [micro-seconds]
Sorting of an integer array (size : 170) :  Quick_Sort took     11.55 [micro-seconds],   Selection_Sort took     32.59 [micro-seconds]
Sorting of an integer array (size : 175) :  Quick_Sort took     12.32 [micro-seconds],   Selection_Sort took     34.64 [micro-seconds]
Sorting of an integer array (size : 180) :  Quick_Sort took     12.83 [micro-seconds],   Selection_Sort took     36.18 [micro-seconds]
Sorting of an integer array (size : 185) :  Quick_Sort took     13.34 [micro-seconds],   Selection_Sort took     38.23 [micro-seconds]
Sorting of an integer array (size : 190) :  Quick_Sort took     13.34 [micro-seconds],   Selection_Sort took     40.28 [micro-seconds]
Sorting of an integer array (size : 195) :  Quick_Sort took     13.60 [micro-seconds],   Selection_Sort took     42.34 [micro-seconds]
Sorting of an integer array (size : 200) :  Quick_Sort took     13.86 [micro-seconds],   Selection_Sort took     44.39 [micro-seconds]
```

# 다양한 Scenario별 기능 시험을 위한 main() 함수

```
int main()
{
	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)
	{
		printf("\nTest Array Algorithms  :\n");
		printf("   1: Performance Comparison of Selection Sort and Quick Sort for
		    Small Integer Array\n");
		printf("   2: Test Big Rand Array (Array Size: 1,000,000 ~ 10,000,000)\n");
		printf("   3: Performance Measurements of hybrid_QS_SS for Integer Array\n");
		printf("Input menu (-1 to terminate) : ");
		scanf("%d", &menu);
		//printf("\n");
		if (menu == -1)
			break;
```

# 다양한 Scenario별 기능 시험을 위한 main() 함수

```
            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;
}
```

# Oral Test

# Oral Test

Q6.1 동적 메모리 할당의 필요성에 대하여 설명하고, 동적 메모리 할당을 사용하여 동적 배열을 생성하는 방법에 대하여 설명하라.

Q6.2 RAND_MAX (32,767) 보다 큰 값인 배열 크기 size와 offset가 주어지면 (0 ~ size - 1) + offset 범위의 값을 가지며 중복되지 않는 난수 (random number)들을 생성하여 지정된 동적 배열에 담아주는 void genBigRandArray (int *bigArray, int size, int offset) 함수의 동작 원리를 설명하라.

Q6.3 Windows 운영체제에서 제공하는 Performance Counter를 사용하여 함수의 실행시간을 millisecond와 microsecond 단위로 정밀하게 측정하는 방법에 대하여 예를 들어 설명하라.

Q6.4 hybridQuickSelectionSort(int *bigArray, int size) 함수가 어떻게selection sorting 과 quick sorting의 장점을 활용하여 다양한 배열의 크기에 대하여 빠르게 정렬할 수 있는지에 대하여 상세하게 설명하라.