EECE372 HW#3

20190445 허수범

April 25, 2024

1 Preliminaries & Background

Insertion sort는 간단한 sorting algorithm으로, unsorted array의 각 원소를 iterative하게 array의 sorted portion의 적절한 위치에 삽입하는 sorting algorithm입니다. Small N에서는 비교적 우수한 성능을 보이지만, time complexity가 '$O(n^2)$'으로 높기 때문에, N이 커질수록 좋지 않은 성능을 보입니다. 반면 Merge sort는 divide and conquer approach를 사용하여 input array를 재귀적으로 divide하고, 각 subarray를 sorting하며 merge하는 sorting algorithm입니다. Merge sort의 time complexity는 '$O(n \log n)$'으로, large dataset에서 매우 좋은 성능을 보입니다.

본 assignment는 두 sorting algorithm을 각각 C 언어와 assembly로 구현하여 알고리즘 수행 시간을 비교하는 것입니다. Assembly는 C 언어 내의 inline assembly block에서 사용됩니다. 직접 assembly로 구현을 하는 것과 C 언어를 compiler가 자동으로 번역하는 것은 각각 장단점이 존재합니다. Asembly로 직접 code를 구현하면, 하드웨어를 직접 제어할 수 있어 세밀한 최적화와 효율적인 리소스 사용이 가능해 C에 비해 빠르고 memory efficient한 implementation이 가능합니다. 반면 C 언어로 구현하면, 간편한 코딩이 가능하며, 가독성이 높은 code를 작성할 수 있어 개발하는 데에 assembly에 비해 매우 적은 시간이 소요됩니다.

본 report에서는 위의 내용들을 직접 구현해보고, 실험한 결과를 분석해 이미 알고 있는 지식과 실제 실험 결과 간의 일관성을 확인해 보도록 하겠습니다. 또한, 저는 extra credit을 위해 recursion(function call and return from subroutine)을 모두 하나의 inline assembly block에서 구현하였고, 이에 더해 insertion sort와 merge sort의 assembly 구현에서 C local variable을 선언하지 않았습니다.

2 각 Algorithm에 대한 설명

다음은 각 algorithm에 대한 설명입니다.

2.1 Insertion Sort

2.1.1 Insertion Sort in C

'insertion_C'는 Listing 1과 같이 구현하였습니다. 구현 당시 수업자료의 예시 코드를 참고하였습니다. Array의 두 번째 원소부터 iterate하여, 각 원소(key)를 왼쪽의 sorted array의 적절한 위치에 삽입하도록 하였습니다. Array implementation이기 때문에, while loop를 활용하여 key의 적절한 삽입 위치를 찾는 동안 각 sorted 원소들을 오른쪽으로 한 칸씩 밀어 주었습니다.

```
void insertion_C(int* array, int N) {
    int i, j, key;
    for(int i = 1; i < N; i++){
        key = array[i]; // take an unsorted element
        j = i - 1;

        while(array[j] > key && j >= 0){ // loop over sorted elements
            array[j + 1] = array[j]; // move elements to the right
            j--;
        }
        array[j + 1] = key; // insert element into the correct position
    }

    return;
}
```

Listing 1: 'insertion_C' function.

2.1.2 Insertion Sort in Assembly

'insertion_ASM'은 Listing 2과 같이 구현하였습니다. 위의 C implementation의 각 줄을 assembly로 번역하는 방식으로 구현하였습니다. Local variable을 C로 따로 선언해주지 않고, register r0의 값을 'N', r1을 i, r2을 key, r3를 j로 설정하였고, branch instruction을 활용해 각 loop를 구현해주었습니다. 'array'의 주소는 따로 register에 load 하지 않고, 각 원소의 값에 접근하거나, 원소의 값을 바꿔줄 때마다 square bracket과 offset(+LSL instruction - 이는 'int'의 크기에 맞춰 indexing하기 위해 사용되었습니다)을 활용하였습니다. 각 loop에서 cmp instruction과 add, sub instruction을 활용해 조건을 제대로 확인하고, 조건에 사용되는 variable을 적절히 increment 혹은 decrement하는 것이 중요했습니다.

2.2 Merge Sort

Merge sort는 수업 자료의 내의 코드를 더 효율적인 방식으로 수정하여 구현하였습니다.

2.2.1 Merge Sort in C

'mergesort_C'는 Listing 3와 같이 merge sort의 과정을 재귀함수를 활용하여 구현하였습니다. 입력된 right index의 크기가 left보다 크다면, 즉 subarray의 element 개수가 2 이상이면, middle index를 구하고, 각각 left subarray와 right subarray에 대해 'mergesort_C'를 재귀적으로 호출합니다. 이후, left subarray와 right subarray를 merge하는 'merge_C' 함수를 호출합니다. 이 과정이 끝났거나, 입력된 subarray의 element가 1개 이하라면, return합니다.

```c
void insertion_ASM(int* array, int N) {
    asm(
        "ldr r0, %[N]\n\t" // r0 = N
        "mov r1, #1\n\t" // r1 = i = 1

    "outer_loop:\n\t"
        "cmp r1, r0\n\t" // compare i with N
        "bge end_outer_loop\n\t" // exit loop if i >= N

        "ldr r2, [%[array], r1, LSL #2]\n\t" // r2 = array[i]

        "sub r3, r1, #1\n\t" // r3 = i - 1

    "inner_loop:\n\t"
        "ldr r4, [%[array], r3, LSL #2]\n\t" // r4 = array[j]
        "cmp r4, r2\n\t" // compare array[j] with key
        "ble end_inner_loop\n\t" // exit loop if array[j] <= key

        "cmp r3, #0\n\t" // compare j with 0
        "blt end_inner_loop\n\t" // exit loop if j < 0

        "add r5, r3, #1\n\t" // r5 = j + 1
        "str r4, [%[array], r5, LSL #2]\n\t" // array[j+1] = array[j]

        "sub r3, r3, #1\n\t" // decrement j
        "b inner_loop\n\t" // continue inner loop

    "end_inner_loop:\n\t"
        "add r5, r3, #1\n\t" // r5 = j + 1
        "str r2, [%[array], r5, LSL #2]\n\t" // array[j + 1] = key

        "add r1, r1, #1\n\t" // increment i by 1
        "b outer_loop\n\t" // continue outer loop

    "end_outer_loop:\n\t"

        : // no output operands
        : [array] "r" (array), [N] "m" (N) // input operands
        : "memory", "r0", "r1", "r2", "r3", "r4", "r5" // clobbered registers
    );

    return;
}
```

Listing 2: 'insertion_ASM' function.

```c
void mergesort_C(int *array, int left, int right) {
    if(right > left){ // continue when subarray has more than one element
        int middle = (left + right) / 2; // find the middle point
        mergesort_C(array, left, middle); // call mergesort for left half
        mergesort_C(array, middle + 1, right); // call mergesort for right half

        merge_C(array, left, middle, right); // merge two halves
    }

    return;
}
```

Listing 3: 'mergesort_C' function.

'merge_C'는 Listing 4와 같이 구현하였습니다. 'merge_C'의 경우, 강의 자료의 코드와 다르게, 하나의 'temp_array'만을 사용하여 구현하였습니다. 이는 이후 Assembly로 구현할 때, local variable로 static temp array를 선언하지 않기 위한 조치였습니다. Stack pointer를 subtract해서 temp array로 사용하려 했는데, 이 방법으로는 두 개의 temp array를 사용하기 까다롭다고 판단했기 때문입니다.

Original array의 주소와 left, middle, right index의 값이 입력되면, left index와 right index의 값을 이용해 필요한 'temp_array'의 길이를 구하고, 'temp_array'를 statically allocate합니다. 이후 left subarray의 start index인 'i', right subarray의 start index인 'j', 'temp_array'의 start index인 'k'를 선언해주고, 각 subarray의 원소들을 오름차순으로 'temp_array'에 복사해주는 첫 while loop를 실행합니다. 첫 while loop가 끝난 후에는 각 subarray의 남은 원소들을 'temp_array'에 복사하는 두 while loop를 통과하여 'temp_array'를 완성해 줍니다. 마지막으로 'merge'는 'temp_array'의 모든 원소를 original array의 해당되는 위치에 복사해주고 끝납니다.

```c
void merge_C(int *array, int left, int middle, int right) {
    int i, j, k;

    int temp_array_length = right - left + 1; // calculate the length of temporary
    array
    int temp_array[temp_array_length]; // allocate temporary array

    i = left; // start index for the first subarray
    j = middle + 1; // start index for the second subarray
    k = 0; // start index for the temporary array

    while (i <= middle && j <= right) { // copy datas to temp array
        if (array[i] <= array[j]) {
            temp_array[k] = array[i];
            k++;
            i++;
        } else {
            temp_array[k] = array[j];
            k++;
            j++;
        }
    }

    // copy remaining elements from the left subarray
    while (i <= middle) {
        temp_array[k] = array[i];
        k++;
        i++;
    }

    // copy remaining elements from the right subarray
    while (j <= right) {
        temp_array[k] = array[j];
        k++;
        j++;
    }

    // reconstruct the original array
    for (i = 0; i < temp_array_length; i++) {
        array[left + i] = temp_array[i];
    }

    return;
}
```

Listing 4: 'merge_C' function.

2.2.2 Merge Sort in Assembly(Extra Credit)

'mergesort_ASM'은 'insertion_ASM'과 마찬가지로 C counterpart의 각 줄을 assembly로 번역하여 구현하였습니다. 또한, 위에서도 언급하였다시피, extra credit을 위해 recursion(function call and return from subroutine)을 모두 하나의 inline assembly block에서 구현하였습니다. 여기서도 local variable은 따로 선언해주지 않고, register들과 input argument들만을 활용하였습니다.

Inline assembly block의 시작은 'merge_sort' call을 위한 register setting입니다. Listing 5이 register setting과 branch instruction을 보여주는 code snippet입니다. Register r0를 left, r1을 right으로 설정해 준 후, bl instruction을 활용해 'merge_sort'를 call합니다. 'merge_sort'가 끝난 후에는 다시 이 위치로 돌아와 'end' label로 branch하도록 하였습니다.

```
asm (
    "mov r0, #0\n\t" // initilaize left = 0
    "ldr r1, %[N]\n\t" // r1 = N
    "sub r1, r1, #1\n\t" // r1 = right = N - 1
    "bl merge_sort\n\t" // call merge_sort
    "b end\n\t" // end mergesort
```

Listing 5: 'mergesort_ASM' setting.

'merge_sort'는 Listing 6에서 확인 할 수 있습니다. 이 부분은 recursive subroutine call과 epilog를 구현해 놓은 snippet입니다. 위에서 설정해주었다시피 r0와 r1은 각각 left와 right입니다. 먼저 prolog에서 r0, r1, lr를 push해주어 preserved register를 save하고, activation record를 setup합니다. 그 다음에는 r0와 r1의 값, 즉 left와 right의 크기를 비교하여, subarray의 크기가 적절한지 판단합니다. subarray의 크기가 2 미만이라면, 'merge_sort_end'로 branch해줍니다. 이후에는 push와 pop instruction을 활용해 register 내의 값을 보존해주며 각 subarray를 위한 'merge_sort'를 call 해줍니다. 이후 'merge'를 call하기 이전에 r0, r1, r2의 값을 각각 left, middle, right로 설정해주고, 'merge' label로 branch합니다. 'merge_sort_end'는 epilog로, pop instruction을 통해 preserved register들을 복원하고, calling function으로 return합니다. 그 밑의 부분은 inline assembly block의 마지막에 input, out operand들과 clobbered register들을 표시해주는 부분으로, output operand는 없고, input operand는 각각 array의 주소와 N의 값 뿐인 것을 확인할 수 있습니다.

'merge'의 prolog와 register setting은 Listing 7과 같이 구현하였습니다. push instruction을 활용해 link register의 값을 저장해 주었고, i, j, k의 값을 각각 r3, r4, r5에 저장해 주었습니다. 그 밑이 중요한 부분인데, r9 register를 활용해 temporary array의 length를 계산해 해당 값을 sp의 값에서 subtract해주어 'temp_array'에 필요한 memory를 stack에 할당해 주었습니다.

'merge'의 merge operation은 Listing 8와 같이 구현하였습니다. 각 loop를 cmp와 branch instruction을 활용하여 C counterpart와 동일하게 구현해 주었습니다. Temporary array가 하나 뿐이었기에 복잡한 번역 과정 없이 assembly로 변환할 수 있었습니다. 'merge_end'에서는 sp의 값을 원래대로 되돌려주어 temporary array에 할당된 memory를 deallocate해 주고, 'pop pc' instruction을 통해 calling function으로 return하도록 구현해 주었습니다.

```
"merge_sort:\n\t"
// r0 = left
// r1 = right
    "push {r0, r1, lr}\n\t" // save previous variables

    "cmp r0, r1\n\t" // compare left and right
    "bge merge_sort_end\n\t" // end merge_sort if array has less than 2 elements

    "sub r2, r1, r0\n\t" // r2 = right - left
    "lsr r2, r2, #1\n\t" // r2 = (right - left) / 2
    "add r2, r0, r2\n\t" // r2 = middle = left + (right - left) / 2

    "push {r1}\n\t" // right saved, r0 = left r2 = middle
    "mov r1, r2\n\t" // r0 = left, r1 = middle
    "bl merge_sort\n\t" // recursive call on left subarray
    "pop {r2}\n\t" // r2 = right

    "push {r0}\n\t" // left saved
    "mov r0, r1\n\t" // r0 = middle
    "add r0, r0, #1\n\t" // r0 = middle + 1
    "mov r1, r2\n\t" // r1 = right
    "bl merge_sort\n\t" // recursive call on right subarray

    "mov r2, r1\n\t" // r2 = right
    "mov r1, r0\n\t" // r1 = middle + 1
    "sub r1, r1, #1\n\t" // r1 = middle
    "pop {r0}\n\t" // r0 = left

    // r0 = left, r1 = middle, r2 = right
    "bl merge\n\t" // call merge

"merge_sort_end:\n\t"
    "pop {r0, r1, pc}\n\t" // restore original variables and return
// merge_sort ends here
//----------------------------------------
"end:\n\t"

    : // no output operands
    : [array] "r" (array), [N] "m" (N) // input operands
    : "memory", "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r8", "r9", "pc", "lr" //
    clobbered registers
);
```

Listing 6: 'mergesort_ASM' recursive subroutine call and epilog.

```
"merge:\n\t"
    "push {lr}\n\t" // save return address
    "mov r3, r0\n\t" // i = left
    "add r4, r1, #1\n\t" // j = middle + 1
    "mov r5, #0\n\t" // k = 0

    "sub r9, r2, r0\n\t" // r9 = right - left
    "add r9, r9, #1\n\t" // r9 = temp_array_length
    "lsl r9, r9, #2\n\t" // r9 = temp_array_length * sizeof(int)
    "sub sp, sp, r9\n\t" // allocate array

    "lsr r9, r9, #2\n\t" // r9 = temp_array_length
```

Listing 7: 'mergesort_ASM' merge prolog & setting.

```
    // copy datas to temp array
    // while (i <= middle && j <= right)
"loop1:\n\t"
    "cmp r3, r1\n\t" // compare i and middle
    "bgt copy_remnant_left\n\t" // end loop1
    "cmp r4, r2\n\t" // compare j and right
    "bgt copy_remnant_left\n\t" // end loop1

    // if(array[i] <= array[j])
    "ldr r6, [%[array], r3, LSL #2]\n\t" // r6 = array[i]
    "ldr r8, [%[array], r4, LSL #2]\n\t" // r8 = array[j]

    "cmp r6, r8\n\t" // compare array[i] and array[j]
    "bgt if2\n\t" // else
    // if (array[i] <= array[j])
"if1:\n\t"
    "str r6, [sp, r5, LSL #2]\n\t" // temp_array[k] = array[i]
    "add r5, #1\n\t" // k++
    "add r3, #1\n\t" // i++
    "b loop1\n\t"
    // else
"if2:\n\t"
    "str r8, [sp, r5, LSL #2]\n\t" // temp_array[k] = array[j]
    "add r5, #1\n\t" // k++
    "add r4, #1\n\t" // j++
    "b loop1\n\t"

    // copy remaining elements from left
    // while(i <= middle)
"copy_remnant_left:\n\t"
    "cmp r3, r1\n\t"
    "bgt copy_remnant_right\n\t"
    "ldr r6, [%[array], r3, LSL #2]\n\t" // r6 = array[i]
    "str r6, [sp, r5, LSL #2]\n\t" // temp_array[k] = array[i]
    "add r5, #1\n\t" // k++
    "add r3, #1\n\t" // i++
    "b copy_remnant_left\n\t"

    // copy remaining elements from right
    // while (j <= right)
"copy_remnant_right:\n\t"
    "cmp r4, r2\n\t"
    "bgt copy_end\n\t"
    "ldr r6, [%[array], r4, LSL #2]\n\t" // r6 = array[j]
    "str r6, [sp, r5, LSL #2]\n\t" // temp_array[k] = array[j]
    "add r5, #1\n\t" // k++
    "add r4, #1\n\t" // j++
    "b copy_remnant_right\n\t"

"copy_end:\n\t"
    "mov r5, #0\n\t" // i = 0
    // reconstruct original array
"reconstruct:\n\t"
    "cmp r5, r9\n\t" // compare i and temp_array_length
    "bge merge_end\n\t" // return merge if i >= temp_array_length
    "ldr r6, [sp, r5, LSL #2]\n\t" // r6 = temp_array[i]
    "add r4, r0, r5\n\t" // left + i
    "str r6, [%[array], r4, LSL #2]\n\t" // array[left + i] = temp_array[i]
    "add r5, r5, #1\n\t" // i++
    "b reconstruct\n\t"

"merge_end:\n\t"
    "lsl r9, r9, #2\n\t" // r9 = temp_array_length * sizeof(int)
    "add sp, sp, r9\n\t" // deallocate array
    "pop {pc}\n\t" // return
```

Listing 8: 'mergesort_ASM' merge operation.

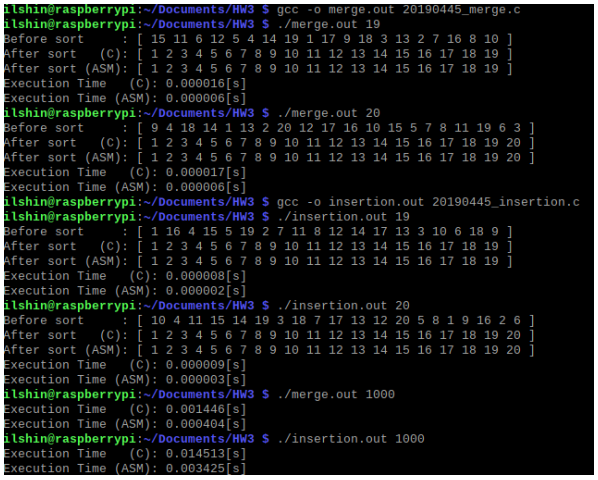


Figure 1: Result examples.

3 Result

C 및 assembly 코드가 정상적으로 동작하고 있음을 보이는 result example은 Figure 1에서 확인할 수 있습니다. 매 실행마다 random한 number가 생성되며, ASM과 C 모두에서 sorting이 정상적으로 작동하며, execution time 역시 정상적으로 확인되는 것을 볼 수 있습니다.

4 Comparative Analysis

Sorting에 소요된 시간은 각 language마다, 각 algorithm마다 큰 차이를 보여주었습니다. 이를 확인하기 위해 program을 여러 차례 실행한 결과는 Figure 2와 같습니다.

4.1 Between Different 'N'

'N' 값에 따른 각 sorting algorithm의 execution time을 비교하기 위해 plot한 graph들은 Figures 3입니다. 추세를 정확히 확인하기 위해 위의 두 그래프는 각 축을 logarithmic scale로 변환하였습니다. 먼저 insertion sort의 plot 결과를 보면, assembly와 C의 경우 모두 'x^2'에 가까운 추세선을 가지는 것을 확인할 수 있습니다. 이는 insertion sort의 theoretical time complexity인 '$O(n^2)$'에 부합하는 결과라 볼 수 있습니다. Merge sort의 경우, assembly와 C 모두 'x^1'과 'x^2' 사이의 추세선을 가지는 것으로 보아, merge sort의 theoretical time complexity인 '$O(n \log n)$'와 비슷한 결과가 나왔다고 볼 수 있습니다.

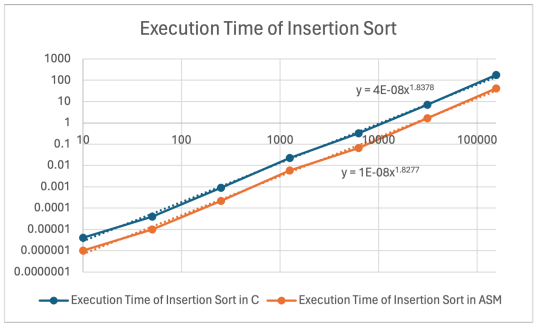
4.2 Between Languages

4.2.1 Insertion Sort

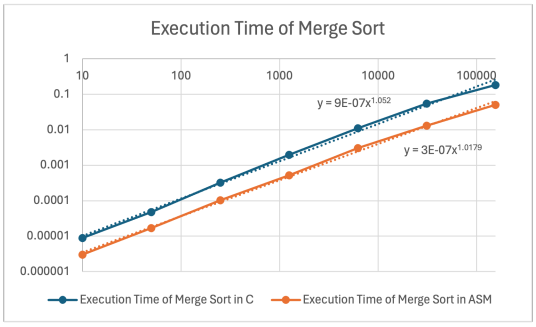
Insertion sort의 두 language 사이의 execution time을 비교한 그래프는 Figure 3c입니다. 두 language 사이의 실행시간에 큰 차이가 있는 것을 확인할 수 있습니다. 정확한 analysis를 위해 Figure 3a를 확인해 보면, assembly implementation이 C implementation에 비해 약 4배 빠른 것을 확인할 수 있습니다.

```
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 10
Before sort     : [ 9 3 10 5 6 7 4 1 2 8 ]
After sort   (C): [ 1 2 3 4 5 6 7 8 9 10 ]
After sort (ASM): [ 1 2 3 4 5 6 7 8 9 10 ]
Execution Time   (C): 0.000004[s]
Execution Time (ASM): 0.000001[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 50
Execution Time   (C): 0.000040[s]
Execution Time (ASM): 0.000010[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 250
Execution Time   (C): 0.000901[s]
Execution Time (ASM): 0.000215[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 1250
Execution Time   (C): 0.022772[s]
Execution Time (ASM): 0.005794[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 6250
Execution Time   (C): 0.328257[s]
Execution Time (ASM): 0.065414[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 31250
Execution Time   (C): 6.965692[s]
Execution Time (ASM): 1.634329[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./insertion.out 156250
Execution Time   (C): 175.295198[s]
Execution Time (ASM): 41.707893[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 10
Before sort     : [ 4 6 10 2 9 3 5 1 7 8 ]
After sort   (C): [ 1 2 3 4 5 6 7 8 9 10 ]
After sort (ASM): [ 1 2 3 4 5 6 7 8 9 10 ]
Execution Time   (C): 0.000009[s]
Execution Time (ASM): 0.000003[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 50
Execution Time   (C): 0.000048[s]
Execution Time (ASM): 0.000017[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 250
Execution Time   (C): 0.000324[s]
Execution Time (ASM): 0.000104[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 1250
Execution Time   (C): 0.001994[s]
Execution Time (ASM): 0.000521[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 6250
Execution Time   (C): 0.011069[s]
Execution Time (ASM): 0.003051[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 31250
Execution Time   (C): 0.055409[s]
Execution Time (ASM): 0.013165[s]
ilshin@raspberrypi:~/Documents/HW3 $ ./merge.out 156250
Execution Time   (C): 0.183960[s]
Execution Time (ASM): 0.050374[s]
```

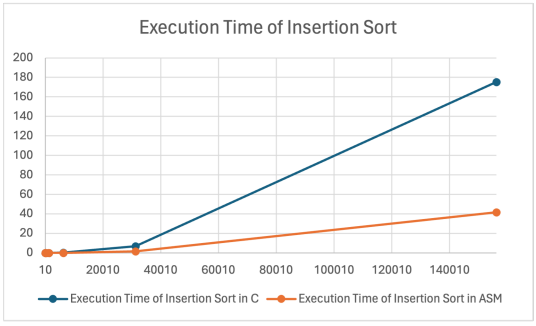
Figure 2: Result for comparison.



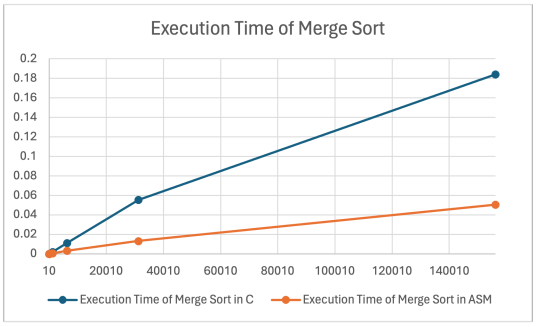
(a) Execution Time of Insertion Sort with Trendline(Logarithmic Scale).



(b) Execution Time of Merge Sort with Trendline(Logarithmic Scale).

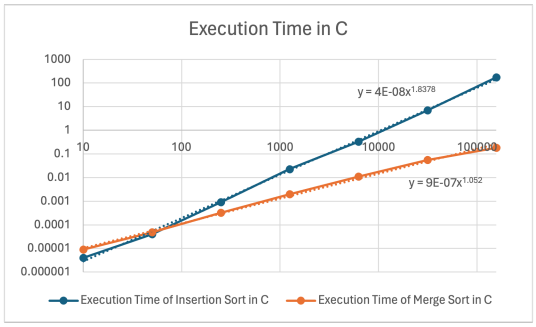


(c) Execution Time of Insertion Sort.

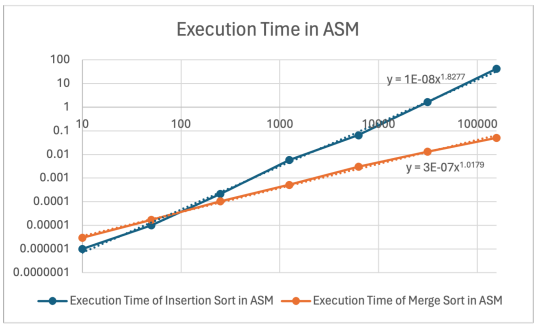


(d) Execution Time of Merge Sort.

Figure 3: Execution Time of Each Sorting Algorithm.



(a) Execution Time of C Implementation for Each Algorithm.



(b) Execution Time of ASM Implementation for Each Algorithm.

Figure 4: Execution Time for Each Language.

4.2.2 Merge Sort

Merge sort의 두 language 사이의 execution time을 비교한 그래프는 Figure 3d입니다. Merge sort에서도 insertion sort와 마찬가지로 두 language 사이의 실행시간에 큰 차이가 있는 것을 확인할 수 있습니다. 정확한 analysis를 위해 Figure 3b를 확인해 보면, assembly implementation이 C implementation에 비해 약 3배 빠른 것을 확인할 수 있습니다.

4.3 Between Sorting Algorithms

4.3.1 C Language

각 sorting algorithm의 C implementation의 execution time plot 결과는 Figure 4a와 같습니다. Insertion sort의 execution time이 압도적으로 커 각 축을 log scale로 하지 않고서는 merge sort의 추세를 확인하기 힘들어 두 축을 모두 logarithmic scale로 하여 plot하였습니다. 'N'이 100 이상일 때는 큰 차이로 merge sort가 빠르지만, small N에서는 insertion sort가 근소하게 빠르거나, 비슷한 것을 확인할 수 있습니다. 수업에서 언급된 것과 마찬가지로, insertion sort는 small N에서 좋은 성능을 보이지만, time complexity($O(n^2)$)가 merge sort의 그것($O(n \log n)$)에 비해 크기 때문에, N이 커질수록 merge sort가 크게 우수한 성능을 보이는 것을 확인할 수 있습니다.

4.3.2 Assembly

각 sorting algorithm의 assembly implementation의 execution time plot 결과는 Figure 4b과 같습니다. 이 역시 C language graph와 같은 이유로 각 축을 logarithmic scale로 plot하였습니다. Assembly implementation에서 역시 'N'이 100 이상일 때는 큰 차이로 merge sort가 빠르지만, small N에서는 insertion sort가 근소하게 빠르거나, 비슷한 것을 확인할 수 있습니다. 이는 위의 C implementation의 분석 결과와 동일합니다.

5 Discussion

실제로 각 sorting algorithm을 C와 assmebly로 구현하고, 실행 시간을 측정하여 비교해 본 결과, Section 1의 내용이 실제 실험 결과와도 일관되는 것을 확인할 수 있었습니다. 먼저 Subsection 4.1에서는 각 sorting algorithm의 N에 따른 실행 시간의 변화가 이미 알려진 각 sorting algorithm의 theoretical time complexity와 유사한 추세를 보이는 것을 확인하였습니다. 둘째로, Subsection 4.2에서는 assembly implementation이 C implementation에 비해 수 배 빠른 실행 시간을 보인다는 것을 확인하였습니다. 세 번째로는 Subsection 4.3에서 small N에서는 insertion sort가 빠른 실행시간을 보이지만, N이 커질수록 merge sort가 압도적으로 빠른 실행 시간을 보이는 것을 확인하였습니다. 또한, 본 assignment를 수행하며, C 언어를 활용한 구현은 단시간 내에 해냈지만, assembly의 경우, 익숙하지 않았던 탓도 있었겠지만, 매우 오랜 시간 동안 코딩하여 algorithm을 구현해야 했습니다.

본 assignment를 통해, 수업 시간에 다룬 내용들이 실제 구현과 실험 시에도 부합하는 것을 확인할 수 있었습니다.