**알고리즘 설계와 분석 실험 보고서**

**20181210 유창호**

**1. My Sorting Implementation (My Quick Sort)**

I optimized the Quick Sort algorithm using median of three method, tail recursion and insertion sort. The pivot in the original Quick Sort was always selected from the back of the list. This restriction in the selection of pivot may not give optimal results as we can observe the time complexity is O(n^2) for the worst-case input.

Selecting the pivot as the median value of three different index can solve the problem. I selected the pivot as the median value by comparing the first, middle, end index of the list. In doing so we increase the change that the pivot will partition the list more equally.

int m = (p + r ) / 2;//middle value

int min, median;

if(arr[p] <= arr[m] && arr[m] <= arr[r]){//get median

median = m;

}

else if(arr[p] <= arr[r] && arr[r] <= arr[m]){

median = r;

}

else if(arr[r] <= arr[p] && arr[p] <= arr[m]){

median = p;

}

else if(arr[r] <= arr[m] && arr[m] <= arr[p]){

median = m;

}

else if(arr[m] <= arr[r] && arr[r] <= arr[p]){

median = r;

}

else{

median = p;

}

int pivot = median;

**get pivot as median of three**

Next, I implemented tail recursion to reduce stack usage for the recursive function calls. The original quicksort() function called two different quicksort() functions, one for the lower part and one for the higher part. Since one function calls two new recursive functions, it takes up a lot of stack space and the constant operation time for push and pop of the stack can hurt optimization. To solve this I made the quicksort() function to only call a single new function thus reduced stack usage.

while(p < r){//tail recursion

int q = partition2(arr, p, r);

if(q - p < r - q){

quickSort2(arr, p, q-1);

p = q + 1;

}

else{

quickSort2(arr, q+1, r);

r = q - 1;

}

}

**Tail Recursion**

Lastly, as the list gets smaller by partition, when it gets smaller than a certain threshold, I applied Insertion Sort. This is because insertion sort compares and swaps elements less than quicksort in case of small arrays. After comparing the performance of Insertion Sort and Quick Sort for small arrays, I chose 20 as the threshold.

if(r-p <= 20){//if smaller than 20, do insertionsort

insertionSort(arr,p,r);

return;

}

**In quicksort()**

**2. Experiment Environment**

These are the specifications of environmental settings and setups when conducting the experiment.

|  |  |
| --- | --- |
| Hardware | MacBook Air |
| OS | macOS Big Sur ver. 11.6.7 |
| CPU | Apple M1 |
| Memory | LPDDR4 16GB |

The test inputs were generated using srand() and rand() function. It generates input examples ranging from 0 to RAND\_MAX (2147483647). I generated two sets of test inputs, one with random array pattern and one sorted in a non-increasing order. The input size for random array pattern ranges from 10 at the smallest and 1,000,000 at the largest. The input size for non-increasing array ranges from 10 at the smallest and 100,000 at the largest. Time calculation was done using the clock() function and its measurement is calculated in seconds.

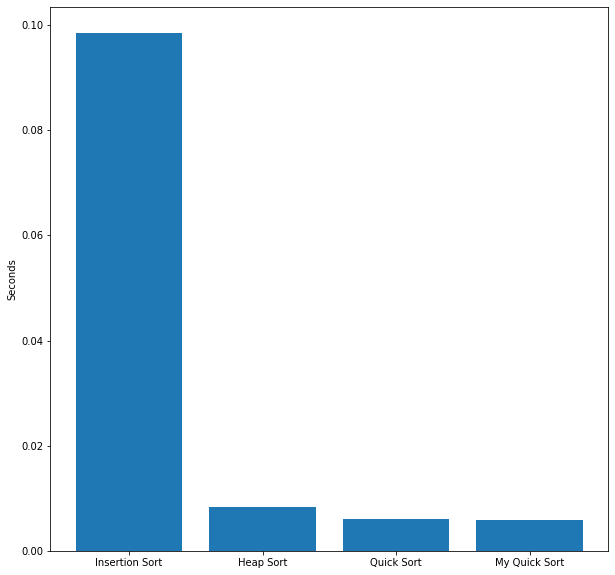
**3. Performance Comparison**

These are the performance results of each algorithm. Each algorithm was tested in two cases, random generated inputs and worst-case inputs.

**Performance of Randomly Generated Inputs**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm  Input size | 1. Insertion Sort | 2. Quick Sort | 3. Heap Sort | 4. My Quick Sort |
| 10 | 0.000131 sec | 0.000132 sec | 0.000205 sec | 0.000184 sec |
| 100 | 0.000293 sec | 0.000241 sec | 0.000233 sec | 0.000235 sec |
| 1000 | 0.002124 sec | 0.000661 sec | 0.000848 sec | 0.000663 sec |
| 10000 | 0.098505 sec | 0.006017 sec | 0.008281 sec | 0.005880 sec |
| 50000 | 1.478402 sec | 0.022832 sec | 0.029878 sec | 0.021585 sec |
| 100000 | 5.802386 sec | 0.040990 sec | 0.059168 sec | 0.038890 sec |
| 1000000 | 579.631024 sec | 0.260884 sec | 0.364382 sec | 0.238652 sec |

**Algorithms in order of Performance**



**When Input Size = 10000**

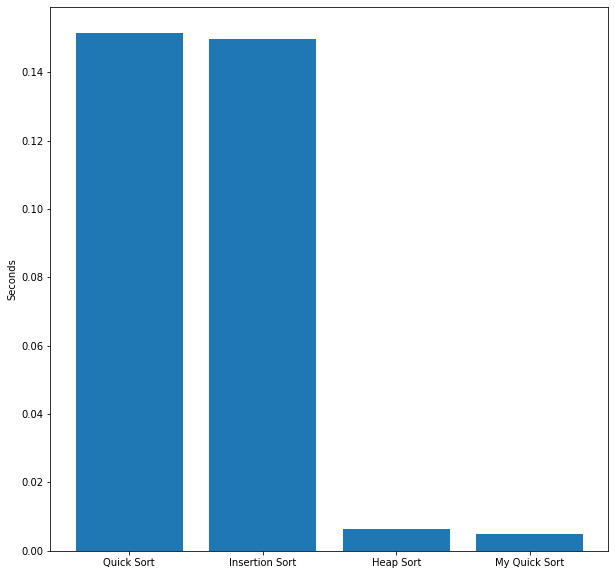
The performance of random generated inputs was calculated as an average time of five different randomly generated cases. In the case of small input size, we don’t see a significant difference in performance. However, as the input size gets larger, we start to notice a difference. Insertion Sort was the least optimal sort as its time complexity is O(n^2), exponentially increasing regard to input size. The next efficient algorithm was Heap Sort with time complexity of O(nlogn) on average. Quick Sort and My Quick Sort was the most efficient sorting algorithm of the four. The average time complexity of Quick Sort is O(nlogn) which is the same as Heap Sort but constant operations such as swapping happens less so it its more efficient. My Quick Sort performed the best as it was optimized with median of three, tail recursion and use of insertion sort.

**Performance of Worst Case Inputs**

These are the performance results of worst case inputs (array of non-increasing order).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm  Input size | 1. Insertion Sort | 2. Quick Sort | 3. Heap Sort | 4. My Quick Sort |
| 10 | 0.000270 sec | 0.000087 sec | 0.000138 sec | 0.000091 sec |
| 100 | 0.000496 sec | 0.000188 sec | 0.000311 sec | 0.000180 sec |
| 1000 | 0.004214 sec | 0.004585 sec | 0.000853 sec | 0.000593 sec |
| 10000 | 0.149719 sec | 0.151477 sec | 0.006488 sec | 0.005000 sec |
| 50000 | 2.923810 sec | 3.101220 sec | 0.022590 sec | 0.022996 sec |
| 100000 | 11.600099 sec | 12.278338 sec | 0.046852 sec | 0.035765 sec |

**Algorithms in order of Performance**



**When Input Size = 10000**

The order of performance is the same (Insertion Sort < Heap Sort < Quick Sort / My Quick Sort) for the small sized inputs of 10 and 100. However, as the input size gets larger the performance of Quick Sort drastically decreases, even worse than Insertion Sort. This is because in a worst case input, Quick Sort always chooses the index of the smallest value as partition. Thus, a lot of swapping is involved in each recursive call giving overhead in computation. My Quick Sort on the other hand solved this issue by implementing median of three in choosing partition, avoiding the worst case partition in each recursive call.

**Conclusion**

The performance result of the two experiments are as follows.

**Average Case : Insertion Sort < Heap Sort < Quick Sort / My Quick Sort**

**Worst Case : Quick Sort < Insertion Sort < Heap Sort < My Quick Sort**

On average case, Quick Sort performed the best with time complexity of O(nlogn). However, in the worst case it performed the worst with time complexity of O(n^2). To guarantee the performance of an average case we can optimize Quick Sort by implementing median of three. Other methods such as tail recursion and use of insertion sort can further strengthen performance as we see in My Quick Sort.