|  |  |  |
| --- | --- | --- |
| **Algorithms analysis** | Section | 02 |
| Student number | 21600635 |
| **Homework 3 – Sorting algorithms** | Name | Jung, Bo Moon |

**□ Explanation of Insertion sort**

Insertion sort is the comparison of all the elements of the array with the already sorted subarray and inserting them at appropriate positions. The feature of Insertion sort is that it is easy to write code. And it is fast for small input and nearly - sorted input. The reason for this is quite simple, just because the number of comparisons is reduced.

If so, let's analyze it by checking the correctness and efficiency of the insertion sort algorithm. To check the correctness of the insertion sort, we need to check the loop invariant.

Correctness:

* Initialization: when j = 2, A[1 ~ J – 1] consists of the single element A[1].
* Maintenance: subarray A[1 ~ j] consists of the elements of originally in A[1 ~ j], but in sorted order.
* Termination: ends of the loop, A[1 ~ n] is sorted order.

So insertion sort has correctness.

In time complexity:

* worst case(reverse):
* best case(sorted):

Because of this characteristic, the more sorted inputs, the better the time complexity. Looking at Figure 1, you can see how much more efficient with n. Additionally, this sorting algorithm do not need more memory space beyond array size, so it can be called 'sort in place'.

|  |  |
| --- | --- |
| 테이블이(가) 표시된 사진  자동 생성된 설명  ***Figure 1. sudocode of insertion sort*** | ***Figure 2. graph of time complexity*** |

**□ Explanation of Heap sort**

Heap sort is an algorithm that sorts through heapify using an array as a complete binary tree. Heapify has Max-Heapify and Min-Heapify. We can decide which heapify to use according to the sorting method. For convenience, taking ascending sort as an example, all items in the array should be Max-Heapify. Here, the characteristic of Max-Heapify is that the parent node has a larger value than the child node. Max-Heapify's sudo code is as shown in figure 3 and the time complexity is .

If so, let's analyze it by checking the correctness and efficiency of the heap sort algorithm. To check the correctness of the heap sort, we need to check the loop invariant.

Correctness:

* Initialization: when i = length(A), n = length(A). Then, A[i + 1 ~ n] has no element.
* Maintenance: In A[1 ~ i], A[0] is the largest number and is smaller than the element in A[i + 1 ~ n]. Also, A[i ~ n] is in an ordered state.
* Termination: After the function ends, A[1 to n] is sorted, and A[0] is the smallest element.

So heap sort has correctness.

In time complexity:

Now, looking at how heap sort is done in detail, the entire array should be Max-Heapify. Therefore, we use the Build-Max-Heap algorithm using Max-Heapify described above. The sudo code of Build-Max-Heap is as shown in figure 4, and the time complexity is because it depends on the height of the node. Finally, Heapsort is made into max heap through Build-Max-Heap and then sorted. Therefore, the time complexity is on the first line and on the 3rd to 5th lines. So time complexity of Heapsort is

Heapsort's sudo code is shown in figure 5. Additionally, this sorting algorithm do not need more memory space beyond array size, so it can be called 'sort in place'.

|  |  |  |
| --- | --- | --- |
|  | **텍스트이(가) 표시된 사진  자동 생성된 설명** | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| ***Figure 3. sudocode of Max-Heapify*** | ***Figure 4. sudocode of Build-Max-Heapify*** | ***Figure 5. sudocode of Heapsort*** |

**□ Explanation of Quick sort**

The quick sort algorithm creates two subarrays based on the pivot through divide. When dividing based on the ascending sort, values ​​smaller than the pivot go to the left subarray, and the larger values ​​go to the right subarray. If the divide is over, sort the two subarrays through a recursive conquer. The characteristic of this algorithm is that it has a very fast execution speed on average. However, the reason that the modifier 'average' is included is because the subarray is unevenly divided according to the size of the pivot when dividing.

The divide is implemented with an algorithm called partition, and figure6 is the sudo code. Since there is a for loop, the time complexity is . Also, conquer uses recursion and sudo code is as shown in figure 7. Quick sort uses these two algorithms together. Conquer is a quick sort itself, so we'll look at the time complexity later.

If so, let's analyze it by checking the correctness and efficiency of the quick sort algorithm. To check the correctness of the quick sort, we need to check the loop invariant. (assume first index of array is 0)

Correctness:

* Initialization: Since i = -1 and j = 0, the first A[p ~ i] has no value, and A[I + 1 ~ j-1] also has no value.
* Maintenance: Since i = p-1 and j = p, maintain A[p ~ i] <= pivot, A[i + 1 ~ j-1]> pivot.
* Termination: When j = r, the left subarray <= pivot, right subarray> pivot is maintained..

So quick sort has correctness.

In time complexity:

* Worst case (pivot is largest value):
* Best case (pivot is middle value):
* Average case (The actual execution of quick sort is a mixture of bad and good splits):

|  |  |
| --- | --- |
| 텍스트이(가) 표시된 사진  자동 생성된 설명 | 텍스트이(가) 표시된 사진  자동 생성된 설명 |
| ***Figure 6. sudocode of Partition*** | ***Figure 7. sudocode of Quicksort*** |

**□ 10 arrays**

|  |  |
| --- | --- |
| Size | Elements |
| 10 | 10 11 15 12 1 1 10 17 18 1 |
| 20 | 7 9 33 18 10 32 24 38 3 29 0 5 12 2 27 23 7 9 0 12 |
| 30 | 14 38 39 16 13 4 48 9 39 31 0 3 17 4 54 39 7 38 20 44 6 31 11 27 13 59 18 29 52 43 |
| 40 | 21 67 45 47 23 56 72 60 75 73 73 48 22 6 41 22 7 67 73 76 62 53 40 17 26 32 17 67 41 71 77 24 56 41 56 19 10 73 43 16 |
| 50 | 28 96 51 85 26 88 76 18 98 95 13 66 27 68 48 18 14 16 13 48 65 82 42 87 99 38 96 18 17 46 47 63 1 68 46 85 22 55 75 37 68 47 78 3 46 80 18 23 29 81 |
| 60 | 35 5 64 116 36 33 0 49 34 37 113 11 32 10 15 101 14 5 73 20 61 104 91 117 52 91 15 9 26 14 97 109 33 95 109 38 74 44 107 98 28 12 104 99 4 85 79 67 83 51 64 71 13 73 13 10 46 72 62 35 |
| 70 | 42 14 50 34 19 105 124 100 30 99 6 9 24 92 42 97 134 94 106 32 64 26 73 107 85 57 134 107 82 22 47 28 125 22 72 71 53 86 79 92 35 37 10 108 9 30 7 131 77 108 78 84 41 117 54 60 10 121 45 98 137 95 13 17 71 4 55 28 133 18 |
| 80 | 49 23 116 105 49 57 8 118 153 1 106 54 89 14 69 20 101 23 106 44 107 155 95 137 45 150 13 158 58 117 117 67 57 69 115 137 105 128 91 133 122 22 156 157 127 135 88 115 111 118 52 10 29 41 142 143 7 50 28 81 12 41 102 154 5 70 152 16 34 51 105 133 158 129 114 157 39 152 140 8 |
| 90 | 176 145 142 103 172 9 12 149 149 163 159 132 114 136 36 156 141 92 179 116 63 57 104 27 98 116 12 49 7 105 67 46 2 116 5 70 57 110 83 94 169 27 129 66 65 160 29 79 118 155 166 103 37 78 150 93 111 39 71 104 67 160 104 98 39 23 142 144 95 164 136 107 126 175 152 11 15 108 178 10 78 145 72 153 154 125 25 77 119 95 |
| 100 | 63 194 175 134 182 1 96 20 25 5 19 97 99 178 83 139 61 161 19 108 186 39 146 37 111 9 91 67 3 133 97 152 114 3 8 3 176 59 35 48 9 72 55 142 130 105 17 90 52 72 127 189 78 2 71 163 128 188 194 127 122 6 153 122 33 89 99 72 76 197 94 88 21 141 197 138 191 24 143 139 106 199 61 27 124 120 63 154 83 153 198 13 4 150 133 165 103 104 48 155 |

□ Screenshots of your program running

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

□ Discussion about the results

First, I mentioned earlier that each insertion sort algorithm is fast when n <50. So, looking at the results, the measurement times of the three sort algorithms with the size of the array up to 50 are similar. However, when the size of n is greater than 50, the measurement time of insertion sort and quick sort gradually starts to differ. Therefore, from the results, it can be seen that the measurement time of insertion sort increases as the size of n increases due to time complexity.

However, looking at the results, there is one question. When comparing insertion sort and heap sort time complexity, the time complexity of heap sort is better. But why is the measurement time difference so small, and rather, the performance of the insertion sort is better? I think there are two causes. First, in fact, insertion sort has extremely fast time complexity of O(n) in the case of the best case. So it means it's much faster than the other two algorithms. Looking at the array elements, nothing is completely reversed. Therefore, depending on the data situation, insertion sort may be faster or slower. Second, heap sort has the same time complexity as quick sort, so you can think that it should be similar to quick sort. However, it is known to be slower than quick sort by actual measurement. The reason for this is that heap sort is more influenced by the state of data than any other algorithm. Therefore, there is a tendency that safety is not guaranteed.

On the other hand, the reason why the quick sort algorithm is the fastest is that as the size of the array increases, the probability of becoming the worst case is very small. The reason is that the probability that the selected pivot is the largest value is 1/n.

In conclusion, as the size of the array increases, the speed of quick sort will be faster than that of the other two algorithms, and the speed of insertion sort and heap sort will vary depending on the state of data.

□ Codes

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void insertion\_sort(int \*a, int size);

void heap\_sort(int \*a, int size);

void build\_max\_heap(int \*a, int size);

void max\_heapify(int \*a, int i, int size);

void quick\_sort(int \*a, int start, int end);

int partition(int \*a, int start, int end);

void init\_array(int \*a, int i);

void print\_array(int \*\*a);

void print\_line(int \*a, int size);

int main(){

int \*\*array = malloc(sizeof(int \*) \* 10);

clock\_t start, finish;

double time = 0;

for(int i = 0; i < 10; i++){

array[i] = malloc(sizeof(int) \* (i + 1) \* 10);

}

int i = 0;

while(i < 10){

srand(i);

for(int j = 0; j < (i + 1) \* 10; j++){

array[i][j] = rand() % ((i + 1) \* 20);

}

i++;

}

print\_array(array);

for(i = 0; i < 10; i++){

int size = (i + 1) \* 10;

printf("-----------------------\n");

printf("size : %d\n", size);

//insertion sort

start = clock();

insertion\_sort(array[i], size);

finish = clock();

time = (double)(finish - start) / 1000;

printf("(insertion sort : %f sec)\n", time);

print\_line(array[i], size);

init\_array(array[i], i);

//heap sort

start = clock();

heap\_sort(array[i], size);

finish = clock();

time = (double)(finish - start) / 1000;

printf("(Heap sort : %f sec)\n", time);

print\_line(array[i], size);

init\_array(array[i], i);

//quick sort

start = clock();

quick\_sort(array[i], 0, size - 1);

finish = clock();

time = (double)(finish - start) / 1000;

printf("(Quick sort : %f sec)\n", time);

print\_line(array[i], size);

printf("-----------------------\n");

}

for(i = 0; i < 10; i++){

free(array[i]);

}

free(array);

return 0;

}

void insertion\_sort(int \*a, int size){

for(int i = 1; i < size; i++){

int key = a[i];

int j = i - 1;

while(j >= 0 && a[j] > key){

a[j + 1] = a[j];

j--;

}

a[j + 1] = key;

}

return;

}

void heap\_sort(int \*a, int size){

build\_max\_heap(a, size);

for(int i = size - 1; i >= 0; i--){

int temp = a[0];

a[0] = a[i];

a[i] = temp;

size--;

max\_heapify(a, 0, size);

}

return;

}

void build\_max\_heap(int \*a, int size){

for(int i = size / 2 - 1; i >= 0; i--){

max\_heapify(a, i, size);

}

return;

}

void max\_heapify(int \*a, int i, int size){

int left = i \* 2 + 1;

int right = i \* 2 + 2;

int max = i;

if(left < size && a[left] > a[i])

max = left;

if(right < size && a[right] > a[max])

max = right;

if(max != i){

int temp = a[i];

a[i] = a[max];

a[max] = temp;

max\_heapify(a, max, size);

}

return;

}

void quick\_sort(int \*a, int start, int end){

if(start < end){

int q = partition(a, start, end);

quick\_sort(a, start, q - 1);

quick\_sort(a, q + 1, end);

}

return;

}

int partition(int \*a, int start, int end){

int x = a[end];

int i = start - 1;

for(int j = start; j < end; j++){

if(a[j] <= x){

i = i + 1;

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

int temp = a[i + 1];

a[i + 1] = a[end];

a[end] = temp;

return i + 1;

}

void init\_array(int \*a, int i){

srand(i);

for(int j = 0; j < (i + 1) \* 10; j++){

a[j] = rand() % ((i + 1) \* 20);

}

return;

}

void print\_array(int \*\*a){

//array출력

for(int i = 0; i < 10; i++){

for(int j = 0; j < (i + 1) \* 10; j++){

printf("%d ", a[i][j]);

}

printf("\n");

}

return;

}

void print\_line(int \*a, int size){

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

printf("%d ", a[i]);

}

printf("\n\n");

return;

}