|  |  |  |
| --- | --- | --- |
| **Algorithms analysis** | Section | 02 |
| Student number | 21900371 |
| **Homework 3 – Sorting algorithms** | Name | Seo, Hyo Gyeong |

*If your explanation is less informative and insufficient, then you may not get any points.*

*Also, you should provide discussion, otherwise you will get penalty.*

**□ Explanation of Insertion sort**

Insertion sort is an algorithm that compares all elements of an array in order to the ordered portion of an array and sorts elements in an appropriate location.  
Specific algorithms are as follows. (Suppose that an array of sizes n is sorted in ascending order.)

1. Assume that the first data are already sorted and compare elements from the second element with the immediately preceding elements.

2. If the value of key is greater than the i-th value of an ordered array, move the value of i to i+1.

3. sorting all the way up to the (n-1)th, element will align the entire array.

<Time Complexity>  
If the entire array is sorted, comparisons are made without movement, so time is needed just for comparisons of n-1. Therefore, in the best case, the time complexity is very efficient with O(n). If the input data are in reverse order, the total number of comparisons is n(n-1)/2, so the worst time complexity is O(n^2). Therefore, the insertion alignment has a large deviation in performance depending on the data.

In my code, I implemented insertion sort through below function.

**void** insertion\_sort(vector<**int**> \*res, **int** size)

In this function, while outer loop iterate, value of each index is key and if the former index value is bigger than the value move the former value to the next index. And if not, break the inner loop and store key value to that position.

**□ Explanation of Heap sort**

Heap sort is an algorithm that creates and sorts a maximum heap or minimum heap tree in the form of a full binary tree.  
Specific algorithm for heap sort is as follows.

1. Make the array the maximum heap.

- Inserts a new node into the heap as the last node.  
- Compare the new nodes with the parent nodes and exchange them to match the properties of the heap.

2. Remove the elements one by one from the heap and store them from the back in the array.

- Delete the root node with the maximum value.  
- Gets the last node of the heap to the deleted root node location.  
- Reconstruct the heap.

3. Sort them in ascending order.

<Time Complexity>  
Heap sort consists of a fully binary tree form, which takes as long as logn to reconstruct the heap. Since the number of array elements is n, the time complexity is O(nlogn).

In my code, I implemented heap sort through these three functions.

**void** heap\_sort(vector<**int**> \*res, **int** size);

**void** build\_max\_heap(vector<**int**> &res, **int** size);

**void** max\_heapify(vector<**int**> &res, **int** parent, **int** size);

First of all, it builds max heap. From array\_size/2-1 it makes max heap from bottom area so after iterating the loop, it make one max heap. For making max heap I used max\_heapify. In this code, It finds the max value among the parent, left child and right child value. If max value is different from parent node value, swap the max value and parent value and using recursion, reconstruct the child node’s heap. After making max heap, by swapping the top node and last node and reconstruct the size-1 heap repeatedly, sort the array in ascending order.

**□ Explanation of Quick sort**

Quick sort is an alignment algorithm using divide and conquest methods. First, we divide the array into two subarrays based on pivot. The subarray is ordered by segmentation until it is sufficiently small. After sorting, partial arrays are merged to form another array.  
Specific algorithms are as follows.

1. Select one element from an array and designate it as a pivot.

2. Elements smaller than pivot move to the left of the pivot and values greater than the pivot to the right of the pivot.

3. Sort two separate arrays by Pivot.

- Use recursion to divide and sort arrays until each array is small enough.

4. When sorting is completed by splitting, the array is combined by return.

<Time Complexity>  
Quick sort use recursion where the depth is logn, and the time complexity is O(nlogn) at best, since n comparative operations are made at each step. In the worst-case case, the array is disproportionately divided and the depth of the recursion is n, so the time complexity becomes O(n^2), which can take a long time. However, quick alignment is efficient because once determined pivots are excluded from computation and maximize unnecessary data movement, requiring no additional memory and faster than algorithms with the same time complexity. Therefore, it is important to choose the appropriate pivot so that the data can be divided equally.

In my code, I implemented quick sort through below two function.

**void** quick\_sort(vector<**int**> &res, **int** start, **int** end);

**int** partition(vector<**int**> &res, **int** start, **int** end);

In quick\_sort function, it uses recursion. By using partition function, determine pivot(initial pivot is last element) and divide in two array, first part is that end point is pivot-1 and second part is that start point is pivot+1, so through recursion it divided more smaller part and sorted. In partition function, it compare the pivot and values. If the value is same or less than pivot, swap the value and increase the index. After iteration, swap the index+1’s value and last value, and return the index+1 for pivot.

**□ 10 arrays**

|  |  |
| --- | --- |
| Size | Elements |
| 10 | [351,69,212,167,179,131,407,50,300,189] |
| 20 | [201,490,276,186,210,114,313,452,452,230,415,4,500,260,466,157,352,401,415,319] |
| 30 | [51,58,340,205,388,97,72,354,104,271,122,432,73,66,375,338,351,204,122,331,337,210,415,118,479,338,440,295,287,316] |
| 40 | [401,479,404,224,419,80,478,256,256,459,329,360,499,19,431,166,203,154,329,490,400,162,102,59,158,166,155,491,500,421,147,357,257,78,392,46,279,104,195,470] |
| 50 | [251,47,115,243,450,63,384,158,408,500,36,288,72,325,340,347,202,104,36,149,463,114,142,147,484,494,17,40,360,26,382,71,446,134,453,197,25,468,442,374,284,486,291,365,437,105,422,264,5,228] |
| 60 | [101,468,179,262,128,46,143,60,60,41,96,216,498,131,249,175,54,407,96,161,26,419,329,88,310,175,232,89,73,131,470,285,135,43,161,495,418,479,42,278,223,383,149,79,336,414,118,346,294,491,138,380,464,147,172,298,386,258,17,255] |
| 70 | [451,389,243,281,159,29,49,462,212,82,303,144,71,437,305,356,53,357,303,320,89,371,369,176,489,3,447,285,286,236,205,499,324,99,222,293,164,343,289,182,309,280,7,440,235,76,314,428,436,107,455,2,100,289,426,475,411,443,255,464,249,229,191,84,136,267,280,244,144,376] |
| 80 | [301,457,307,300,190,12,455,364,364,270,10,72,497,390,214,184,405,160,10,479,299,323,56,117,315,331,309,334,499,341,293,213,13,155,283,444,410,207,389,439,395,30,218,154,281,238,157,10,78,223,419,124,236,78,180,299,83,128,140,173,234,354,319,432,483,242,219,409,488,438,301,410,331,46,484,406,429,461,313,325] |
| 90 | [151,378,371,319,368,495,214,266,16,311,217,500,70,196,123,365,257,110,217,491,362,128,96,205,141,12,24,383,212,446,28,427,55,64,344,242,303,71,136,343,334,427,76,368,180,47,353,445,367,339,383,393,19,220,434,123,255,460,378,382,72,332,447,280,330,217,158,221,479,500,26,167,23,434,265,78,170,467,477,189,67,69,22,28,321,150,94,379,448,448] |
| 100 | [1,446,82,338,399,478,120,168,168,352,424,428,496,2,179,193,256,60,424,150,425,80,283,146,320,340,386,79,72,51,116,494,244,120,405,393,49,435,236,247,420,324,434,229,226,209,196,27,9,455,200,15,155,362,188,300,280,145,263,91,57,457,75,128,30,339,450,386,470,209,251,277,215,469,193,397,411,326,141,406,290,387,472,433,203,343,405,287,148,233,449,10,276,369,136,467,219,261,303,72] |

**□ Screenshots of your program running**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

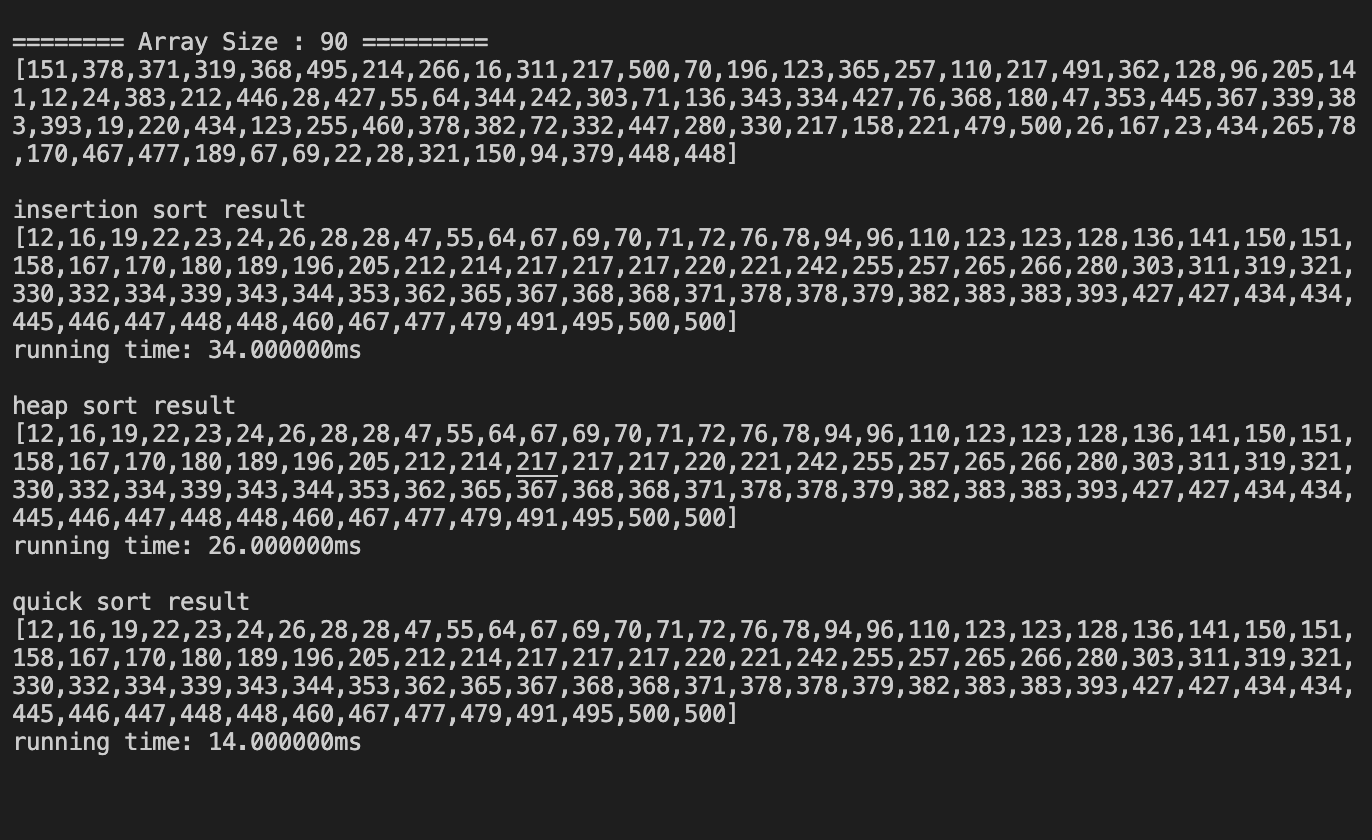
자동 생성된 설명

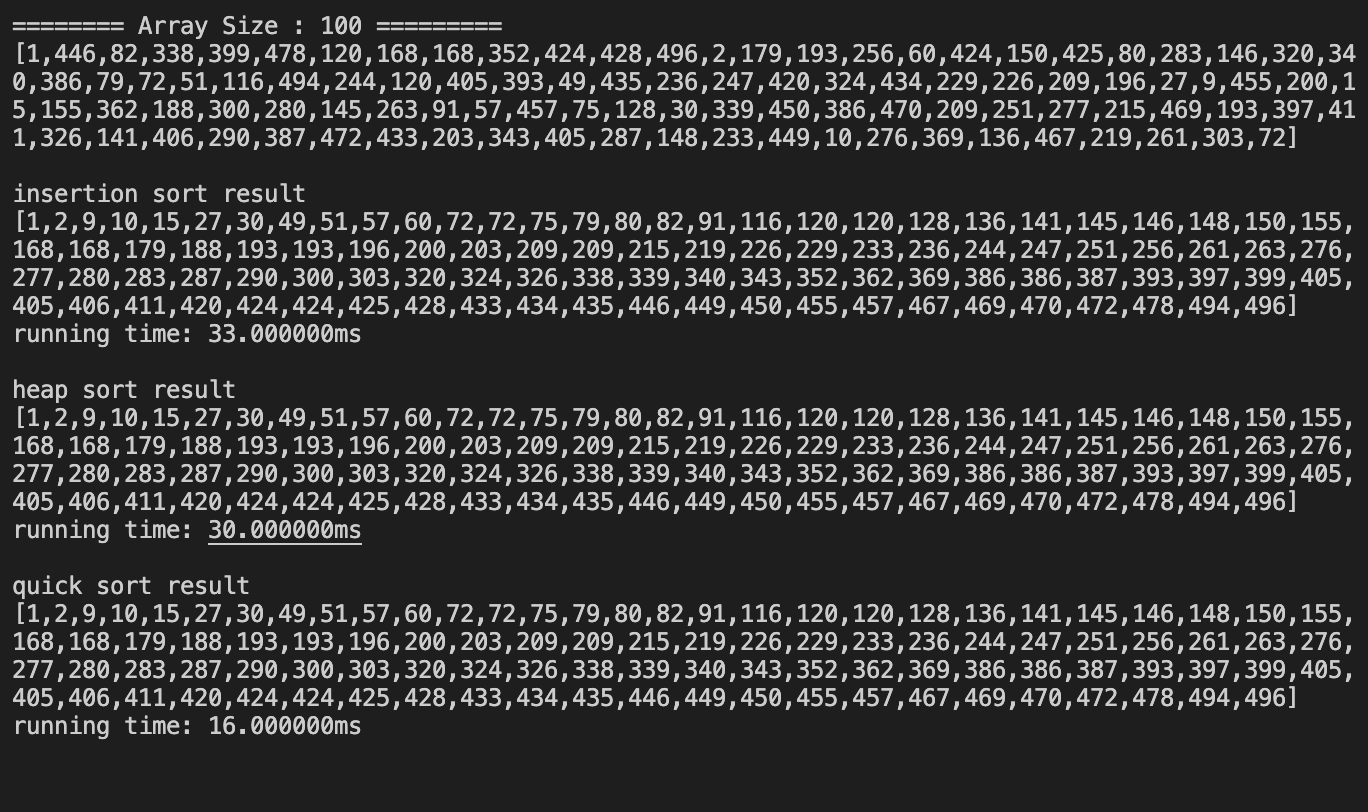
텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명





**□ Discussion about the results**

- The clock() function in #include <time.h> is used to calculate the time it takes to sort according to each algorithm. The clock function allows us to know the time in ‘ms’, which is used to calculate the running time. The shorter the running time, the more efficient it is. Therefore, I compared which algorithms are efficient using running time.

- Comparing the running time between heap sort and insertion sort, the heap sort was slightly faster when the array was larger, but there was no significant difference. Given the average time complexity, it is thought that heap sort with time complexity O(nlogn) should be faster than insertion sort with time complexity O(n^2), but the insertion sort can be faster at most O(n) depending on how the array is aligned.

- When comparing the running time of the same heap sort with the average time complexity of O(nlogn), the running time of the quick sort was generally short. Quick sort has the time complexity of O(n^2) as the worst case scenario when disproportionately divided by pivot. In this code, elements of the array are specified as random values, so they are generally divided into proportions. Therefore, in general, it will have average time complexity. However, the heap sort must initially construct the heap, but the quick sort takes advantage of the array structure. Therefore, it takes time to construct the heap, and even with the same time complexity, the heap sort takes longer.

- Based on the results of implementing this code alone, it can be thought that quick sort is more efficient than heap alignment, but heap sort can be more efficient depending on the situation we use. Quick sort is useful when data is quickly sorted once. However, when data continues to be added or deleted, heap sort remains aligned, so heap sort may be more efficient in situations requiring multiple sorting.

- I set the initial element as the pivot to be the baseline in code implementing quick sort. However, care must be taken not to set the pivot to a biased value, since in quick sort, it is efficient when the pivot has a median value and is equally partitioned.

- In this code, the array size is small and experiments are done in limited situations, so there is no significant difference in running time and one algorithm seems efficient. However, since each algorithm has its pros and cons and there can be a deviation in the time for sorting depending on the data, it is necessary to apply the appropriate algorithm depending on the situation where sorting is required.

**□ Codes** // you should also submit the separate executable C or C++ files, TA will try run your code.

#include <iostream>

#include <vector>

#include <cstdlib>

#include <time.h>

#define SEED 5

using namespace std;

void generate\_array(vector<int> \*arr, int size); // generate array

void print\_array(vector<int> \*arr, int size); // print generated array

void insertion\_sort(vector<int> \*res, int size); //Insertion sort

void heap\_sort(vector<int> \*res, int size); // Heap Sort

void build\_max\_heap(vector<int> &res, int size); //build max heap (Heap Sort)

void max\_heapify(vector<int> &res, int parent, int size); //reconstruct the heap.

void quick\_sort(vector<int> &res, int start, int end);// Quick sort

int partition(vector<int> &res, int start, int end); // partition (Quick sort)

int main(){

vector<int> arr;

vector<int> res;

/\* for calculating running time ( 1clock = 1ms ) \*/

clock\_t i\_start, i\_end;

clock\_t h\_start, h\_end;

clock\_t q\_start, q\_end;

for(int i=1; i<=10;i++){

printf("======== Array Size : %d =========\n",i\*10); // print array size

generate\_array(&arr,i\*10); //generate array

print\_array(&arr,i\*10); //print generated array

printf("\n");

/\* insertion sort \*/

i\_start = clock();

res.resize((int)(arr.size()));

copy(arr.begin(),arr.end(),res.begin()); //copy array to result vector

insertion\_sort(&res, i\*10); //insertion sort

i\_end = clock();

printf("insertion sort result\n");

print\_array(&res,i\*10); //print result of insertion sort

printf("running time: %lfms\n\n",(double)(i\_end - i\_start)); //caculate run time

res.clear(); //clear result vector

/\* Heap sort \*/

h\_start = clock();

res.resize((int)(arr.size()));

copy(arr.begin(),arr.end(),res.begin()); //copy array to result vector

heap\_sort(&res, i\*10); // Heap Sort

h\_end = clock();

printf("heap sort result\n");

print\_array(&res,i\*10); //print result of heap sort

printf("running time: %lfms\n\n",(double)(h\_end - h\_start)); //caculate run time

res.clear(); //clear result vector

/\* Quick sort \*/

q\_start = clock();

res.resize((int)(arr.size()));

copy(arr.begin(),arr.end(),res.begin()); //copy array to result vector

quick\_sort(res,0, i\*10-1); // quick Sort

q\_end = clock();

printf("quick sort result\n");

print\_array(&res,i\*10); //print result of quick sort

printf("running time: %lfms\n\n",(double)(q\_end - q\_start)); //caculate run time

res.clear(); //clear result vector

arr.clear(); //clear vector

printf("\n\n");

}

return 0;

}

/\* generate array \*/

void generate\_array(vector<int> \*arr, int size){

srand(SEED \* size); //set random seed (seed \* size)

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

int r = rand()%500+1; //range = [1,500]

arr->push\_back(r); //save random number in vector

}

}

/\* print generated array \*/

void print\_array(vector<int> \*arr, int size){

printf("[");

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

printf("%d",arr->at(i)); //print elements

if(i!=size-1){

printf(",");

}else{

printf("]\n");

}

}

}

/\* Insertion Sort \*/

void insertion\_sort(vector<int> \*res, int size){

int j, key;

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

key = res->at(i);

for(j=i-1; j>=0; j--){

if(res->at(j) > key){ //if value is bigger than key

res->at(j+1) = res->at(j); //move the value to index+1

}else{

break; //if not exit the loop

}

}

res->at(j+1) = key; //move key to proper position

}

}

/\* Heap Sort \*/

void heap\_sort(vector<int> \*res, int size){

build\_max\_heap(\*res, size); //build max heap

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

swap(res->at(0),res->at(i)); //swap top node and last node

size--;

max\_heapify(\*res,0,size); // reconstruct the remained heap

}

}

void build\_max\_heap(vector<int> &res, int size){ //make array to heap

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

max\_heapify(res,i,size); //make heap from the bottom

}

}

void max\_heapify(vector<int> &res, int parent, int size){ //reconstruct heap

int left, right, max;

max = parent;

left = parent \* 2;

right = parent \* 2 + 1;

/\* find the max among parent, left child, right child \*/

if(left<size && res.at(left) > res.at(max)){

max = left;

}

if(right<size && res.at(right) > res.at(max)){

max = right;

}

if(max != parent){ //if max value is not parent value

swap(res.at(parent),res.at(max)); //swap max value to parent position

max\_heapify(res,max,size); //reconstruct the child's heap

}

}

/\* Quick sort \*/

void quick\_sort(vector<int> &res, int start, int end){

if(start<end){

int pivot = partition(res, start, end); //recieve pivot

quick\_sort(res,start,pivot-1); // left part of pivot

quick\_sort(res,pivot+1,end); // right part of pivot

}

}

int partition(vector<int> &res, int start, int end){ // return the pivot's index

int pivot = res.at(end); //set last element to pivot

int i = start - 1;

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

if(res.at(j)<=pivot){ // if j's value is same or less than pivot

i = i+1;

swap(res.at(i),res.at(j)); //swap the value to i's value

}

}

swap(res.at(i+1),res.at(end)); //swap the last element value and i+1's value

return i+1; //return pivot's index

}