Algorithms PA1 report

B02703036 財金四 林耘寬

Figure 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Input size | IS | | MS | | HS | |
|  | CPU time  (s) | Memory  (KB) | CPU time  (s) | Memory  (KB) | CPU time  (s) | Memory  (KB) |
| 4000.bc | 0ms | 11696KB | 0ms | 11696KB | 0ms | 11696KB |
| 4000.wc | 12.001ms | 11696KB | 4ms | 11696KB | 0ms | 11696KB |
| 4000.ac | 4ms | 11696KB | 0ms | 11696KB | 0ms | 11696KB |
| 16000.bc | 0ms | 11840KB | 8ms | 11840KB | 4ms | 11840KB |
| 16000.wc | 188.012ms | 11840KB | 8ms | 11840KB | 4ms | 11840KB |
| 16000.ac | 96.006ms | 11840KB | 8.001ms | 11840KB | 4ms | 11840KB |
| 32000.bc | 0ms | 11840KB | 16.001ms | 11968KB | 4ms | 11840KB |
| 32000.wc | 648.041ms | 11840KB | 16.001ms | 11968KB | 4.001ms | 11840KB |
| 32000.ac | 360.022ms | 11840KB | 16.001ms | 11968KB | 4ms | 11840KB |
| 1000000.bc | 0ms | 17860KB | 428.027ms | 24004KB | 108.007ms | 17860KB |
| 1000000.wc | 607246ms | 17860KB | 408.025ms | 24004KB | 108.007ms | 17860KB |
| 1000000.ac | 302227ms | 17860KB | 484.029ms | 24004KB | 188.011ms | 17860KB |

Figure 2

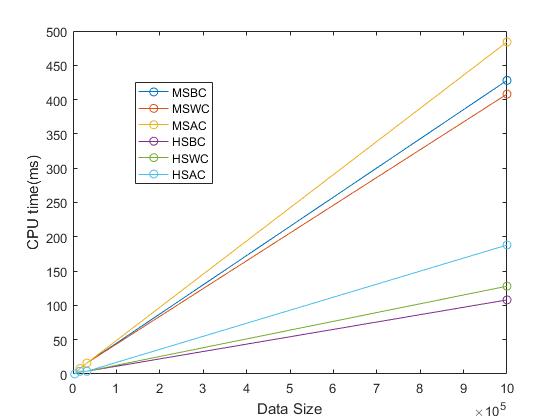
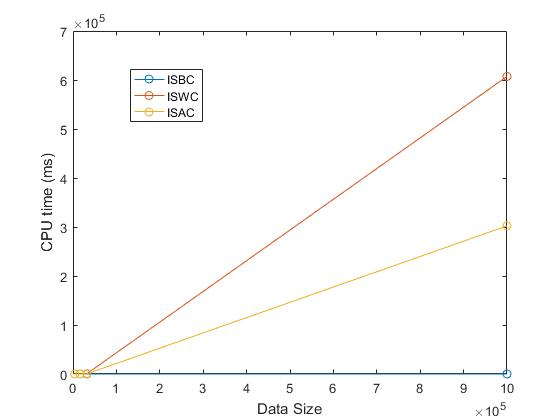


Figure 3



Discussion:

1. By Figure 1 above, we can find that the best case of insertion sort is faster than both merge sort and heap sort, which the time complexity are O(nlogn). And the time complexity of insertion sort is O(n) if the data is the best case.
2. We can easily find out that if the data size is large, both the average case and worst case, the insertion sort is much slower than the merge sort and heap sort. It supports the theorem that in average case and worst case, the time complexity of insertion sort is O(n2) and the time complexity of merge sort and heap sort is O(nlgn).
3. Since merge sort needs another array to store the data temporarily, the memory uses in merge sort is larger than heap sort and insertion sort if the data size is large enough.

Explanation:

I have made a subtle change in sorttool.cpp. I have changed the function max\_heapify from Max\_Heapify(vector<int>& data, int root)

to Max\_Heapify(vector<int>& data, int root, int n) because I don’t notice that the heapsize variable in sorttool.h at first. I add another variable n to represent the current heapsize.

Appendix

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// File [sort\_tool.cpp]

// Author [Yun Kuan, Lin]

// Synopsis [The implementation of the SortTool Class]

// Modify [2017/03/21 ]

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include "sort\_tool.h"

#include<iostream>

// Constructor

SortTool::SortTool() {}

// Insertsion sort method

void SortTool::InsertionSort(vector<int>& data) {

// Function : Insertion sort

// TODO : Please complete insertion sort code here

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

int flag = data[i];

int j = i - 1;

while (j >= 0 && data[j] > flag){

data[j + 1] = data[j];

j--;

}

data[j + 1] = flag;

}

}

// Merge sort method

void SortTool::MergeSort(vector<int>& data){

SortSubVector(data, 0, data.size() - 1);

}

// Sort subvector

void SortTool::SortSubVector(vector<int>& data, int low, int high) {

// Function : Sort subvector

// TODO : Please complete SortSubVector code here

// Hint : recursively call itself

// Merge function is needed

if(low != high){

SortSubVector(data, low, (low + high) / 2);

SortSubVector(data, (low + high) / 2 + 1, high);

Merge(data, low, (low + high) / 2, (low + high) / 2 + 1, high);

}

}

// Merge

void SortTool::Merge(vector<int>& data, int low, int middle1, int middle2, int high) {

// Function : Merge two sorted subvector

// TODO : Please complete the function

vector<int> v;

int start = low;

int end = high;

while (low <= middle1 && middle2 <= high){

if (data[low] <= data[middle2]){

v.push\_back(data[low]);

low++;

}

else{

v.push\_back(data[middle2]);

middle2++;

}

}

while (low <= middle1){

v.push\_back(data[low]);

low++;

}

while (middle2 <= high){

v.push\_back(data[middle2]);

middle2++;

}

int i = 0;

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

data[item] = v[i];

i++;

}

}

//Max heapify

void SortTool::Max\_Heapify(vector<int>& data, int root, int n) {

// Function : Make tree with given root be a max-heap

// if both right and left sub-tree are max-heap

// TODO : Please complete max-heapify code here

if (2 \* root + 1 <= n){

int largerchild = 2 \* root + 1;

if (2 \* root + 2 <= n){

int rightchild = 2 \* root + 2;

if(data[rightchild] > data[largerchild])

largerchild = rightchild;

}

if (data[root] < data[largerchild]){

int temp = data[root];

data[root] = data[largerchild];

data[largerchild] = temp;

int next\_root = largerchild;

Max\_Heapify(data, next\_root, n);

}

}

}

//Build max heap

void SortTool::Build\_Max\_Heap(vector<int>& data) {

// initialize heap size

// Function : Make input data become a max-heap

// TODO : Please complete Build\_Max\_Heap code here

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

Max\_Heapify(data, i, data.size()-1);

}

}

// Heap sort method

void SortTool::HeapSort(vector<int>& data) {

// Build Max-Heap

Build\_Max\_Heap(data);

// 1. Swap data[0] which is max value and data[i] so that the max value will be in correct location

// 2. Do max-heapify for data[0]

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

int temp = data[i];

data[i] = data[0];

data[0] = temp;

Max\_Heapify(data, 0, i-1);

}

}