# California State University, Fresno

# DEPARTMENT OF COMPUTER SCIENCE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class: | **Algorithms & Data Structures** | | | Semester: | **Spring 2022** |
|  | | | | | |
| Points |  | Document author: | **MingkuanPang** | | |
|  | Author’s email: | **Yafking20 @mail.fresnostate.edu email** | | |
| Laboratory number: | **03** | | |
|  | | | | | |

**1. Statement of Objectives**

This lab asks for implementing two algorithms: insert sort and merge sort. Then analyzing their time complexities in different cases and comparing them to one another.

**2. Experimental Procedure**

**selection sort:**

Text

Description automatically generated

For the selection sort, set up the index for the minimum variable to be 0, and min to be Array [0].

Then travel the array start from Start. Every time when Array[i] is less than min, set min to be Array[i]. That way, when reach the end of the array, min will be the minimum variablein the array.

Then swap the value between Array [Start] and min. After finish swapping value, increase Start position by 1 for the next array travel. And set min\_index to be Start and min to be v[Start]. Keep doing this operation until Start reach the end of the array.

**Merge:**

Text

Description automatically generated

For the merge function, we first divide the array into two parts. (A[l]…A[m]) and (A[m+1] …A[r]).

Then we compare the elements in those two-sub array, then choose the smaller one and inert it into array A from first position till its end.

**Merge Sort:**

Text

Description automatically generated

For merge sort, **l** will be the left boundary of the input array, and the **r** will be the right boundary of the input array. When right boundary is at least **1** greater than the left boundary (**r**-**l**>1), we divide the array evenly until it is undividable, then we use the merge function to combine them from the bottom to the top. Finally, the array will be sorted.

**3. Analysis**

**selection sort:**

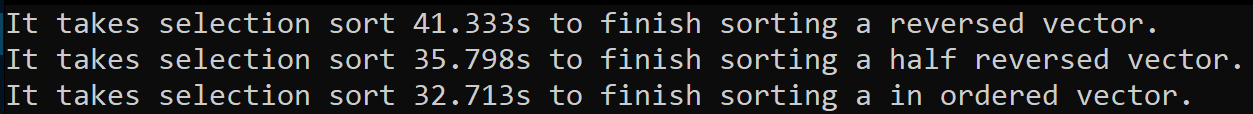
**Input:**

Graphical user interface

Description automatically generated

First, created three types of arrays: reversed, half-sorted and completely sorted with the size of 50000. Then using the insert sort to sort them.

**Output:**



From the result, there is big difference between those three cases of input. Even the array is half reversed, it will be much better than the array that fully reversed.

**O(n) of selection sort:**

void selection\_sort(vector<int>& v)

{

if (v.size() > 1)

{

int min\_index = 0, Start = 0, min = v[0];

while (Start < v.size() - 1) **----------------------🡪 n**

{

for (int i =Start; i < v.size(); i++)**----------🡪**

{

if (v[i] < min) ------------🡪 **C**

{

min = v[i];

min\_index = i;

}

}

swap(v[min\_index], v[Start]); **-----------🡪 C**

Start++; **-----------🡪 C**

min\_index = Start; **-----------🡪 C**

min = v[Start]; **-----------🡪 C**

}

}

}

1. For T(n) of the while loop, T(n) = n-1+1=n.
2. For T(n) of the for loop, T(n) = c\*(n+n-1+n-2+…. +1) =c·

For the rest of part, T(n) = c+c+c+c = 4c, therefore, T(n) for the selection sort will be:

O(n) = c· + 4c = c· + c

**For the case of input is a sorted array:**

Because the array is already sorted, the for loop does nothing. Therefore,

T(n) =1· = , for the rest will remain the same. Therefore, in this case

T(n) = +4c = +c

**For the case of input is a half-sorted array:**

Because the array is half sorted, the code in the for loop only execute half of the time.

Therefore, T(n) = · = , therefore, in this case

T(n) = + 4c = + c

**For the case of input is a reversed array:**

That will be the worst case, T(n) = O(n). Therefore, in this case

T(n) = c· + c

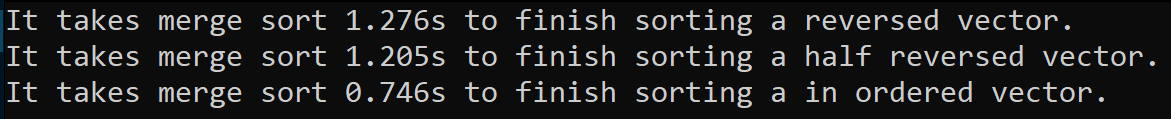
**Merge sort:**

**Input:**

Text

Description automatically generated

**Output:**



Merge sort is much efficient than the selection sort from comparing their outputs. Merge sort only needs 1.3 seconds to sort a complete reversed array with size 50000 while selection sort needs 41 seconds.

**O(n) of merge sort:**

void merge(vector<int>& v, int l, int m, int r)

{

if (v.size() > 1)

{

vector<int>lv;

vector<int>rv;

int len1 = m - l;

int len2 = r - m;

for (int i = 0; i < len1; i++) **--------------🡪 n**

{

lv.push\_back(v[i + l]);

}

for (int i = 0; i < len2; i++) **--------------🡪 n**

{

rv.push\_back(v[i + m]);

}

lv.push\_back(INT\_MAX); **------------🡪 c**

rv.push\_back(INT\_MAX); **-----------🡪 c**

int j = 0, k = 0, pos = l;

for (int i = l; i < r; i++) **---------------🡪 n**

{

if (lv[j] <= rv[k])

v[i] = lv[j++];

else

v[i] = rv[k++];

}

}

}

void merge\_sort(vector<int>& v, int l, int r)

{

if (r - l > 1)

{

int middle = (l + r) / 2;

merge\_sort(v, l, middle);

merge\_sort(v, middle, r);

merge (v, l, middle, r); **---------🡪 3**c **·** n + c

}

}

**Merge:**

For line 1, T(n) = c **·** (n-1+1) = c **·** n.

For line 2, T(n) = c **·** (n-1+1) = c **·** n.

For line 3, T(n) = c **·** (n-1+1) = c **·** n.

For the rest of part of merge function, T(n) = c+c = 2c = c

Therefore, for the merge function,

O(n) = c **·** n + c **·** n + c **·** n + c = 3 c **·** n + c = c **·** n = n

**Merge Sort:**

In the merge sort, we divide the size of input in half in every recursive step. Therefore, we have

log(n) steps in total. In every step, we run merge function once. Therefore, for the merge sort:

O(n) = n **·** log(n)

**For the case of input is a sorted array:**

In this case, the code the third for loop will not be executed, the rest will remain the same. Therefore, T(n) for the merge function in this case:

T(n) = c **·** n + c **·** n + n +2c = (2c+1) n+2c

Therefore, T(n) for the merge sort in this case:

T(n) = (3n+2c) · log(n).

**For the case of input is a half-sorted array:**

In this case, the code the third for loop will only be executed half of the time, and the rest will remain the same. Therefore, T(n) for the merge function in this case:

T(n) = cn + cn + + +2c = + 2c

Therefore, T(n) for the merge sort in this case:

T(n) = + 2c) · log(n).

**For the case of input is a reversed array:**

That will be the worst case, therefore:

T(n) for the merge function in this case:

T(n) = c **·** n + c **·** n + c **·** n +2c = 3cn + 2c

Therefore, T(n) for the merge sort in this case:

T(n) = (3n+2) c · log(n).

**4. Encountered Problems**

When I tried to write the merge function, I made the recursion keep going while l is less than r. It will not make the merge function work correctly. Because when l = 2, r =3, the mid will be

=2, then mid will be equal to l, that will make the merge divide the array into one sub array has 0 element, and the other only has one element. That will make the merge function work wrongly. Therefore, I make the recursion keep going while r is at least 2 greater than l. because that is not necessarily to execute if r is equal to or only 1 greater than l.

**5. Conclusions**

Even merge sort is written in recursive way, and it consists of two functions, it is still much better than selection sort. However, selection sort takes less movement in sorting array than merge sort and it easier to implement. Therefore, for sorting the array with small size, we can use selection sort. For sorting large size of array, merge sort will be a good choice!

**6. References**

I did not use any references in this report.