Lab Report

CMPT 225

Junchen Li

301385486

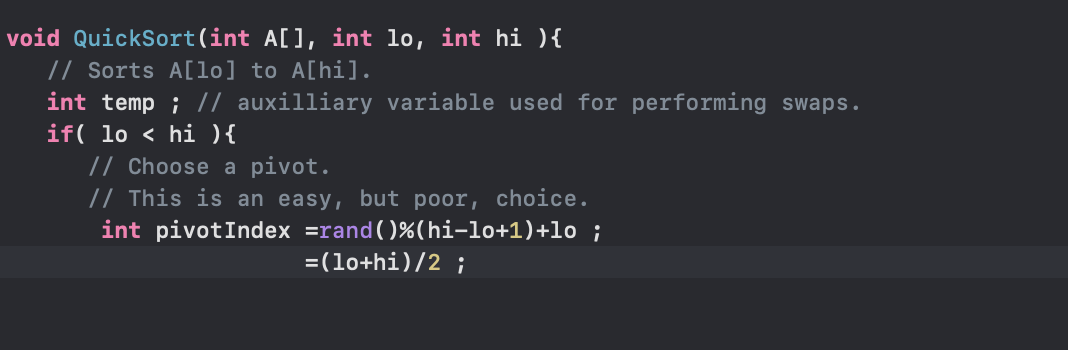
2020/4/5

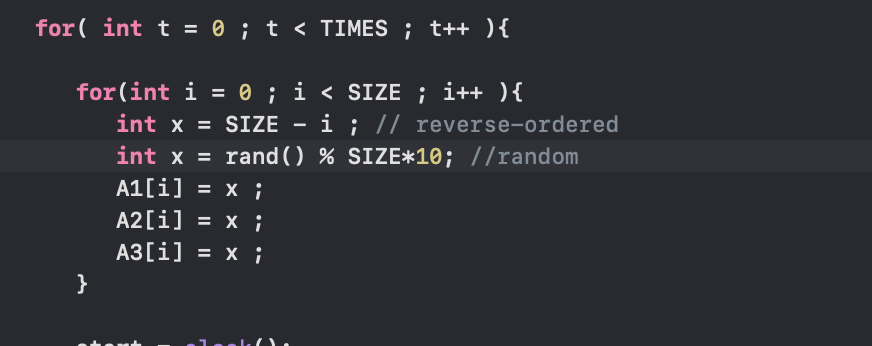
Part one:

Purpose:

I choose six different size of data (10000,20000,30000,40000,50000,60000) as our x variable. We test two different data type (integer and char) each one we divided into two different orders of data (we call it data family) (random and reverse). We try to use three different sorting methods (C++ sort, Quick sort, Insert sort) to test their time efficiency. What’s more, when we using Quick sort we also need two different pivot point positions (random and mid). Our main goal is figure out how different pivot point positions will influence the Quick sort running time.

Code of two different pivot point positions:



Code of two different ways of choosing data order:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***random integer*** data type array  ***Mid pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.4708 | 1.0077 | 1.6385 | 2.1596 | | 2.7638 | 3.3995 | 1.90665 |
| Quick sort | 0.974 | 2.2027 | 3.3895 | 4.5398 | | 5.9044 | 7.2741 | 4.04741 |
| Insert sort | 58.6448 | 244.755 | 552.843 | 988.029 | | 1531.92 | 2225.7 | 933.6486 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***random integer*** data type array  ***Random pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.476 | 1.0097 | 1.5785 | 2.2429 | | 2.7588 | 3.3833 | 1.9082 |
| Quick sort | 1.028 | 2.2389 | 3.4924 | 4.6691 | | 6.0347 | 7.3232 | 4.13105 |
| Insert sort | 62.6012 | 249.702 | 561.882 | 996.392 | | 1558.92 | 2242.97 | 951.451 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***reverse integer*** data type array  ***Mid pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.0322 | 0.0706 | 0.0866 | 0.1164 | | 0.1464 | 0.1707 | 0.10381 |
| Quick sort | 0.4465 | 0.9927 | 1.3863 | 1.9198 | | 2.4323 | 2.9054 | 1.6805 |
| Insert sort | 117.968 | 474.497 | 1054.43 | 1914.34 | | 3036.17 | 4168.87 | 1794.37 |

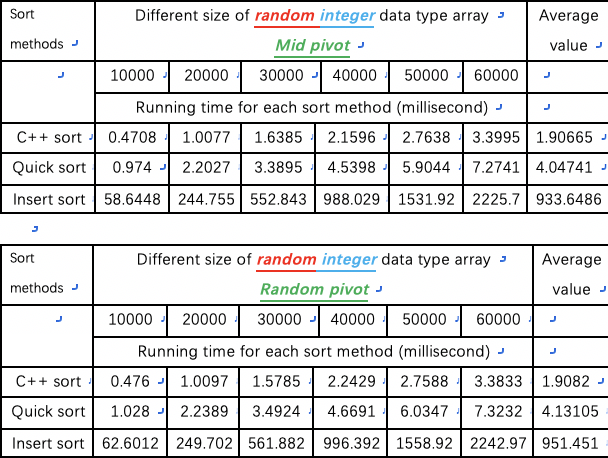
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***reverse integer*** data type array  ***Random pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.0327 | 0.0643 | 0.0925 | 0.1202 | | 0.1499 | 0.2055 | 0.11085 |
| Quick sort | 0.6729 | 1.4143 | 2.1988 | 3.0474 | | 3.9808 | 4.6256 | 2.65663 |
| Insert sort | 124.259 | 498.879 | 1118.46 | 1990.28 | | 3106.33 | 4485.04 | 1887.20 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***random char*** data type array  ***Mid pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.3043 | 0.5706 | 0.861 | 1.1413 | | 1.3957 | 1.6775 | 0.99173 |
| Quick sort | 1.3381 | 4.1645 | 8.1795 | 13.6428 | | 20.7467 | 29.4812 | 12.9254 |
| Insert sort | 58.817 | 236.061 | 523.221 | 920.036 | | 1441.95 | 2079.55 | 876.605 |

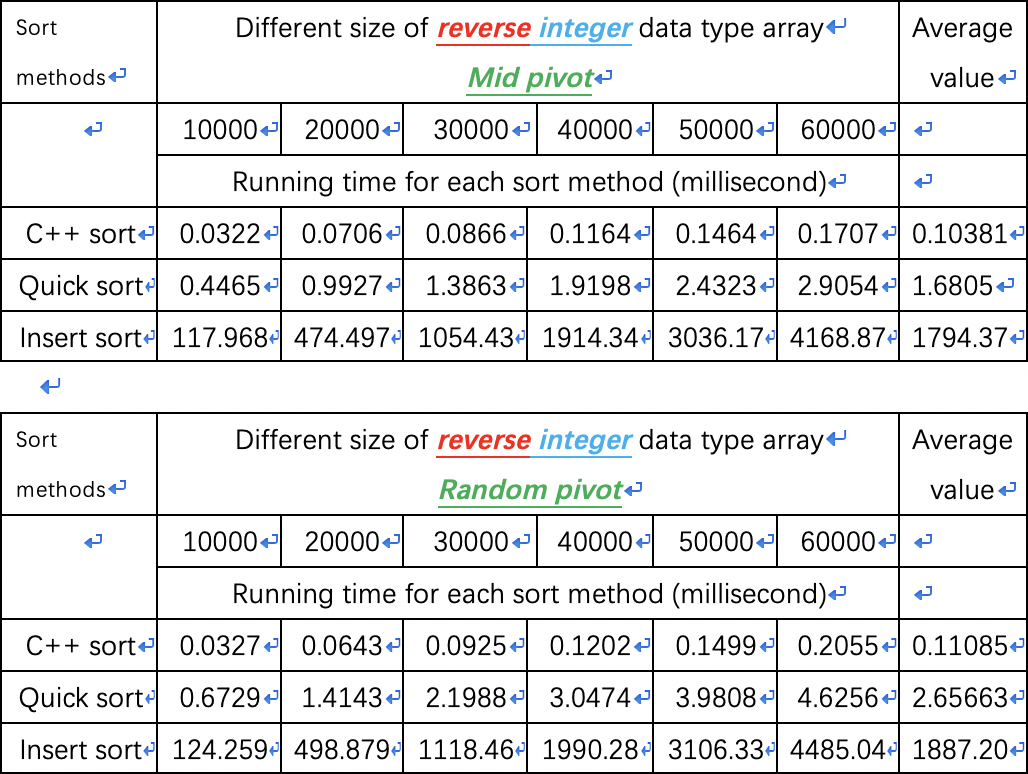
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***random char*** data type array  ***Random pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.291 | 0.5826 | 0.7641 | 1.1881 | | 1.416 | 1.6787 | 0.98675 |
| Quick sort | 1.0601 | 4.4773 | 8.6816 | 14.3953 | | 21.9042 | 30.259 | 13.4629 |
| Insert sort | 61.5937 | 246.213 | 548.267 | 983.879 | | 1516.09 | 2182.45 | 923.082 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***reverse char*** data type array  ***Mid pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.1665 | 0.3125 | 0.4533 | 0.6013 | | 0.7329 | 0.8748 | 0.52355 |
| Quick sort | 0.8307 | 2.7192 | 4.653 | 8.2488 | | 12.172 | 16.2737 | 7.4829 |
| Insert sort | 61.1482 | 244.83 | 543.92 | 972.051 | | 1510.72 | 2071.02 | 900.614 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of ***reverse char*** data type array  ***Random pivot*** | | | | | | | Average value |
|  | 10000 | 20000 | 30000 | | 40000 | 50000 | 60000 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| C++ sort | 0.1636 | 0.3192 | 0.4536 | 0.6521 | | 0.764 | 0.8943 | 0.54113 |
| Quick sort | 0.878 | 2.4514 | 5.0807 | 8.0254 | | 11.8067 | 16.5059 | 7.45801 |
| Insert sort | 64.4763 | 257.298 | 574.065 | 1020.43 | | 1591.41 | 2293.54 | 966.869 |



millisecond



the gap of each method become larger and larger.

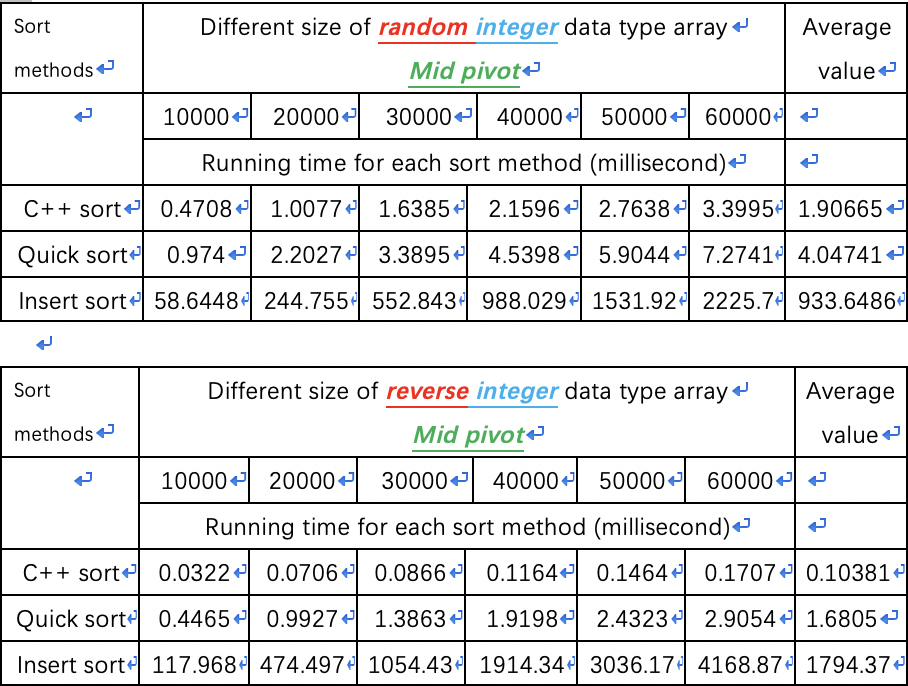
Next, let’s see if we can observe some difference between same choice for pivot but different data family.

For the reverse integer, as the size of data become bigger and bigger, the random choice pivot should need more time than mid pivot. Especially, after 30000 size of data, the difference

For the rest of all data and comparisons, when data type, the way of choose data family are same, the time efficiency is almost same for them. So we do not show all plot for the same result. However, when talking about integer and we use reverse order of data, results are quite different.

millisecond

Due to the large result in insert sort for both different case, we move it to independent plot. As we can observe in these two graph, for the quick sort method that we used, the result for both mid pivot choosing and random pivot choosing is almost same (we talking about the integer case). For handling with a large size of data, the C++ sort is the most efficient way, and then Quick sort finally the insert sort.



Due to the large data, we also move insert sort to other plot. We can find reverse order data of C++ sort is most efficient way to sort data. Random order data of C++ sort and Reverse order data of Quick sort are almost same before around 22000. After that, random data of C++ sort will be better. Unfortunately, random order of Quick sort will take pretty long time to sort them.

When we dealing with the insert sort with a large size of data, the reverse order data need more time than random order data. Especially when bigger than 30000, the reverse type growing faster than before.

All in all, when we use the C++ sort and Quick sort, the reverse order data will greater than random. Among them, the C++ is an excellent choice. However, when we using the insert sort, the random order data one will be better than random one. So the best case for integer is reverse order data by using C++ sort. The worst case is reverse order data by using insert sort.

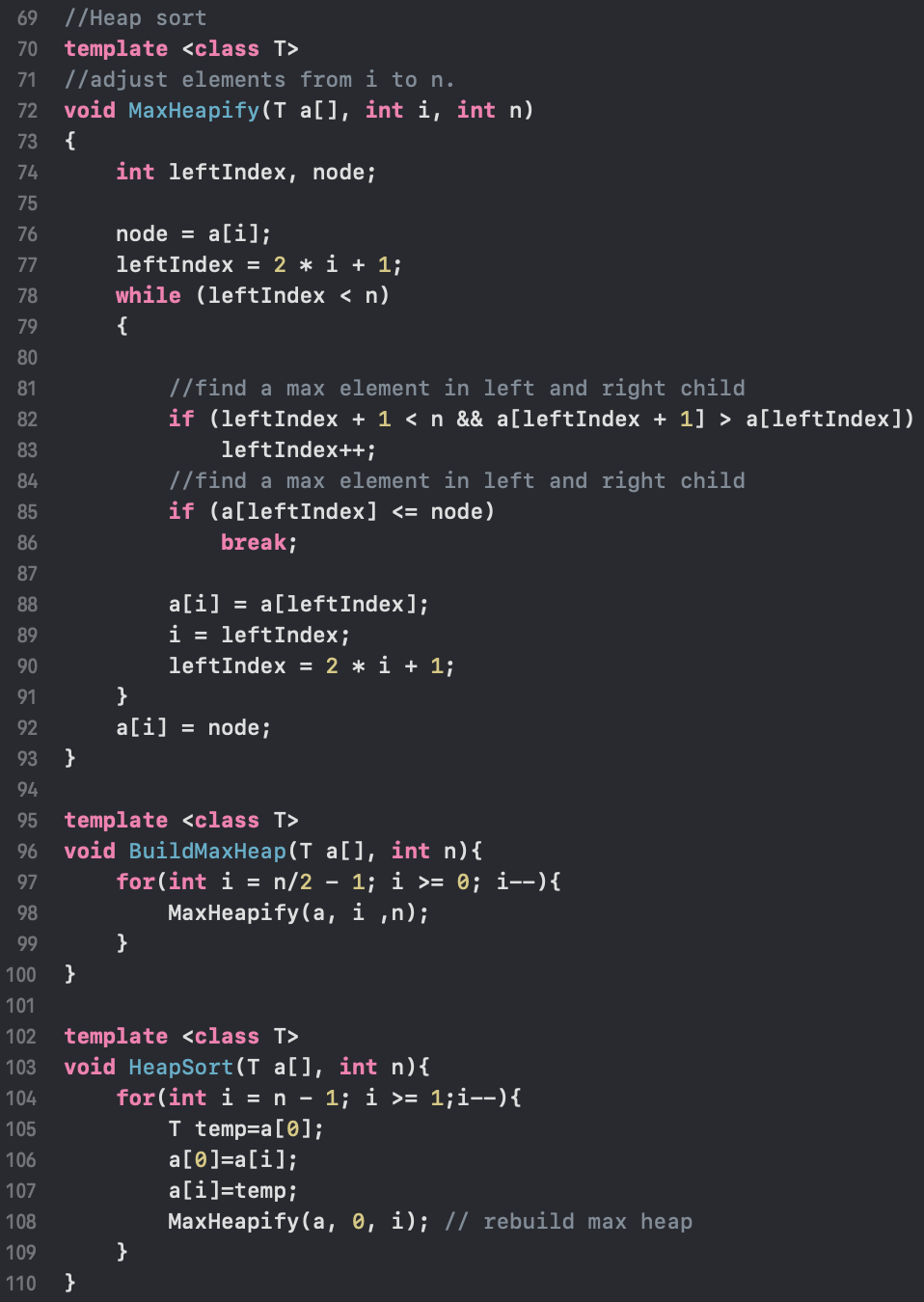
In this section, we add one more data type (char type) into our research. In general, we observe that the regularity of char type almost as same as the int type. So we do not repeat every details in the class of char. We collect them together and make a larger plot. As we can see in the above plot, firstly whatever the data type is, the C++ sort is most efficient way to sort data; the quick sort in random integer need more time than other situations. Random data family always need more time than reverse data family from the plot. Most importantly, whatever the data type is, random pivot position always need more time than mid pivot position in the Quick sort.

As we can see in the left plot, general when we dealing with the char data type, it should use more time than the integer data type. So the extremely case of best time efficiency is reverse order of integer by using C++ sort method to sort. Interesting thing is the worst case is reverse order of integer by using Insert sort.

Part two:

Before we testing all the sort, we need to find a sorting way that actually faster than on small inputs. We test (insert sort, selection sort, and heap sort) these sort first to see which one is best sort we can form our “simple sort”.

This is the code of Heap sort:



It has three parts one is main function called HeapSort. In HeapSort, it has BuildMaxHeap and MaxHeapity these two helper functions to complete the functioning.

Once we build our “simple sort”, we need to locate the amount of S. We need to think about when we need to use the simple sort.

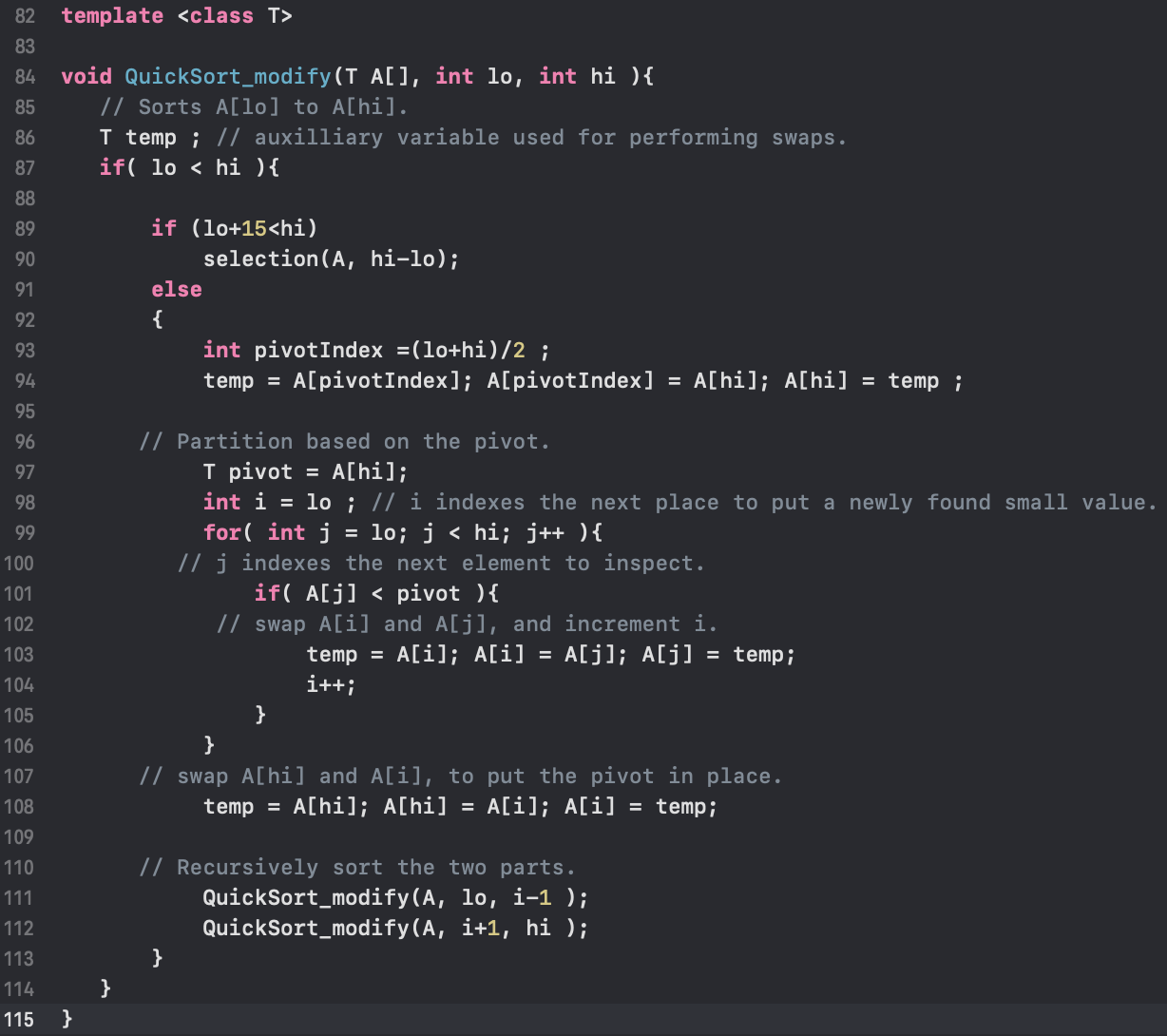
Choosing a sort method: get a roughly “S”

Because of we talking about the small amount of data size, we will choose (2,5,7,15,17,20).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort methods | Different size of small size data  in Insert sort, selection sort and heapsort | | | | | | | Average value |
|  | 2 | 5 | 7 | | 15 | 17 | 20 |  |
| Running time for each sort method (millisecond) | | | | | | |  |
| Insert sort | 0.0006 | 0.0006 | 0.0008 | 0.0008 | | 0.0011 | 0.001 | 0.00083 |
| Selection sort | 0.0004 | 0.0004 | 0.0006 | 0.0007 | | 0.0009 | 0.0013 | 0.00065 |
| Heapsort | 0.0005 | 0.0008 | 0.001 | 0.0007 | | 0.0006 | 0.0011 | 0.00091 |

After the experiment, we can see there is an intersection point when data size is fifteen. Before fifteen, selection is our best choice when we dealing with the small size data. After that, the heapsort is our excellent choice of sorting data. Also, we know the point of fifteen is a critical point. Our simple sort is selection sort.

**This is just a roughly S !!!**

So we add simple sort into our Quick sort modify and we will use C++ sort, Quick sort-standard, Quick sort-modify, Insertion sort, simple sort to finish our final test. For the testing purpose, we will not change the pivot point position for both quick sort.

Also, due to the better observation we want to use different range of size.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort methods | ***Random*** order integer with five different sort methods | | | | | |
|  | 2 | 5 | 7 | 15 | 17 | 20 |
| Running time for each sort method (millisecond) | | | | | |
| C++ sort | 0.0009 | 0.0008 | 0.0008 | 0.0011 | 0.0008 | 0.0009 |
| Simple sort | 0.0006 | 0.0006 | 0.0007 | 0.0009 | 0.0008 | 0.0012 |
| Insert sort | 0.0006 | 0.0004 | 0.0007 | 0.0007 | 0.0005 | 0.001 |
| Standard Quick | 0.0007 | 0.0007 | 0.0008 | 0.0012 | 0.0011 | 0.0012 |
| Modify Quick | 0.0004 | 0.0006 | 0.0007 | 0.0008 | 0.0009 | 0.0011 |

First at all, we talk about range of size (2 , 5 , 7 , 15 , 17 , 20) in the random order :

We talk about range of size (30 , 50 , 70 , 90 , 100 , 120) :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort methods | ***Random*** order integer with five different sort methods | | | | | |
|  | 30 | 50 | 70 | 90 | 100 | 120 |
| Running time for each sort method (millisecond) | | | | | |
| C++ sort | 0.0013 | 0.0017 | 0.002 | 0.0026 | 0.0031 | 0.0034 |
| Simple sort | 0.0021 | 0.0042 | 0.0076 | 0.011 | 0.0131 | 0.0182 |
| Insert sort | 0.0014 | 0.0028 | 0.0046 | 0.0068 | 0.0089 | 0.0116 |
| Standard Quick | 0.0016 | 0.0028 | 0.0041 | 0.0053 | 0.0064 | 0.0071 |
| Modify Quick | 0.0012 | 0.0021 | 0.0032 | 0.0042 | 0.0046 | 0.0058 |

As we can see in the above two plots, at the beginning (2~20) the simple sort is always better than Standard Quicksort. We also can know the insert sort show great performance. However, during the (20~120) the simple sort become slower than Standard Quicksort. So we started to focus on taking all the data in this interval.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort methods | ***Random*** order integer with five different sort methods | | | | | |
|  | 20 | 22 | 24 | 26 | 28 | 30 |
| Running time for each sort method (millisecond) | | | | | |
| C++ sort | 0.0009 | 0.0013 | 0.001 | 0.0014 | 0.0013 | 0.0015 |
| Simple sort | 0.0012 | 0.0015 | 0.0016 | 0.0018 | 0.0019 | 0.0021 |
| Insert sort | 0.001 | 0.0014 | 0.0012 | 0.0014 | 0.0014 | 0.0014 |
| Standard Quick | 0.0013 | 0.0015 | 0.0015 | 0.0017 | 0.0017 | 0.0016 |
| Modify Quick | 0.0011 | 0.0014 | 0.0012 | 0.0014 | 0.0015 | 0.0015 |

We only focus on the simple sort and Standard Quick sort at this time. We notice the relationship change between size of 22 and 24. So we conclusion that the S for random order integer around 23. (Srandom=23)

Mentioned one more detail is we tested selection sort has best functional running when small amount of data. However, when we look at the graph of range 2~20, insert sort looks like a best case. This is because unstable performance of the computer will affect the data and the data may be subject to fluctuations due to the small accuracy of the data.

Let’s turn to the reverse order integer to see if the S will be the same.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort methods | ***Reverse*** order integer with five different sort methods | | | | | |
|  | 2 | 5 | 7 | 15 | 17 | 20 |
| Running time for each sort method (millisecond) | | | | | |
| C++ sort | 0.0009 | 0.001 | 0.0009 | 0.0011 | 0.001 | 0.001 |
| Simple sort | 0.0005 | 0.0003 | 0.0007 | 0.0006 | 0.0008 | 0.0009 |
| Insert sort | 0.0004 | 0.0004 | 0.0004 | 0.0007 | 0.0016 | 0.0013 |
| Standard Quick | 0.0007 | 0.0007 | 0.0007 | 0.0008 | 0.0011 | 0.0007 |
| Modify Quick | 0.0006 | 0.0006 | 0.0003 | 0.0012 | 0.0013 | 0.001 |

This time we found during the 17~20 the relationship changed. So we zoom that gap and also we put two more bigger data to confirm we are in the right place.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sort methods | ***Reverse*** order integer with five different sort methods | | | | | |
|  | 17 | 18 | 19 | 20 | 25 | 30 |
| Running time for each sort method (millisecond) | | | | | |
| C++ sort | 0.0008 | 0.0016 | 0.0017 | 0.0009 | 0.001 | 0.0012 |
| Simple sort | 0.0006 | 0.0009 | 0.0011 | 0.001 | 0.0012 | 0.0015 |
| Insert sort | 0.001 | 0.0011 | 0.0011 | 0.001 | 0.0014 | 0.0022 |
| Standard Quick | 0.0007 | 0.0007 | 0.001 | 0.0009 | 0.0008 | 0.0013 |
| Modify Quick | 0.0007 | 0.0007 | 0.0008 | 0.0006 | 0.0008 | 0.0012 |

As we can see the red part, there is a intersection point. Due to the above table, we can see the S fir reverse integer is around 18

(Sreverse=18)

As the data size become larger and larger, the C++ sort shows its great performance of sorting.

Conclusion:

For the part one, we compare different data size, data family, data type and different pivot point position for Quick sort.

* We conclude whatever the data type and data family is, the random pivot point position is always need more time than reverse pivot point position
* The char data type need more time than integer data type.
* Random of data family is always need more time than reverse data family
* As the size become larger, the C++ sort method is always best choice of sorting.

In part two, we try to locate the size of S in two different data family and see if S are same or not.

* We conclude Srandom=23 and Sreverse=18 for the integer.
* As the size become larger, the C++ sort method is always best choice of sorting.
* Generally, after modifying Quick sort (or we can call it Intro sort) is much better than standard Quick sort.
* From the internet, I know for the Intro sort when size over max limitation of depth usually we will use the heapsort. ( We define the maximum depth limit as 2\*log(N) ). And when size in the middle, we can choose to use Quick sort.