Caroline Dang
July 18, 2023
CS 311

Comparison of Quicksort and Insertion Sort Algorithms: An Experimental Analysis

Introduction:

This experiment compares two sorting algorithms, Quicksort and Insertion Sort. Their runtime will be compared on three array types of sizes 10,000 to 500,000. The three arrays types are integer arrays of random order, ascending order, and descending order. This refers to the order of integers in the arrays. I hypothesize that the Quicksort will be faster than the Insertion sort for arrays of descending order and random order. The Insertion sort will be faster than Quicksort for arrays in ascending order. This experiment is an opportunity to apply knowledge gained in the course and gain practical insights into the behavior of fundamental algorithms.

Hypothesis

Quicksort average time will be less than Insertion Sort on random arrays, as Quicksort's average time complexity is O(n log n) compared to Insertion Sort's O(n^2).

Insertion Sort average time will be less than Quicksort on arrays already in ascending order, as Insertion Sort's best-case time complexity is O(n), which is better than Quicksort's O(n log n) in this scenario.

Quicksort average time will be less than Insertion Sort on arrays in descending order, as Quicksort's time complexity depends on the pivot selection strategy and still performs relatively well compared to Insertion Sort's O(n^2) worst-case time complexity.

Experimental design

This experiment aims to compare the runtime of two sorting algorithms, Quicksort and Insertion Sort, on three different types of arrays: random order, ascending order, and descending order. The independent variables in this experiment are the sorting algorithm used (Quicksort or

Insertion Sort) and the order of the arrays. The dependent variable is the runtime or execution time of the sorting algorithms. The experiment will be conducted using a computer with standard hardware and software. The sorting algorithms will be implemented in C++ programming language, and the runtime will be measured using the chrono library to ensure accuracy.

To conduct the experiment, arrays of various sizes ranging from 10,000 to 500,000 will be generated for each array type. For random arrays, random integers will be generated as elements. For ascending and descending arrays, the random arrays will be sorted accordingly. The runtime of both sorting algorithms will be measured for each array size and type.

To maintain consistency, the experiment will control other factors such as the same hardware and software setup for each trial. The collected data will be analyzed by calculating the average runtime for each array type and size, as well as the standard deviation to assess the variability of the results.

However, it is essential to acknowledge potential limitations of the experiment. Factors like background processes running on the computer and system load may slightly impact the results. Furthermore, the efficiency of the sorting algorithms may depend on the implementation and choice of pivot in Quicksort. Despite these potential limitations, this experimental design aims to provide valuable insights into the runtime performance of Quicksort and Insertion Sort on different array types and sizes, contributing to a better understanding of these fundamental algorithms.

Data collection

The data collection for this experiment involves measuring the runtime of the Quicksort and Insertion Sort algorithms on arrays of different sizes and types. The three array types are random order, ascending order, and descending order. The experiment will be conducted using a single run for each array size and type.

The independent variables in this data collection are the sorting algorithm used (Quicksort or Insertion Sort) and the order of the arrays. The dependent variable is the runtime of the sorting algorithms for each array type and size.

The runtime of the sorting algorithms will be measured in seconds using the chrono library in C++. For each array size and type, the runtime will be recorded once, and the data will be immediately collected and logged.

To ensure accuracy, the stack size for the program will be set to a sufficient value before the experiment starts, preventing potential stack overflow issues during the sorting process.

The experiment will cover a range of array sizes from 10,000 to 500,000, with increments of 10,000. For each array size, both Quicksort and Insertion Sort will be tested on arrays of random, ascending, and descending orders.

During the data collection process, steps will be taken to minimize any external factors that could influence the data. This includes ensuring that no other resource-intensive applications are running concurrently.

The data collected will be analyzed and used to compare the runtime of Quicksort and Insertion Sort for each array type and size. The average runtime for each sorting algorithm will be calculated based on the single run for each array type and size. The data will be presented in tabular and graphical formats to aid in clear visualization and interpretation.

Due to the computational complexity of sorting large arrays and the time-consuming nature of each run, conducting multiple runs for each array size and type was not feasible within the given time constraints. As a result, the experiment will be conducted using a single run for each array size and type to collect data on the runtime of the sorting algorithms.

While conducting multiple runs would enhance the statistical significance of the results and provide more robust conclusions, the single-run approach will still yield valuable insights into the performance of the sorting algorithms for different array types and sizes.

Data

Test Number	Array Size	Insertion Sort (Random) (seconds)	Quick Sort (Random) (seconds)	Insertion Sort (Ascending) (seconds)	Quick Sort (Ascending) (seconds)	Insertion Sort (Descending) (seconds)	Quick Sort (Descending) (seconds)
1	10000	0.170345	0.0003943	0.0000232	0.0005014	0.333299	0.0004204
2	20000	0.657714	0.0010189	0.000047	0.0010269	1.28612	0.0008322
3	30000	1.46265	0.0014248	0.0000681	0.0014713	2.9083	0.0013484
4	40000	2.58812	0.0018586	0.00001248	0.0017405	5.1723	0.0020792
5	50000	4.09217	0.0027923	0.00001148	0.0025925	8.49202	0.0025524

6	60000	5.89177	0.0033984	0.00001438	0.0038668	11.8407	0.0034019
7	70000	8.06002	0.0038902	0.00002001	0.0034003	17.0362	0.0036273
8	80000	10.5129	0.0045558	0.00003071	0.0038352	20.9508	0.0046365
9	90000	13.347	0.0042764	0.00002086	0.0044754	26.5419	0.0047877
10	100000	16.1842	0.0048604	0.00002422	0.0052811	34.1065	0.0048226
11	110000	20.3528	0.0066157	0.00002602	0.0066058	39.3418	0.0064434
12	120000	23.4197	0.0071076	0.00003253	0.0060244	49.4613	0.0059716
13	130000	27.7608	0.0066997	0.00003331	0.0066385	54.2484	0.006592
14	140000	31.4216	0.0068679	0.00003242	0.007291	62.9407	0.0076484
15	150000	36.101	0.0075599	0.00004701	0.0076109	72.1653	0.0074677
16	160000	41.1282	0.0086976	0.00005627	0.0080425	83.1646	0.0091256
17	170000	46.3671	0.0084915	0.00004054	0.0098133	92.7552	0.0087103
18	180000	53.6754	0.0092666	0.00004265	0.01039	104.772	0.0092729
19	190000	58.1344	0.0104261	0.00004797	0.0099438	115.787	0.0098624

20	200000	64.3501	0.010381	0.00004916	0.010499	130.906	0.0107906
21	210000	72.7598	0.0119434	0.00004915	0.0110835	144.757	0.0107984
22	220000	79.0573	0.0115534	0.00005055	0.0116659	155.597	0.0118929
23	230000	87.1756	0.0122018	0.00008666	0.0121978	173.08	0.0148674
24	240000	94.7337	0.0143982	0.00007414	0.0136916	189.391	0.0124569
25	250000	102.075	0.0140845	0.00005726	0.013616	204.338	0.0135679
26	260000	109.911	0.0154383	0.00006179	0.0142244	218.807	0.0138194
27	270000	117.235	0.0162019	0.00008666	0.015808	237.39	0.0154586
28	280000	126.883	0.0150224	0.00008357	0.0160741	253.228	0.0151922
29	290000	136.525	0.0158858	0.00006783	0.015824	274.009	0.0159268
30	300000	148.122	0.0164468	0.0000687	0.0174324	296.03	0.0164626
31	310000	157.914	0.0172819	0.00007145	0.0181395	312.072	0.0174571
32	320000	167.019	0.0188366	0.00015627	0.0179957	345.664	0.0181343
33	330000	182.515	0.0191908	0.00010413	0.0188561	356.926	0.0197525

34	340000	190.424	0.0232497	0.00005726	0.0224637	383.314	0.0216273
35	350000	200.649	0.0235785	0.00012795	0.0228544	405.005	0.0214596
36	360000	213.64	0.0218742	0.00008414	0.0209578	427.018	0.0208891
37	370000	224.317	0.0214503	0.00008753	0.0227119	449.129	0.0210252
38	380000	238.876	0.0224606	0.00009292	0.02256	464.747	0.0224357
39	390000	243.957	0.0243247	0.00010413	0.0226883	495.437	0.0224771
40	400000	259.547	0.0235809	0.00011297	0.0245002	527.161	0.0237517
41	410000	277.932	0.0246646	0.00010765	0.0245447	551.056	0.0245436
42	420000	288.069	0.0249161	0.00010271	0.0254105	576.761	0.0250622
43	430000	296.569	0.0272513	0.00010413	0.025909	596.487	0.0260855
44	440000	317.319	0.0275631	0.0001121	0.0274292	625.442	0.0265567
45	450000	324.314	0.0288985	0.00010425	0.0302539	650.938	0.02808
46	460000	340.038	0.0279227	0.0001221	0.0284978	679.084	0.0274748
47	470000	354.778	0.0291695	0.0001073	0.0285354	1357.36	0.0290965

48	480000	375.151	0.0306932	0.00012064	0.0304534	748.818	0.0290457
49	490000	390.382	0.0316178	0.0001156	0.0306875	826.374	0.0300637
50	500000	401.083	0.0306525	0.00012703	0.0323862	803.106	0.0308044

Table 1

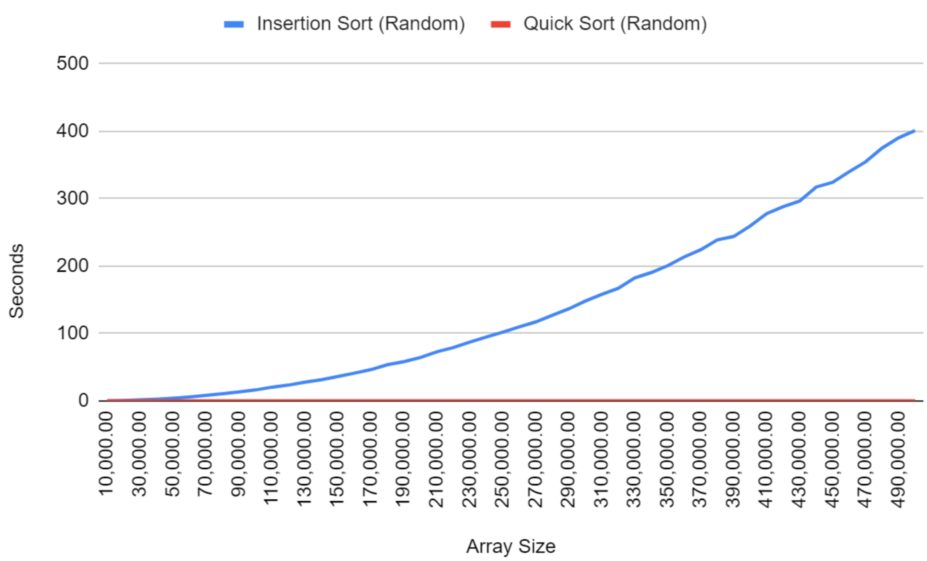

Chart 1

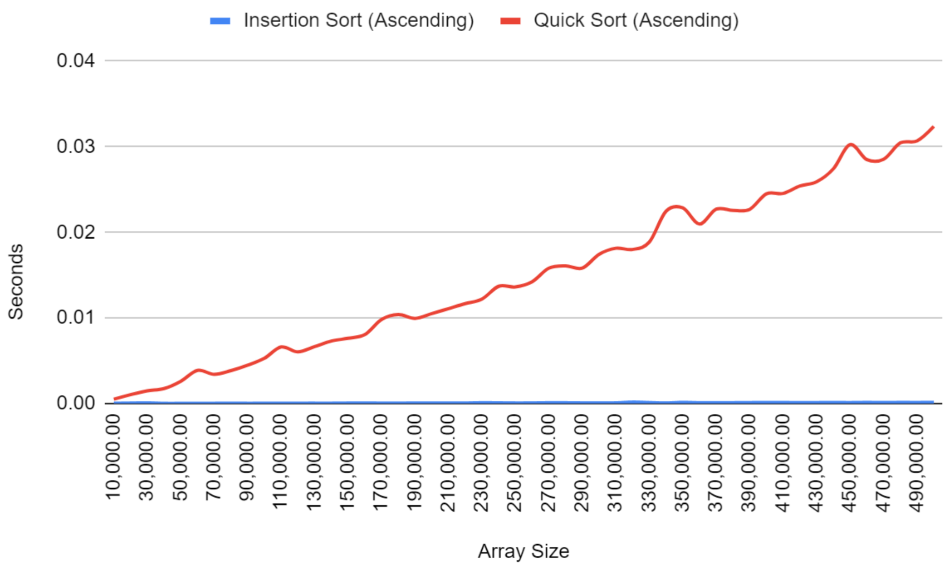

Chart 2

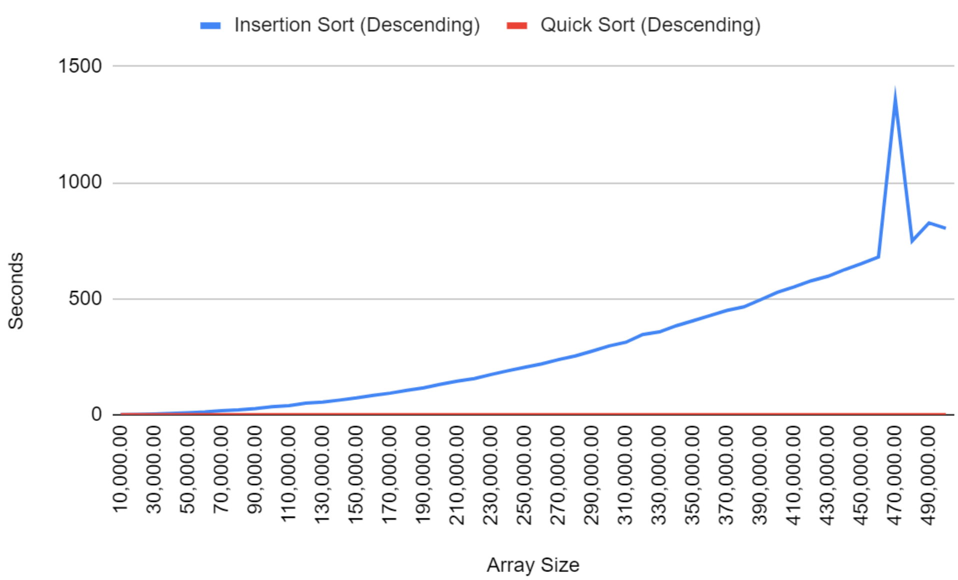

Chart 3

	Insertion Sort (Random)	Quick Sort (Random)	Insertion Sort (Ascending)	Quick Sort (Ascending)	Insertion Sort (Descending)	Quick Sort (Descending)
Average (seconds)	139.6929678	0.015058754	0.0000697778	0.01501007	293.2547288	0.014733186
Minimum (seconds)	0.170345	0.0003943	0.00001148	0.0005014	0.333299	0.0004204
Maximum	401.083	0.0316178	0.00015627	0.0323862	1357.36	0.0308044

(seconds)						

Table 2

Analysis and Interpretation

Interpretation of Charts and Tables

For random arrays: Quicksort consistently outperforms Insertion Sort, as expected. Quicksort's average time complexity of O(n log n) is more efficient than Insertion Sort's O(n^2) for random data.

For ascending arrays: As hypothesized, Insertion Sort is faster than Quicksort. Insertion Sort's best-case time complexity of O(n) performs better than Quicksort's O(n log n) when the data is already sorted in ascending order.

For descending arrays: As per the hypothesis, Quicksort is indeed faster than Insertion Sort. Contrary to random and ascending arrays, Quicksort's time complexity performs relatively better than Insertion Sort's O(n^2) in the case of descending data.

For all array types, the runtime of both algorithms increases as the array size grows. This aligns with the expected behavior since larger arrays require more operations to sort.
The growth rate of the runtime is more pronounced for Insertion Sort, especially for larger array sizes. This is due to its quadratic time complexity, which leads to slower performance as the data size increases.

Quicksort consistently exhibits lower minimum and maximum runtimes for all array types compared to Insertion Sort. This reinforces the finding that Quicksort generally performs better on average.The range between the minimum and maximum runtimes is more significant for Insertion Sort, indicating higher variability in its performance compared to Quicksort.

Hypothesis Evaluation

Based on the data and analysis, let's evaluate the initial hypothesis:

Quicksort's average time will be less than Insertion Sort on random arrays: Supported. The data shows that Quicksort significantly outperforms Insertion Sort for random arrays of various sizes.

Insertion Sort's average time will be less than Quicksort on arrays already in ascending order: Supported. The experiment confirms that Insertion Sort is faster than Quicksort for arrays sorted in ascending order.

Quicksort's average time will be less than Insertion Sort on arrays in descending order: Supported. The experiment demonstrates that Quicksort is faster than Insertion Sort for descending arrays.

Comparison of Algorithms

Considering all array types and sizes, Quicksort demonstrates better performance than Insertion Sort on average for random, ascending, and descending arrays.

Limitations and Possible Improvements

The experiment has a few limitations worth mentioning:

Single run: Due to time constraints, the experiment used a single run for each array size and type. Conducting multiple runs would enhance the statistical significance and reduce the impact of potential outliers.

Hardware and system variations: The experiment was conducted on a specific computer system, which may have affected the results. Performing the experiment on multiple systems would provide more robust insights.

Discussion of Insights

The experiment reaffirms the theoretical understanding of sorting algorithms' time complexities. Quicksort's average-case $O(n \log n)$ performance outperforms Insertion Sort's average-case $O(n^2)$ for random and ascending order data. Additionally, Quicksort's time complexity performs better than Insertion Sort for descending order data as well.

Significance and Implications

The experiment provides practical insights into the performance of Quicksort and Insertion Sort on different types and sizes of arrays. The findings can guide algorithm selection based on the characteristics of the data to be sorted.

For real-world applications, developers can utilize the knowledge gained from this experiment to optimize sorting operations based on the specific array types they encounter.

Conclusion

This experiment compared the runtime performance of Quicksort and Insertion Sort on arrays of different sizes and types: random order, ascending order, and descending order. The hypotheses were evaluated, and the results confirmed the following:

Quicksort's average time was faster than Insertion Sort for random arrays due to its O(n log n) average-case time complexity.

Insertion Sort outperformed Quicksort for arrays already sorted in ascending order, benefiting from its O(n) best-case time complexity.

Quicksort also outperformed Insertion Sort for arrays in descending order, demonstrating its efficiency compared to Insertion Sort's O(n^2) worst-case time complexity.

Overall, Quicksort displayed better performance than Insertion Sort on average for all array types and sizes, making it a favorable choice for general sorting tasks.

While the experiment had limitations, the findings provide valuable insights for developers and engineers when selecting sorting algorithms based on specific scenarios. This experiment contributes to a better understanding of fundamental sorting algorithms and their practical implications, facilitating their efficient application in real-world scenarios.

Appendix

Test3.cpp

```
#include "sorting.h"
#include "print_array.h"
#include <chrono>
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
// Function to print the run time for a specific sorting algorithm
auto printRuntime = [](const string &sortType, double runtime) {
cout << "Time for " << sortType << ": " << runtime << " seconds" << endl;
};
```

```cpp
const int numTests = 50;
const int maxArraySize = 500000;

for (int testNum = 1; testNum <= numTests; ++testNum) {
int arraySize = maxArraySize * testNum / numTests;

cout << "**** Test " << testNum << " ****" << endl;
cout << "Array size: " << arraySize << endl;

// Create random arrays with random integers
int *randomArray1= new int[arraySize];
int *randomArray2 = new int [arraySize];
for (int i = 0; i < arraySize; i++) {
randomArray1[i] = rand() % arraySize;
}
// randomArray2 for quicksort
copy(randomArray1, randomArray1 + arraySize,randomArray2 );

// Measure the time for insertion sort on the random array
auto t1 = chrono::high_resolution_clock::now();
insertionSort(randomArray1, 0, arraySize - 1, false);
auto t2 = chrono::high_resolution_clock::now();
chrono::duration<double> runtime =
chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("insertion sort (random)", runtime.count());

// Measure the time for quick sort on the random array
t1 = chrono::high_resolution_clock::now();
quickSort(randomArray2, 0, arraySize - 1, false);
t2 = chrono::high_resolution_clock::now();
runtime = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("quick sort (random)", runtime.count());

// Measure the time for insertion sort on the ascending array (randarr1)
t1 = chrono::high_resolution_clock::now();
insertionSort(randomArray1, 0, arraySize - 1, false);
t2 = chrono::high_resolution_clock::now();
runtime = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("insertion sort (ascending)", runtime.count());

// Measure the time for quick sort on the ascending array (randarr2)
t1 = chrono::high_resolution_clock::now();
quickSort(randomArray2, 0, arraySize - 1, false);
t2 = chrono::high_resolution_clock::now();
runtime = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("quick sort (ascending)", runtime.count());

// Reverse random arrays (ascending) to descending
reverse(randomArray1, randomArray1 + arraySize);
```

```
reverse(randomArray2, randomArray2 + arraySize);

// Measure the time for insertion sort on the descending array
t1 = chrono::high_resolution_clock::now();
insertionSort(randomArray1, 0, arraySize - 1, false);
t2 = chrono::high_resolution_clock::now();
runtime = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("insertion sort (descending)", runtime.count());

// Measure the time for quick sort on the descending array
t1 = chrono::high_resolution_clock::now();
quickSort(randomArray2, 0, arraySize - 1, false);
t2 = chrono::high_resolution_clock::now();
runtime = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
printRuntime("quick sort (descending)", runtime.count());

// Delete arrays to free up space
delete[] randomArray1
delete[] randomArray2
}
}
```

Sorting_basic.cpp

```
/**
* Implemention of selected sorting algorithms
* @file sorting.cpp
*/

#include "sorting.h"
#include <bits/stdc++.h>

using namespace std;

/**
* Implement the insertionSort algorithm correctly
*/
void insertionSort(int array[], int lowindex, int highindex, bool reversed) {
int key,j;
if (!reversed)
{
for (int i = lowindex; i <= highindex; i++) {
key = array[i];
j = i - 1; //j is item before key
while (key < array[j] && j >= lowindex) {
swap(array[j], array[j+1]); //start with key=j+1, and shift key to the left
until it key>arr[j]
```

```
j--;
}
}
}else
{
for (int i = lowindex; i <= highindex; i++) {
key = array[i];
j = i - 1;
while (key > array[j] && j >= lowindex) {
swap(array[j], array[j+1]);
j--;
}
}
}
}

/**
* @brief Implementation of the partition function used by quick sort
*
*/
int partition(int array[], int lowindex, int highindex, bool reversed) {
// TODO: Add your code here
int mid = lowindex + (highindex - lowindex) / 2;

if ((array[lowindex] > array[mid] && array[lowindex] < array[highindex]) ||
(array[lowindex] < array[mid] && array[lowindex] > array[highindex])) {
swap(array[lowindex], array[mid]);
} else if ((array[mid] > array[lowindex] && array[mid] < array[highindex]) ||
(array[mid] < array[lowindex] && array[mid] > array[highindex])) {
swap(array[mid], array[highindex]);
}

int pivot = array[highindex];
int lo = lowindex; //lo pt to lowindex
int hi = highindex-1; //hi pt to highindex-1(exclude pivot)

if(!reversed) {
while (lo <= hi) //
{
while (array[lo] <= pivot && lo<=hi) //incr lo until element at lo <=pivot
lo++;
while (array[hi] >= pivot && lo<=hi) //decr hi until arr[hi] >pivot
hi--;
if (lo < hi)
swap(array[lo], array[hi]);

}
}else{
while (lo <= hi) //
```

```
{
while (lo <= hi && array[lo] >= pivot)
lo++;
while (lo <= hi && array[hi] < pivot)
hi--;
if (lo < hi)
swap(array[lo], array[hi]);

}

}
swap(array[lo], array[highindex]);
return lo;//returb pivot index
}

/**
* Implement the quickSort algorithm correctly
*/
void quickSort(int array[], int lowindex, int highindex, bool reversed) {
// TODO: Add your code here
if (lowindex<highindex){
int pivotIndex = partition(array, lowindex,highindex,reversed);
//recursive call
quickSort(array,lowindex,pivotIndex-1,reversed);
quickSort(array, pivotIndex+1, highindex, reversed);
}

}
```