Douglas Calmes

Project Report

1. Expected/theoretical performance for each algorithm
   1. Insertion Sort
      1. Insertion sort is an in-place sorting algorithm, but it has an average running time of Ɵ(n^2).

Insertion sort

For j = 2 to a.length c1 n

key = a[j] c2 n-1

I = j – 1 c3 n-1

While i>0 and a[i]>key c4 tJ

a[i+1] = a[i] c5 tJ-1

i = I -1 c6 tJ-1

a[i+1] = key c7 n-1

* 1. Quick Sort
     1. Quick sort is an in-place sorting algorithm, but it has an average running time of Ɵ(n log n).
     2. Quick sort is also a divide and conquer algorithm

Quicksort(A,p,r)

If p < r c1 1

q=partition(a,p,r) an+b 1

quicksort(a,p,q-1) T(q-p) 1

quicksort(a,q+1,r) T(r-q) 1

partition(a,p,r)

x = A[r] c1 1

i = p-1 c2 1

for j=p to r-1 c3 n

if a[j] <= x c4 n-1

i = i+1 c5 n-1

exchange a[i] with a[r] c6 n-1

exchange a[i+1] with a[r] c7 1

return i+1 c8 1

T(n) = an+b

* 1. Heap Sort
     1. Heap Sort is an in-place sorting algorithm, but it has an average running time of Ɵ(n log n).

Heapsort(a)

Build-max-heap(a) nlgn 1

For I = a.length downto 2 c2 n

Exchange a[l] with a[i] c3 n-1

a.heapsize = a.heapsize -1 c4 n-1

max-heapify(a,l) lg n n-1

1. A table summarizing the observed results of each algorithm

|  |  |  |
| --- | --- | --- |
| Insertion Sort | | |
| Size of Data | Comparisons | Time |
| 10,000 Random | 25053546 | 671234 |
| 10,000 Sorted | 10000 | 358 |
| 10,000 Identical | 10000 | 359 |
| 10,000 Reverse | 50004942 | 1345135 |
| 20,000 Random | 99247616 | 2672855 |
| 20,000 Sorted | 20000 | 719 |
| 20,000 Identical | 20000 | 727 |
| 20,000 Reverse | 200009799 | 5426146 |
| 40,000 Random | 400184216 | 10955821 |
| 40,000 Sorted | 40000 | 1441 |
| 40,000 Identical | 40000 | 1446 |
| 40,000 Reverse | 800019189 | 22004262 |
| 80,000 Random | 1600809475 | 44011554 |
| 80,000 Sorted | 80000 | 3001 |
| 80,000 Identical | 80000 | 3008 |
| 80,000 Reverse | -1094930538 | 87823449 |

To summarize, insertion sort is very fast when it deals with data that is already sorted or is identical. This is because it only needs to compare each element one time, realizing it is sorted already it does not need to do any changes to the data. It is when the data is random or in reverse when it takes a long time to calculate. This means the data is consistently swapping, more so when dealing with completely reverse data when it needs to compare maximum times.

|  |  |  |
| --- | --- | --- |
| Quick Sort | | |
| Size of Data | Comparisons | Time |
| 10,000 Random | 2293777 | 6187 |
| 10,000 Sorted | 49995000 | 2293777 |
| 10,000 Identical | 49995000 | 2300713 |
| 10,000 Reverse | 49760234 | 1744855 |
| 20,000 Random | 349846 | 13466 |
| 20,000 Sorted | 199990000 | 9157142 |
| 20,000 Identical | 199990000 | 9175437 |
| 20,000 Reverse | 198023012 | 6896218 |
| 40,000 Random | 773847 | 30149 |
| 40,000 Sorted | 799980000 | 36999232 |
| 40,000 Identical | 799980000 | 37037182 |
| 40,000 Reverse | 784561504 | 27501008 |
| 80,000 Random | 1637365 | 63467 |
| 80,000 Sorted | -1095007296 | 148909412 |
| 80,000 Identical | -1095007296 | 149015782 |
| 80,000 Reverse | -1219293157 | 108606903 |

To summarize, it is clear to see that the data is sorted the fastest in quick sort when the data is randomized. This is because of how quick sort works, randomized data has a better chance to choose better pivots when dealing with comparisons. When the data is sorted, identical, or reverse there is a better chance that the pivot is a bad pivot, leading to a ton of comparisons. I purposely left the comparison counter as an long integer to show that there are so many comparisons that it can not even be calculated properly. You would need to use a larger value type in order to actually see how many comparisons are being made.

|  |  |  |
| --- | --- | --- |
| Heap Sort | | |
| Size of Data | Comparisons | Time |
| 10,000 Random | 387519 | 16903 |
| 10,000 Sorted | 410013 | 17742 |
| 10,000 Identical | 44997 | 1233 |
| 10,000 Reverse | 365091 | 16048 |
| 20,000 Random | 835104 | 37724 |
| 20,000 Sorted | 879765 | 39486 |
| 20,000 Identical | 89997 | 2530 |
| 20,000 Reverse | 793026 | 36112 |
| 40,000 Random | 1790604 | 83355 |
| 40,000 Sorted | 1876452 | 86276 |
| 40,000 Identical | 179997 | 5016 |
| 40,000 Reverse | 1703001 | 79849 |
| 80,000 Random | 3820767 | 182487 |
| 80,000 Sorted | 3994506 | 188256 |
| 80,000 Identical | 359997 | 10224 |
| 80,000 Reverse | 3642150 | 175035 |

To summarize, it is clear to see that heapsort is probably the most efficient algorithm used in this assignment. No matter how the data is stored, it runs very efficiently and almost identical in the grand scheme of things, aside from being stored with identical data. With identical data it runs a faster than the other datasets, but not by a extremely large margin.

1. Analysis and Discussion of results
   1. Are the theoretical and actual performance results consistent?
      1. The results are fairly consistent between the theoretical and actual performance.
   2. Any anomalies?
      1. Quicksort has a hard time efficiently sorting data that is stored reverse, identical, and sorted. It caused my comparisons to miscalculate when using long integer value.
   3. What surprised me?
      1. I was really surprised to see how fast insertion sort runs on data that is already sorted or is identical. I understood that it would run very fast and only compare once per element before noticing it was sorted, but the speed in which it did so was mind blowing. It was very cool to see this!
      2. I was also very surprised by how poor of a job quick sort was at running on any set of data that was not random. While knowing it would take a longer amount of time due to the pivot, it was shocking to actually see how poor it manages to produce a sorted set of data.
      3. I was not really too surprised to see how well heap sort was at being efficient, but there was one thing that exceeded my expectations. I knew that it would run faster on identical data, but it ran a little bit faster that I expected.