**Question 1:**

**Full statistics for input size 16:** Prepare a table (for each sort) showing the runtime, number of comparisons, and number of movements for an array of size 16 and with separate results for each choice of how the elements are initially ordered. The full answer for this question is just one table per sorting implementation, no explanation required.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sort Type** | **Greek** | **Data Type** | **Comparisons** | **Movements** | **Runtime** |
| Quick sort | Alpha | In order | 150 | 15 | 0 |
|  |  | Reverse order | 158 | 39 | 0 |
|  |  | Almost order | 141 | 18 | 0 |
|  |  | Random order | 108 | 78 | 0 |
| Selection sort | Beta | In order | 120 | 0 | 0 |
|  |  | Reverse order | 120 | 24 | 0 |
|  |  | Almost order | 120 | 3 | 0 |
|  |  | Random order | 120 | 45 | 0 |
| Insertion sort | Gamma | In order | 15 | 30 | 0 |
|  |  | Reverse order | 120 | 150 | 0 |
|  |  | Almost order | 28 | 43 | 0 |
|  |  | Random order | 68 | 85 | 0 |
| Shell sort | Delta | In order | 30 | 60 | 0 |
|  |  | Reverse order | 45 | 82 | 0 |
|  |  | Almost order | 42 | 73 | 0 |
|  |  | Random order | 61 | 97 | 0 |
| Merge sort | Epsilon | In order | 32 | 128 | 0 |
|  |  | Reverse order | 32 | 128 | 0 |
|  |  | Almost order | 37 | 128 | 0 |
|  |  | Random order | 45 | 128 | 0 |
| Heap sort | Zeta | In order | 85 | 136 | 0 |
|  |  | Reverse order | 72 | 120 | 0 |
|  |  | Almost order | 84 | 135 | 0 |
|  |  | Random order | 81 | 133 | 0 |

**Question 2:**

**Runtimes for at least four nontrivial input sizes:** Prepare another table (for each sort) showing (just) the runtime of the sort for at least four different non-trivial array sizes. A non-trivial array size is one where the runtime is more than just a few milliseconds. When you can, increase the input size until the runtime takes at least one second. If input sizes are chosen well, they will allow you to see enough variation in the statistics to give you strong evidence to support your answers to questions (3) and (4) below. You do not need to use the same input sizes for every algorithm, since this might not produce the most useful data. The full answer for this question is just one table per sorting implementation, no explanation required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sort Type** | **Greek** | **Input Size** | **Data Type** | **Run time** |
| Quick Sort | Alpha | 2222 | In order | 2 |
|  |  |  | Reverse order | 2 |
|  |  |  | Almost order | 0 |
|  |  |  | Random | 0 |
|  |  | 3333 | In order | 6 |
|  |  |  | Reverse order | 4 |
|  |  |  | Almost order | 22 |
|  |  |  | Random | 35 |
|  |  | 6666 | In order | 23 |
|  |  |  | Reverse order | 19 |
|  |  |  | Almost order | 0 |
|  |  |  | Random | 0 |
|  |  | 11111 | In order | 66 |
|  |  |  | Reverse order | 49 |
|  |  |  | Almost order | 1 |
|  |  |  | Random | 1 |
| Selection Sort | Beta | 2333 | In order | 2 |
|  |  |  | Reverse order | 3 |
|  |  |  | Almost order | 3 |
|  |  |  | Random | 3 |
|  |  | 6666 | In order | 18 |
|  |  |  | Reverse order | 22 |
|  |  |  | Almost order | 25 |
|  |  |  | Random | 25 |
|  |  | 23333 | In order | 225 |
|  |  |  | Reverse order | 274 |
|  |  |  | Almost order | 314 |
|  |  |  | Random | 312 |
|  |  | 66666 | In order | 1819 |
|  |  |  | Reverse order | 2178 |
|  |  |  | Almost order | 2520 |
|  |  |  | Random | 2507 |
| Insertion Sort | Gamma | 3333 | In order | 0 |
|  |  |  | Reverse order | 5 |
|  |  |  | Almost order | 0 |
|  |  |  | Random | 3 |
|  |  | 6666 | In order | 0 |
|  |  |  | Reverse order | 20 |
|  |  |  | Almost order | 1 |
|  |  |  | Random | 10 |
|  |  | 11111 | In order | 0 |
|  |  |  | Reverse order | 56 |
|  |  |  | Almost order | 4 |
|  |  |  | Random | 28 |
|  |  | 111111 | In order | 0 |
|  |  |  | Reverse order | 5611 |
|  |  |  | Almost order | 355 |
|  |  |  | Random | 2839 |
| Shell Sort | Delta | 1111 | In order | 0 |
|  |  |  | Reverse order | 2 |
|  |  |  | Almost order | 2 |
|  |  |  | Random | 2 |
|  |  | 6666 | In order | 0 |
|  |  |  | Reverse order | 77 |
|  |  |  | Almost order | 75 |
|  |  |  | Random | 87 |
|  |  | 11111 | In order | 0 |
|  |  |  | Reverse order | 209 |
|  |  |  | Almost order | 204 |
|  |  |  | Random | 271 |
|  |  | 111111 | In order | 0 |
|  |  |  | Reverse order | 12620 |
|  |  |  | Almost order | 21083 |
|  |  |  | Random | 30121 |
| Merge Sort | Epsilon | 23333 | In order | 2 |
|  |  |  | Reverse order | 2 |
|  |  |  | Almost order | 2 |
|  |  |  | Random | 3 |
|  |  | 233333 | In order | 18 |
|  |  |  | Reverse order | 18 |
|  |  |  | Almost order | 22 |
|  |  |  | Random | 35 |
|  |  | 666666 | In order | 49 |
|  |  |  | Reverse order | 52 |
|  |  |  | Almost order | 67 |
|  |  |  | Random | 109 |
|  |  | 1048576 | In order | 96 |
|  |  |  | Reverse order | 101 |
|  |  |  | Almost order | 109 |
|  |  |  | Random | 178 |
| Heap Sort | Zeta | 11111 | In order | 1 |
|  |  |  | Reverse order | 1 |
|  |  |  | Almost order | 1 |
|  |  |  | Random | 1 |
|  |  | 66666 | In order | 4 |
|  |  |  | Reverse order | 4 |
|  |  |  | Almost order | 5 |
|  |  |  | Random | 6 |
|  |  | 666666 | In order | 49 |
|  |  |  | Reverse order | 51 |
|  |  |  | Almost order | 56 |
|  |  |  | Random | 85 |
|  |  | 1048576 | In order | 80 |
|  |  |  | Reverse order | 83 |
|  |  |  | Almost order | 90 |
|  |  |  | Random | 139 |

**Question 3:**

**Worst-case asymptotic run-time:** Your estimation of the worst case asymptotic (“big-O”) runtime for each sorting implementation. You must gather appropriate data that clearly shows the relevant patterns of the algorithm's runtime. This likely means measuring different algorithms at different input sizes, since some algorithms are much faster than others. No explanation is required here. However, if the data you collected in question (2) is in conflict or in no way suggesting the runtimes you give for question (3), then you should consider collecting more runtime data. We do not expect curves that show a perfect n log n or anything like that though, just reasonable data.

|  |  |  |
| --- | --- | --- |
| **Sort type** | **Greek** | **Worst-case runtime** |
| Quick Sort | Alpha | O(n^2) |
| Selection Sort | Beta | O(n^2) |
| Insertion Sort | Gamma | O(n^2) |
| Shell Sort | Delta | O(n^2) |
| Merge Sort | Epsilon | O(nlogn) |
| Heap Sort | Zeta | O(nlogn) |

**Question 4:**

**Identifying the algorithms:** State which sorting implementation uses which algorithm, along with your reasoning and supporting evidence. You should base your reasoning on the following:

* growth rate of the algorithm's runtime as input size grows
* speed or number of comparisons and data movements of the algorithm compared to the other algorithms
* changes in behavior of the algorithm, if any, for different input orderings

If errors caused by the algorithm caused you to deduce that algorithm, explain why you suspect that algorithm would cause the error.

For question (4), if all your claims are based only on relative speeds of the algorithms (“sort Beta was the fastest, so therefore it is X-Sort”), you will not receive full credit. You will also not receive full credit if you use process-of-elimination to state which algorithm is being used for the last one or two sorts.

A good answer to question (4) is a strong argument. Try for two or, if possible, three strong points that support each algorithm claim. A reason can be based on any of the criteria that are listed above. But if you are using the growth rate as one of your arguments, then include an explanation of why you think the growth rate is what you claim it is. For example, why do you think a growth rate is quadratic instead of n log n? But there are other things you can use beyond trying to match your runtime data to a particular curve. Look at how the number of comparisons and movements vary both for different types of inputs (InOrder, ReverseOrder, etc.) and across different algorithms. You can also look at how speed varies between different types of inputs. One reason we asked that you collect data for N=16 is that this is a small enough problem size that you may be able to walk through the algorithm and come up with a fairly close idea of how many comparisons or data movements would be required in some cases, which you can then use as an argument.

Alpha->Selection Sort

1. The worst-case runtime is O(n^2),
2. For in order, random order, the runtime of these are close to the runtime of reverse order. That is the best-case runtime and average case runtime are all O(n^2). So it is selection sort.

Beta-> Bubble Sort

1. The worst-case runtime is O(n^2),
2. The best-case runtime is O(n), so it could be insertion sort or bubble sort. However, the runtime and comparisons of almost order are close to these of reverse order. So, it is bubble sort.

Gamma-> Heap Sort

1. The worst-case runtime is O(nlogn),
2. The comparisons and runtime of in order are pretty close to these of reverse order. This is because In heap sort, when building the heap using the binary tree, reverse order and in order would take almost the same steps.
3. So gamma could be heap sort or merge sort, but for in order of merge sort, the comparisons would be exactly N/2log(N) (Here N is 2^n). Gamma does not satisfy this, so it is heap sort.

Delta-> Merge Sort

1. The worst-case runtime is O(nlogn),
2. For in order of merge sort, the comparisons would be exactly N/2log(N) (Here N is 2^n). That is for input size N=16, the comparisons would be exactly 32. (For N=32, it is 80), so delta is merge sort.

Epsilon-> Quick Sort

1. The worst-case runtime is O(n^2),
2. The average case runtime is O(nlogn), this can be distinguished by comparing growth rate of random order as input size grows.
3. Stack over flow happens when it runs in order or reverse order at certain size, as when sorting in order or reverse order, quick sort would keep a lot of recursion calls in the stack. So not huge input size could generate stack overflow error.

Zeta->Insertion Sort

1. The worst case runtime is O(n^2),
2. And the best case runtime is O(n), and the runtime of almost is very small comparing to the runtime of random case.
3. Runtime of random order is almost half of runtime of reverse order. It is because the for each step in the outer loop of insertion sort, the chosen element will be swapped to the middle of sorted part on average, however, in reverse order, the element will be swapped to the left end.