Table 1 (on the next page) shows the CPU times, in seconds, of all programs submitted this semester for the second programming assignment, listed anonymously. This was the fifth semester that I used this specific assignment, although the virtual machine I was using changed after the first three semesters, so times of programs from the first three semesters are not directly comparable. I did not change the code of the baseline program or my program since they were created; I just recompiled and reran them when I switched to a new virtual machine.

The "T1" through "T4" columns show the times for sorting the four datasets. The "Sum" column shows the sum of the "T1" through "T4" times. The row labeled "Instructor" (colored green) shows the times of my program. The row labeled "Baseline" (colored blue) shows the times of a simple program that was created quickly using the sort member function that is part of the "list" class with a straight-forward comparison function. A maximum cutoff time equal to the baseline time plus one second was used as an upper bound for each column; these are shown in the row labeled "Upper Bound", colored in red. Any program that crashed, hung, produced incorrect output, or took longer than the upper bound time for any test case would be assigned this upper bound time. Those times are also shown in red, followed by an explanation (I = incorrect output; H = hung; S = slow; C = crash). If a program had fallen into multiple categories (e.g., if it took longer than the cutoff time and had incorrect output), only one code would be shown. This semester, no student had a program that fell into the "hung" or "incorrect output" categories. Although no student this semester beat my program's times overall, during the previous semesters that I assigned the same assignment, a total of three students in total have beaten my times. This semester, several students beat me on T3, and one beat me on both T3 and T4.

As indicated on the program handout, all programs were compiled using the g++ compiler, without any compiler optimizations, on an Ubuntu 20.04 virtual machine with 2 GB of allocated RAM. For students who sent a presubmission, and who did not send a further submission after that, I did not rerun the program, and instead I used the times from the presubmission. If students sent an updated version, even if the only change was adding comments, I recompiled and reran the final programs. Even if you don't change the code, the times will not typically be exactly the same, but they will be very similar, and any difference would be unlikely to affect the grade.

I spent one day, four semesters ago, developing my program, and I was reasonably happy with my times. When I developed my program, I developed my sorts using the same sample datasets that have been posted on the class website. Only after I decided I was finished with development did I rerun the scripts to randomly generate the final datasets (the official T1 through T4), and then I ran my sorts on those only once. (If there had been bugs in my program which showed up for the final datasets but not the sample datasets, I would have fixed them, but that did not turn out to be necessary.) I did not make any changes to my code since then. Even when I switched to a new virtual machine last semester, I only recompiled and reran my program to determine my updated times for the four datasets, without changing the code.

Later in this document, on the pages after the chart, I discuss the strategies that I used for my sorts (including attempts that I tried but did not ultimately use), as well as some additional ideas

that I would have liked to try if I had more time. I also discuss other strategies that were, or would likely be, fast for these datasets.

**Colin Hwang:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0.169988** |  | **2.47588** |  | **0.246932** |  | **0.576862** |  | **3.469662** | **92** |

*Table 1: Times and grades, not including late penalties, for all submissions of program 2 in DSA I during the spring 2021 semester (shown anonymously)*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Name** | **T1** |  | **T2** |  | **T3** |  | **T4** |  | **Sum** | **Score** |
| Instructor | 0.073225 |  | 0.815538 |  | 0.274263 |  | 0.229091 |  | 1.392117 |  |
|  | 0.108955 |  | 1.58347 |  | 0.247586 |  | 0.194528 |  | 2.134539 | 97 |
|  | 0.1085 |  | 1.1906 |  | 0.586039 |  | 0.487649 |  | 2.372788 | 96 |
|  | 0.113412 |  | 1.8055 |  | 0.220551 |  | 0.526909 |  | 2.666372 | 95 |
|  | 0.108174 |  | 1.54809 |  | 0.23373 |  | 0.960389 |  | 2.850383 | 94 |
|  | 0.111613 |  | 1.57258 |  | 0.22827 |  | 1.00987 |  | 2.922333 | 94 |
|  | 0.144758 |  | 1.6811 |  | 0.235329 |  | 1.02486 |  | 3.086047 | 93 |
|  | 0.169988 |  | 2.47588 |  | 0.246932 |  | 0.576862 |  | 3.469662 | 92 |
|  | 0.120204 |  | 1.3801 |  | 0.940769 |  | 1.23361 |  | 3.674683 | 92 |
|  | 0.081199 |  | 0.904797 |  | 0.778501 |  | 2.10247 |  | 3.866967 | 91 |
|  | 0.130492 |  | 1.66338 |  | 1.1183 |  | 1.07625 |  | 3.988422 | 91 |
|  | 0.185573 |  | 2.29128 |  | 1.63069 |  | 0.146856 |  | 4.254399 | 90 |
|  | 0.110278 |  | 1.60362 |  | 1.19017 |  | 1.37957 |  | 4.283638 | 90 |
|  | 0.128987 |  | 1.86727 |  | 1.295 |  | 1.03618 |  | 4.327437 | 89 |
|  | 0.209567 |  | 2.76249 |  | 0.410698 |  | 1.02365 |  | 4.406405 | 89 |
|  | 0.12416 |  | 1.78575 |  | 1.20613 |  | 1.43938 |  | 4.55542 | 88 |
|  | 0.238003 |  | 3.27918 |  | 0.468912 |  | 0.989824 |  | 4.975919 | 87 |
|  | 0.265141 |  | 3.41777 |  | 0.261748 |  | 1.05164 |  | 4.996299 | 87 |
|  | 0.253997 |  | 3.40296 |  | 0.364815 |  | 1.03149 |  | 5.053262 | 87 |
|  | 0.21541 |  | 3.00648 |  | 1.19492 |  | 0.991739 |  | 5.408549 | 85 |
|  | 0.229701 |  | 3.07009 |  | 1.44583 |  | 0.987284 |  | 5.732905 | 83 |
|  | 0.168726 |  | 2.36462 |  | 1.18362 |  | 2.02916 |  | 5.746126 | 83 |
|  | 0.21748 |  | 2.98941 |  | 1.64803 |  | 0.96526 |  | 5.82018 | 82 |
|  | 0.243025 |  | 3.34446 |  | 1.20353 |  | 1.20353 |  | 5.994545 | 82 |
|  | 0.220709 |  | 2.9611 |  | 2.36347 |  | 1.26168 |  | 6.806959 | 80 |
|  | 0.215476 |  | 2.98788 |  | 1.43941 |  | 2.16783 |  | 6.810596 | 80 |
|  | 0.218428 |  | 3.07271 |  | 1.46525 |  | 2.19705 |  | 6.953438 | 80 |
|  | 0.261918 |  | 3.55205 |  | 1.17512 |  | 2.126 |  | 7.115088 | 79 |
|  | 0.272435 |  | 3.48401 |  | 1.84433 |  | 2.12647 |  | 7.727245 | 77 |
|  | 0.258474 |  | 3.11655 |  | 2.13869 |  | 2.57919 |  | 8.092904 | 76 |
|  | 0.214542 |  | 2.97993 |  | 1.675 |  | 4.07102 |  | 8.940492 | 74 |
|  | 0.309699 |  | 4.06319 |  | 1.2366 |  | 3.73641 |  | 9.345899 | 73 |
|  | 1.284695 | (C) | 5.01108 | (C) | 3.55296 | (C) | 1.38676 |  | 11.2355 | 71 |
|  | 0.431348 |  | 5.01108 | (S) | 0.932694 |  | 6.50557 |  | 12.88069 | 70 |
| Baseline | 0.284695 |  | 4.01108 |  | 2.55296 |  | 6.67671 |  | 13.52545 |  |
| Upper Bound | 1.284695 |  | 5.01108 |  | 3.55296 |  | 7.67671 |  | 17.52545 |  |
|  |  |  |  |  |  |  |  |  |  |  |

My first attempt to beat the baseline involved an indirect, modified quicksort, making use of the "sort" function available from the <algorithms> library. I created a global array of structures that was large enough to hold a structure corresponding to every item in the longest possible list. Each structure contained an unsigned integer (a 32-bit int) and a pointer to a Data object (the type in the list being sorted). I used the 32-bit field to store information that was usually enough to compare to items. I based the 32-bit field on the first four characters of the last name of the Data object, being careful to pad correctly for names that contained fewer than four characters. When there was a tie, only then would the pointer to Data be used to break the tie. This initial attempt led to sorts that were approximately twice as fast as the baseline sorts for T1 and T2. I'm sure the method could have been further improved for T1 and T2 without doing anything too tricky (I never intended to use this as my final solution for T3 or T4).

For T3, I then moved on to insertion sort, which was always my intention. Recall that insertion sort can be very fast for sequences that are close to sorted. I implemented two different insertion sorts. The first operated on the original linked list directly. Then I implemented an indirect insertion sort, using the same global array that had been used for the indirect quicksort. To my surprise, this wound up being a bit faster than the direct insertion sort, so I stuck with it. I would guess that all the fastest programs for T3 use some variation of insertion sort, either direct or indirect. (Bubble sort can also be fast for a close-to-sorted sequence, but empirically it would not typically be as fast as insertion sort.)

Although I optimized T4 last, I'll discuss it next. Since we only need to consider one field for T4 (the SSNs), and the number of characters is fixed and not very large, it was my expectation that radix sort would likely be the fastest strategy. I implemented three variations: a three-pass radix sort using three digits each (the dashes can be ignored); a two-pass radix sort using 18 bits each (four digits and half of the bits from the middle digit); and a two-pass radix sort where one pass used five digits and the other pass used four digits). For each implementation, I relied on twodimensional arrays, where one dimension represented the bin, and the other dimension was large enough to make it very unlikely to overflow. A bit to my surprise, the last implementation (two passes using five digits and four digits, respectively) turned out to be slightly faster than the implementation that used 18 bits per pass, so I used that in my final program. The speed of radix sort depends largely on the specifics of the implementation, including how efficiently you determine bins. When possible, I would use bitwise operators instead of more general arithmetic operators. Also, I do not copy back items into the original list in between passes. Rather, as I process one set of bins, resulting from one pass, I insert items directly into another set of bins used for the next pass. After the final pass, only then do items get returned to the original list. I would have liked to try other variations of radix sort, but I did not have time. I would have also liked to try a one-pass radix sort followed by an insertion sort, but I did not have time.

Moving back to T1 and T2, I decided midway through the day (since I limited myself to one day to develop my program) to consider utilizing the fact that we know the exact set of names being used for the first two fields of our Data objects. These lists of names were posted on the course website when the project was assigned, and I said you are allowed to make use of these if you want. At first, I really did not expect to be able to get much, if any, improvement from this, but after thinking about it a bit, I changed my mind.

Ultimately, I made use of the provided C++ unordered\_map class (I included the additional <unordered\_map> header file under the comment specifying that no code changes can occur above the comment). I created a hard-coded global unordered\_map (which is really just a hash table that can map items of one type to another) that maps names (which are strings) to integers. I actually created two of these unordered\_map objects, one for last names and one for first names. The mapping would convert a name to its sorted index of last names or first names. (For example, the mapping for last names would convert "ACOSTA" to 0, "ADAMS" to 1, etc.) I wrote an entirely separate program to produce the code to declare the unordered\_map, and then I copied that global declaration into my sorting program (below the comment that indicates no changes can occur above it). It would not have been too difficult to manually create that declaration, but I think writing code to do it probably took me less time.

My first attempt to use the hash tables (created using the unordered\_map class) to improve my times for T1 and T2 involved modifying the indirect quicksort discussed earlier. The basic algorithm was the same, but the integer value placed in the structure used to compare items was now based on the hash table lookups. I also expanded this field to be an unsigned long long (i.e., a 64-bit value), giving more flexibility, so I could incorporate information using both names and the SSN if I wanted to. The index of the last name occupied higher-order bits than the index of the first name. Only if the first and last names being compared were identical would the sort need to use the pointer to Data objects to consider the SSN. This led to a significant further improvement from my earlier quicksort. Next, I incorporated the SSN characters into the integer field as well (which was possible, since I had changed this field to be 64 bits), and that achieved a slight additional improvement.

After that, I tried what I would consider a one-pass radix sort including insertions into bins. I used a two-dimensional array of bins, 500x500, and the hash table lookups of last and first names were used to determine the proper bin. When items collided (mapped to the same bin), they would be inserted into the bin in the proper location (as with insertion sort). Then, items (Data pointers) were copied from the bins in the proper order back to the original list. Compared to the fastest indirect quicksort, this interestingly led to significant additional improvement for T2, but it did not lead to any improvement for T1, and was, in fact, slightly slower than the quicksort. (This does make sense, as the newest rendition of a radix sort combined with insertions into small bins is a linear sort, so it will be faster than quicksort when the sequence is long enough. Apparently, with my implementations, it is faster than quicksort for sequences with about one million items, but not for sequences with about one hundred thousand items.)

Next, I tried an implementation where only the hash table for last names was used. Based on last names, each Data pointer was added to one of 500 bins. Unlike the previous implementation, these were not ordered insertions, but rather, new items were added to the ends of bins. Then, I sequentially executed 500 modified quicksorts (again using the provided "sort" function), one for each bin. Then, items were copied back to the original list. This led to my final improvement for T1. For T2, it was almost identical in speed to the previous sort. Based on several tests with the sample file for T2 (not the final T2), it was unclear which sort was faster for that dataset. For my final program, rather than rely on two different sorts for T1 and T2, I used this version for both.

There were several additional things I wanted to try. As previously mentioned, for T4, I would have liked to try a one-pass radix sorts that did not fully sort the column, followed by an insertion sort. For T1 and T2, I wanted to try to come up with faster methods of mapping last names and first names to indexes. (It is worth noting that I did verify that the hash table lookups mapping names to indexes take up a considerable percentage of my overall sorting times for T1 and T2.) Although I am sure that the unordered\_map class is highly optimized, it is also generic. Given that I know I am applying this to short strings with only 500 different values, it may be possible to come up with a quick hash function that is very unlikely to lead to collisions on a large hash table with a fixed size. I also think it may be faster to not use a hash table at all, but rather to use hard-coded arrays of last names and first names, and to apply binary search. (Although that is generally logarithmic instead of average-case constant, since the size is fixed at 500, it would require at most 9 checks.) Alternatively, I could have also tried making 26 different arrays, one for each letter of the alphabet, and performing either a binary search, or even a linear search, in the proper array, based on the first letter of the name (because the length of each array would be short, averaging under 20 items per slot). I would have liked to try all of these things, and I am pretty confident that at least one of them could have shaved additional time off my sorts. However, I spent an entire day working on my program, and when I first originally assigned this assignment during a previous semester, I committed in advance to personally spending only one day on the program.

Ultimately, the sorting times depend on both the general choice of strategy used for each dataset and also the implementation details. This is always the case. Good choices of strategies with somewhat reasonable implementations could lead to a middle-A. To achieve a high A, that likely requires some additional clever tricks. To beat my times, you would likely also have to perform considerable experimentation to optimize your times.