Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

The chart on the next four pages shows CPU times (in seconds) and overall time scores (calculated as indicated on the project handout) of all programs submitted for the second programming assignment, listed anonymously, over the five semesters that this exact assignment was assigned. Grades (not including late penalties) are also indicated for submissions from the current semester. The T1 through T4 columns represent the times that programs achieved for the four test cases; the Score column shows the weighted overall score (the lower, the better). 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, but reasonably efficient, comparison function. A maximum cutoff time of twice the baseline time was used for each test case. Any program that crashed, hung, produced incorrect output, or took longer than the cutoff time for any test case was assigned the cutoff time. Those times are shown in red, followed by an explanation (I = incorrect output; H = hung; S = slow; C = crash). It is possible that some of the programs that appeared to hang ultimately would have completed, perhaps with correct output, but that does not matter. After about 20 seconds, I would stop the program, because whether or not it finished after that point would not affect the score or grade. It is also possible that some programs may have been both too slow and had incorrect output; only one of these two causes would be indicated in the chart, and I was not necessarily consistent in choosing which to indicate. Note that due to the maximum cutoffs, there was a highest possible overall score (and a lowest possible grade) for any submitted program that at least compiled. (In previous semesters, programs that did not compile and could not easily be made to compile were left out of the chart. This semester, all programs compiled, although a few required me to change "stof" or "stoi" to something else; I did not penalize such programs.)

This was the fifth year that I assigned this particular assignment, although I have been doing variations of sorting assignments for some time, so going into it, I had a good idea of what strategies were likely to be fast. Still, I did not have time to attempt several variations that I wanted to. I wrote my program four years ago in one day, and I did not make any modifications to my program since then.

Congratulations to Daniel Nakhimovich, who beat my program's weighted score for the program! I will describe his strategies, as well as my own, and others that are likely to be fast for these data sets. Since I was beaten this year, next year's class will get a different second assignment. It will be a similar assignment in principle, but the datasets will use entirely different data (the type of data, and the variations between the datasets, has not yet been decided).

All programs were tested under the same environment, as was indicated on the program handout. All programs were compiled with g++ without any compiler optimizations. The only flag I added for some submissions was -std=c++11, for those programs that required it (as specified in class, I did not consider it invalid to use C++11 syntax). The programs were compiled and run using Cygwin on my home desktop.

Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

	T1	T2	T3	T4	Score	Grade (no penalty)
	0.015	0.219	0.141	0.093	0.603	100
Instructor	0.031	0.296	0.156	0.062	0.824	
	0.032	0.391	0.125	0.109	0.945	
	0.031	0.328	0.157	0.203	0.998	99
	0.047	0.281	0.124	0.14	1.015	
	0.015	0.453	0.328	0.094	1.025	
	0.031	0.343	0.249	0.187	1.089	
	0.046	0.453	0.125	0.141	1.179	
	0.062	0.375	0.171	0.094	1.26	
	0.031	0.641	0.204	0.125	1.28	98
	0.047	0.453	0.156	0.203	1.282	
	0.047	0.546	0.171	0.124	1.311	
	0.048	0.453	0.157	0.266	1.356	97
	0.062	0.452	0.219	0.093	1.384	
	0.047	0.764	0.094	0.063	1.391	
	0.047	0.453	0.343	0.172	1.438	
	0.047	0.468	0.343	0.171	1.452	
	0.047	0.672	0.203	0.125	1.47	96
	0.047	0.64	0.296	0.093	1.499	
	0.047	0.655	0.296	0.078	1.499	
	0.047	0.577	0.343	0.156	1.546	
	0.047	0.546	0.5	0.078	1.594	95
	0.063	0.719	0.172	0.125	1.646	94
	0.063	0.656	0.297	0.11	1.693	93
	0.063	0.687	0.391	0.125	1.833	92
	0.047	0.672	0.626	0.109	1.877	92
	0.063	0.842	0.312	0.093	1.877	
	0.062	0.656	0.422	0.219	1.917	
	0.063	0.656	0.531	0.125	1.942	
	0.093	0.92	0.203	0.078	2.131	
	0.079	1.17	0.156	0.093	2.209	
	0.063	0.624	0.514	0.452	2.22	
	0.062	0.702	0.452	0.483	2.257	
	0.078	0.889	0.546	0.063	2.278	
	0.109	1.014	0.125	0.094	2.323	
	0.094	1.015	0.281	0.109	2.345	
	0.094	1.029	0.281	0.109	2.359	
	0.094	1.03	0.281	0.109	2.36	
	0.094	1.045	0.281	0.094	2.36	
	0.093	1.03	0.312	0.094	2.366	
	0.109	1.047	0.141	0.094	2.372	

Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

0.063	0.702	0.296		0.765		2.393	
0.093	1.045	0.312		0.109		2.396	
0.063	0.702	0.328		0.75		2.41	
0.094	1.079	0.312		0.094		2.425	
0.063	0.719	0.313		0.766		2.428	
0.063	0.703	0.329		0.797		2.459	
0.063	0.781	0.328		0.766		2.505	
0.016	0.282	0.11		1.966	(I)	2.518	
0.093	1.311	0.172		0.109		2.522	
0.11	1.03	0.327		0.094		2.551	
0.109	1.078	0.297		0.109		2.574	90
0.109	1.14	0.297		0.109		2.636	
0.094	1.311	0.297		0.093		2.641	
0.11	1.11	0.359		0.094		2.663	
0.078	0.905	0.89		0.094		2.669	
0.125	1.188	0.156		0.094		2.688	89
0.093	1.375	0.328		0.094		2.727	89
0.109	1.185	0.343		0.109		2.727	
0.078	0.796	0.796		0.389		2.761	
0.109	1.295	0.328		0.094		2.807	
0.078	0.782	0.344		0.906		2.812	
0.094	0.753	0.328		0.813		2.834	
0.109	1.328	0.328		0.094		2.84	
0.078	0.765	0.609		0.704		2.858	
0.078	0.904	0.827		0.359		2.87	
0.125	1.313	0.203		0.125		2.891	88
0.047	0.701	0.858		0.905		2.934	
0.032	0.312	0.391		1.966	(H)	2.989	88
0.015	0.516	0.359		1.966	(S)	2.991	
0.14	1.313	0.219		0.109		3.041	
0.078	0.858	0.733		0.671		3.042	
0.063	1.092	0.967		0.375		3.064	
0.109	1.778	0.125		0.093		3.086	
0.078	1.454	0.781		0.156		3.171	87
0.125	1.531	0.328		0.094		3.203	
0.063	0.827	0.733		1.014		3.204	
0.109	1.763	0.281		0.093		3.227	
0.109	1.763	0.281		0.109		3.243	
0.094	1.062	1.186		0.109		3.297	
0.031	0.359	2.528	(I)	0.188		3.385	
0.094	1.591	0.141		0.827		3.499	
0.032	0.436	2.528	(I)	0.218		3.502	
0.125	2.047	0.188		0.094		3.579	86

Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

	0.125		1.888		0.359		0.109		3.606	
	0.047		0.5		2.528	(C)	0.109		3.607	
	0.125		1.984		0.328		0.094		3.656	
	0.047		0.64		0.594		1.966	(H)	3.67	
	0.11		1.578		0.891		0.125		3.694	
	0.172		1.872		0.218		0.125		3.935	
	0.094		1.623		1.31		0.094		3.967	
	0.094		0.905		0.921		1.31		4.076	
	0.109		1.778		0.874		0.452		4.194	
	0.125		1.841		1.108		0.094		4.293	
	0.078		0.813		2.528	(I)	0.172		4.293	84
	0.109		1.578		1.344		0.36		4.372	
	0.125		1.576		1.404		0.171		4.401	
	0.078		1.062		2.528	(S)	0.109		4.479	83
	0.094		1.015		2.528	(I)	0.125		4.608	
	0.172		2.125		0.391		0.422		4.658	82
	0.109		1.031		2.528	(I)	0.141		4.79	
	0.093		1.232		2.528	(I)	0.11		4.8	
	0.312	(I)	1.11		0.484		0.125		4.839	81
	0.11		1.093		2.528	(I)	0.125		4.846	81
	0.109		1.809		1.451		0.655		5.005	
	0.11		1.328		2.528	(I)	0.109		5.065	81
	0.109		1.139		2.528	(H)	0.374		5.131	
	0.125		1.701		1.544		0.67		5.165	
	0.047		4.368	(C)	0.296		0.094		5.228	
	0.11		1.763		1.669		0.702		5.234	
	0.156		2.372		0.935		0.546		5.413	
	0.125		1.654		2.418		0.094		5.416	
	0.094		1.89		2.528	(I)	0.078		5.436	80
	0.14		1.95		1.685		0.406		5.441	
	0.109		1.778		2.528	(H)	0.11		5.506	
	0.125		1.872		1.841		0.717		5.68	
	0.125		1.797		2.528	(I)	0.125		5.7	79
	0.141		1.872		1.701		0.717		5.7	
	0.125		1.844		2.528	(H)	0.11		5.732	
	0.156		1.295		1.451		1.513		5.819	
	0.109		1.312		2.528	(H)	0.89		5.82	79
	0.11		1.607		1.56		1.591		5.858	
	0.11		4.368	(I)	0.312		0.109		5.889	
	0.141		1.938		2.528	(C)	0.11		5.986	79
Baseline	0.156		2.184		1.264		0.983		5.991	
	0.125		4.368	(H)	0.312		0.109		6.039	
	0.125		1.747		2.528	(I)	0.64		6.165	

Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

0.156		1.467		1.311		1.966	(S)	6.304	
0.125		1.887		2.528	(C)	0.702		6.367	
0.14		1.919		2.528	(C)	0.717		6.564	
0.125		2.028		1.748		1.544		6.57	
0.282		3.813		0.156		0.125		6.914	76
0.188		2.153		2.528	(S)	0.358		6.919	
0.282		3.906		0.172		0.125		7.023	76
0.14		2.059		2.528	(S)	1.077		7.064	
0.172		2.512		2.528	(I)	0.343		7.103	
0.125		1.872		2.06		1.966	(S)	7.148	
0.109		1.919		2.528	(S)	1.966	(S)	7.503	
0.203		2.891		2.438		0.157		7.516	
0.203		2.06		2.528	(H)	0.936		7.554	
0.125		1.969		2.528	(I)	1.966	(H)	7.713	74
0.312	(I)	4.368	(I)	0.125		0.156		7.769	
0.203		2.641		2.528	(C)	0.641		7.84	74
0.312	(C)	4.368	(C)	0.281		0.093		7.862	
0.312	(I)	4.368	(I)	0.344		0.094		7.926	73
0.14		2.059		2.528	(I)	1.966	(S)	7.953	
0.109		4.368	(H)	2.528	(I)	0.14		8.126	73
0.172		2.625		2.528	(S)	1.266		8.139	73
0.204		2.266		2.528	(I)	1.328		8.162	
0.312	(I)	1.528		1.669		1.966	(I)	8.283	
0.187		2.34		2.528	(C)	1.966	(H)	8.704	
0.11		4.368	(H)	2.528	(I)	0.906		8.902	72
0.203		2.672		2.528	(S)	1.966	(H)	9.196	72
0.312	(I)	4.368	(I)	2.528	(C)	0.094		10.11	70
0.312	(I)	4.368	(I)	2.528	(I)	0.11		10.126	70
0.312	(I)	4.368	(I)	2.528	(C)	0.124		10.14	
0.312	(I)	4.368	(I)	2.528	(I)	0.686		10.702	
0.312	(I)	4.368	(I)	2.528	(H)	1.966	(H)	11.982	
0.312	(C)	4.368	(C)	2.528	(C)	1.966	(C)	11.982	
0.312	(I)	4.368	(I)	2.528	(I)	1.966	(I)	11.982	
0.312	(H)	4.368	(H)	2.528	(H)	1.966	(H)	11.982	
0.312	(S)	4.368	(S)	2.528	(S)	1.966	(I)	11.982	
0.312	(I)	4.368	(I)	2.528	(I)	1.966	(I)	11.982	
0.312	(C)	4.368	(C)	2.528	(C)	1.966	(C)	11.982	
0.312	(I)	4.368	(I)	2.528	(C)	1.966	(C)	11.982	
0.312	(S)	4.368	(S)	2.528	(S)	1.966	(S)	11.982	
0.312	(I)	4.368	(I)	2.528	(I)	1.966	(I)	11.982	65
0.312	(I)	4.368	(I)	2.528	(I)	1.966	(I)	11.982	65

Data Structures and Algorithms I
Spring 2017
Programming Assignment #2 - Grade Sheet

The student who beat me times used what could roughly be called a one-pass most-significant digit radix sort, optionally followed by a series of insertion sorts depending on the data set. All of the sorts relied on a two-dimensional array, where the row represented the bucket, and the columns stored the items (pointers to Data) placed into the bucket. The number of columns was always 10, and the number of rows was approximately equal to the size of the linked list, so the average number of items in each bin would be 1, and overflows would be very unlikely. More specifically, for T1, there were 100,000 buckets, and the five left-most digits (taking into account stripped 0s) were used to determine the bucket. For T2, there were 1,000,000 buckets, and the left six-most digits were used to determine the bucket. For both T1 and T2, each bucket was sorted using an insertion sort before items (pointers to Data) were placed back in the original linked list. For T3, there were 1,000,000 buckets, and all digits were used, so no insertion sorts were necessary (the items place within a bucket were guaranteed to be identical). For T4, assuming that I am understanding the code correctly, the same strategy as for T2 was used. Since the left-most digits are not going to change (this is true with extremely high probability), all items would be placed in the same bucket, thus overflowing the bucket. However, since rows in a 2D array are sequential in memory, and the other buckets have 0 items, this won't cause a problem, and ultimately, the insertion sort of the bucket would be an insertion sort of the entire sequence of pointers. I actually think there was a reasonable chance that this could have overflowed the entire 2D array, but it was known based on pre-submissions that this was not an issue. Also, there are ways to shave off time from the insertion sort (e.g., see the description of my T4 solution). Not surprisingly, this is the one dataset where my sort is faster.

I will now discuss the strategies I used for my sorts, and then discuss other techniques that, in general, are also fast.

For T1 and T2, I used an indirect sort, and took advantage of the provided "sort" function that is part of the algorithms library (a standard C++ library with a header file that is already included in the provided code). There is no guaranteed strategy that it uses, but it is almost always a modified quicksort. As with the "sort" member function of the list class, you need to provide a comparison function. The simplest way to do the indirect sort would be to create a simple array of pointers to Data objects and pass the sort function the bounds of the array and a comparator. However, that would not be as fast as possible, because the sort would constantly have to use indirection to obtain the strings being compared. Therefore, my indirect array is an array of structures such that each structure contains an unsigned integer that can be used to approximate the magnitude of the Data object, plus a pointer to the Data object which is used to break ties.

The conversion of the original linked list to the indirect array does take significant time relative to the sort, so it is important to make the conversion as efficient as possible. To set the integer values, I use the eight left-most characters (representing digits) of the string in most cases, but less than that for cases in which leading 0s were stripped. To determine if leading 0s have been stripped, you need to locate the decimal point in the string. Rather than walk left to right, which would be relatively slow, I start looking at the 21st character (which will be the location of the decimal point approximately 9 out of 10 times) and then walk left to find the decimal point (each necessary step indicates a stripped 0 digit). It is also important to convert the left-most digits to bits efficiently. I wound up using only bit operators (and increments for a pointer to the digits). At first, I thought I would have to subtract '0' (the ASCII value of the character zero) from each character in the string. Referring to an ASCII chart, however (I still don't have this stuff memorized), I realized that the digits range from 0x30 (hexadecimal 30 = decimal 48) to 0x39, so a bitwise-and with 0xF is all that is necessary to convert a digit character to the appropriate four bits. Bit shifts and bitwise-ors can then be used to combine the new bits from each digit to obtain an unsigned integer. Note that in terms of storage, this is a bit wasteful; four bits can store 16 values, and I am using them to store one of ten values (based on a decimal digit). It is possible to use a 32-bit unsigned integer to keep track of nine decimal digits, but to do that, the conversion would be slower, and I am convinced it would not be worth it.

In any case, it is quite likely (I believe almost a certainty) that there will be duplicates of the first eight digits in some cases for both T1 and T2. (If you don't agree, look up the "birthday paradox" and you should be convinced.) Therefore, my comparison function takes this into account; if the unsigned integers for the objects being compared are equal, it does a full string compare using the pointers to the original Data objects. Although that is much slower, it occurs very rarely.

It is probably possible to implement a quicksort that would be a bit faster than the "sort" function in the algorithms library. That sort is highly optimized, and it is doubtfully possible to beat it by much for general sequences without special properties. However, we would have one advantage, which is that we could include the comparison in our own code, rather than have the sort call a function for every comparison. I know that in pervious semesters, some students have been able to create quicksorts that marginally beat the available sort function. I did not have time to try.

A more likely method that has the potential to sort T1 and T2 faster is to use a radix sort. (Of course, this semester, the students that beat my weighted score did this, but I will discuss the strategy more generally here.) There are a few things that make it difficult to create an efficient radix sort for this data. First, there is no way to go through every digit and still have the sort be fast enough. (If the data sets were big enough, any linear sort would beat quicksort, but with one million elements, it needs to be implemented extremely efficiently.) Therefore, we need to apply radix sort using only the high-order digits; but this might lead to a list that is not quite sorted. One possible way to fix this is to follow the radix sort by an insertion sort. The insertion sort should be very fast, since the list will be close to sorted by that time (see my time for T4 compared to my time for T2), so the combination of a radix sort followed by an insertion sort might be fast enough. Perhaps it might be even faster to combine the insertion sort with the last phase of radix sort by inserting the elements into their bins in sorted order, as opposed to just plugging them onto the ends. Also, if the radix sort has more than one pass, you would typically have to copy elements back to the original list in between passes. Faster, though, is to use a separate set of bins for each pass, and copy directly from one set of bins to the other (based on different bits, of course). Most importantly, you want to avoid dynamic memory. Figure out the average number of items that will fall into each bin, and pre-allocate the size of each bin to be much bigger than that. (Although there is some theoretical chance that this will fail, if you make each bin, say, twice as large as the average bin for a multiple-pass radix sort, the chance of this failing for randomly generated data is going to be extremely small.) Unfortunately, I did not have time to implement my own radix sort, but I expect there is a good chance I would be able to beat my current times for T1 and T2 with this strategy if I put in enough effort, and again, the student that beat me this semester used a similar strategy.

Moving on to T3, I used a counting sort. Since there are only one million possible distinct values, it is possible to count how many times each value occurs. There were a lot of string processing tricks I used to convert each string to a bin (a.k.a. bucket). In fact, I used more buckets than necessary. For each of the six digits in the number (including stripped leading and trailing zeros), I used four bits, and so all together, I created $(2^4)^6 = 2^{24}$ (≈ 16.8 million) buckets. This gave me the advantage of being able to determine the appropriate bucket very fast, but on the other hand, I had more buckets to loop through in the second phase of the sort. It is possible that a counting sort using one million buckets could be faster (due to slower conversion but fewer buckets), but I did not have a chance to try it. I am positive I could have sped up my counting sort by only looping through the buckets that might have counts greater than zero (using separate loops for each such stretch of buckets), but once again, I didn't have time to implement this. One other trick is that I didn't want to convert the bucket number back to the string representation for the array. Since identical strings are indistinguishable, I just stored one pointer associated with each bucket that had a count greater than zero, and I used these pointers to populate the linked list while looping through the buckets during the second part of the sort. This is quite hacky, since it means that the pointers used to write the output file are not identical to the pointers in the original list, but it is well within the rules of the assignment, and it produces the correct output.

For T4, I never considered anything other than an insertion sort. It tends to be the fastest sort for close to sorted lists. I wasn't sure which would be faster between an insertion sort applied directly to the linked list versus an indirect insertion sort using an array. I only had time to implement one, and I chose an indirect insertion sort. Although this adds an extra phase at the end of the sort to restore the original list, it allows me to use a method similar to what I did in T1 and T2, storing an integer along with each pointer to a Data object copied over. This way, as each item walks to the left (during the insertion of the item), I could usually get away with comparing integers instead of strings. Note that I did not have to create the indirect array first and then apply the insertion sort; each item gets inserted appropriately as it is first copied over. As with T1 and T2, I was limited to how many digits I could use; I again used eight digits, converted to 32-bit unsigned integers, and broke rare ties with a general string compare.

The trickiest part of the insertion sort was figuring out which digits to use. Using the left-most digits would be useless, since they would be the same for every item in the list. At first, I used one digit to the left of the decimal point and seven to the right. (You only have to locate the decimal point once, for the first item; it will be in the same position for every item in the list. The chance of gaining an extra digit anywhere in T4 is much smaller than one in a million.) I knew there would be cases where a subtraction changes the ones digit from a 0 to a 9, and I had to be careful to handle this case correctly in my comparison. What is a bit trickier is that there are cases in which multiple subtractions will change a 0 to an 8, maybe even a 7, and handling all the possible cases took extra time. Ultimately, I decided to use two digits to the left of the decimal point and six digits to the right of the decimal point. I did not calculate the odds, but I am fairly confident that there is less than a one in a million chance that there will ever be enough subtractions in a small range of the list to change the hundreds digit from a 0 to an 8, so the only special case that had to be considered was a 0 changing to a 9. My gut instinct is that my T4 sort might be the hardest to beat; but then again, some students have come close, and perhaps an insertion sort applied directly to the linked list (as opposed to my indirect method) could be faster if implemented just right.