CS380 Algorithm Design & Analysis

Assignment 3: Sorting Large Files

Date Assigned: Monday, February 28, 2011
Part 1 Due: Monday, March 7, 2011 @ 11:59pm
Part 2 Due: Friday, March 11, 2011 @ 11:59pm
Total Points: 45pts (20 points for part 1, 25 points for part 2)

One common problem with designing a computer program is resource constraints. For example, let's say you had a really big file containing people's names that you wanted to sort. The most common way to do this is to read the entire file into a big array or vector of strings, and then sort the array, and finally save the sorted array into a new file.

But files can be really big, and they may contain more information than can be stored in memory. My computer, for example, has a 300GB hard drive and 2.4 GB of memory. Theoretically, I could have a 300GB file. If I tried to read in such a large file into an array (or vector), my program would run out of memory and throw an "out of memory" error. The program is constrained by the available memory resources.

For this assignment, you will design and implement a program that sorts a file and still works correctly, even when there is not enough memory to store an entire input file. The file you are to sort will contain strings, for example:

> Raindrops on roses and whiskers on kittens, bright copper kettles and warm woolen mittens. Brown paper packages tied up with string. These are a few of my favorite things.

Your program would then produce a new file containing all of the words sorted by their ASCII value:

> Brown Raindrops These a and and are bright copper favorite few kettles kittens, mittens. my of on on packages paper roses string. things. tied up warm whiskers with woollen

Of course the above example only sorts a 1KB file. Your program should also be able to sort a 6GB file or even larger!

Your program will implement a hybrid of the MergeSort algorithm and three other sorting algorithms (insertion sort, quicksort, and heapsort) as a two-phase process, as described below.

Phase 1: Breaking the input file into manageable chunks

In the first phase, the program will read in a fixed number of words from the input file, and store that into a vector. It should pick a number that will fit into memory. Next, it will sort that array using one of the three sorting algorithms (start with insertion sort).

It will then store the sorted vector in a temporary file called “temp_1_1.txt".

For this assignment, a “word" is anything that is delimited by whitespace, regardless of whether it contains punctuation or not.

The program will repeatedly read in chunks until it has read in the entire contents of the input file. Each time it reads in a chunk of words from the input file, it stores that chunk in a vector, sorts the vector, and saves the vector in another temporary file (“temp_1_2.txt", “temp_1_3.txt", ... “temp_1_n.txt").

Notice that it is reading in an amount that will fit into memory each time, so that it does not run out of memory.

After this first phase is complete, each of the “temp_1_i.txt" files is in sorted order, and together they contain all of the words that were in the input file. The second phase must merge the files together, while making sure that the merged files remain in sorted order.

Phase 2: Merging the chunks together

The program should begin phase 2 by merging “temp_1_2.txt" and “temp_1_2.txt", and saving it to a new file called “temp_2_1.txt". Next, it will merge “temp_1_3.txt" with “temp_1_4.txt", and save the merged file as “temp_2_2.txt". This will repeat until there are no more “temp_1_i.txt" files left.

If there are an odd number of these files, the last one will have nothing to merge with. That's ok; it can be merged in later iterations.

After merging pairs of the “temp_1_i.txt" files, the sort method needs to begin merging pairs of the “temp_2_i.txt" files. It will begin by merging “temp_2_1.txt" and “temp_2_2.txt", and saving the result to “temp_3_1.txt". Then, it will merge “temp_2_3.txt" and “temp_2_4.txt", and save the result to “temp_3_2.txt", and so on.

Each time through the set of temporary files, the merging process cuts the number of temporary files roughly in half, because it merges two files into one. (I say “roughly" because there may be an odd number of files, in which case the last file does not get merged with anything.) This phase of the sorting must keep repeating, until there are only two temporary files left. When that happens, it should merge those temporary files, which will contain the final sorted file. Then the sort is finished.

Details on how to merge a pair of files (assuming each file is sorted)

Recall from class that the MergeSort algorithm operates by keeping indexes to two arrays. When you are merging two files, you will be doing essentially the same thing.

The algorithm for merging the two files is relatively simple: you should begin by reading in one word from each file into a variable, say word1 and word2. If word1 is smaller than word2 (it comes alphabetically before word2), then print out word1 to the output file and read in another word from the first file into word1. Otherwise, print out word2 to the output file, and read in another word from the second file into word2. When either one of the files is finished, write the contents of the other file to the output file.

At first, try reading in around 1/20 of the words in each chunk during the first phase. Your program should work no matter what setting is used for the number of words to read in at a time, so long as that many words can fit into memory. Test your program with several different settings.

What to Complete for Part 1

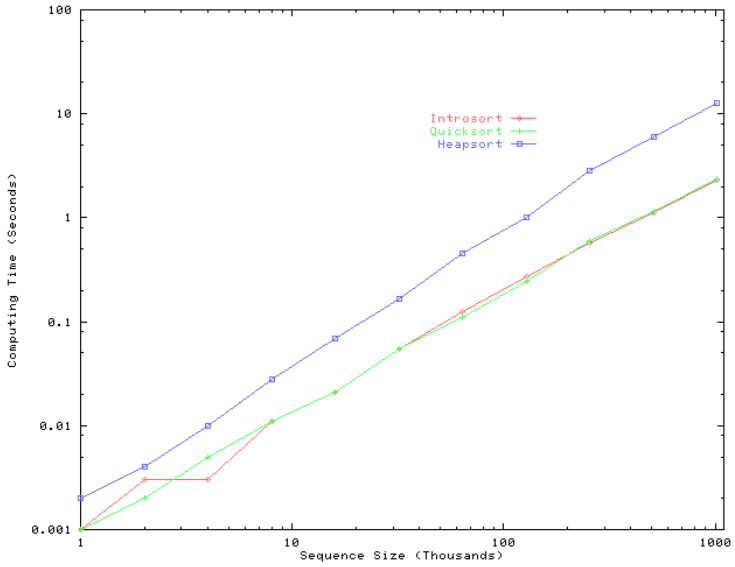
- Complete Phase 1 as described above. In other words, your program should read in the file and break it up into files of manageable chunks, and each file is sorted. I will be testing this phase against both a small file and a large file.
- The naming of the temporary files should be handled automatically.
- Design your program using good object-oriented techniques. Use the insertion sort algorithm from your first assignment, and make sure that your program is modular as you go along.

What to Submit for Part 1

- Submit an electronic copy of your project by 11:59pm on the day that it is due. Name your project "03PUNET-Large-1", replacing PUNET with your PU Net ID (i.e. khoj0332).
- A summary of the time that you spent working on this assignment, and what slowed you down the most. Submit this document electronically as a Google Document called "03PUNet-Large-1". Create the Google document and share it with me at ShereenKhoja@gmail.com.

What to Complete for Part 2

- Complete Phase 2 as described above. In other words, your program should merge the sorted files together into one file.
- Write a report on the performance of the three sorting algorithms you have used. In order to do this, you should use some kind of timing function to work out how long each sort took and test it on files of various sizes. Once you have obtained multiple running times, you should use Excel to create a graph similar to the one below:



- Hints on using timers in C++:

```
clock_t start, finish;
start = clock();
sort();     // Call your sorting algorithm
finish = clock();
cout << "Time for sort (seconds): "
    << ((double)(finish - start))/CLOCKS_PER_SEC;
```

- The report should contain a table of running times, the graph, what algorithms is suited for different values of n, and a description of how the times compare to the theoretical running times of the algorithms.

What to Submit for Part 2

- Submit an electronic copy of your project by 11:59pm on the day that it is due. Name your project "03PUNET-Large-2", replacing PUNET with your PU Net ID (i.e. khoj0332).
- The report described above and a summary of the time that you spent working on this assignment, and what slowed you down the most. Submit this document electronically as a Google Document called "03PUNet-Large-2" for example "03khoj0332Part2". Create the Google document and share it with me at ShereenKhoja@gmail.com.