Splitting input into parts

split –n10 data_in

This creates 10 new files a1 a2 a3... each with 1/10th of the original file.

It does no delete the original file

Sorting original file to get a verification file Verifying your sorted output

- Sorting original file to get verification file
- ``time sort <original_file > verification_file

- Verifying your output file is correct
  - Diff your_sorted_file verification file
    - If there are errors check that the split did not split a line. Across the output files

evaluating sorting algorithms

- Perform timing estimates on your sorting algorithm
- 1- time your_sorting_algorithm using the time command line command “time program …”
  - this prints out the total time for your **program** to work
  - you will have to re-run it for each algorithm
- 2- time your_sorting_algorithm using the coarse timer function used earlier

  get start time; sort; get end time; compute the difference

  this measures the total execution time of your_sorting_algorithm you can test several algorithms at the same time

3- time your sorting algorithm using the fine grain timing function

check several algorithms that look promising (better than nsquared)

check against a nsquared algorithm to confirm your expectations

determine largest data set you can sort

- starting with candidate sorting algorithms determine the maximum size array that can be sorted. The ideal size would be 5 million data items. The limits are due to allocations on the stack and appear as memory addressing errors and cause the program to crash. You may be able to get the faster algorithms to work with larger arrays through careful analysis of the algorithms. Moving the allocations to the global memory (by moving the declaration outside the main()) should help.

sort each disk file separately using your favorite algorithm

apply your favorite algorithm to each of the 10 (or more) files separately producing 10 (or more) fully sorted files. You do not need to use mergesort to do this sorting but you cannot use the sort command.

merge the individual file into a single file
one word at a time (inefficient)

1- read one value in from each file

2- find the smallest value

3- write the smallest value out to the output file

4- read a replacement value for the one just used up

5- repeat until all files are empty

note not all files will empty at the same time.

expect some files to be empty while others still have data

read in blocks of data from the disk file (more efficient)

- for the previous case, each time you read one word from a disk file it takes several milleseconds to get the data from the operating system. A more efficient way would be to read in a block of data from each file, buffer it in your program, process the blocks in the same way you processed the files. In this case you need to keep track how many data items are in the buffer and refill it when it is empty