**CS 325 Fall 17**

**HW 1 – 30 points**

1. (3 pts) Describe a Θ (*n* lg*n*) time algorithm that, given a set S of *n* integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Explain why the running time is Θ (*n* lg*n*).

We can use MergeSort to sort the array. MergeSort recursively splits the array into upper 0…n/2 and lower n/2+1…n halves. On each call to MergeSort the two halves are merged, comparing the left-most values and inserting them into an array in proper order. MergeSort is

Θ (*n* lg*n*).

We can search a sorted array for a specific value using BinarySearch. A BinarySearch starts in the middle of the array at array[n/2], checks if the search term exists in that index, and if not, it checks if the search term is larger or smaller than the value in the current index. If the value is larger, it continues by calling BinarySearch on the left side of the array from the current index. This continues until the search term is found, or there are no more indexes left to search. In order to find whether two elements exist whose sum is x, we can cycle through each index of the array, subtract x from the value in that index, and BinarySearch the array for the result.

BinarySearch is O(lg(n)). In the worst case, we are running BinarySearch n times when we cycle through each item of the array. This results in O(n lgn) time. We then add the time that MergeSort takes to the time that BinarySearch takes: (n lgn) + (n lgn) = 2(n lgn). Dropping the constant and accounting for the fact that MergeSort will always take *n* lg*n* time no matter the input (different from BinarySearch), we get a final result of Θ (*n* lg*n*).

1. (3 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.
   * 1. f(n) = n0.25; g(n) = n0.5

**f(n) = O(g(n)**

g(n) is growing faster

the limit of f(n)/g(n) as n approaches infinity is 0

* + 1. f(n) = log n2; g(n) = ln n

**f(n) = Θ(g(n))**

log n2 and ln n only differ by constants

* + 1. f(n) = nlog n; g(n) =*n*

**f(n) = O(g(n)**

g(n) is growing faster

the limit of f(n)/g(n) as n approaches infinity is 0

* + 1. f(n) = 4n; g(n) = 3n

**f(n) =** Ω **(g(n))**

f(n) is growing faster

the limit of f(n)/g(n) as n approaches infinity is infinity

* + 1. f(n) = 2n; g(n) = 2n+1

**f(n) = Θ(g(n))**

f(n)/g(n) = 1/2

* + 1. f(n) = 2n; g(n) = n!

**f(n) = O(g(n)**

g(n) is growing faster than f(n)

the limit of f(n)/g(n) as n approaches infinity is 0

1. (4 pts) Let f1 and f2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

* + 1. If *f1*(*n*) *=* O(*g*(*n*)) and *f2*(*n*) *=* O(*g(n*))then *f1*(*n*)*+ f2(n*) = O(*g(n*)).

**Proof:**

**Assume that *f1*(*n*) *=* O(*g*(*n*)) and *f2*(*n*) *=* O(*g(n*)).**

**There are 3 possibilities for f1(n) and f2(n): 1) f­1(n) is growing faster than f2(n). 2) f2(n) is growing faster than f1(n). 3) f1(n) and f2(n) are growing at the same rate.**

**For the first possibility, if f1(n) is growing faster than f2(n), then the highest degree term of f2(n) must be lower than the highest degree term of f1(n). For f1(n) + f2(n), the highest degree term has not changed. Therefore, based on the definition of big O notation, f1(n) + f2(n) must be equal to O(g(n)).**

**Then, of course, the second possibility will have the same result. The only difference being that the highest degree term comes from f2(n) rather than f1(n). Therefore, f1(n) + f2(n) must be equal to O(g(n)).**

**For the third possibility, if f1(n) and f2(n) are growing at the same rate, then they must have terms with the same leading degree. Since we are not concerned with the lower order terms, they can be dropped, and only the highest order terms need to be added together. Based on the rules of mathematics, adding two terms with same degree results in another term with the same degree.**

**Then, as n increases, there must be a point n0 at which f2(n) will never be larger than f1(n).**

* + 1. If *f*(*n*) *=* O(*g1*(*n*)) and *f*(*n*) *=* O(*g2*(*n*))then *g1*(*n*) *=* Θ (*g2*(*n*) )

**Counter example:**

**Suppose that f(n) = n2, g1(n) = n3 and g2(n) = n­2. Then, based on the definition of big O notation, f(n) = O(*g1*(*n*)) and *f*(*n*) *=* O(*g2*(*n*)). Based on the definition of Θ notation, there must exist positive constants c1, c2, and n0 such that 0 ≤ c1g2(n) ≤ g1(n) ≤ c2g2(n) for all n ≥ n0. There is no point n0 at which x2 will be larger than x3 for all n ≥ n0. Therefore, it is not true that “If *f*(*n*) *=* O(*g1*(*n*)) and *f*(*n*) *=* O(*g2*(*n*))then *g1*(*n*) *=* Θ (*g2*(*n*) )”.**

1. (10 pts) **Merge Sort and Insertion Sort Programs**

Implement merge sort and insertion sort to sort an array/vector of integers. You may implement the algorithms in the language of your choice, name one program “mergesort” and the other “insertsort”. Your programs should be able to read inputs from a file called “data.txt” where the first value of each line is the number of integers that need to be sorted, followed by the integers.

Example values for data.txt:

4 19 2 5 11

8 1 2 3 4 5 6 1 2

The output will be written to files called “merge.out” and “insert.out”.

For the above example the output would be:

2 5 11 19

1 1 2 2 3 4 5 6

***Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will test execution with an input file named data.txt.***

**CS 325 Fall 17**

**HW 1 – 30 points**

1. (10 pts) **Merge Sort vs Insertion Sort Running time analysis**

The goal of this problem is to compare the experimental running times of the two sorting algorithms.

* 1. Now that you have proven that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from a file to sort, you will now generate arrays of size n containing random integer values from 0 to 10,000 and then time how long it takes to sort the arrays. We will not be executing the code that generates the running time data so it does not have to be submitted to TEACH or even execute on flip. Include a “text” copy of the modified code in the written HW submitted in Canvas.

* 1. Use the system clock to record the running times of each algorithm for n = 1000, 2000, 5000, 10,000, …. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. If you program in C your algorithm will run faster than if you use python. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the same algorithm you may want to take the average time of several runs for each value of n.
  2. For each algorithm plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. Also plot the data from both algorithms together on a combined graph. Which graphs represent the data best?
  3. What type of curve best fits each data set? Again you can use Excel, Matlab, any software or a graphing calculator to calculate a regression equation. Give the equation of the curve that best “fits” the data and draw that curve on the graphs of created in part c).
  4. How do your experimental running times compare to the theoretical running times of the algorithms? Remember, the experimental running times were “average case” since the input arrays contained random integers.

**EXTRA CREDIT:** *It was the best of times, it was the worst of times…*

Generate best case and worst case input for both algorithms and repeat the analysis in parts b) to d) above. Discuss your results and submit your code to TEACH.