CSCD320 Homework1

Ian Kaiserman, EWU ID: 00867173

Problem 1 (5 points)**.** *Based on your learning from the CSCD300 Data Structures course, describe your understanding of the connection and difference between the "data structures" and "algorithms". Say your opinions in your own language. Any reasonable opinion is welcome.*

Problem 2 (10 points)**.** *Show:* $5000n^2 + n \log n = O(n^2)$

Problem 3 (10 points)**.** *Show:* $5000n^2 + n \log n = o(n^3)$

Problem 4 (10 points)**.** *Show:* $5000n^2 + n \log n = \Omega(n^2)$

Problem 5 (10 points)**.** *Show:* $5000n^2 + n \log n = \omega(n)$

Problem 6 (15 points)**.** *Let* $f(n)$ *be a nonnegative increasing function. Is* $f(n) = \Theta(f(2n))$ *always true ? If yes, prove it; otherwise, give a counter example. Hint: try many different such possible functions for f.*

Problem 7 (25 points)**.** *The idea of Merge Sort that we have discussed in the class is to split the input sequence of size n into two subsequences of each sized n/2, recursively sort each subsequence, and merge the two sorted subsequences into a sorted version of the original sequence. Now, someone proposed the following new idea: why don't we split the input sequence into more subsequences of smaller size, so that the algorithm can reach the exit condition (which is when the sequence size becomes 1) more quickly and thus make the algorithm faster? Your job:*

1. *Change the algorithm and give the pseudocode for this proposed new Merge Sort, where the input sequence is divided into 4 subsequences of each sized* $n/4$*.*
2. *Analyze the time complexity of this new algorithm and present the result using the* Θ *notation.*
3. *Is this new algorithm asymptotically faster than the one in the textbook ? Justify your answer.*
4. *Can this algorithm be faster in practice than the one in the textbook ? Justify your answer.*
5. *Let's think of the extreme case. We split the input sequence into* n *subsequences of each sized just 1. Can this algorithm be asymptotically faster than the one in the textbook ? Justify your answer.*
6. *Do you get any insight why the textbook Merge Sort only splits the input sequence into two halves ?*

Problem 8 (15 points total; 5 points for each algorithm)**.** *Search and learn three existing algorithms that use the divide-conquer strategy. For each algorithm, in your own language, concisely and clearly describe:*

1. *the problem statement*
2. *the algorithmic idea in the solution (don't just copy the code or the text on the webpage to me)*
3. *the time complexity*
4. *the condition, on which the worst-case running time appears.*
5. *the source of your finding. For example, the url of the webpages, the title and page of a book, the title/author/year of an article, etc.*

Note: If you just copy and paste or with small modification without your own understanding, you will get zero for this problem.

Solution for Problem 1.

From my understanding, data structures are the ways in which data is stored and expressed, while algorithms are the ways in which said data is manipulated and used for certain purposes. data structures can influence how algorithms are written depending on their needs and how it is structured, and algorithms often directly change data contained in data structures.

Solution for Problem 2.

Proof. For a function to be in the family of the time complexity, there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ (loose upper bound)

$f(n) = 5000n^2 + n \log n$

$g(n) = n^2$

Find c and n_0 such that when $n \geq n_0, 0 \leq 5000n^2 + n \log n \leq cn^2$

$5000 + \frac{\log n}{n} \leq c$

Let $c = 5001$ and $n_0 = 5$

$\frac{\log n}{n} \leq 1$

When $n \geq n_0$ starting from 5, the left side gets closer and closer to 0, making the inequality true and showing that $O(n^2)$ is correct for the polynomial. □

Solution for Problem 3.

Proof. For a function to be in the family of the time complexity, there must exist a positive constant n_0 for any positive constant c, such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ (strictly loose upper bound)

$f(n) = 5000n^2 + n \log n$

$g(n) = n^3$

Show that for any positive constant c, there exists a positive constant n_0 such that when $n \geq n_0$, $0 \leq f(n) < cg(n)$.

$5000n^2 + n \log n < cn^3$

$\frac{5000}{n} + \frac{\log n}{n^2} < c$

As n becomes larger, the entire left side approaches 0.

Therefore, for any positive constant c, there always exists a large enough n_0 such that when $n \geq n_0$, the inequality is true. □

Solution for Problem 4.

Proof. For a function to be in the family of the time complexity, there exists positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ (loose lower bound)

$f(n) = 5000n^2 + n \log n$

$g(n) = n^2$

Find c and n_0 such that when $n \geq n_0, 0 \leq cn^2 \leq 5000n^2 + n \log n$

$c \leq 5000 + \frac{\log n}{n}$

Let $c = 5000$ and $n_0 = 200$

$0 \leq \frac{\log n}{n}$

For all $n \geq n_0$, the right side gets closer to 0 but never truely reaches 0, making it always greater than or equal to the left side, proving the inequality true.

□

Solution for Problem 5.

Proof. For a function to be in the family of the time complexity, there must exist a positive constant n_0 for any positive constant c, such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ (strictly loose lower bound)

$f(n) = 5000n^2 + n \log n$

$g(n) = n$

Show that for any positive constant c, there exists a positive constant n_0 such that when $n \geq n_0$, $0 \leq cg(n) < f(n)$.

$cn < 5000n^2 + n \log n$

$c < 5000n + \log n$

Since both sides share the same largest degree (c and n, degree of 1), for every possible positive constant c, a positive constant n_0 can be chosen to make the right side larger, and therefore when $n \geq n_0$, the inequality is always true. □

Solution for Problem 6.

The statement is false.

Proof. For $f(n)$ to be part of the family of functions of $\Theta f(2n)$, The upper and lower bounds must be strictly tight on both sides for all $f(n)$.

However, in the case of $f(n) = c^n$ where c is any natural number, that makes $f(2n) = c^{2n}$ which has a different degree than $f(n)$ and therefore a different growth rate, making the use of Θ incorrect. □

Solution for Problem 7.

1. Pseudocode for $n/4$ merge sort algorithm

```
merge_sort(A, p, t) {
  if(p<s) {
    quarter = (t-p)/4
    q = floor(p+quarter)
    r = floor(p+2quarter)
    s = floor(p+3quarter)

    merge_sort(A, p, q)
    merge_sort(A, q+1, r)
    merge_sort(A, r+1, s)
    merge_sort(A, s+1, t)

    merge(A, p, q, r, s, t)
  }
}
```

2. Looking at the pseudocode given above, each recursive merge sort call is called roughly $n/4$ times due to the array being split into 4 parts, and the merge call is still being called n times. Compared to the traditional merge sort, the visual tree is split into 4 pieces each time, which means instead of $2^x = n$ being the number of elements in the tree, it is now $4^x = n$. This converts into $\log_4 n$. Each line in the tree still adds up to cn because the number of items in each line of the tree increases by the same factor that each item is being divided by. Therefore, the resulting time complexity is $\Theta(n \log_4 n)$.

3. Yes, the new algorithm is asymtotically faster than the one in the text book. This is because base 4 logarithm grows at a slower rate than base 2, meaning the time spent grows less for the former than the latter as n gets larger.

4. This algorithm can be faster in practice, especially for much larger array sizes. When the size of the input array gets extremely large, so does the gap between the base 2 and base 4 logarithm functions. This means the 4 way merge sort gets more efficient the larger n gets.

5. In the extreme case, when the subsequences are very small, this is where this algorithm proves to be slower overall, and isn't faster asymptotically than the textbook example. This is because once the subsequences get smaller than size 4, the algorithm can't efficiently split the subarrays into 4 pieces, meaning it will end up checking the same subarrays several times. For example, when looking at a subarray from index 1 to 2, it will be recursively calling merge sort for the subarrays 1 to 1, 1 to 1, 2 to 2, 2 to 2, meaning there are two duplicate steps there. For much larger arrays, this is a large waste of time and memory when a typical merge sort would be much more efficient.

6. The textbook Merge Sort splits into two halves because it is the simplest and most efficient way of using merge sort in ALL cases, not just smaller or larger arrays. It minimizes the amount of needless steps like the 4 way version has, while still having a relatively efficient design due to the nature of logarithm time complexity not causing an exponentially increasing number of steps.

Solution for Problem 8.

1. QuickSort (I know this was mentioned in class, but I couldn't remember if I'd learned it or not, and even if I did it would be nice to get a refresher on it)

    - Problem statement: Given an array of n numbers, sort the array in ascending order.
    - Algorithmic idea: Starting with the original array, a "pivot" needs to be chosen from the list of numbers. One example is using the last element in the array. The array will be sorted by comparing each element to this pivot to determine which side of the pivot this element should be on (larger or smaller).

      The array is recursively split on the pivot element, and these two parts are sent to a "partition" function, which keeps selecting a pivot element and splitting accordingly until the subarrays only have one number. As the pivoting is one, unlike Merge Sort, all of the changes made to the order are done on the original array instead of a copy. Therefore, as the pivoting is done to divide elements to the left and right of the pivot, the array naturally sorts itself as it reaches these subarrays of one element.

- Time complexity: The time complexity of Quick Sort varies based on the array input. This is because instead of always splitting the array into two even parts, the splitting here entirely depends on what the pivot is, meaning it could be split as few times as Merge Sort, or as many times as going over the entire list for every element. In the average case however, the splitting will look similar to merge sort (about the middle on average) and will similarly have a time complexity of $O(n \log n)$.
- As mentioned before, the worst case is that the entire list has to be analyzed for every element of the list. This is because it's possible that every time a pivot is analyzed, every single element needs to be moved to the other side of it. Therefore the worst case scenario time complexity is $O(n^2)$.
- Source: `https://www.geeksforgeeks.org/quick-sort/`

2. Binary Search (divide and conquer method)

   - Problem statement: Given a sorted array of n numbers, and a particular number to find, return the location of the number being searched for.
   - Algorithmic idea: a function is created that takes the sorted array, the number being searched for, and a low and high index. A middle point is found and compared to the search number. if found, return; if not, determine if it's on the left or right side of the middle element. Recursively call the same search function, this time using the corresponding side of the array as the low and high index points to search in. the search value will continue to be compared to the middle value of the subarray, and if it doesn't match it will be determined to be on the left or right side of the sorted array and it will be further divided.
   - Time Complexity: The time complexity of this algorithm is similar to that of Merge Sort, but with one key difference. the muiltiplication of $\log n$ by n is not necessary, because the second half of each subarray will never need to be searched. This is because the array is already sorted, so if the search value is less than or greater than the middle value, the left right side ONLY will need to be searched, respectively. Therefore, the time complexity is only $O(\log n)$.
   - The worst case scenario for this algorithm is if the array has to be divided all the way until the subarray searched is of size 1. This only ends up having a time complexity of $O(\log n)$ which is the same as the general complexity of the algorithm, and much more tame than the previous example.
   - Source: `https://www.geeksforgeeks.org/binary-search/`

3. Integer Multiplication

   - Problem Statement: Given two positive integers of the same length, find the product of these integers.
   - Algorithmic idea: A function takes both of the input integers, and splits the length of the integers in half. These 4 halves (2 for each integer) are sent recursively into the same function in two-digit multiplication style (bottom right*top right, bottom right*top left, bottom left*top right, bottom left*top left), continuously split until each input integer has a length of 1, where they are then multiplied together and returned. Once each recursive call has those set of 4 results, they are then multiplied by an exponent of 10 depending on the number of digits in the two numbers and added together, similar to what is done on paper when multiplying. Eventually these results make their way up the recursive tree and the final result is achieved.
   - Time Complexity: The time complexity of this algorithm is $O(n^2)$, since each digit of the first number needs to be dealt with once for every digit of the second number.
   - There isn't really a worst case scenario with this algorithm because the numbers need to be fully multiplied out no matter what, as that is the point of the algorithm
   - Source: `https://www.cs.cmu.edu/~cburch/pgss99/lecture/0721-divide.html`