# CS260 - Data Structures - Written Assignment 1

Fitsum Alebachew

Question 1 : 10 points

Determine the most accurate $\Theta(\cdots)$ for each function and explain how you calculated the tight bound. Assume that the functions take an input array of size $n$.

(a) [3 points] The `max` method is $\Theta(\quad)$.

```
int max(int* a, int size)
{
    if(size < 2)
        return -1;
    if(a[0] > a[1])
        return a[0];
    else
        return a[1];
}
```

**Solution:** This will do a maximum of 2 comparisons. It is $\Theta(1)$.

(b) [3 points] The `maxElement` method is $\Theta(\quad)$.

```
int maxElement(int* a, int size)
{
    int max = a[0];
    for(int i=1; i<size; i++)
        if(a[i] > max)
            max = a[i];
    return max;
}
```

**Solution:** This will iterate through the list once. It is $\Theta(n)$.

(c) [4 points] The `maxSubseqSum` method is $\Theta(\quad)$.

```
int maxSubseqSum(int* a, int size)
{
    max=a[0];
    for(int i=1; i<size; i++)
    {
        sum = 0
        for(int j=i; j<size; j++)
        {
            sum+=a[j];
            if(sum > max)
                max=sum;
        }
    }
    return max;
}
```

**Solution:** In the nested for loop, for each initial iteration in the list, it iterates through the whole list again. That makes it $\Theta(n^2)$.

Question 2 : 10 points

Consider the following functions:

- $\log_2 x$
- $x \cdot \log_2 x$
- $(3/2)^x$
- $x/\log_2 x$
- $2^x$
- $\sqrt{x}$
- $x^2$
- $(\log_2 x)^2$
- $(1/3)^x$

You are told that, functions above represent the number of operations carried out by a set of algorithms that do the same job, albeit in different ways. Order the functions from slowest growth rate to the fastest growth rate (i.e., sort the algorithms from fastest to slowest). Explain your reasoning for consecutively ordered functions, possibly by using the limit method, to get credit.

**Solution:** $(1/3)^x < \log_2(x) < (\log_2(x))^2 < \sqrt{x} < x/\log_2 x < x < x \cdot \log_2(x) < x^2 < 2^x$.

$(1/3)^x$ is a decreasing function, so it will have a negative growth rate. The second growing functions are the log functions(ordered by degree). Even powered to any number, they grow slower than squareroot functions of any degree. Then the polynomials ordered to their degree or the lesser growing functions they're being multiplied/divided by if they have the same degrees. Then exponentials, which grow faster than polynomials.

Question 3 : 10 points

Professor Bond has developed the following algorithm that he claims is going to be a break through in computer science.

Assuming array is zero indexed, A[0] is the first and A[n-1] is the last element, the algorithm is as follows:

```
mysteryFunction(A, n)
{
    int lastPos=n-1;
    while(lastPos > 0)
    {
        int maxPos=0;
        for(int i=0; i <= lastPos; i++)
            if(A[i] > A[maxPos])
                maxPos = i;
        swap(A, maxPos, lastPos);
        lastPos = lastPos - 1;
    }
}
```

(a) [2 points] Explain what the algorithm does in plain English.

**Solution:** It finds the maximum element, swaps it to the end of the array, then does the same for the remainder of the list until the whole list is sorted.

(b) [2 points] Why does the algorithm run for all elements but the last, rather than for all elements?

**Solution:** After the first run, the last element is sorted and also the largest in the array, so for the algorithm to work, it needs to exclude it from the comparing process.

(c) [3 points] What is the best case of this algorithm and what is its best case running time in $\Theta$ notation?

**Solution:** The algorithm must run through each element going down by one for each iteration $(1 + 2 + 3 + ... + n) = n(n + 1)/2$. This is $\Theta(n^2)$.

(d) [3 points] What is the worst case of this algorithm and what is its worst case running time in $\Theta$ notation?

**Solution:** The algorithm will run through each element going down by one for each iteration $(1 + 2 + 3 + ... + n) = n(n + 1)/2$. This is $\Theta(n^2)$. The same! The kind of input, like presorting does not change the run time, only the size of input matters.

Question 4 : 20 points

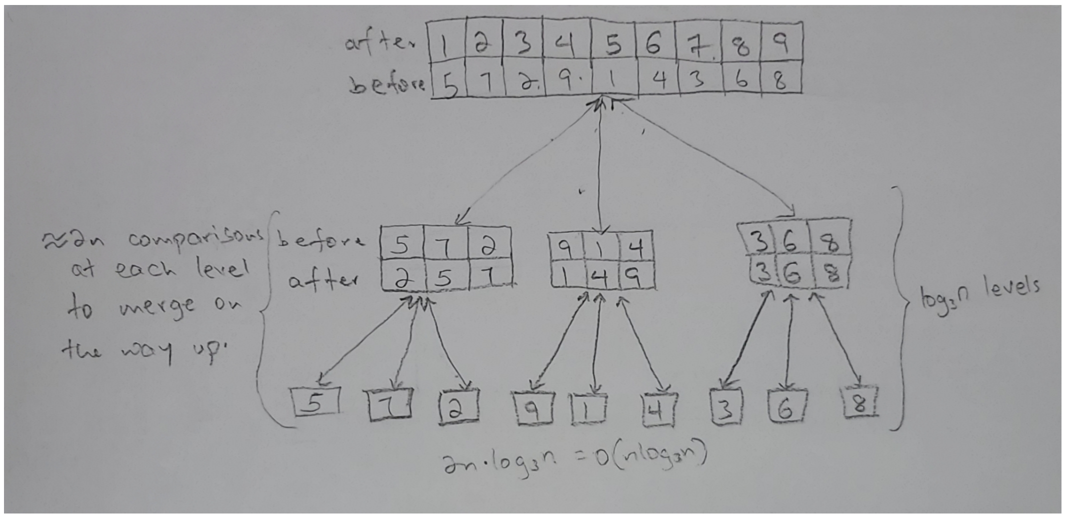
Professor Holmes came up with the idea of a sorting algorithm that he calls *Trinary Sort* which he claims is asymptotically faster than merge sort, despite being similar in logic. Unlike merge sort, trinary sort splits the input list into three roughly equal parts at each step of the recursion as long as the list is splittable (i.e., has at least 3 elements in this case). The merge operation, similar to what it does in mergeSort, takes three already sorted subarrays, and merges them.

(a) [4 points] In merge sort, merge operation takes make $n - 1$ comparisons in total to merge two lists of size $n/2$, which takes $O(n)$ time. How many comparisons will the merge operation of Trinary sort make to merge three sublists of size $(n/3)$? What would be the asymptotic bound?

**Solution:** When merging 3 sublists into one, two comparisons will have to be made to find the largest out of three values. The total number of comparisons will be $2n - 3$ since the last two elements will need 1 or no comparisons needed. This is also $\Theta(n)$.

(b) [5 points] What is the total running time of the Trinary Search algorithm? Show it using the **tree expansion method** as we have done in the class.

**Solution:**



(c) [2 points] What is the recurrence relation for the Trinary Search algorithm?

**Solution:** $T(n) = 3T(n/3) + 2n$

(d) [4 points] Calculate the total running time of Trinary search, this time by **expanding the recurrence relation** you suggested above.

**Solution:** $T(n) = O(n \log_3 n)$.

(e) [5 points] Is Professor Holmes right in his claim that Trinary search is asymptotically faster than Merge Sort? Why/why not? Prove your point. (Hint: Use limit method that was suggested in class and exemplified in the exercise posted, to compare upper/lower bounds for algorithms)

**Solution:** No. Both of their runtimes are asymptotically equivalent, because $\log_3 n$ and $\log_2 n$ are only different by a constant multiple ($\log_3 n / \log_2 n = \log 2 / \log 3$).

Question 5 : 15 points

Calculate an upper bound (tight bound, if possible) for each of the following recurrence relations. Show your work by expanding the relation as was done in class.

(a) [5 points] $T(n) = T(n-1) + n$

**Solution:** $T(n) = T(n-2) + 2n = T(n-3) + 3n = T(n-k) + kn$
Stopping condition: $k = n - 1$
$T(n) = T(1) + n(n-1) = n^2 - n + 1 = \Theta(n^2)$

(b) [5 points] $T(n) = T(n/2) + 1$

**Solution:** $T(n) = T(n/4) + 2 = T(n/8) + 3 = T(n/2^k) + k$
Stopping condition: $2^k = n, k = \log_2 n$
$T(n) = T(1) + log_2 n = 1 + log_2 n = \Theta(log_2 n)$

(c) [5 points] $T(n) = 4T(n/3) + n$

**Solution:** $T(n) = 4(4T(n/9) + n/3) + n = 16T(n/9) + 7n/3$
$= 4(4(4T(n/27) + n/9) + n/3) + n = 64T(n/27) + 37n/9$
$= 4^k T(n/3^k) + n((4/3)^{k-1} + (4/3)^{k-2} + ... + 1)$
Stopping condition: $n = 3^k, k = \log_3 n$
$T(n) = c_1 n + c_2 n \log_3 n$ where c's are constants independent of n $= \Theta(n \log_3 n)$

Question 6 : 10 points

You applied to Google for an internship and they wanted to test your abilities to come up with novel data structures. The company is currently developing tiny robots that has very limited memory and computational resources. They are in need of data structures to be used in embedded system implementation of the robot. Since it is easier to handle and process arrays than linked lists, the company want **the data structures to be array based**, which will be desirable for a limited system. Answer the questions below accordingly.

(a) [5 points] You are being asked to design a new stack that Google wants to name as *Amphisbaena* (why on earth would they want to call it such a weird name!!). Amphisbaena will actually be two stacks on a single array of size $n$, without the two overflowing each other unless there are $n$ elements in total among both stacks. In plain English, explain a data structure that will dynamically and efficiently hold two stacks together in a single structure.

**Solution:** Two stacks can be implemented on a single array if one begins from the start of the array and one from the end. Elements pushed and popped are done so accordingly using two stack pointers pointing to the top of each stack. When the pointers point to adjacent locations, the array will have a combined n elements and is full.

(b) [5 points] You need to design a queue, weirdly enough named as *Ourobors*, to be used in the robot's operating system. Since, the runtime of dequeue operation of a standard queue would be inefficient for the robot, Ouroboros should support a constant time dequeue operation. In plain English, suggest and explain such a queue structure.

**Solution:** We can implement a circular array where added elements don't need to be shifted, but the front and rear pointers are shifted to the necessary locations when an item is added or removed. With this implementation, the dequeue operation will have a runtime of $O(1)$.

Question 7 : 10 points

Your interview at Google was not too bad, and you are expecting to hear from them for a second interview. While waiting for it, thinking that it will not hurt to have alternatives at hand, you applied for an internship at Microsoft. They gave you a singly linked list where there is a pointer that points to the head of the list, and each element of the list points to the next element (as is the case in singly linked lists), with the exception of last element pointing to NIL. They are asking you to develop a function called *reverse* that takes the list as input and reverses the order of objects in $O(n)$ time while using $O(1)$ amount of extra memory.

(a) [5 points] Write the pseudocode of your *reverse* function.

**Solution:**

```
reverse(node **head){
    node *next, *prev = NULL;
    for (node *cur = *head; cur != NULL;cur = next){
        next=cur->next;
        cur->next = prev;
        prev = cur;
    }
    *head = prev;
}
```

(b) [5 points] Explain the running time of your algorithm.

**Solution:** The algorithm runs through the linked list once and uses two additional variables to keep track of next and previous nodes since we are changing the next values in the iterations. This gives it the required $O(n)$ running time and $O(1)$ extra memory usage.

Question 8 : 5 points

Dictionaries are data structures specialized in storing, finding, and removing records from the collection, that keeps record of **unique** elements. You are being asked to design a a dictionary data structure for a specific input set, where you are guaranteed that the input set is going to be integers drawn from the range [1,k] for some some finite integer n. Design a data structure where search, insertion, and deletion operations are done in O(1) time in the worst case. (Note: initialization can take O(n) time)

**Solution:** We can use a hash table where each value has a key determined by a hash function. Inserting, searching and deleting will take $O(1)$ time since the hashing function will give us the address of the element in focus. When there are collisions, the dictionary can be rehashed (initialized again) to a bigger size until there are no collisions.

Question 9 : 10 points

You got an offer letter from Google for your internship and accepted it without thinking too much... very soon to realize that this was the beginning of a nightmare (as they will keep asking you challenging algorithms questions throughout your internship)!! Your group leader wants to test your knowledge on hash tables as well as your algorithm design skills with the following question: We have a list of unique integers. We would like to find whether there exist two pairs of integers, such that their summations are equal, To be more precise, determine whether there exists pairs (p, q) and (r, t) in the list which satisfies p+q = r+t. You can print any such pairs if there exists more than one correct answer.

(a) [5 points] A naive algorithm to solve this problem can achieve $O(n^4)$ run time. In your own words, explain such a naive approach that will do the job.

**Solution:** We can write a quadruple nested for loop to check the combination of every two pairs of numbers against eachother. The first two loops would loop through p and q values, and the other two will loop through r and t values.

(b) [5 points] Using ha data structure that would allow a better performance, suggest a $O(n^2)$ run time algorithm that tackles the problem.

**Solution:** A run through the list can be done to find every sum value of possible two number pairs and store that in a dictionary data structure, this takes $O(n^2)$ time. While storing the values, we can check if the sum is a member of the dictionary(which is done either way). If matches are found, they can be stored in a list holding the matches and returned to the user.