CS 344 Exam 1: Asymptotics, Divide and Conquer Algorithms

Kanghwi Lee

1. (Honor Pledge) - I pledge on my honor that I have neither received nor given any help on this exam

- Yes

2. (Comparing functions)

(a) True

Statement: 2^sqrt(n) = O((sqrt(2))^n)

Justification:

- To prove this, we need to show that there exist constants C and n0 such that 2^sqrt(n) <= C * (sqrt(2))^n for all n >= n0.
- Take the logarithm base 2 of both sides: log(2^sqrt(n)) <= log(C * (sqrt(2))^n) sqrt(n) <= log(C) + n * log(sqrt(2))
- Now, for large enough n, the log(C) term becomes negligible compared to the n * log(sqrt(2)) term, and hence: sqrt(n) <= n * log(sqrt(2))
- Divide both sides by n * sqrt(2): sqrt(n) / (n * sqrt(2)) <= log(sqrt(2))
- As n approaches infinity, the left side approaches 0, and the right side is constant. Therefore, the inequality holds for large enough n, which means 2^sqrt(n) is O((sqrt(2))^n).

(b) False

Statement: n^sqrt(n) = O((log n)^n)

Justification:

- To disprove this, we need to show that for any constant C and n0, n^sqrt(n) > C * (log n)^n for infinitely many n >= n0.
- Let's consider the limit of n^sqrt(n) / (log n)^n as n approaches infinity: lim(n -> ∞) (n^sqrt(n) / (log n)^n)
- By applying L'Hôpital's rule multiple times, it can be shown that this limit approaches infinity, meaning that n^sqrt(n) grows faster than (log n)^n, and hence n^sqrt(n) is not O((log n)^n).

(c) False

Statement: n^(n!) = Θ((n!)^n)

Justification:

- N^n! = multiplies n by itself n! times and (n!)^n multiplies itself by n times (n!)^n will grow faster than n^n!
- N^n! isn't bounded above or below by (n!)^n for all n>n0 because (n!)^n grows faster than n^n!
- N^n! isn't Θ((n!)^n)
- N^n! is O(n!)^n) as (n!)^n doesn't have an upper bound on n^n! but the reverse isn't true, therefore **False**

3. (Recurrence resolution)

- **(a)** The Master theorem tells us that T(n) = O(n ^ r * log(n)) if 0 < d < 1, O(n ^ r) if d = 1, and O(n ^ r * log(log(n))) if d > 1. We have b ^ r = b, r > 0, and 0 < d < 1. Therefore, the Master theorem tells us that T(n) = O(n ^ r * log(n)).
- **(b)** The value of d used in the previous part was 0. This is the smallest possible value of d to which the Master theorem applies.
- **(c)** The tightest upper bound that the Master theorem allows us to provide for T(n) is O(n ^ r * log(n)). This is because the Master theorem only applies when 0 < d < 1.
- **(d)** The lower bound for T(n) is O(n ^ r). This is because the time to solve the base case is O(n ^ r), and the time to solve each recursive sub-problem is O(n ^ r).
- **(e)** The recurrence tree for T(n) is shown below.

```
     T(n)

    /   \

   n ^ r   n ^ r * log(n)

  /    \ /    \

 n ^ r   n ^ r * log(n)   n ^ r * log(n)

/    \ /    \ /    \

...    ... ...    ... ...
```

The number of levels in the recurrence tree is log(n) / log(b). The cost at each level is n ^ r. Therefore, the Theta bound for T(n) is O(n ^ r * log(n)).

4. (Note picking)

- **(a)** An O(n ^ 2) algorithm to solve this problem is as follows:

```
def pick_note(notes):

  max_value = 0

  max_index = 0
  for i in range(len(notes)):
    if notes[i] > max_value:
      max_value = notes[i]
      max_index = i
  return max_index
```

This algorithm works by looping through the notes array and keeping track of the maximum value and its index. The maximum value is returned at the end of the loop. **->O(n^2)**

- **(b)** A more efficient algorithm to solve this problem is as follows:

```
def pick_note(notes):
  max_value = notes[0]
  max_index = 0
  for i in range(1, len(notes)):
    if notes[i] > max_value:
      max_value = notes[i]
      max_index = i
  return max_index
```

This algorithm is more efficient because it only needs to loop through the notes array once. The maximum value is then returned at the end of the loop. **->O(n)**

5. (Function anomalies)

- **(a)** The number of function anomalies for the array A-[13 71 19 7 3 5] is 10.
- **(b)** A simple condition for when an index pair (i, j) is a function anomaly for f for a given array A is if i < j and f(a_i) > f(a_j) and f(a_i) <= f(a_k) for all k in the range [i+1, j-1].

  - > A[i] > A[j]
- **(c)** An O(n ^ 2) algorithm that computes the number of function anomalies for f on a given array A is as follows:

```
def anomalies(A, f):
  anomalies = 0
  for i in range(len(A)):
    for j in range(i + 1, len(A)):
```

```
    if i < j and f(A[i]) > f(A[j]) and f(A[i]) <= f(A[k]) for k in range(i + 1, j - 1):
      anomalies += 1
  return anomalies
```

This algorithm works by looping through the array A twice. The first loop checks each pair of consecutive elements in the array. The second loop checks each element in the array, and then checks all of the elements that come after it. The number of function anomalies is then returned.

- **(d)** A more efficient algorithm that solves the above problem is as follows:

```
def anomalies(A, f):
  anomalies = 0
  for i in range(len(A)):
    j = i + 1
    while j < len(A) and f(A[i]) > f(A[j]):
      if f(A[i]) <= f(A[k]) for k in range(i + 1, j):
        anomalies += 1
      j += 1
  return anomalies
```

This algorithm works by looping through the array A once. The algorithm starts at the beginning of the array and keeps track of the most significant element it has seen. The algorithm then loops through the rest of the array, checking if each element is larger than the largest element it has seen. If it is, then the algorithm increments the number of function anomalies. The number of function anomalies is then returned.

6. (k-Selects Sort)

(a) Proving correctness of k-Selects Sort algorithm

The k-Selects Sort algorithm does not sort the entire array A but selects specific elements from A. It picks elements at ranks n/k, (2n)/k, (3n)/k, ..., kn/k, and then applies Merge sort to sort each of the k groups B_1, B_2, ..., B_k.

While the algorithm does not sort the entire array, it correctly selects the elements at the specified ranks, and within each group, the Merge sort algorithm correctly sorts the elements. Therefore, the k-Selects Sort algorithm is correct in terms of selecting the specified elements and sorting them within their respective groups.

(b) Analyzing time complexity

Let n be the size of the array A.

Using SELECT to pick elements at ranks n/k, (2n)/k, ..., kn/k takes O(n) time.

Partitioning the array A into k groups also takes O(n) time.

Applying Merge sort to each k group takes O(n * log(n/k)) time.

Overall, the time complexity of the k-Selects Sort algorithm is O(n) + O(n) + k * O(n * log(n/k)) = O(n * log(n/k)).

(c) Efficiency comparison with Merge sort

For what values of k would k-Selects Sort be as efficient as Merge sort?

In the worst case, Merge sort has a time complexity of O(n * log(n)), and k-Selects Sort has a time complexity of O(n * log(n/k)).

To make k-Selects Sort as efficient as Merge sort, we want the time complexity of both algorithms to be the same:

O(n * log(n/k)) = O(n * log(n))

Solving for k:

log(n/k) = log(n)

n/k = n

k = 1

For k = 1, k-Selects Sort would be as efficient as Merge sort. However, this would defeat the purpose of using k-Selects Sort as we are essentially selecting and sorting the entire array A. In practice, k-Selects

7. (Coconut Breaking):

(a) An O(n) time algorithm to find the earliest floor from which dropping coconut results in it breaking is as follows:

Python

```
def earliest_breaking_floor(n):
  for i in range(1, n + 1):
    if coconut_breaks(i):
      return i
  return n + 1
```

This algorithm works by looping through the floors from 1 to n. The algorithm checks if the coconut breaks if it is dropped from the current floor. If it does, then the algorithm returns the current floor. If it does not, then the algorithm continues looping. The algorithm returns n + 1 if the coconut does not break from the floors.

However, this algorithm needs to be corrected. This is because the coconut may break if it is dropped from a floor that is higher than the earliest breaking floor. For example, if the coconut breaks if it is dropped from a floor of 10, then the algorithm will return 9 as the earliest breaking floor, even though the coconut will also break if it is dropped from a floor of 10.

(b) A more efficient algorithm to find the earliest breaking floor is as follows:

Python

```
def earliest_breaking_floor(n):
  for i in range(1, n + 1):
    coconut_dropped = True
    for j in range(i, n + 1):
      if not coconut_breaks(j):
        coconut_dropped = False
        break
    if coconut_dropped:
      return i
  return n + 1
```

This algorithm works by looping through the floors from 1 to n. The algorithm keeps track of whether or not the coconut has broken if it is dropped from any of the floors that have been checked so far. If the coconut has broken, then the algorithm returns the current floor. If the coconut has not broken, then the algorithm continues looping. The algorithm returns n + 1 if the coconut does not break from any of the floors.

This algorithm is correct because it ensures that the earliest breaking floor is always returned. This is because the algorithm keeps track of whether or not the coconut has broken if it is dropped from any of the floors that have been checked so far. If the coconut has broken, then the algorithm returns the current floor. If the coconut has not broken, then the algorithm continues looping. The algorithm will eventually reach a floor where the coconut does break, and the algorithm will return to that floor as the earliest breaking floor.

The time complexity of this algorithm is O(n). This is because the algorithm loops through the floors from 1 to n. The total time complexity is then O(n).

8. Extra credit: False

If f(n), g(n), h(n) are all strictly increasing functions satisfying f(n) = O(g(n)) and g(n) = O(h(n)) then f(g(n)) = O(f(h(n))).

Prove:

Since f(n) = O(g(n)), there exists a constant c and an n_0 such that f(n) <= c * g(n) for all n >= n_0.

Since g(n) = O(h(n)), there exists a constant d and an n_1 such that g(n) <= d * h(n) for all n >= n_1.

Then, for all n >= max(n_0, n_1), we have:

f(g(n)) <= c * g(n) <= c * d * h(n) = c * d * f(h(n))

Therefore, f(g(n)) = O(f(h(n))).

Disprove:

The proposition is false if f(n) is not a monotonic function. For example, let f(n) = 1 if n is even and f(n) = 0 if n is odd. Then f(n) = O(g(n)) for all strictly increasing functions g(n). However, f(g(n)) = 0 for all n, while f(h(n)) is non-zero for some values of n. Therefore, f(g(n)) is not O(f(h(n))).

In conclusion, the proposition is true if f(n) is a monotonic function, but it is false if f(n) is not a monotonic function.