CS 344 Exam 1: Asymptotics, Divide & Conquer Algorithms

1. I pledge on my honor that I have neither received nor given help on this exam.

2. True or False. Justify.

a) $2^{\sqrt{n}} = O((\sqrt{2})^n)$ **True**

We can take the limit of the ratio of both functions as they approach infinity

$$\lim_{n\to\infty}\left(\frac{2^{\sqrt{n}}}{\sqrt{2}^n}\right) = \lim_{n\to\infty}\left(\frac{2^{\sqrt{n}}}{2^{\frac{n}{2}}}\right) = \lim_{n\to\infty}\left(2^{\sqrt{n}-\frac{n}{2}}\right)$$

as n approaches infinity, the term $\sqrt{n}-\frac{n}{2}$ dominates, but grows slower than n because it is the exponent. Therefore the limit approaches 0, meaning $2^{\sqrt{n}}$ is bounded by $\sqrt{2}^n$, making the statement true.

b) $n^{\sqrt{n}} = O((\log n)^n)$ **False**

we can take the limit of the ratio of both functions as n approaches infinity again

$$\lim_{n\to\infty}\left(\frac{n^{\sqrt{n}}}{(\log n)^n}\right) = \lim_{n\to\infty}\left(\frac{\frac{d}{dn}(n^{\sqrt{n}})}{\frac{d}{dn}(\log n)^n}\right) = \lim_{n\to\infty}\left(\frac{\sqrt{n}\cdot n^{\sqrt{n}-1}}{(\log n)^n}\right)$$

AS n approaches infinity, the exponent $(\sqrt{n}-1)$ grows slower than $\sqrt{n}$, and $\log n$ grows slower than n. Hence, the limit on the numerator grows faster than the limit in the denominator, making the limit approach infinity. This means $n^{\sqrt{n}}$ is not bounded by $(\log n)^n$, making the statement false.

c) $n^{n!} = \theta((n!)^n)$ **False**

In order to prove this, we need to show that $n^{n!}$ neither grows faster nor slower than $n!^n$, which means we need to prove both $n^{n!} \neq O((n!)^n)$ and $n^{n!} \neq \Omega((n!)^n)$.

- $n^{n!} \neq O((n!)^n)$
  For larger values of n, n! grows faster than n, so $n^{n!}$ grows much faster than $n!^n$, so this is proven.
- $n^{n!} \neq \Omega((n!)^n)$
  For larger values of n, $n!^n$ grows faster than n, so $n^{n!}$ will grow much slower than $n!^n$, so this also is proven

Since $n^{n!}$ is neither $O((n!)^n)$ nor $\Omega((n!)^n)$, the statement $n^{n!} = \Theta((n!)^n)$ must be false.

3. $T(n) = b^r T(\frac{n}{b}) + n^r \log n$

a) use master theorem for a polynomial upper bound for T(n)

The master theorem tells us that if $0 < d < 1 : T(n) = O(n^r \log n)$, if $d = 1 : T(n) = O(n^r)$, and if $d > 1 : O(n^r \log(\log n))$. we also have $b^r = b$, $r > 0$, and $0 < d < 1$. Hence, master theorem gives us $T(n) = O(n^r \log n)$.

b) What was the value of d? Can it be improved to give T(n) a tighter upper bound?

$d = 0$ in the last part, which is also the smallest value of d in which the master theorem applies.

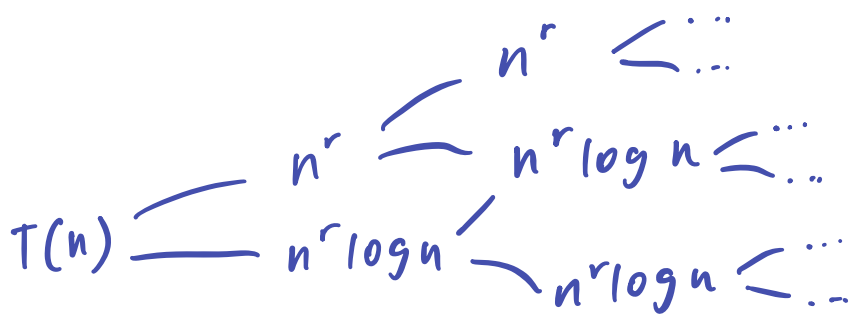
c) what is the tightest upper bound that the master theorem allows you to provide for T(n)?

$O(n^r \log n)$, because the master theorem only applies when $0 < d < 1$.

d) Provide a lower bound for T(n)

$O(n^r)$, because it is the time to solve both the base case as well as the recursive sub-problem.

e) solve the recurrence tree



the number of levels in the recurrence tree can be determined by: $\frac{n}{b^i} = 1$

$\rightarrow n = b^i \rightarrow \log n = i \log b$

$i = \frac{\log n}{\log b}$ levels.

The cost at level i is: $\left(\frac{n}{b^i}\right)^r \cdot \log\left(\frac{n}{b^i}\right) = \left(\frac{n^r}{b^{ir}}\right)(\log n - i \log b)$

The total cost can be represented as: $\sum_{i=1}^{i=r} \left(\frac{n^r}{b^{ir}}\right)(\log n - i \log b)$

$= \left(\frac{n^r}{b^r}\right)\left(r \log n - \frac{(r \log b)(r+1)}{2}\right)$

The θ bound for $T(n)$ is

$T(n) = \theta\left(\left(\frac{n^r}{b^r}\right)(r \log n)\right)$

4. Alguss has an array of n currency notes. If the specific pattern is picked, they get to take it home. But only 1 pattern is allowed to be picked. maximize the take home amount.

a) Provide a $O(n^2)$ algorithm to help solve this problem

1. Initialize a variable to store the maximum take home amount set to 0.
2. Iterate through each denomination in the array
3. For each denomination, calculate the amount Alguss would take home if that denomination is picked:
   - Initialize a take home variable
   - Iterate through the array and sum up the notes of the selected denomination into the take home variable initialized
   - If the take home variable is more than the max take home, update the max take home to the take home value
4. Return the maximum take home variable

The time complexity is $O(n^2)$ because for each of the n denominations we iterate through all n currency notes in the array.

b) Give a more efficient algorithm to solve this problem

1. Initialize a hash table to keep track of the total amount for each denomination. The keys of the table will be the denominations, and the values will be the total amount for that denomination
2. Iterate through the array of currency notes
3. For each denomination, update the corresponding value in the table
4. After iterating through all of the notes, find the denomination with the maximum amount in the hash table
5. Return the denomination with the maximum amount

The time complexity is $O(n)$ because we iterate through the array once to calculate the total amount for each denomination using the hash table

5. Let $f: \mathbb{Z} \to \mathbb{Z}$ be a strictly increasing function. Consider an array of n integers. We call an index pair (i,j) a function anomaly if $i<j$ & $f(a_i) > f(a_j)$.

a) calculate function anomalies for array $[13, 71, 19, 7, 3, 5]$

to calculate these anomalies, we can directly check all pairs (i,j) where $i<j$ & $f(a_i) > f(a_j)$.

$A = [13, 71, 19, 7, 3, 5]$

indeces: 1 2 3 4 5 6

$13 > 7, 3, 5 \rightarrow$ 3 elements

1 < 4 5 6 +

$71 > 19, 7, 3, 5 \rightarrow$ 4 elements

2 < 3 4 5 6 +

$19 > 7, 3, 5 \rightarrow$ 3 elements

3 < 4 5 6 +

$7 > 3, 5 \rightarrow$ 2 elements

$4 < 5, 6$ +

$3 \not> 5 \rightarrow$ 0 elements

$5 < 6$

$3 + 4 + 3 + 2 + 0 + 0 = \underline{12}$

The elements that satisfy the definition for a function anomaly are shown on the right. When they are all added together, there are a total of 12 function anomalies

b) come up with a simple condition for when an index pair is an anomaly for a given array.

We can observe that for any array A, an index pair (i, j) is a function anomaly for f if and only if $i < j$ & $A[i] > A[j]$

c) provide an $O(n^2)$ algorithm using the information above that computes the number of function anomalies on a given array

1. Initialize a counter variable and set it to 0
2. Iterate through each element in the array ($A[i]$)
   - For each $A[i]$ iterate through all elements ($A[j]$) where $j > i$
   - Increment the counter every time $A[i] > A[j]$
3. After iterating through all elements, return count, which is the number of function anomalies

d) Come up with a more efficient algorithm using divide & conquer

1. Use merge sort but count the number of inversions during the merge step
2. if $A[i] > A[j]$ (i = index of left subarray & j = index of right subarray) during the merge step, it is a function anomaly
3. Count the total number of anomalies during the merge step
4. Recursively apply the merge sort algorithm on the left and right subarrays
5. Return the total count of anomalies during the merge steps

the time complexity of this algorithm is $O(n \log n)$ as merge sort takes $O(n \log n)$ time and counting the function anomalies during the merge step takes $O(n)$ time

6. consider an array of n integers. sort this array with the SELECT subroutine:
- use SELECT to pick out elements having ranks $\frac{n}{k}, \frac{2n}{k}, \frac{3n}{k}, \ldots, \frac{kn}{k}$
- Use elements picked out in the previous step as pivots, partition the array into k groups
- Apply mergesort to sort each of the B_i's

a) Prove/disprove this algorithm actually sorts an array

Try with an example array $A = [5,1,4,2,3]$ $n=5$, $k=2$

Step 1: Use SELECT with $k=2$, pick elements at ranks $\frac{5}{2}=2$ & $2(\frac{5}{2})=5$, which are $A[2]=1$ & $A[5]=3$

Step 2: Partition the array into 2 groups B_i for $1 \leq i \leq k$:

$B_1 = \{5,1\}$ (elements $A[5]=3$ & $A[1]=1$)

$B_2 = \{4,2\}$ (elements $A[4]=2$ & $A[2]=1$)

Step 3: Apply mergesort to sort each of the B_i's:

$B_1 : \{1,5\}$

$B_2 : \{2,4\}$

Yet after completing the sorting steps, the final array will be $[1,5,2,4,3]$, which is still not sorted. This algorithm partitions the array based on the selected pivots but does not guarentee correct relative ordering of elements within each partition, not fully sorting the array. But, this algorithm is correct in terms of selecting specific elements and sorting them in their selected groups.

b) Analyze the time complexity of the algorithm above as a function of k & n.

- SELECT: $O(n)$, as it finds the kth order statistic in linear time
- partitioning: $O(n)$, partitioning the array into k groups requires iterating through all n elements in the array once and placing them in the corresponding group
- Sorting B_i: $O(\frac{n}{k} \cdot \log(\frac{n}{k}))$, because mergesort is being applied to each of the k groups, each group has approximately $\frac{n}{k}$ elements, and mergesort has a time complexity of $O(n \log n)$ for an array of size n.

the dominating step in the whole algorithm is the sorting of each B_i, which has a time complexity of $O(\frac{n}{k} \cdot \log(\frac{n}{k}))$ time. The overall time complexity would approximately be $O(k \cdot (\frac{n}{k} \cdot \log(\frac{n}{k})))$
$= O(n \cdot \log(\frac{n}{k}))$

C) What values of k would k-selects sort be as efficient as mergesort?

The time complexity should be comparable to $O(n \log n)$ to be as efficient as mergesort. To acheive this, we need to have a k which makes $O(n \cdot \log(\frac{n}{k})) \approx O(n \log n)$. solving for k:
$n \log(\frac{n}{k}) \approx n \log \rightarrow \log(\frac{n}{k}) \approx \log n \rightarrow \frac{n}{k} \approx n \rightarrow k \approx 1$.
when k is around 1, the SELECT algorithm approaches the efficiency of mergesort.

7. Algyss lives in a building with n floors. wants to throw coconuts off. It is safe to throw coconuts off until floor x, $1 \leq x \leq n$ without breaking, but thrown (x+1) onwards will break the coconut.

a) Provide an O(n) time algorithm to find the earliest floor to break the coconut

1. Initialize a floor variable and set it to 1
2. Drop the coconut from the value of floor
3. If coconut breaks, return floor
4. If coconut does not break, increment floor by 1 and repeat steps 2 & 3.

The algorithm terminates when the coconut breaks, leaving us with a time complexity of O(n), the worst case scenario being dropping the coconut from all floors 1 to n.

b) Provide a more efficient algorithm to do the same task as before. Can Algyss execute this algorithm using a single coconut?

1. Initialize variables called low and high, setting low to 1 and high to n, representing the lowest and highest floors the coconut can be dropped without breaking
2. while low is less than high:
   - Initialize a variable called mid which is the middle floor of low and high
   - Drop the coconut from the floor of mid
   - If the coconut breaks, update high to mid
   - If the coconut does not break, update low to mid+1
3. Return the floor when low is equal to high

Algyss may not be able to use a single coconut for this algorithm, because the search space is halved whether the coconut breaks or not. This is the binary search algorithm, and it runs until the space between high and low becomes 0. It has a time complexity of O(log n) since the search space is halved in each iteration.

8. (Extra Credit) Prove/Disprove: $f(n), g(n), h(n)$ are all strictly increasing functions, and $f(n) = O(g(n))$, $g(n) = O(h(n))$, then $f(g(n)) = O(f(h(n)))$

To prove this statement, we need to show that there exists positive constants c & K such that: $|f(g(n))| \leq c \cdot |f(h(n))|$ for all $n > K$.

Because $f(n) = O(g(n))$ & $g(n) = O(h(n))$, there exists positive constants c_1, c_2, k_1, k_2 such that: $|f(n)| \leq c_1 \cdot |g(n)|$ for all $n > k_1$ and $|g(n)| \leq c_2 \cdot |h(n)|$ for all $n > k_2$. Considering $f(g(n))$,

$$|f(g(n))| \leq c_1 \cdot |g(n)| \leq c_1 \cdot (c_2 \cdot |h(n)|) = (c_1 \cdot c_2) \cdot |h(n)|$$

(first inequality) (second inequality)

$(c_1 \cdot c_2)$ is a positive constant, and $|f(g(n))|$ is bounded above by this constant multiplied into $|h(n)|$ for all n greater than some k (which is the maximum of k_1 and k_2). Hence, we can conclude that $|f(g(n))| = O(|h(n)|)$, which in turn proves the statement $f(g(n)) = O(f(h(n)))$.