Brendan Semeraro 7/23/23

Exam 1

1) I Pledge my honor I have neither received nor given any help on this exam.

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

$\ln(2^{\sqrt{n}}) \leq \ln(C*(\sqrt{2})^n)$

$\ln(a^b) = b*\ln(a)$

$(\sqrt{n}) \cdot \ln(2) \leq n*\ln(C \cdot \sqrt{2})$

$(\frac{\sqrt{n}}{n}) \cdot \ln(2) \leq \ln(C \cdot (\sqrt{2}))$

as n approaches infinity the left side will converge to 0 and the $\sqrt{n}$ divided by n is 0, so $0 \leq \ln(C \cdot (\sqrt{2}))$

$\ln(C \cdot (\sqrt{2}))$ is constant so $0 \leq D$

This is always true no matter the value of D

$2^{\sqrt{n}} = O((\sqrt{2})^n) =$ true

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

must check growth rates of the functions

$n^{\sqrt{n}}$ will grow faster than $(\log n)^n$

log of $n^{\sqrt{n}} = \sqrt{n} \cdot \log(n)$

log of $(\log n)^n = n \cdot \log(\log n)$

as n approaches infinity, $\sqrt{n} \cdot \log(n)$ grows much faster than $n \cdot \log(\log n)$ because $\log(n)$ will grow faster than $\log(\log n)$ and $\sqrt{n}$ grows faster than n

so, $n^{\sqrt{n}} = o((\log n)^n)$ is FALSE

c) $n^{n!} = \Theta((n!)^n)$ FALSE

$n^{n!}$ 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 $\Theta((n!)^n)$

$n^{n!}$ is $O((n!)^n)$ as $(n!)^n$ doesn't have an upper bound on $n^{n!}$, but the reverse isn't true, therefore FALSE

4) a) function MaxDenomination(array)
max_amount = 0
max_denomination = 0
for i = 0 to n-1
denomination = array[i]
current_amount = 0
for j = 0 to n-1
if array[j] == denomination
current_amount += array[i]
if current_amount > max_amount
max_amount = current_amount
max_denomination = denomination
return max_denomination

algo compares each denom with all other denom's in the list. If current denom is same as denom it's compared to, adds the value of denom to current amount after checking all denom's if current amount is higher than max amount, max amount and max denom get updated, at end, max denom is returned

b) function MaxDenomination(array)
denominationMap = new Map()
for i = 0 to n-1
denomination = array[i]
if denominationMap contains denomination
denominationMap[denomination] += denomination
else denominationMap[denomination] = denomination
max_Denomination = getMaxKey(denominationMap)
return max_denomination
function getMaxKey(map)
max_key = null
max_value = -1

... Continued

...Continued

```
for each key in map
  if map[key] > max_value
      max_value = map[key]
      max_key = key
      return max_key
```

algorithm works by creating a map, the keys are the denominations and the values are the total amount for each domination, next finds key with the max value in the map and returns it This denomination gives Alguss the maximum amount

5) a) (13,7) 13>7 index 13 < index 7 — 10 anomalies in array
(13,3) 13>3 index 13 < index 3 — A=[13,71,19,7,3,5]
(13,5) 13>5 index 13 < index 5
(71,19) 71>19 index 71 < index 19
(71,7) 71>7 index 71 < index 7
(71,3) 71>3 index 71 < index 3
(71,5) 71>5 index 71 < index 5
(19,7) 19>7 index 19 < index 7
(19,3) 19>3 index 19 < index 5
(19,5) 19>5 index 19 < index 5

b) simple condition where an index pair (i,j) is a function anamoly for f for a given array A is:
a pair (i,j) will be a function of anamoly if i<j and A[i] > A[j]

c)
```
def count_anomalies(A):
    n = len(A)
    anomalies = 0
    For i in range(n):
        for j in range(i+1, n):
            if A[i] > A[j]:
                anomalies += 1
    return anomalies
```

algorithm works by count-anamolies taking array A as input, length of array is stored in n and counter for anamolies is set to 0 loop over array with index i for every i we start another loop from "i+1" til end of array with index j. Each pair (i,j) checks A[i] > A[j] if yes, anamoly found and counter increases after all pairs checked, returns total count of anomalies

d)
```
def merge_count(array,start,mid,end):
    temp= [0] * (end- start +1)
     i= start
     j=mid+1
     k=0
     inv_ count=0
While  i<= mid  and j <= end:
 if array[i] <= array[j]:
  temp [k] =array[i]
   k+ =1
   i +=1
While j<= end:
temp[k] = array [j]
  k+= 1
  j+=1
for  i in range(start, end +1):
  array[i] = temp[i- start]
  return  inv_count
  def merge_sort_and_count(array, start, end):
  inv_count =0
  if start < end:
   mid =(start +end)//2
   inv_count +=merge_sort_and_count(array, start, mid)
   inv_count  +=merge_sort_and_count( array, mid+1, end)
inv_count += merge_ and_count(array, start, mid, end)
  return inv
  A=[13,71, 19,7, 3, 5]
    print (merge_sort_and_count(A,0, len(A)-1))
```

algorithm divides first array into two halves, counts number of inversions in each half recursively and then counts inversions while it merges the two halves

6) a) The algorithm DOES sort the input array A because the Select subroutine is used to find elements with ranks $\frac{n}{k}, \frac{2n}{k}, \ldots, \frac{kn}{k}$ and these elements divide the array into k segments, each with $\frac{n}{k}$ elements, each segment has all element ranks that are between two consecutive selected elements and all elements in one segment are less than the elements in the next segment and then each segment gets sorted individually via mergesort which satisfies all elements within each segment are sorted

b) The time complexity k-select sort algorithm had 3 key components using SELECT to find elements of certain ranks, SELECT is the slowest case time complexity $O(n)$, performed k times so it is $O(kn)$ and then it partitions the array into k segments and iterates through the array and compares each element to the selected pivots and can be done with $O(n)$ time using merge sort, which has time complexity of $O(kn) + O(n) + O(n \log(\frac{n}{k}))$ and simplifies to $O(kn + n \log(\frac{n}{k}))$

c) Merge sort time complexity $O(n \log n)$, k-select algorithm will be as efficient when it's time complexity is also $O(n \log n)$ $O(n \log n) = O(kn + n \log(\frac{n}{k}))$, ignore constants let $\log n = kn + \log(\frac{n}{k})$ for $k=1$, both sides are equal, so k-select is as efficient as merge sort when $k=1$

7) a)
```
def find_breaking_floor(n, is_safe):
    for floor in range(1, n+1):
        if not is_safe(floor):
            return floor
    return n
```

algorithm works by is_safe returns true if a coconut thrown from the floor doesn't break and false otherwise

The function find_breaking_floor throws a coconut from each starting floor from 1 and returns first floor in which it breaks

6) def find_breaking_floors(n, is_safe):
    low = 1
    high = n
    while low < high:
        mid = (low + high) // 2
        if is_safe(mid):
            low = mid + 1
        else:
            high = mid
    return low

this binary search algo is more effective than the linearsearch, but this algo can only be used when there is more than one coconut because if there is only one, once broken binary search won't work

8) false because for $f(n)=n$ $g(n)=n^2$ and $h(n)=n^3$ so $f(n) = O(g(n))$ and $g(n) = O(h(n))$ are true but $f(g(n))=n^2$ and $f(h(n))=n^3$ so $f(g(n)) = O(f(h(n)))$ does n^4 hold true

3) a) $T(n) = O(n^r \cdot \log(n))$ if $0<d<1$, $O(n^r)$ if $d=1$ and $O(n^r \cdot \log(\log(n)))$ if $d>1$. $b^r = b$ $r>0$ and $0<d<1$ so $T(n) = O(n^r \cdot \log(n))$

b) $D=0$ was used because it's the smallest value the Master's Theorem applies to

c) $O(n^r \cdot \log(n))$ for when $0<d<1$ tightest upper bound

d) lower bound is $T(n)$ is $O(n^r)$ because best case is $O(n^r)$ and sub-recursive is $o(n^r)$

e) $T(n)$
n^r $n^r \cdot \log(n)$
n^r n^r $r \times \log(n)$ $n^r \cdot \log(n)$

number of recurrence levels is $\frac{\log(n)}{\log(b)}$

cost at each level is n^r

Theta bound $T(n) = O(n^r \cdot \log(n))$