CSC373 A1: Dinosaurian Dilemmas

Yuchen Shen 1004610664
Zhongliu Liu 1004706936
Yunfei Ouyang 1004284231

October 2020

Q1. Fruit Frenzy

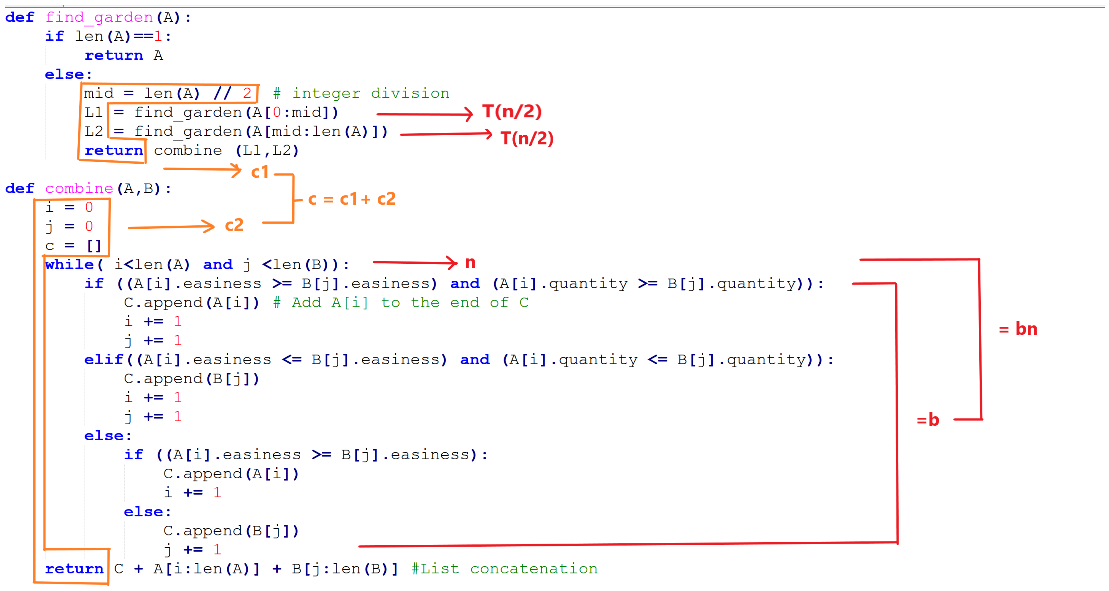
(a) We can put all n fruit gardens into an array $A[1 \cdots n]$
Step 1: Divide A into two equal subarrays like $A[1 \cdots mid]$ and $A[mid + 1 \cdots n]$, and round down, each of size roughly $\frac{n}{2}$.
This will cost constant time $c = O(1)$

Step 2: Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarrays by easiness of reaching in descending order and by number of fruits in ascending order.
For example, if a group has only 1 garden i, then keep it as [i];
If it has two gardens (i and j), then compare their easiness and quantity, if $(e_j > e_i)$ and $(q_j < q_i)$, then keep both in an array [j,i], otherwise, discard the one with both values are smaller (i.e., if $(e_j > e_i)$ and $(q_j > q_i)$, then keep only j in an array [j])
This is $2T(\frac{n}{2})$.

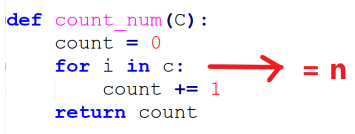
Step 3: Combine by merging the two sorted subarrays back into the single sorted array, since the worse case is no garden discarded in step 2 and we have to go through all n gardens when we merge two subarrays.
This will cost $n \times b = O(n)$ (b is a constant)
Step 4: Now we have final array for all gardens that Bob can pick, so for the number of gardens we just go through down the array and count the total number.
This will cost $O(n)$

Therefore, the total runtime is $T(n) = 2T(\frac{n}{2}) + n \times b + c + n = 2T(\frac{n}{2}) + 2O(n) + O(1)$, i.e., $T(n) = O(n \log n)$

(b) The function showed as below:



```
def find_garden(A):
    if len(A)==1:
        return A
    else:
        mid = len(A) // 2  # integer division
        L1 = find_garden(A[0:mid])
        L2 = find_garden(A[mid:len(A)])
        return combine (L1,L2)

def combine(A,B):
    i = 0
    j = 0
    c = []
    while( i<len(A) and j <len(B)):
        if ((A[i].easiness >= B[j].easiness) and (A[i].quantity >= B[j].quantity)):
            C.append(A[i]) # Add A[i] to the end of C
            i += 1
            j += 1
        elif((A[i].easiness <= B[j].easiness) and (A[i].quantity <= B[j].quantity)):
            C.append(B[j])
            i += 1
            j += 1
        else:
            if ((A[i].easiness >= B[j].easiness):
                C.append(A[i])
                i += 1
            else:
                C.append(B[j])
                j += 1
    return C + A[i:len(A)] + B[j:len(B)] #List concatenation
```



```
def count_num(C):
    count = 0
    for i in c:
        count += 1
    return count
```

According to the above code, there are three functions, first called *find_garden* is designed for step 1, which divides the array to the small subproblems. Then recursively calling function *combine* to sort each small arrays as step 2 described and return the combined single arrays. After these two programs are done, it will return the final array which contains the all gardens Bob can pick. The last function *count_num* will return the number of gardens in final array.
It is easy to find that there are two division costing $2T(\frac{n}{2})$, the while loop for $A + B = n$ total costs bn, and plus the counting part which cost n, so $T(n) = 2T(\frac{n}{2}) + nb + c + n = 2T(\frac{n}{2}) + 2O(n)$, which yields $T(n) = O(n \log n)$

Q2. Missing Member

Since age rangs from set $\{0, 1, \ldots, 2^n - 1\}$, which means all n bits are supposed to have same number of 0s and 1s.

Step 1: Walk up to every dinosaur and ask the least significant bit (say 1-th bit of their age) and split them into two groups depending on whether that bit is a 1 or a 0. Since total number of dinosaurs is an even number $m = 2^n$, we will get one group with $\frac{m}{2}$ dinosaurs, and the other group has $\frac{m}{2} - 1$ dinosaurs, and each of group dinosaurs have the same 1th bit. Then, discard all dinosaurs in the bigger group, and this also tells that the age of missing dinosaur's least significant bit is a 1 or a 0.

Step 2: Now, the smaller group is left ($\frac{m}{2} - 1$ dinosaurs), and we will consider the 2nd bit for their ages. Do the same as Step 1 described, splitting them into two groups again, discarding the bigger group and the smaller of the two group will tell us whether this bit should be a 1 or a 0.

Step 3: Continue with the above steps and keep the groups become smaller and smaller till we have worked out what all the bits are, which is the age of the missing dinosaur.

For the number of queries: $m - 1 + \frac{m}{2} - 1 + \cdots + 1 = 2m + c$ (c is a constent), thus finding the age of the missing dinosaur in $O(m)$ queries.

For the runtime: since each time to splitting them into two parts cost $O(1)$, telling each bit whether a 0 or a 1 from smaller group also cost $O(1)$, plus $O(m)$ times queries, so total run time is $O(m)$ queries.

Q3. Building Hospital

(a) Each society i live in the interval $[s_i\ ,\ f_i]$, where $1 \leq i \leq n$.
Let L be an array consists of n elements $[s_i,\ f_i]$.

Step 1: Sort all societies in array L by the ascending order of f_i, i.e., $f_1\ \leq f_2 \cdots \leq f_i \cdots \leq f_n$. Loop through each element in L.

Step 2: Set the first society as current society and build a hospital at the ending position ($f_{current}$) of the current society, here is at f_0.

Step 3: Do the loop:
Check starting position (s_i) of all rest societies. For each s_i with $i > 1$, if $s_i \leq f_{current}$, which mean this society overlap with the current society, and we do not have to build a new hospital for it, until we find all kinds of overlapped societies. we will skip all of them, and start from the next society (or the leftmost remaining one, i.e., $s_i > f_{current}$) that is set as the new current society and build a new hospital at its $f_{current}$

The following is python pseudocode for this greedy algorithm:

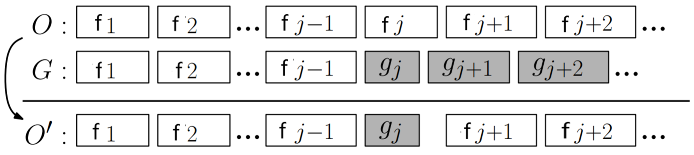
```
num_hospital = 0;
hospital_location = {}
last_hospital_position = - inf

min_num_hospital(L):
    sort L according to fi
    for (1 <= i<= n ):
        if si > last_hospital_position
            hospital_location.append(fi)
            last_hospital_position = fi
            num_hospital += 1
    return num_hospital, hospital_location
```

The time complexity of this greedy algorithm is $O(n \log n)$, because sorting L takes $O(n \log n)$ time (usually, the time complexity of a good sort costs $O(n \log n)$, like merge sort, quick sort, etc), and the for loop on line 5 (loop i from 1 to n) costs $O(n)$. So total is $O(n \log n + n) = O(n \log n)$.

(b) *Proof.*
Claim: This greedy algorithm provides an optimal solution to interval scheduling.
Let $O = (f_1, f_2, \cdots f_k)$ be the optimal solution for locations of hospitals depending on a listed in increasing order of f_i, and let $G = (g_1, g_2, \cdots g_{k'})$ be the solution by our greedy algorithm.
We have to prove O converts to G (i.e., $O = G$).
Since O is optimal, it must contain at least as many hospitals as the greedy algorithm has, and hence there is a first index j (or say leftmost hospital) where these two location of hospital differ. That is, we have:

$$O = (f_1, f_2, \cdots f_{j-1}, f_j, \cdots f_k)$$
$$G = (f_1, f_2, \cdots f_{j-1}, g_j, \cdots g_{k'})$$

where $g_j \neq f_j$. Here, $j \leq k$, since otherwise G would have less hospitals than O, which would contradict O's optimality. Our greedy algorithm selects the locations with the earliest finish time that does not conflict with any earlier hospital. Thus, we know that g_j does not conflict with any earlier hospital selection, and it finishes no later than f_j finishes.So we can replace f_j with g_j in O and get the greedier optimal O' (shows in the following fig)



O : f 1 | f 2 | ... | f $j-1$ | f j | f $j+1$ | f $j+2$ | ...
G : f 1 | f 2 | ... | f $j-1$ | g_j | g_{j+1} | g_{j+2} | ...
O' : f 1 | f 2 | ... | f $j-1$ | g_j | f $j+1$ | f $j+2$ | ...

Obviously, since g_j finishes no later than f_j and it cannot create any new conflicts. Besides, O' has the same number of hospitals as O has, then O' is still valid solution and it is at least as good with respect to our optimization criterion (the optimal solution whose leftmost hospital is exactly where our greedy leftmost hospital is).
Inductive Step: Let $m \in \mathbb{N}$ and $m > 1$, by repeating above process and assume $P(m)$ holds.
$P(m)$: we can replace m leftmost hospitals in O and get the newest O_{new}, all m hospitals are from G (IH).
WTP, $P(m+1)$ holds.
By IH, we conclude that optimal solution O_{new} replaced m hospitals, which means there is no solution to find less replaced hospitals than m in the left side. Thus there is no solution to find less replaced hospitals less that $m+1$.
Hence, $m+1$ hospitals have been matched, i.e., $P(m+1)$ holds, and by induction ,we prove $P(m) \Rightarrow P(m+1)$.
Therefore,we will eventually convert O into G without decreasing the number of hospitals, so G is optimal.

■

Q4. Traveller's Dilemma

a. Let array d be an array consists of n elements d[i] which $1 \leq$ i $\leq$ n. and d[i] stores the distance from the ith charging station to the starting point d[0]. for each d[i], d[i-1] - d[i] $\leq$ 200. Let d[0] = 0. Let charging station list be an empty array C

Step 1: Let the first charging station the dinosaur should visit be d[1], if d[1] == 200, put d[1] into C and move on to the next element in d. Else if d[1] < 200, loop through the array d to find an element d[a] where d[a] < 200 and d[a+1] > 200. Put d[a] into C. Let the current location d[curr] be d[a]/d[1] (the first element in C).

Step 2: Find the next charging station d[i] where d[i] - d[curr] $\leq$ 200 and d[i+1] - d[curr] > 200. Put d[i] into charging station array C.

Step 3: Repeat step 2 and until d[n-1] is checked

The output is charging station array C. Since we get through each element in d, the worse case run time is $O(n)$

The following is python pseudocode for this greedy algorithm:

```
find_charging_locations(d):
    i = 1
    charging_station = []
    curr = 1
    while (i < = n - 1) and (curr <= n - 1):

        if d[i] - d[curr] <= 200 and d[i+1] > 200
```

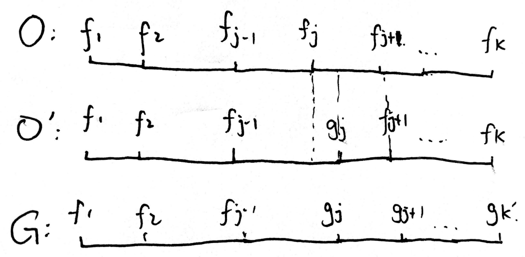
```
            charging_station.append(d[i])
            curr = i
            i = curr
        else
            curr++


    return charging_station
```

Since the while loop at most go through n times, and for each iteration, the cost is constant, so the total worst-case running time is $O(n)$

b. *Proof.*
Claim: This greedy algorithm provides an optimal solution to interval scheduling.
Let $O = (f_1, f_2, \cdots f_k)$ be the optimal solution for choosing the location of charging stations, and let $G = (g_1, g_2, \cdots g_{k'})$ be the solution by our greedy algorithm.
We have to prove O converts to G (i.e.,$O = G$).
Since O is optimal, it must contain at least as many charging stations as the greedy algorithm has, and hence there is a first index j where these two location of charging station differ. That is, we have:

$$O = (f_1, f_2, \cdots f_{j-1}, f_j, \cdots f_k)$$
$$G = (f_1, f_2, \cdots f_{j-1}, g_j, \cdots g_{k'})$$

where $g_j \neq f_j$. Here, $j \leq k$, since otherwise G would have less charging stations than O, which would contradict O's optimality. Our greedy algorithm selects the locations with the maximum total distance that does not conflict with any earlier charging locations. Thus, we know that g_j does not conflict with any earlier charging station selection, and its location is no closer than f_j.So we can replace f_j with g_j in O and get the greedier optimal O' (shows in the following fig)


O: f_1 f_2 f_{j-1} f_j f_{j+1} ... f_k
O': f_1 f_2 f_{j-1} g_j f_{j+1} ... f_k
G: f_1 f_2 f_{j-1} g_j g_{j+1} ... $g_{k'}$

Since g_j can be no closer to starting point compare to f_j which means $g_j \geq f_j$ because the interval that regulates distance of g_j compare to f_{j-1} is the maximum distance and it cannot create any new conflicts to the following locations of O' since the distance between g_j and f_{j+1} must be smaller or equal to 200. The proof is : f_{j+1} - $f_j \leq 200$ and $g_j \geq f_j$ so that f_{j+1} - $g_j \leq 200$. Therefore, the dinosaur can travel successfully from charging station g_j to f_{j+1} without running our of battery. Besides, O' has the same number of charging stations as O has, then O' is still valid solution and it is at least as good with respect to our optimization criterion.

Inductive Step: Let $m \in \mathbb{N}$ and $m > 1$, by repeating above process and assume $P(m)$ holds.
$P(m)$: we can replace m leftmost charging stations in O and get the newest O_{new}, all m charging station locations are from G (IH).
WTP, $P(m+1)$ holds.

By IH, we conclude that optimal solution O_{new} replaced m charging stations, which means there is no solution to find less replaced charging stations than m in the left side. Thus there is no solution to find less replaced hospitals less that $m+1$.
Hence, $m+1$ charging stations have been matched, i.e., $P(m+1)$ holds, and by induction ,we prove $P(m) \Rightarrow P(m+1)$.
Therefore,we will eventually convert O into G without increasing the number of charging stations , so G is optimal.

■

c. For instance, let's assume an array d' with d'[1] = 199, d'[2] = 200, and c[1] = 200, c[2] = 1000, The greedy algorithm tell us d'[2] is charging station to stop, but d'[1] is much better and choosing d'[2] would add up 800 more in total cost. Applying to the same rules in every element in array d', the greedy algorithm wants to make sure for the between each two consecutive stops should maximize as possible so that we can make sure there's the least number of charging stations we need to visit, whereas there might be stops between the two stops we choose that possess lower cost, so the greedy algorithm described above does not minimize the total cost of travel.

Q5. No More Slaps, Please!

We will design the algorithm by first design an algorithm for k = 1.
For $k = 1$: i.e. I can only withstand at most 1 dino-slaps, that is if any dinosaur slapped me for the second time, I am done. Let L be a list of dinosaurs sorted in a non decreasing order of their age and that $len(L) = n$

Step 1: We will first record m = $\lceil\sqrt{n}\rceil$, then L can be separate into at most n/m parts, where each part is at most m length.
i.e. the first part is $L[: m]$, the second part is $L[m+1 : 2m]$, ... the last part is $L[n\%m+1 :]$.

Step 2: We can start asking once we split the list. We will ask the last dinosaur in the first part of the list, so it is the dinosaur at $L[: m][-1]$. We asks: "Are you x years old?".
there are three cases:
case 1: Dinosaur says: "Oh my, I'm flattered. But no, I'm younger than x."
Then we proceed to the next part of the list, ask the last dinosaur in the next part the same question.
case 2: Dinosaur says: "Why yes! How did you know that?"
Then we get the answer, this dinosaur is the dinosaur with x age.
case 3: Dinosaur gets offended and slaps you
The dinosaur slaps me, so $k-1=0$. I can withstand at most 0 dino-slaps now.
Since the dinosaur slapped me, we know that he is older than the age x, thus we can ask other dinosaurs in that part of the list. We begin our ask to the first dinosaur with the ascending order of the index. We will find the dinosaur with age x with at most $\lceil\sqrt{n}\rceil$ asks.

The following is the pseudo code for k = 1:

```
L = [list of dinosaurs sorted in a non decreasing order of their age] where Len(L) = n

find_dinosaur_age(x):
    m = ceil(sqrt{n})
    subList = []

    for i = 0, i < ceil(n / m), i ++:
        last_dino = L[i*m: (i+1)*m][-1]
        if last_dino.age == x:
            return last_dino
        else if last_dino.age > x:
            k -= 1
            subList = L[i*m: (i+1)*m]
            break
        else:
            pass

    for d in subList:
        if d.age == x:
            return d
```

Run time:
For the first loop, it runs at most $\lceil \sqrt{n} \rceil + 1$ times.
For the second loop, it runs at most $\lceil \sqrt{n} \rceil + 1$ times.
Therefore the run time of this algorithm is $(2\lceil \sqrt{n} \rceil + 2) \in O(\sqrt{n})$.

Now, we can generalize this case, and write a recursive algorithm for arbitrary k in $1 < k < log(n)$. Note that for $k >= log(n)$, we can use binary search which is $O(log(n))$.

Step 1: Let $m = \lceil n^{(2^k-1)/2^k} \rceil$. Then similarly to the case $k = 1$, L can be partition into at most n/m parts where each part is at most m length.

Step 2: The following recursive function takes (x, k, L) as the initial input.
Then for each time we got slapped, i.e. $k-1$. We recursive call the function on sub list where we got slapped with parameters (x, $k-1$, and that sub list's location), thus this recursive algorithm will guide us to the dinosaur with age x within the k slaps.

The following is the pseudo code for $1 < k < log(n)$:

```
L = [list of dinosaurs sorted in a non decreasing order of their age] where Len(L) = n

find_dinosaur_age(x, k, L):
    n = len(L)
    m = ceil(n^{(2^{k}-1)/2^{k}})

    for i = 0, i < ceil(n/m), i++:
        last_dino = L[i*m: (i+1)*m][-1]
        if last_dino.age == x:
            return last_dino
        else if last_dino.age > x:
            subList = L[i*m: (i+1)*m]
            find_dinosaur_age(x, k - 1, subList)
        else:
            pass

    for d in subList:
        if d.age == x:
            return d
```

Run time:
Since the recursive call is inside the for loop, I do not know if the run time of this algorithm can be generalized by applying the Master theorem. Therefore I will try to find a pattern by looking at the run time of this algorithm for k = 2, k = 3, k = 4...

For k = 2: The length m of each sub list is $m = \lceil n^{\frac{2^k-1}{2^k}} \rceil = \lceil n^{\frac{3}{4}} \rceil$. Then i iterates from 0 to $\lceil n/m \rceil$ which is at most $\lceil n^{\frac{1}{4}} \rceil$, at the last iteration of i, the algorithm recursive call find_dinosaur_age(x, 1, subList) where subList is m length.
- Recursive call for k = 1: Since in our pseudo code for k = 1, we have the run time of $O(\sqrt{n})$, therefore the recursive call runs $O(\sqrt{n^{\frac{3}{4}}}) = O(n^{\frac{3}{8}})$ where $n^{\frac{3}{4}}$ is the sublist length in k = 2.

Thus the total run time of this algorithm for k = 2 is $O(n^{\frac{1}{4}} + n^{\frac{3}{8}}) \in O(n^{\frac{3}{8}})$.
Similarly, for k = 3, the run time is $O(n^{\frac{21}{64}})$.
For k = 4, the run time is $O(n^{\frac{315}{1024}})$.
...
By inspection, we can deduce a general run time for this algorithm.
For $k = m$, the run time is $O(n^C)$ where $c = \prod_{i=1}^{m}(\frac{2^i-1}{2^i})$.
Since $\prod_{i=1}^{m}(\frac{2^i-1}{2^i}) \leq \frac{1}{2}\forall m$, we have the run time of the above recursive algorithm $O(\sqrt{n})$.
Last, I believe that this algorithm will get very close to $O(log(n))$ time similarly to binary search when k is arbitrarily large.