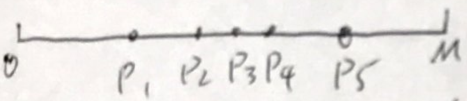
Liang Liu
153182750

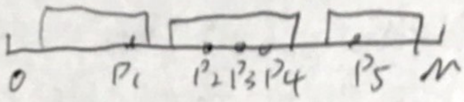
312 A4

1. (a) Suppose there are 5 points on the line $0 < P_1 < P_2 < P_3 < P_4 < P_5 < M$ and ~~$P_4 - P_3 - P_2$~~ $P_4 - P_2 < P_5 - P_4 = P_2 - P_1$ $\in$ ~~in~~ $[x_j, x_{j+1}]$ which means P_2, P_3, P_4 are close point, the graph shown as:



By greedy algorithm select an interval cover lagest number of still-uncovered points, so first interval would be $P_2, P_3 P_4 \in I_1 \in [x_j, x_{j+1}]$.
second interval would be either P_1 or P_5
and same as thrid interval

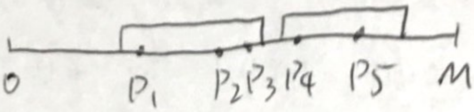
graph for greedy algorithm: G_1



3 intervals

But there's a better way, because $P_3 - P_1 \in [x_j, x_j + 1]$ & $P_5 - P_5 \in [x_j, x_{j+1}]$

so graph for this way: G_2

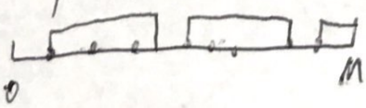


2 interals

So ~~Int~~ number of interval in G_2 less than G_1

(b) what I thought for the greedy algorithm is:
the start of interval always select the most ~~ri~~ left point that for still-uncovered points

graph shown as:



correctness: for greedy algorithm in (a), when all the points are selected in n interval followed by lorgest number of still-uncovered points, if there're more points, the greedy algorithm in (a) must select the start of interval before next point, but if we choose the start of interval on the left most uncovered point, then no other points ~~is~~ covered by interval using greedy algorithm in (a) can not be covered in greedy algorithm in (b), so this greedy algorithm is an optimal solution.

2. (a) if A = [2 , 1 , 3 , 50 , 10]

the optimal solution : A[0] + A[3] = 52

(b) A = [1, 70, 100, 80, 10]

greedy algorithm: A[2] = 100 , A[0] = 1 , A[4] = 10

Total = 100 + 1 + 10 = 111

optimal solution: A[1] = 70 , A[3] = 80

Total = 70 + 80 = 150 > 111

so ~~opt~~ greedy algorithm can not always give the best solution

(c) ~~Assume A [0, n~~ ~~A has n elements~~

Assume Function f(n) ~~the~~ n is the number of index in array A when we go last three elements, we compare the last second and last third which are f(n-1), f(n-2). And then we need to add A[n] to f(n-2) because we can't add on f(n-1) because of rule.

(d) recurrence rule:

n = len (A)

if (n = 0) then

f(n) = 0

else

f(n) = max(f(n-1) , f(n-2) + A[n])

(e) use bottom to up to create algorithm

O(1)
```
B = [n]
B[0] = A[0]
B[1] = max(A[0], A[1])
```

O(n)
```
for i from 2 upto n-1
        B[i] = max( B[i-1] , A[i] + B[i-2] )
return B[i-1]
```

(f) T(n) = # calls · t = O(n)

3. (a) Assume X is a majority element in A, which means the times of X appears $> \frac{n}{2}$

if X is not a majority element in the first half on the array ~~or~~ or second half of the array, the times of X appears $< \frac{n}{4}$

so the times of X appears in A $< \frac{n}{2}$, which is not $> \frac{n}{2}$

so if X is a majority element in A, it must be a majority element in first half of A or second half of A

(b)
```
n = len(A); a=0; b=0;
majority(A[n])
mid = n/2 - 1                                   — O(1)
Left = majority(A[0, mid]        ]
right = majority(A[mid+1, n]     ] 2T(n/2)
if Left = right then             ]
    return Left                  ] O(1)
else
    for i=0 to mid do            ]
        if A[i] = Left then      ] O(n)
            a++                  ]
    od
    for j=mid+1 to n do          ]
        if A[j] = right then     ] O(n)
            b++                  ]
    od
    if a > b then                ]
        return Left              ] O(1)
    else
        return right
```

~~the recursion part will execute~~

$T(n) = 2T(\frac{n}{2}) + O(n)$

use master THM

Time complexity is $O(n\log n)$

First split array into 2 subarray of half size then do linear search by using 2 for loops to find the majority element then return.

由 扫描全能王 扫描创建

4. Three operations can be use in algorithm

Root(T), LeftChild(T, v), rightChild(T, v)

Basically I'm thinking doing like BST to solve this problem, but the change is when it gose to left most of unupdate nood it will update the ~~height~~ diameter by finding the maxium of its ~~diameter~~ height and the sum of its children plus one, and return. The main point is to find the longest path pass the current root.

~~D=0~~

~~Diameter(Root(T), D)~~

~~LH = Diamete~~

~~if (Root(T) = NULL) then~~

~~return 0~~

~~LH = Diameter(leftChild(T, v))~~

~~RH = Diameter(rightChild(T, v))~~

~~D = max(D, LH + RH + 1)~~

~~return max(LH, RH) + 1~~

n = Root(T)

Diameter(n)

$O(1)$ | if n is null

return 0

~~path~~

d = 0

path(n, d)

$O(1)$ | if n is null

return 0

$2T(\frac{n}{2})$ | LH = path(leftchild(T,v), d)

RH = path(rightChild(T,v), d)

$O(1)$ | d = max(d, LH + RH + 1)

return (max(LH, RH) + 1)

return (d)

$T(n) = 2T(\frac{n}{2}) + O(1)$

By master THM

$T(n) \in O(n)$