CS 6150: HW 2 – Divide & Conquer, Dynamic Programming

Submission date: Tuesday, Sep 22, 2023 (11:59 PM)

This assignment has 5 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Binary search reloaded	12	
Sub-division in the selection algorithm	8	
Finding a frequent element	10	
Let them eat cake	10	
Maximum Weight Independent Set	10	
Total:	50	

Question 1: Binary search reloaded ..... **[12]**

For both the parts below, let $A[0], A[1], \ldots, A[n-1]$ be an array of size n, where $n > 2$.

(a) **[6]** First, suppose $A[i]$ is either $+1$ or -1 for every i. Let $A[0] = +1$ and $A[n-1] = -1$. The goal is to find an index i such that $A[i] = +1$ and $A[i+1] = -1$. (If there are many such indices, finding one suffices.) Give an algorithm for this task that makes only $O(\log n)$ queries to the array A.

Ans)

Let $l = 0$ *and* $h = n - 1$
$search(l, h)$
$mid = (l + h)/2$
let $i = mid$, $i + 1 = mid + 1$
$if\ A[i] = 1$ && $A[i + 1] = -1$
$return\ i$
$else\ if\ A[i] = 1$ && $A[i + 1] = 1$
let $l = mid + 1$
$return\ search(l, h)$
$else\ if\ A[i] = -1$ && $A[i + 1] = -1$
let $h = mid - 1$
$return\ search(l, h)$

(b) **[6]** Next, suppose that A is an array containing distinct integers. The goal is to find a "local minimum" in the array, specifically, an array element that is smaller than its neighbors (i.e., i such that $A[i-1] > A[i] < A[i+1]$). For the end points of the array, we only need to compare with one neighbor (i.e., $A[0]$ counts as a valid solution if $A[0] < A[1]$ and likewise with $A[n-1]$).

Give an algorithm that makes only $O(\log n)$ queries to the array and finds some local minimum in the array. [*Hint:* can you use part (a)?]

Ans)

Let $l = 0$ *and* $h = n - 1$
$Search(l, h)$
$mid = (l + h)/2$
let $i = mid$, $i + 1 = mid + 1, i - 1 = mid - 1$
$if\ A[i] < A[i + 1]$ && $A[i] < A[i - 1]$
$return\ i$
$else\ if\ A[i] < A[i + 1]$&& $A[i] > A[i - 1]$
$h = mid - 1$
return (search(l,h))
$else\ if\ A[i] > A[i + 1]$ &&$A[i] < A[i - 1]$
$l = mid + 1$
return (search(l,h))

Question 2: Sub-division in the selection algorithm ..... **[8]**

Recall the linear time selection algorithm we saw in class (median-of-medians, lecture 5) and answer the following questions. Please provide detailed justification.

(a) **[4]** Instead of dividing the array into groups of 5, suppose we divided the array into groups of 7. Now sorting each group is slower. But what would be the recurrence we obtain? [*Hint:* What would the size of the sub-problems now be?]

(Bonus [3]) Analyze the runtime of your new recurrence. Is it still linear?

Ans)

We divide this problem in to 7 sub problems.
Number of sub problems created would be n/7.
Median would be near to 1/2*(N/7).
Median would be near n/14.
Let the median be x.
The elements smaller than x would be 4n/14.

Elements greater than x would be 4n/14 as well.
The position of that would be n-4n/14=10n/14.
so x lies between 4n/14 and 10n/14.
Taking the worst case where k is near 4n/14 and we have to search in the whole line. That would be 10n/14.
T(n)=T(n/7)+T(10n/14)+cn
Here T(n/7) would be the time for finding median
T(10n/14) would be time to find median in the worst case.
cn would be time to sort elements.
$T(n) = T(n/7) + T(5n/7) + n$
Using Akra Bazzi
$\Sigma_{i=1}^{k} a_i T(b_i n) + g(n)$
$k = 2, a_1 = 2, b_1 = 1/7, a_2 = 1, b_2 = 5/7, g(n) = n$
$\Sigma_{i=1}^{k} a_i * {b_i}^p = 1$
$a_1(b_1)^p + a_2(b_2)^p = 1$
$1(1/7)^p + 1(5/7)^p = 1$
$p \simeq 0.7632$
$T(n) = \theta(n^p(1 + \int_1^n g(x)/x^{p+1}))dx$
$T(n) = \theta(n^{0.7632}(1 + \int_1^n x/x^{0.7632+1}))dx$
$T(n) = \theta(n^{0.7632}(1 + \int_1^n x^{-0.7632}))dx$
$T(n) = \theta(n^{0.7632}(1 + 4.22(n^{0.2368} - 1)$
$T(n) = \theta(n^{0.7632} + 4.22n - 4.22n^{0.7632})$
$T(n) = \theta(n)$

(b) [**4**] Say we defined "almost median" differently, and we ended up with a recurrence $T(n) = T(5n/7) + T(n/5) + cn$. Does this still lead to a linear running time?
Ans)
$T(n) = T(5n/7) + T(n/5) + n$
Using Akra Bazzi
$\Sigma_{i=1}^{k} a_i T(b_i n) + g(n)$
$k = 2, a_1 = 1, b_1 = 5/7, a_2 = 1, b_2 = 1/5, g(n) = n$
$\Sigma_{i=1}^{k} a_i * {b_i}^p = 1$
$a_1(b_1)^p + a_2(b_2)^p = 1$
$1(5/7)^p + 1(1/5)^p = 1$
$p \simeq 0.8589$
$T(n) = \theta(n^p(1 + \int_1^n g(x)/x^{p+1}))dx$
$T(n) = \theta(n^{0.8589}(1 + \int_1^n x/x^{0.8589+1}))dx$
$T(n) = \theta(n^{0.8589}(1 + \int_1^n x^{-0.8589}))dx$
$T(n) = \theta(n^{0.8589}(1 + 7.08(n^{0.1411} - 1))$
$T(n) = \theta(n^{0.8589} + 7.08n - 7.08n^{0.8589})$
$T(n) = \theta(n)$

Question 3: Finding a frequent element ..... **[10]**

Suppose we have an array $A[0], A[1], \ldots, A[n-1]$ consisting of objects for which there is an efficient "equality test", but no comparison. More precisely, we can tell if $A[i] = A[j]$ in constant time, but do not have access to any other operations. Now, suppose that the array has a "majority element", i.e., suppose there is some x such that $A[i] = x$ for $> n/2$ distinct indices i. The goal is to find such an element.

(a) [**4**] Consider the following algorithm: pick r elements of the array at random (uniformly, with replacement), and for each chosen element $A[j]$, count the number of indices i such that $A[i] = A[j]$. If one of the elements occurs $> n/2$ times, output it, else output FAIL. This takes $O(nr)$ time. Prove that the probability of the algorithm outputting FAIL is $< \frac{1}{2^r}$. (Remember that we are promised that the array has a majority element.)

Ans)
Majority element mean that the probability of choosing that will be more than 0.5..
Success of major would be more than 0.5
Therefore probability of selecting minor would be less than 0.5
taking r trials would mean that probability would be less than $(1/2)^r$
Let's take an example
A=[1,1,1,1,1,1,4,5,6,8]
P(1)=6/10 and P(4)=P(5)=P(6)=P(8)=1/10
Now we create a new random array which in which we take random elements from the first array.
Here r represents number of times elements taken from the original array copied and put back.
Fail condition is when the element chosen is not majority.
P(4)=1/10
Taking 3 tries
$P(4)^r = (1/10)^3$
P(4)=0.001 which is less than 0.5

(b) [**6**] Now give a deterministic algorithm (one that always finds the right answer) for the problem that runs in time $O(n \log n)$. [Hint: divide and conquer.]

Ans)
Since We cannot compare elements with each other, we can't use sorting in this question.
We will apply something same as merge sort to this.
We are going to divide the array of say size n.
We will divide it into 2 parts of size n/2 and n/2.
We will then put them in function f which would return the element with highest frequency.
Say f would return element m1 for first half and m2 for second half.
Important: consider an array A[1,1,1,1,1,1,4,5,6]
If we divide the array the sub-arrays would be[1,1,1,1,1] and[1,4,5,6,8]
So m1 here would be 1 for first array and m2 for other array would be 1/4/5/6 as they have the same frequency.
We will merge this array and we will get the same result, 1.

Algorithm
Majorelement(A,l,h)
m=(l+h)/2
left.major.element=Majorelement(A,l,m)
right.major.element=Majorelement(A,m+1,h)
if left.major.element==right.major.element
return left.major.element
left.major.freq=getfreq(A,left.major.element)
right.major.freq=getfreq(A,right.major.element)
if left.major.freq¿n/2
return left.major.freq
else
return right.major.freq
Complexity would be O(nlogn)

Question 4: Let them eat cake . . . . . **[10]**

Alice has a cake with k slices. Each day, she can eat some of the slices, and save the rest for later. If she eats j slices on some day, she receives a "satisfaction" value of $\log(1 + j)$ (note the diminishing returns). However, due to imperfect preservation abilities, each passing day results the loss of value by a factor β (some constant smaller than 1, e.g., 0.9). Thus if she eats j slices on day t, she receives a satisfaction of $\beta^{t-1} \log(1 + j)$.

Given that Alice has k slices to start with, given the decay parameter β, and assuming that she only eats an integer number of slices each day, give an algorithm that finds the optimal "schedule" (i.e., how many slices to eat and which days). The algorithm must have running time polynomial in k.

Ans)
We have k slices.They take $\beta^{t-1}\ \log(1+j)$ *time to finish off.*
t=the day, j= number of sl(ices.
k= total number of slices
Let β *be some constant* < 1
Say we have 5 slices
There are many ways to eat it.
5 slices in 1 day
4 slices in 1 day 1 slice in next day
3 slices in 1 day 2 slices in next day
2 slices in 1 day 3 slices in next day
1 slice in 1 day 4 slices in next day
We can see that few steps have their own way.
Like when we eat 2 slices, it can be divided into 2 in 1 day and 0 slices in the next day. As well as 1 slice in 1 day and 1 slice the next day.
Same for 3 it can be divided into 2,1 as well as 1,1,1 as well as 2,1 as well as 3,0
Then same for 2 it can be divided into 1,1 and 2,0
Therefore we can see that subproblems get repeated.
Therefore we apply dynamic programming.

Algorithm
for i 1 to k
for j 1 to i
let variables be max score, max set
score=log(1+j)+$\beta(dp(i-j))$

$if(score > maxscore)$
$maxscore = scoremaxset = dp(i-j)+j$
$dp(i) = maxscore$
$dpset(i) = maxset$
We can see that the algorithm's time complexity would be O(k^2) *since there are two for loops.*

Question 5: Maximum Weight Independent Set . . . . . **[10]**

Let $G = (V, E)$ be a (simple, undirected) graph with n vertices and m edges. Also provided is a weight function w that maps vertices to real valued weights. An independent set is a subset of the vertices with no edges between them. Formally, an independent set is a $S \subseteq V$ such that for all $i, j \in S$, $\{i, j\} \notin E$. The total weight of an independent set is defined to be $\sum_{i \in S} w(i)$.

Now, suppose G is a rooted tree, and suppose we have a weight function on its vertices. Give an algorithm that finds the independent set of largest total weight in G. (If there are many such sets, finding one suffices.) The algorithm should run in time polynomial in n.

Ans)
Algorithm
$dp = [< 0, >, < 0, >, < 0, > ..... < 0, >]$
first part is weight and second part is empty set.
We consider weights to be positive
f(root)
(if dp(root).set is not empty)
return dp(root)
weight and set are returned
if root is leaf
dp(root).set=root
dp(root).weight=weight of root
return dp(root)

includeset=root
includeweight=weight of root
excludeset=empty
excludeweight=0
for grand child of root
weight,set=f(grandchild)
includeweight += weight
includeset=add(set,empty)
for child of root
weight,set=f(child)
excludeweight+=weight
excludeset=add(set,empty)
if includeweight¿excludeweight
dp(root)=include weight,include set
else
dp(root)=exclude weight,exclude set
return(root)

Explanation

A tree would have root,child and grandchild
we would return those nodes which are not connected by edges
if we choose root, we have to exclude child and choose grandchild. Child is directly connected whereas grandchild is not therefore it is included.
f(root)=maxroot U f(grandchild),U f(child)
This means when we put root in function we get root and union recursive function of grand child OR union f(child)
When we choose child we got for child and greatgranchild nodes
We can see the subproblems get repeated when we choose f(child) it gives child and greatgrandchild and f(root) also give child
Therefore we go for DP.