CS 6150: HW 2:Payman Behnam U1078370 – Divide & Conquer, Dynamic Programming

Submission date: Friday, Sep 20, 2019, 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.

Answer:

We can use Binary Search. We choose a middle element and compare it with its neighbors (the element right before and after it).

Suppose that the middle element is A[mid] = +1. If A[mid+1]= -1 we are done. If A[mid+1]= +1, we will look at the right part. We are sure that finally we can find two elements that A[i]=+1 and A[i+1]=-1. The reason is that the most right element $A[n-1] = -1$. So the worst case would be all the element in the right half are +1 except the last one. So we can find at least on pair such that $A[i] = +1$ and $A[i+1] = -1$ (i.e., $A[n-2] = +1$, $A[n-1] = -1$]).

Now suppose that the middle element is A[mid] = -1. If A[mid-1]= +1 we are done. If A[mid-11]= +1, we will look at the left half. We are sure that finally we can find two elements that A[i]=+1 and A[i+1]=-1. The reason is that the most left element $A[0] = -1$. So the worst case would be all the element in the left half are -1 except the last one. So we can find at least on pair such that $A[i] = +1$ and $A[i+1] = -1$ (i.e., $A[0] = +1$, $A[1] = -1$]).

Like binary search we can follow the above rule until we get at least one favorite pair. Since in each step we are diving the interval to two, like binary search we get to the $O(\log n)$. With minor modification, algorithm in 1-b works for 1-a as well.

(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)?]

Answer: We can use Binary Search like part "a". We choose a middle element and compare it with its neighbors (the element right before and after it). Since the elements are distinct, If middle one is less than its neighbors, then we can return it and the algorithm is finished. If the middle element is less than its left neighbor but greater than its right neighbor, then the local minimum is in right half. The reason is that if we just look at the right half, the two possibilities may happen. Either we have a full descending function or a combination of a descending and ascending function. If this is a full descending function, the last element A[n-1] is the local minimum considering that for the end points of the array, we only need to compare with one neighbor. Since this is a full descending function, the last element would be less than its neighbor (i.e., the right one). If there is a combination of descending and ascending, we have for sure a local minimum since there would be a point that connects descending and ascending (i.e., like a minimum point of a concave).

If the middle element is greater than its left neighbor, but less than its right neighbor, then the local minimum is in left half. The reason is kind of the same as previous one. The reason is that if we just look at the left half, the two possibilities may happen. Either we have a full ascending function or a combination of a descending and ascending function. If this is a full ascending function , the first element A[0] is the local minimum considering that for the end points of the array, we only need to compare with one neighbor. Since this is a ascending function, the fist element would be lass than its neighbor. If there is a combination of descending and ascending, we have for sure a local minimum since there would be a point that connects ascending and descending (i.e., like a minimum point of a concave).

Like binary search we can follow the above rule until we get at least one local minimum point. Since in each step we are diving the interval to the two, like binary search we get to the $O(\log n)$.

```
localMinimumFunction(int arr[], int low, int high, int n){
Find index of middle element
int mid = low + (high - low)/2; /* (low + high)/2 */
if ((mid == 0 || arr[mid-1] > arr[mid]) && (mid == n-1 || arr[mid+1] > arr[mid])) then
  | return mid
else
  | if(mid > 0 && arr[mid-1] < arr[mid]
  | return localMinimumFunction(arr, low, (mid -1), n)
end
return localMinimumFunction(arr, (mid + 1), high, n);
}
```

Algorithm 1: Local Minimum Search

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 3. Now sorting each group is faster. But what would be the recurrence we obtain? [*Hint:* What would the size of the sub-problems now be?]

Answer:

If the number of elements in each group is 5 and we have $\frac{n}{5}$ groups then the time complexity of the median-of-medians algorithm is $O(n)$. If K is the set of all medians of each group, then K has $\frac{n}{5}$ members. We know that $\frac{1}{2}$ element of the set K is smaller than median of the set K which equals to $\frac{1}{2} \times \frac{n}{5} = \frac{n}{10}$. Since each of $\frac{n}{10}$ member in the list are the median of a 5- member group and two elements are less then them , there would be $\frac{2n}{10} + \frac{n}{10} = \frac{3n}{10}$ in the set K that are less then the median of set K. Hence, in the worst case, the algorithm need to do recursion on the remaining $\frac{7n}{10}$. The overall recursion is $T(n) \leq T(n/5) + T(7n/10) + O(n)$. According to the class lecture, the complexity of $T(n)$ is $O(n)$.

The median-of-medians divides a list into group of 5-member to have an linear running time. If the size of each group is three, the running time is not linear.

If each group has 3 members instead of 5, following the above procedure, we will end up with $T(n) = T(n/3) + T(2n/3) + O(n)$. If K is the set of all medians of each group, then K has $\frac{n}{3}$ members. We know that $\frac{1}{2}$ element of the set K is smaller than median of the set K which equals to $\frac{1}{2} \times \frac{n}{3} = \frac{n}{6}$. Since each of $\frac{n}{6}$ member in the list are the median of a 3- member group and 1 element is less then them in each group, there would be $\frac{n}{6} + \frac{n}{6} = \frac{n}{3}$ in the set K that are less then the median of set K. Hence, in the worst case, the algorithm need to do recursion on the remaining $\frac{2n}{3}$. The overall recursion is $T(n) \leq T(n/3) + T(2n/3) + O(n)$.

If we can try Akra-Bazzi theorem : $(1/3)^p + (2/3)^p = 1$ if p=1 then we have $\frac{3}{3} = 1$;

$T(x) = \theta(x^p(1 + \int_1^x \frac{g(u)du}{u^{p+1}}) = \theta(x^p(1 + \int_1^x \frac{1}{u}du)$. So the asymptotic complexity would be $\theta(xlnx)$

Note: In the special case of the master-theorem is that if we have $\frac{an}{b} + \frac{cn}{d} + \frac{en}{f} + ...O(n)$ and $\frac{a}{b} + \frac{c}{d} + \frac{e}{f} + ... \geq 1$ and $T(1) = const$ the complexity of $T(n)$ is $O(nlogn)$ and if $\frac{a}{b} + \frac{c}{d} + \frac{e}{f} + ... < 1$, the complexity of $T(n)$ is $O(n)$.

(I discussed it with the professor and a TA and they said no need to prove this especial case. `http://people.eecs.berkeley.edu/~luca/w4231/fall99/slides/l3.pdf`)

(b) **[4]** Say we defined "almost median" differently, and we ended up with a recurrence $T(n) = T(5n/6) + T(n/5) + cn$. Does this still lead to a linear running time?

According to the above notes in part 1, since $\frac{5}{6} + \frac{1}{5} > 1$, the complexity would not be linear. If we can try Akra-Bazzi theorem : $(5/6)^p + (1/5)^p = 1$ if p=1 then we have $\frac{31}{30} > 1$; So $1 < p < 2$. .

$T(x) = \theta(x^p(1+\int_1^x \frac{g(u)du}{u^{p+1}}) = \theta(x^p(1+\int_1^x u^{-p}du) = \theta(x^p + \frac{x}{1-p} - \frac{x^p}{1-p})$. So the asymptotic complexity would be $\theta(x^p)$ when $1 < p < 2$ which is not a linear.

p is a little bit greater than 1.

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.)

Answer:

Since we know that the set has majority, it means that there is an element that occurs at least $\frac{n}{2}+1$ times. Now, we can consider the set with a majority element that has at least $\frac{n}{2}+1$ occurrences and the other elements with $\frac{n}{2}-1$ occurrences. If we pick an element the possibility that the selected element is not a majority is $\frac{n/2-1}{n}$. The algorithm fails if in all r times, we fail. Since the experiments are independent of each other we can have :

The probability of the fail in r times $= (\frac{n/2-1}{n} \times \frac{n/2-1}{n} \times \frac{n/2-1}{n} .... \frac{n/2-1}{n}) < (\frac{n/2}{n} \times \frac{n/2}{n} \times \frac{n/2}{n} .... \frac{n/2}{n}) = \frac{1}{2^r}$. Pay attention the number of terms is r.

(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.]

Answer: We can sort the array first. This takes $O(n \log n)$. If it has a majority element with more than $n/2$ occurrences, what we can do is that we can choose the middle element and compare it with the element before and element after that. If it passes the "equality test " for one of them, that chosen element is the majority. If it doesn't , It doesn't have the majority element. The intuition behind that is that majority elements has $n/2+1$ occurrences. If we sort the array, and it has a majority element, the middle element should be that one.

Another solution would be divide and conquer. We can follow something similar to merge sort. Now the base case would be the an 2-member subsets. If the members are equal, we have a majority. The next step after the base case, would 4-member subsets. For that, we can try 2-member subsets that already have passed equality test and have majority and compare them (choose one element of each of them and compare them). We can keep going for 4, 8, ... If we have majority after $\log n$-1 we can have a subset with a subset with $\frac{n}{2}$ that passes that has passed majority test. So, in the last stage by comparing (equality test) those that have majority with an element of this subset we can decide either it has majority element or not. Like mergesort, the complexity is $O(nlogn)$.

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.

Answer:

Alice is having the cake. Suppose that the initial amount of cake is k, the amount of the consumption in time t is c_t and the utility function is $\mathrm{u}(\mathrm{c_t})$. Assuming T is a time period that Alice finishes the cake, The total utility is given by

$$\sum_{t=1}^{T} \beta^{t-1} u(c_{\mathrm{t}})$$

Since the initial value is K, we have:

$$K = \sum_{t=1}^{T} c_{\mathrm{t}}$$

We need to resolve the following equation:

$$max \sum_{t=1}^{T} \beta^{t-1} u(c_{\mathrm{t}})$$

and we know that :

$$K = \sum_{t=1}^{T} c_{\mathrm{t}}$$

This is a dynamic optimization problem. To solve it we can use the dynamic programming approach to decompose it into some sub-problems or use analytical way.

Now, for the given question the $u(c_{\mathrm{t}})$ = log (1+$\mathrm{j_t}$). (I just change a notation to show the time t is involved).

If we rewrite the equation we have:

$$max \sum_{t=1}^{T} \beta^{t-1} \log(1 + j_{\mathrm{t}})$$

$y_{\mathrm{t+1}} = y_{\mathrm{t}} - j_{\mathrm{t}}$ $1 \leq t \leq T$

$j \geq 0, y_{\mathrm{t+1}} \geq 0$ $1 \leq t \leq T$

$y_0 = K \geq 0$

$\mathrm{y_t}$ is the remaining one in time t

We can follow the Bellman equation using following documents to solve this problem.

(Uploaded notes in Canvas, Wikipeida: `https://en.wikipedia.org/wiki/Bellman_equatio` and book chapter in Introduction to Computational Economics Using Fortran)

In the Bellman, we can writes a value "V" of a decision problem at a certain time in terms of reward from that decision and the value 'V" of the remaining decision. In other words, in general, we can have such formula:

$$V(x_0) = max_{\mathrm{a0}}\{Reward(x_0, a_0) + \beta V(x_1)\}$$

In the given question, we can have the following equation:

$$V(y, n) = max_{\mathrm{j}}\{log(1 + j_{\mathrm{t}}) + \beta V(y - j_{\mathrm{t}}, n - 1)\}$$

subject to:

$$0 \leq j \leq y$$

while $V(y, 0) = 0$ and n is the number of remaining days until we reach to the T.

In other words, we can solve the problem of decision making at time t using following scenario: Onwards we determine what is the maximum reward gained from time *t+1* while we know a remaining amount of cake. Having solve for this, we solve for the optimum amount at time t considering the fact that

$y_{t+1} = y_t - j_t \ 1 \leq t \leq T$

$j \geq 0, y_{t+1} \geq 0 \ 1 \leq t \leq T$

$y_0 = K \geq 0$ This equations means that in each time t, we can look at previous days to see how much Alice has eaten until that time and how much she will get reward if she eats j_t. In other words, if we store the decision in each step in a table, considering n slices has been remained, j can varies from 1 to n. In time t, if the best decision for (n+j, t-1) is stored in the table already, we assign it to the a variable H. Otherwise it would be new optimal solution that we compute and store in the table. The H+ $\beta^{t-1} \log(1 + j_t)$ will be return Intuitively we have three variables:

1) Number of days that change from (1 to k)

2) The reaming slices that changes from (1 to k)

3) The amount of eating per day from remaining slices that varies from (1 to k)

One involving parameter the reward that Alice gets each day. The reward is $\log(1+j_t)$. If j_t is zero, it means that that we have log 1 which is zero. Since we are doing max operation with some non-negative values , having 0 for one of them means that it cannot contributes to V(y,n). For example, if we have only one slice, the satisfaction is maximized only if Alice eat it on the first day.

Since there are three parameters ranging form 1 to K that we need to consider (e.g. like 3 nested), the complexity would be $O(K^3)$. By solving Bellman equation we can have:

$j_t(A) = \frac{(1-\beta)\beta^{t-1}}{1-\beta^K} \ 1 \leq t \leq K$

$y_t(A) = \frac{\beta^{t-1}\beta^K}{1-\beta^K} \ 1 \leq t \leq K$

$v(y,n) = \frac{(1-\beta^n)}{1-\beta} \log(\frac{(1-\beta)}{1-\beta^n} y) + [\frac{\beta(1-\beta^{n-1})}{(1-\beta)^2} - \frac{(n-1)(\beta^n)}{(1-\beta)}] \log(\beta) \ 1 \leq t \leq K$

Resource: `http://ciep.itam.mx/~rtorres/progdin/computing-the-cake-eating-problem.pdf`

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.

Answer:

Let G and Ti denote a tree and the subtree of tree G rooted at the vertex i. W(I) of an independent set I is sum of the weights of all the vertices in set I.

$W(I) = \sum_{i \in I} w(i)$

We define the followings:

M(i) = $Max\{w(I) | I$ is an independent set in Ti and $i \in I\}$

M‘(i) = $Max\{w(I) | I$ is an independent set in Ti and $i \notin I\}$

Suppose j is a child of i. These values can be computed recursively by following equations:

$M(i) = w(i) + \sum_j M‘(j)$

$M‘(i) = \sum_j max\{M‘(j), M(j)\}$

Based on the above equation, for the leaves we have $M(i)$= w(i), $M‘(i) = 0$.

We can follow the dynamic programming approach and find the maximum weight independent set in two steps

step 1: We compute M(i) and M'(i) for each vertex i, in bottom up approach using the above equation (leaves to root)

step 2. We find a maximum weight independent set by examining the tree G, in top-down approach (root to leaves). if M(root)>M'(root), the root will be included in I. For other vertices like j, it is included in I if the parent is not in I and M(j) > M'(j).

We compute M(i) and M'(i) for each vertex in phase1. In phase 2, each vertex is examined once. So, the complexity would be O($|V|$). For each vertex j, the algorithm looked at that three times: I itself, *j's parent* and *j's grandchild*. Since each vertex is visited constant number of times, the complexity would be O($|V|$) = O(n).

Resource: Chen, G.H., Kuo, M.T. and Sheu, J.P., 1988. An optimal time algorithm for finding a maximum weight independent set in a tree. BIT Numerical Mathematics, 28(2), pp.353-356. `http://hscc.cs.nthu.edu.tw/~sheujp/public/journal/3.pdf`