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

Submission date: TBD

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.

SOLUTION:

Algorithm 1 +/- Transition Finder

```
1: function FindTransition(start, end):
2:     mid ← ⌊(start + end)/2⌋
3:     if A[mid] = +1 and A[mid + 1] = −1 then
4:         return mid
5:     if A[mid] = +1 then
6:         return FindTransition(mid + 1, end)
7:     else
8:         return FindTransition(start, mid)
```

Because $A[0]$ and $A[n-1]$ are $+1$ and -1 respectively, there must be at least one transition point between the two, a string of $+1$ starting at $A[0]$ must eventually run into a -1, even if it's at the very end of the array. Knowing this, we can adapt a binary search to this problem: if the center point is a transition, return it. Otherwise, if the center point is a $+1$, there must be a transition point between this one and the end, so we recurse on the top half only. If the center point is a -1, there must be a transition point between this and the start, so we recurse on the bottom half only. This has the same structure as binary search, with the comparisons simply looking for different things, so the runtime is the same: $O(\log n)$.

(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)?]

SOLUTION:

We can modify our algorithm from part (a) slightly to fit this new problem. For any element $A[i]$, we can treat it like a $+1$ if $A[i] > A[i+1]$, and a -1 if $A[i] < A[i+1]$. Since $A[n-1]$ has no next number, we can treat it like a -1. If $A[0] < A[1]$, then we can return A[0] since it's less than its next neighbor and has no previous neighbor. Otherwise, we meet the necessary conditions to use the algorithm from part (a) ($A[0]$" $= $ " $+1$ and $A[n-1]$" $= $ " -1), so we run it, changing lines 3 and 5 to use our new definition of ± 1. The algorithm gives us the index, k, such that $A[k] > A[k+1]$ and $A[k+1] < A[k+2]$. This means $A[k+1]$ is a local minimum, so we return it.

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?]

SOLUTION:

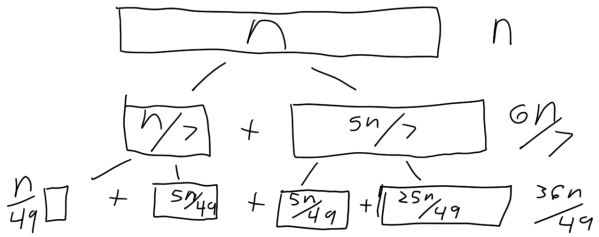
The original recurrence is $T(n) = T(n/5) + T(7n/10) + O(n)$ (in the worst case), because this strategy uses groups of 5 to find the median of medians (mom), and discards at least 3/10 elements afterwards at each step. Our new strategy uses groups of 7 to find the mom, so the recursive step to find it involves looking at 1/7 of all items instead of 1/5. 4/7 elements in the lower half of these

groups is less than our new mom, so we discard at least 4/14 (2/7) elements at each step and recurse on the remaining 5/7. Our new recurrence would be $T(n) = T(n/7) + T(5n/7) + O(n)$.

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

SOLUTION:

If we examine the recursion tree, the first layer does n work, the second layer does $6n/7$, the third does $36n/49$ work, and the i^th level does $n(6/7)^{i-1}$ work. This is a geometric series $n + n\frac{6}{7} + n(\frac{6}{7})^2 + n(\frac{6}{7})^3 + \ldots + n(\frac{6}{7})^{k-1} = n((\frac{6}{7})^k - 1)/(\frac{6}{7} - 1)$, causing the term n to dominate. Therefore, the runtime is still $O(n)$ despite the larger groups.



ALTERNATIVE SOLUTION:

Assume $T(n) = A\lfloor n \rfloor^p$ for some parameter p. We can see that $An^p = T(n) = T(n/7) + T(5n/7) + O(n) \leq A(n/7)^p + A(5n/7)^p + cn$ (for some c). Then we can see that, $n^{p-1}\frac{A}{c}\left(1 - \frac{1}{7^p} - \frac{5^p}{7^p}\right) \leq 1$. Since $T(n) \in \Omega(n)$ we can see that $p \geq 1$. We can also see that $\left(1 - \frac{1}{7^p} - \frac{5^p}{7^p}\right) > 0$. Therefore, for the inequality to hold, $p = 1$ (otherwise n^{p-1} will grow with n). Therefore, we get $T(n) \in O(n)$.

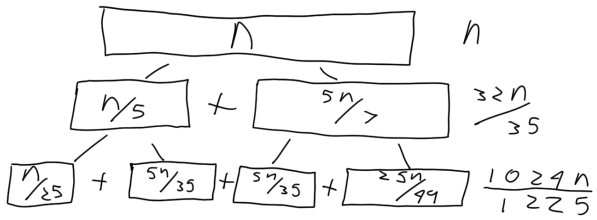
ALTERNATIVE SOLUTION:

We can use the guess-and-check method to solve this. Assume $T(k) \leq Ak$ for all $k < n$ and let $T(n) = T(5n/7) + T(n/7) + cn$. Then we can see that $T(5n/7) + T(n/7) + cn \leq \frac{5A}{7}n + \frac{A}{7}n + cn$ which gives us $T(n) \leq \left(\frac{6A}{7} + c\right)n$. Now we can see that by picking an $A \geq 7c$, we would have $T(n) \leq An$. Therefore, by guess-and-check, we get that $T(n) \in O(n)$

(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?

SOLUTION:

Once again examining the recursion tree, the first layer does n work, the second layer does $32n/35$, and the third does $1024n/1225$. This series exponentially decays by a factor of 32/35, causing the root (n) to dominate. Our runtime is still $O(n)$.



ALTERNATIVE SOLUTION:

Assume $T(n) = A\lfloor n \rfloor^p$ for some parameter p. We can see that $An^p = T(n) = T(n/5) + T(5n/7) + O(n) \leq A(n/5)^p + A(5n/7)^p + cn$ (for some c). Then we can see that, $n^{p-1}\frac{A}{c}\left(1 - \frac{1}{5^p} - \frac{5^p}{7^p}\right) \leq 1$. Since $T(n) \in \Omega(n)$ we can see that $p \geq 1$. We can also see that $\left(1 - \frac{1}{5^p} - \frac{5^p}{7^p}\right) > 0$. Therefore, for the inequality to hold, $p = 1$ (otherwise n^{p-1} will grow with n). Therefore, we get $T(n) \in O(n)$.

ALTERNATIVE SOLUTION:

We can use the guess-and-check method to solve this. Assume $T(k) \leq Ak$ for all $k < n$ and let $T(n) = T(5n/7) + T(n/5) + cn$. Then we can see that $T(5n/7) + T(n/5) + cn \leq \frac{5A}{7}n + \frac{A}{5}n + cn$ which gives us $T(n) \leq \left(\frac{32A}{35} + c\right)n$. Now we can see that by picking an $A \geq \frac{35}{3}c$, we would have $T(n) \leq An$. Therefore, by guess-and-check, we get that $T(n) \in O(n)$.

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.)

SOLUTION:

Because a majority element is guaranteed to appear in this array, the chances of choosing a non-majority element is $< 1/2$. It could be much smaller depending on the size of the majority, but it can't be larger than $1/2$ or the majority would not be a majority. We choose r elements uniformly and independently at random, so the chances that none of them are the majority element are $< 1/2 * 1/2 * 1/2...$ or $1/2^r$.

(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.]

SOLUTION:

Algorithm 2 Non-Comparison Majority Element

```
1:  function GetMajority(start, end):
2:      if start = end then                                  ▷ base case
3:          return A[start]
4:      mid ← (start + end)/2                                ▷ recursion or "divide" phase
5:      X ← GetMajority(start, mid − 1)
6:      Y ← GetMajority(mid, end)
7:      if X = Y then return X                               ▷ evaluation or "conquer" phase
8:      Xcount ← 0
9:      Ycount ← 0
10:     for element e in A[start to end] do
11:         if e = X then Xcount++
12:         if e = Y then Ycount++
13:     if Xcount > Ycount then return X
14:     elsereturn Y
```

First, we can observe that if we were to split the array in half, one or both of those two halves would have the same majority element as the whole array. If the bottom half had a majority element a and the top half had a majority element b, then that means a has $> 1/4$ total copies, and b has $> 1/4$ total copies, and together they have $> 1/2$ copies split between them, so it's not possible for a third element, c to have enough copies to be the overall majority element.

Now we can divide the problem into two halves, the bottom half and the top half, and solve those subproblems recursively. If those two subproblems have the same answer, our decision is easy and we return that answer as well. If they give different answers, count how many occurrences of each

appear in the list, and return whichever is more common. If we only have one element, that element is by definition the majority of that single element set, so we return it. Because we always choose the more common element when given a choice between the majority of the two halves, only the most common element will be returned at the very end, which is guaranteed to be the majority element since one must exist. Only the = operator is used on the elements themselves, the > used at the end compares counts of elements, which is legal.

We make 2 recursive calls, each getting 1/2 the input size of the original. After recursion, we make $2n$ comparisons to do the counting up. Our recurrence is $T(n) = 2T(n/2) + O(n)$. This is identical to mergesort's recurrence so we already know the runtime of this algorithm, which is $O(n \log n)$.

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.

SOLUTION:

First, we should consider what our choices are. For each day, we have the option of eating just one cake slice, or any amount up to however many are left, so we must consider every possible amount of slices. For our solution, we can express the relationship between subproblems with this recursive relationship:

$$cake(t,k) = \begin{cases} 0 & k \leq 0 \\ \max\limits_{1 \leq i \leq k} [\beta^{t-1}\log(i+1) + cake(t+1, k-i)] & otherwise \end{cases}$$

Using the principles of dynamic programming, we can store answers to these subproblems in a kxk array, where each position in the array corresponds to a subproblem instance of the cake eating problem. t theoretically grows without bound, but there's no advantage gained from not eating at least one slice of cake on a particular day, so the solution will have to take place in k days or less. Subproblems depend on other subproblems in the positive t direction (right) and negative k direction (up), so we fill in the array working from the top right, working down first and then left to ensure all subproblems have their dependencies filled.

the kxk array has k^2 cells to fill, and each one takes $O(k)$ comparisons to fill since it needs to look at every value from 1 to k, so the total runtime is $O(k^3)$

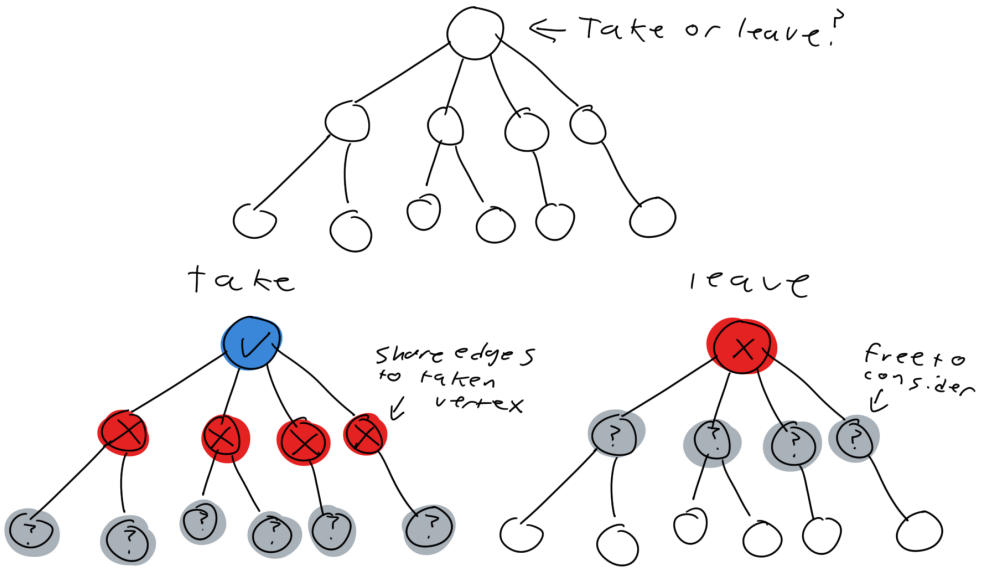
Question 5: Maximum Weight Independent Set ..... **[10]**

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.

SOLUTION:

It may first help to visualize what our choices are in this problem. At each vertex we have the option of adding it to our in-progress vertex cover, or leave it out. The consequences of these two options can be visualized like this:


Take or leave?
take
leave
Share edges to taken vertex
Free to consider

With these two choices, we can more formally express the relationship between subproblems with this recursive relationship:

$$MaxIndSet(v) = \begin{cases} W[v] & children(v) = \varnothing \\ \max \begin{cases} \sum\limits_{v->c} MaxIndSet(c) \\ W[v] + \sum\limits_{v->g} MaxIndSet(g) \end{cases} & otherwise \end{cases}$$

where $v->c$ means "for all children of v" and $v->g$ means "for all grandchildren of v". If the current vertex is a leaf, there's no reason not to to take it, so we return the weight of that vertex. Otherwise, we try taking v and not taking v, and pick the better result. If we don't take v we can evaluate the children, and if we do take v we skip the children and evaluate the grandchildren instead.

Now to apply dynamic programming to this, we can list the vertices in the order they appear, starting with the root, then the root's children, then the root's grandchildren, and so on. $V[0]$ is the root vertex and $W[0]$ is the root's weight. Because each vertex depends on answers from its descendants, we work through this V array backwards, starting at the leaves and working our way up.

We evaluate each vertex once, but the amount of comparisons varies depending on the number of children it has. However, one can observe that when the algorithm processes one vertex v, it only looks at the children and grandchildren of v. Consequently, each vertex v is may be processed three times at most: when the algorithm visits v, the parent of v and the grandparent of v (since the graph is a tree, each vertex has at most one parent and one grandparent). In short, each vertex is processed only a constant number of times, so the runtime of this algorithm is $O(n)$.