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

Submission date: Monday, Sep 27, 2021, 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
Basics – recursion and dynamic programming	10	
Linear time selection	10	
Coin change revisited	10	
Let them eat cake	10	
Conflict-free subsets	10	
Total:	50	

Question 1: Basics – recursion and dynamic programming . . . . . **[10]**

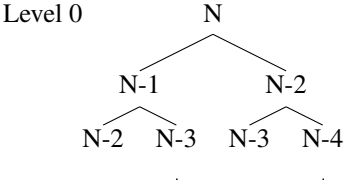
(a) **[4]** Consider the following recursive subroutine for computing the kth Fibonacci number:

```
function FIBONACCI(k):
    if k = 0 or k = 1 then
        return 1
    else
        return Fibonacci(k − 1) + Fibonacci(k − 2)
    end if
end function
```

Implement the subroutine above, and find the Fibonacci numbers for $k = 45, 50, 55$. (You may need to explicitly use 64 bit integers depending on your programming language.) What do you observe as the running time?

Solution. The running time for computing the kth Fibonacci number for k = 45 -> 1836311903, k = 50 -> 20365011074 and k = 55 -> 225851433717 are 5s, 75s and 850s respectively. As the k value increases, the running time increases exponentially.

For computing Fibonacci(N), the recurrence tree is as follows:



Level N ....

This yields a running time that also satisfies the recurrence $T(n) = T(n-1) + T(n-2)$, which solves to $T(n) = \alpha^n$, where α is the golden ratio, roughly 1.6. **Rubric:**

- 3 points for giving running time for k=45,50,55
- 1 points for arguing/guessing the running time

(b) **[2]** Explain the behavior above, and say how you can overcome the issue.

Solution. The run-time of Fibonacci is exponential due to repeated computations,

E.g., even in the short recursive tree shown above, we re-compute the answer for the sub-problems $(N-2)$, $(N-3)$ and $(N-4)$. By using Dynamic Programming, we can store the answers as we go along and retrieve when needed (in constant time).

Rubric:

- 2 points for a method to avoid re-computation

(c) **[4]** Recall the L-hop shortest path problem we saw in class. Here, the procedure `ShortestPath(u, v, L)` involves looking up the values of `ShortestPath(u', v, L-1)` for all out-neighbors u' of u. This takes time equal to $deg(u)$, where deg refers to the out-degree.

Consider the total time needed to compute `ShortestPath(u, v, L)`, for all vertices u in the graph (with v, L remaining fixed, and assuming that the values of `ShortestPath(u', v, L-1)` have all been computed). Show that this total time is $O(m)$, where m is the number of edges in the graph.

Solution. Let $\vec{G} = (V, \vec{E})$ be the given digraph and $m = |\vec{E}|$. The running time of `ShortestPath(u, v, L)` with fixed u, v, L is $\deg^+(u) \cdot O(1) = O(\deg^+(u))$ under the given assumption. Thus, the total time for all vertices is $\sum_{u \in V} O(\deg^+(u)) = O\left(\sum_{u \in V} \deg^+(u)\right) = O(m)$ because the sum of the out-degrees equals the number of edges in the graph. This is because every edge gets counted precisely once in the sum of the out-degrees.

Rubric:

- 2 points for showing that the running time with fixed u is $O(\deg^+(u))$

- 2 points for showing that the out-degree sum equals m

Question 2: Linear time selection . . . . . **[10]**

Recall the linear time selection algorithm we saw in class (median-of-medians, lectures 4 and 5) and answer the following questions. Please provide detailed justification.

(a) **[5]** Instead of dividing the array into groups of 5, suppose we divided the array into groups of 3. Now sorting each group is faster (although both are constant time). But what would be the recurrence we obtain? Is this an improvement over the original algorithm? [Hint: What would the size of the sub-problems now be?]

Solution.

If the array is divided into groups of 3 instead of 5 we can see that the median-of-median step would require sorting an array of size 3 and then taking the median of the medians of the $\frac{n}{3}$ sub-problems. Therefore, the time complexity of that step would be $T(\frac{n}{3})$ instead of $T(\frac{n}{5})$. We can see that $\frac{n}{3} \leq r \leq \frac{2n}{3}$, redefining the almost-median criterion to $\frac{n}{3} \leq r \leq \frac{2n}{3}$, the new recurrence would be, $T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n)$.

Rubric:

- 3 points for Recurrence Relation
- 2 points for correct reasoning

[Some of you may have used the "in class" definition of almost median and ended up with the recurrence $T(n) = T(\frac{3n}{4}) + T(n/3) + O(n)$. This solution will also receive full credit.]

(b) **[5]** The linear time selection algorithm has some non-obvious applications. Consider the following problem. Suppose we are given an array of n integers A, and you are told that there exists some element x that appears at least $n/5$ times in the array. Describe and analyze an $O(n)$ time algorithm to find such an x. (If there are multiple such x, returning any one is OK.) [Hint: Think of a way of using the selection algorithm! I.e., try finding the kth smallest element of the array for a few different values of k.]

Solution. Given for frequent element occurring $n/5$ times in the array

The algorithm is as follows:

1. Find the $(i \cdot n/5)$th smallest element of the array for $i = 1, 2, 3, 4$. Let them be a, b, c, d.
2. For each of a, b, c, d count the number of times they occur in the array and return the most frequent one.

Running time: We make 4 calls to the Select procedure, and counting the number of occurrences each takes n time. Thus the total run time is $O(n)$.

Correctness: Suppose B is the sorted version of A (we don't know B). By definition, a, b, c, d in the algorithm are the $n/5$th, $2n/5$th, $3n/5$th and $4n/5$th elements of B. Now if A has an element repeated $\geq (n/5)$ times, then this element must form a contiguous block of length $\geq n/5$ in B. Thus this block must intersect one of the positions $n/5$, $2n/5$, $3n/5$ and $4n/5$. Thus at least one of a, b, c, d must be the frequent element.

Rubric:

- 2 points for a correct algorithm
- 2 points for arguing correctness
- 1 point for arguing run-time

Question 3: Coin change revisited . . . . . **[10]**

Recall the "coin change" problem, where we have coins of denominations $d_1, d_2, \ldots, d_k$ (an unlimited supply of each), and the goal is to make change for N cents using the minimum number of coins (here $N, d_1, \ldots, d_k$ are given positive integers).

We saw in class that a greedy strategy does not work, and we needed to use dynamic programming.

(a) **[4]** We discussed a dynamic programming algorithm that uses space $O(N)$ and computes the minimum number of coins needed. Give an algorithm that improves the space needed to $O(\max_i(d_i))$.

Solution: Consider the algorithm that uses space $O(N)$ discussed in class. In order to calculate the change needed for some value x, we consult our storage array A at indices $A[x - d_i], \forall i \in [k]$. W.l.o.g., assume $d_k = \max_i(d_i)$. Then to compute a value x, we look at no indices smaller than $x - d_k$ in A. Our algorithm proceeds as follows:

Given some target n, for $x = 1, ..., n$, calculate the number of coins needed for x by storing the minimum of $A[x - d_i]\ \forall i \in [k]$. Store the new value for x in a circular array, so that the array A always contains the entries corresponding to values from $x + 1 - d_k$ to x.

Correctness and Runtime: Since A holds the correct values of making change with our given coin denominations according to the proof in class, this algorithm will return the minimum number of coins needed to make change for x. Since this circular array can be maintained in constant time with a mod-index system, the runtime of this algorithm remains $O(nk)$.

Rubric:

- 2 points for a correct algorithm that uses $O(\max_i d_i)$ space.
- 1 point for arguing why the array does not need to be larger than $O(\max_i d_i)$.
- 1 point for runtime and correctness from class.

(b) **[6]** Design an algorithm that outputs the number of different ways in which change can be obtained for N cents using the given coins. (Two ways are considered different if they differ in the number of coins used of at least one type.) Your algorithm needs to have time and space complexity polynomial in N, k.

Algorithm 1 Number of ways to make change

Input: Target n, Coins $\{d_1, \ldots, d_k\}$.
Output: Number of ways to make target.
Data structure: Create an array A of size $n + 1$ (0-based indexing) and initialize all the elements to 0.

```
1: A[0] ⟵ 1
2: for i in 1, ..., k do
3:     for j in 1, ..., n do
4:         if d_i ≤ j then
5:             A[j] ⟵ A[j] + A[j − d_i]
6:         end if
7:     end for
8: end for
9: return A[n]
```

Solution: See Algorithm 1. $A[j]$ is used to store the number of ways that j can be made into coins.

Correctness: We offer proof by induction. There is exactly 1 way to make change for 0 coins, which is by using 0 coins, as noted on line 1. Now assume that $A[0]$ to $A[j - 1]$ contains the number of ways to make change for j using coins of value at most d_i. Then the number of ways to make j using an additional coin d_i is the same as the number of ways to make $j - d_i$. After iterating from $j = 1$ to n then we can move to a larger denomination d_{i+1}. Again, the number of ways to make j using an additional coin d_{i+1} is the same as the number of ways to make $j - d_{i+1}$. Note that by iterating over the target value j inside of each coin denomination loop i, we do not double count any ways, since any ways that increased the number of a smaller coin has already been considered. By induction, this will return the number of ways to produce change for n.

Running time: With two nested for-loops, with a constant-time operation nested inside, we have a run-time of $O(nk)$.

Rubric:

- 2 points for a correct algorithm
- 2 points for arguing correctness
- 1 point for arguing that no cases are repeated
- 1 point for arguing runtime

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

Alice buys a cake with k slices on day 1. 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 $\sqrt{j}$. However, a cake loses freshness over time,

and so suppose that each passing day results in a loss of value by a factor $\beta = 0.8$. Thus if she eats j slices on day t, she receives a satisfaction of $\beta^{t-1}\sqrt{j}$ on that day.

Given that Alice has k slices at the start of day 1, given the decay parameter $\beta = 0.8$, 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 on days $1, 2, 3, \ldots$). The algorithm must have running time polynomial in k. [Hint: dynamic programming!]

Solution 1.

We can define the problem of optimizing the satisfaction of the cake eating problem as follows,

$$F(k,t) = \max_{1 \le j \le k} (\beta^{t-1}\sqrt{j} + F(k-j, t+1))$$

Where k is the number of slices remaining at the start of time t.

Correctness. The algorithm keeps a data-structure Sat of size $k \times k$. The $Sat[i][j]$ gives the maximum satisfaction when j slices are left at the start of the day i. The algorithm searches through the entire solution space and stores the optimum satisfaction for each sub-problem. So we basically have solved all the possible problems. It is important to note that eating 0 cakes does not provide a good satisfaction so we can actually see that the minimum slices Alice could eat in a day is 1.

Time Complexity. The number of sub-problems is $k \times k$ and for each sub-problem we have to consider at most k possible values so the time complexity is $O(k^3)$.

Algorithm 2 Cake eating schedule

Input: The total number of slices K.
Output: The optimum schedule.
Data structure: Create an array Sat of $k \times k$ and initialize all the elements to -1.

function MAXSATISFACTION(k slices left, t current day)
  if $k = 1$ **then**
    $Sat[t][k] = \beta^{t-1}\sqrt{1}$
  end if
  for each i in 1 to k **do**
    if $Sat[t+1][k-i] = -1$ **then**
      $maxSatisfaction(k-i, t+1)$
    end if
    $SatVal = Sat[t+1][k-i]$
    $Sat[t][k] = \max(\beta^{t-1}\sqrt{i} + SatVal, Sat[t][k])$
  end for
end function

$day = 1$, $Slices = K$, and $Sched = \emptyset$
$maxSatisfaction(Slices, day)$

while $Slices > 0$ **do**
  $s = argmax_{1 \le j \le Slices} \beta^{day-1}\sqrt{j} + Sat[day+1][Slices - j]$
  $Slices = Slices - s$
  $Sched = Sched \cup (day, s)$
  $day = day + 1$
end while

Solution 2.

We can define the problem of optimizing the satisfaction of the cake eating problem as follows,

$$F(k) = \max_{1 \le j \le k} (\sqrt{j} + \beta F(k-j))$$

Where k is the number of slices Alice has.

Correctness. There are two observations in writing the recurrence: (a) eating zero slices on a day is sub-optimal, as $\beta < 1$, and (b) the form of the decay —β^t on day t— allows us to define sub-problems where the only parameter is the number of slices. We then search through the entire solution space and stores the optimum satisfaction for each sub-problem.

Time Complexity. The number of sub-problems is now only k, and for each sub-problem we have to consider at most k possible values so the time complexity is $O(k^2)$

Algorithm 3 Cake eating schedule

Input: The total number of slices K.
Output: The optimum schedule.
Data structure: Create an array Sat of $k+1$ and initialize all the elements to 0 and create an array $Schd$ of $k+1$.

function MAXSATISFACTION(k slices to eat)
    $Sat[0] = 0$ and $Schd[0] = 0$
    for each i in 1 to k **do**
        $Sat[i] = \max_{1 \le j \le i}(\sqrt{j} + \beta Sat[i-j])$
        $Schd[i] = argmax_{1 \le j \le i}(\sqrt{j} + \beta Sat[i-j])$
    end for
end function

$day = 1$, $Slices = K$, and $Sched = \emptyset$
$maxSatisfaction(Slices)$

while $Slices > 0$ **do**
    $s = Schd[Slices]$
    $Slices = Slices - s$
    $Sched = Sched \cup (day, s)$
    $day = day + 1$
end while

Ruberics:

- 5 points for correct algorithm.
  - 1 point for correct recurrence with base case
  - 2 points for using dynamic programming
  - 1 point for finding the max satisfaction value
  - 1 point for finding the schedule to get the max satisfaction value.
- 3 points for correctness.
  - 1 point for pointing out that eating 0 slice is not optimal
  - 1 point for how algorithm exhaustively considers all possibilities
  - 1 point for how dynamic programming is used to save sub-problems
- 2 points for arguing run-time.

Question 5: Conflict-free subsets ..... **[10]**

Suppose we have n people, of which we want to pick a subset to form a team. Every person has a (non-negative) "value" they can bring to the team, and the value of a subset is simply the sum of values of those in the subset. To complicate matters however, some people do not get along with some others, and two people who don't get along cannot both be part of the chosen team.

Suppose that conflicts are represented as an undirected graph G whose vertex set is the n people, and an edge ij represents that i and j do not get along. The goal is now to choose a subset of the people that maximizes the total value, subject to avoiding conflicts (as described above).

(a) **[3]** Consider the following natural algorithm: choose the person who brings the highest value, remove everyone who is conflicted, choose the one remaining with the highest value, remove those conflicted, and so on. Does this algorithm always find the optimal subset (one with the highest total value)? If so, provide formal reasoning, and if not, provide a counter-example.

Solution. No. A counter-example is a path $P = abc$ of three vertices with values $w(a) = w(c) = 2$, $w(b) = 3$. The optimal solution is $\{a, c\}$, which gives a total value of 4, whereas the given algorithm chooses only b, resulting in a lower value, 3.

- 2 points for identifying that this is not an exact algorithm
- 1 point for showing a valid counter-example
- Alternatively, 1 point for a non-trivial attempt to prove the correctness

(b) **[7]** Suppose the graph G of conflicts is a (rooted) tree (i.e., a connected graph with no cycles). Give an algorithm that finds the optimal subset. (If there are many such sets, finding one suffices.) The algorithm should run in time polynomial in n. [Hint: Once again, think of a recursive formulation given the rooted tree structure and use dynamic programming!]

Solution.

Algorithm 4 Maximum weight independent set (first algo)

function MWIS(node v)
Input graph $G = (v, e)$
Input weight array w
Input an empty array Opt of size n
Save the nodes $v_1, v_2, \ldots, v_n$ in the array
  for i in $v_1, v_2, \ldots, v_n$ **do**
    $Opt[i] = \max(w(i)) + \sum_{j \in grandchildren\ of i}^{\#grandchildren} Opt[j], \sum_{j \in children\ of i}^{\#children} Opt[j], \sum_{j \in grandchildren\ of i}^{\#grandchildren} Opt[j])$
  end for
return $Opt[n]$ where n is the root and the corresponding nodes
end function

Correctness. All the solution space is traversed, meaning every node is visited and the possibility of including its weight or not are considered by looking at the root and the its children. We explicitly check to keep the best value. This algorithm starts from the root and for each node it either includes all the children or grandchildren (in the case of including the grandchildren the node itself will also be included and not otherwise). By considering all scenarios in dynamic programming we calculate the maximum weight for each node once.

For the running time we have two sub problems which are dependent on the number of nodes and we go over each sub-problem only once (each nodes look up times is $O(1)$). So adding the time for all nodes, it is polynomial in n.

Note that both proposed pseudo codes for this question will work (one is more concise).

Rubric:

- 3 points for a correct algorithm
- 2 points for arguing correctness
- 2 points for arguing runtime

Algorithm 5 Maximum weight independent set (second algo)

```
1: function MWIS(node v)
2:     if v is NULL then
3:         return No answer
4:     end if
5:     if we have one v (v is leaf) then
         return index and weight of v
6:     end if
7:     for every child of v do
8:         if child has max(index) and max(weight) then
9:             use the stored index and weight
10:        else
11:            MIWS(node child)
12:            I_1.append(nodes chosen for maximum weight)
13:            W_1.append(maximum weight of children)
14:        end if
15:    end for
16:    for every grandchild of v do
17:        if grandchild has max(index) and max(weight) then
18:            use the stored index and weight
19:        else
20:            MIWS(node grandchild)
21:            I_2.append(nodes chosen for maximum weight)
22:            W_2.append(maximum weight of children)
23:        end if
24:    end for
25:    if W_2 > W_1 then
26:        add the node's weight and index to two lists for index and weight: (I_2, W_2)
27:        return (I_2, W_2)
28:    else
29:        add the node's weight and index to I_1 and W_1
30:        return (I_1, W_1)
31:    end if
32: end function
```