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?

Answer.

Below is my implementation in python. I found it takes too long to get results for $k = 40, 45, 50$. I can only get results for $k = 20, 25, 30$, which takes $0.013s, 0.15s, 1.6s$ respectively.

```python
def Fibonacci(k):
    if k==1 or k==0:
        return 1
    else:
        result_cur=Fibonacci(k-1)+Fibonacci(k-2)
        return result_cur
```

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

Answer.

It keeps exploring without any memory mechanism. It will have a running time as $O(2^n)$. We can keep a hash-table to memory the some results. Below is the revised implementation. It can run and get results for $k = 20, 25, 30$, which take $136ns, 136ns, 136ns$much faster than original one.

```python
result={}
def Fibonacci(k):
    if k in result:
        return result[k]
    if k==1 or k==0:
        return 1
    else:
        result_cur=Fibonacci(k-1)+Fibonacci(k-2)
        result[k]=result_cur
        return result_cur
```

(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 defers 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.

Answer.

We use SP to denote ShortestPath.

$$SP(u, v, L) = \min_{u' \in childre(u)} (SP(u', v, L-1) + l(u, u')) \tag{1}$$

Since the value of `ShortestPath(u', v, L-1)` is already known, for each node u, we will consider $deg(u)$ possible value, which will take $O(deg(u))$ time. The total time is $\sum_u O(deg(u)) = O(m)$

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?]

Answer.

The median-of-median step require sorting an array of size 3. Then we need to find the medians of $\frac{n}{3}$ elements (medians of subgroups)., And we can see that $\frac{n}{3} \leq r \leq \frac{2n}{3}$. And we also need $O(n)$ for the partitioning work to create the two sides. So the final recurrence form is

$$T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n) \tag{2}$$

Let $T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + cn$. We can use Akra–Bazzi method to get a asymptotic bound. $(\frac{2}{3})^p + (\frac{1}{3})^p = 1$, we can get $p = 1$.

$$T(n) = \Theta(n^1(1 + \int_1^n cx^{-1}dx) = \Theta(n(1 + \log(n)))) \in O(n\log(n)) \tag{3}$$

So it is actually worse.

(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.]

Answer.

Below is the algorithm.

Running time. We will do linear time selection for 10 times. Since linear time selection takes linear time, the running time here is $10 * (O(n) + c) = O(n)$. Actually, uniformly dividing n to get $m, (m > 5)$ numbers, i.e., $n/m, 2n/m, ..., n$ should work here. Here we choose $m = 10$.

Correctness. Since element x appears at least $n/5$ times, when we cut the n number to m splits, element x will at least show $m/5$ times. When choose the most frequent one, it shows certainly no less than $m/5$ times. So we can get the correct answer.

Algorithm 1 Find-element-one-fifth

```
  Input: original array A
  Output: element appears at least n/5 times

  function Find-element(A)
      ks=[n/10, 2n/10, 3n/10,...,n],                  ▷ 10 k smallest value we want to know.
      ks_value=[]
      for k in ks do
          res_cur=Linear-selection-selection(A,k)     ▷ find the kth smallest value we want to know.
          ks_value.append(res_cur)                    ▷ append it to ks_value.
      end for
      return the most frequent elements in res_cur
  end function
```

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 $O(\max_i(d_i))$.
Answer.
Yes, we certainly can do it. The idea is that we only to memorize the last $\max(d_i)$ number's results. The pseudo code is shown in Algorithm 2
Complexity. Since in current step, we at most need to revisit $n - max(d_k)$, we can reduce the space to $O(\max_i(d_i))$ space. And It takes $O(mk)$ time.
Correctness. We explored all possible situations. If for all $n' < n$ is correct, the current minimum number of coins equals the 1 plus minimum $Memo[n - d_i]$. So it is correct. For base case, it is trivially correct.

Algorithm 2 Coin-change-num

```
Input: N cents, coin denominations $D = [d_1, d_2, \ldots, d_k]$
Output: minimum number of coins
function COUNT(D,N)
    initialize an array $Memo$ of size $|\max(d_i) + 1|$
    Memo[0]=0
    d_max=max(D)
    for each $j$ in [1,2,...N] do
        for each $d_i$ in D do
            cur=Inf                                                    ▷ infinite large number
            if $d_i <= j$ and $Memo[(j - d_i)\%d\_max] > 0$ then
                $Memo[j\%d\_max] = min(cur, 1 + Memo[(j - d_i)\%d\_max])$    ▷ % means taking the
reminder.
                $cur = Memo[j\%d\_max]$
            end if
        end for
    end for
    return Memo[N%d_max]
end function
```

(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.
Answer.
The pseudo algorithm is shown in Algorithm 3
complexity. Time is that we have two loops here, which take $O(Nk)$. And for the space it takes only $O(Nk)$ as well.

Correctness. All possibility have been explored. It is correct.

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.

Algorithm 3 Coin-change-num

```
Input: $N$ cents, coin denominations $D = [d_1, d_2, \dots, d_k]$
Output: the number of different ways of changes
function COUNT(D,N)
    create an array $Table$ of size (n,m)
    set $Table[0, :] = 1$
    for each $i$ in [1,2,...N] do
        for each $d_j$ in D do
            if $d_j <= i$ then
                $Table[i, j] = Table[i - d_j, j] + Table[i, j - 1]$
            end if
        end for
    end for
    return Table[N,m]
end function
```

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 $1, 2, 3, \dots$). The algorithm must have running time polynomial in k. [*Hint:* dynamic programming!]

Answer.
We can define the following recursion,

$$F(k, t) = \max_{1 \leq j \leq k} (\beta^{t-1} \sqrt{j} + F(k - j, t + 1)) \tag{4}$$

where k is the number of slices left at the start of day t. The pseudo code is shwon in Algorithm 4.
Correctness. We explore and store all possible states to $Sat[:]$, which contains the entire solution space. The base case is that eating 0 won't bring any satisfactions. Through induction, it's easy to see its correctness.
Time Complexity. The number of sub-problem is $O(k^2)$, and for each sub-problem we need to consider k possible situations. So the time complexity is $O(k^3)$.

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.
Answer.
No. For example. an undirected graph $A(4) - B(5) - C(4)$, B has the highest value, and we need to remove A and C. We get value of 5. However, if we choose and C, we can get value of 8.

(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!]

Algorithm 4 Cake schedule

```
Input: total number of cake slices $K$
Output: the optimal schedule.
Data structure: create an array $Sat$ of $k \times k$ to store the results and initialize all the elements to -1.
function Satisfaction(k slices left, t current day)
    if k==0 then
        $Sat[t,k] = 0$
    end if
    for each $i$ in [1,2,...,k] do
        if Sat[t+1,k-i]==-1 then
            Satisfaction(k-i,t+1)                                  ▷ We need to update it first.
        end if
        $Sat[t,k] = max\big(Sat[t,k], \beta^{t-1}\sqrt{i} + Sat[t+1,k-i]\big)$ ▷ Select the one with maximum satisfaction.
    end for
end function
t=1,slice=K, Sched=[]
Satisfaction(K,day)
while $slice > 0$ do
    $slice\_cur = argmax_{1 \le j \le slice}\big(\beta^{t-1}\sqrt{j} + Sat[t+1, slice - j]\big)$
    Sched.append([t,slice_cur])
    t+=1
    slice=slice-slice_cur
end while
```

Answer.
The pseudo code is shwon in Algorithm 5.
Correctness. All the solution space is traversed which means that all possibility has been considered. We start from the root and for each node we either include all the children or grandchildren.
Time Complexity. The number of node is n, where each node being visited once. And for each sub-problem we need to consider 2 possible situations, and the number of addition depends on the number of *edge*. So the time complexity should be polynomial in n.

Algorithm 5 Conflict-free subsets

```
Input: graph $G$, Input weight array $w$
Output: The maximum value

function MIS-TREE(G)
    Let $v_1, v_2, ...v_n$ be a post-order traversal of nodes of $G$.    ▷ node_n is the root node.
    Initialize an array M to value 0 of size $n$
    Initialize an array $result$ as []
    for each $i$ in [1,2,...,n] do
        $contain = w[i] + \sum_{v_j \in Grandchildrenofv_i} M[j]$
        $notcontain = \sum_{v_j \in Childrenofv_i} M[j]$
        if $contain > notcontain$ then
            $M[i] = contain$
            $result.append(v_i)$
        else
            $M[i] = notcontain$
        end if
    end for
    return result
end function
```