CS 381 — Introduction to the Analysis of Algorithms
Fall 2021 — Simina Branzei and Alexandros Psomas — HW 3

Due Fri October 15 at 11:59PM

1. (7 points)

   (a) Prove that $\sum_{i=1}^{n} i^4 \le n^5$ for all $n \ge 1$.

   (b) Prove by induction that $\frac{n^5}{5} \le \sum_{i=1}^{n} i^4$ for all $n \ge 1$.

   (c) Conclude that $\sum_{i=1}^{n} i^4$ is $\Theta(n^5)$

   Answer:

   (a) $\sum_{i=1}^{n} i^4 \le \sum_{i=1}^{n} n^4 = n^5$.

   (b) We proceed by induction on n. In the base case of $n = 1$, $1/5 \le 1$ is true. Our inductive hypothesis is that $n^5/5 \le \sum_{i=1}^{n} i^4$ for some $n \ge 1$. In the inductive step, we seek to prove that $(n+1)^5/5 \le \sum_{i=1}^{n+1} i^4$. Starting from the right hand side and using the IH to replace $\sum_{i=1}^{n} i^4$,

$$\sum_{i=1}^{n+1} i^4 = \sum_{i=1}^{n} i^4 + (n+1)^4 \tag{1}$$
$$\ge n^5/5 + (n+1)^4 \tag{2}$$
$$= \frac{1}{5}n^5 + n^4 + 4n^3 + 6n^2 + 4n + 1 \tag{3}$$
$$\ge \frac{1}{5}n^5 + n^4 + 2n^3 + 2n^2 + n + \frac{1}{5} \tag{4}$$
$$= \frac{1}{5}(n+1)^5 \tag{5}$$

   This concludes the induction. Therefore $\frac{n^5}{5} \le \sum_{i=1}^{n} i^4$ for all $n \ge 1$.

   (c) Invoking the results of parts a and b and taking $c_1 = \frac{1}{5}$, $c_2 = 1$, $n_0 = 1$, we have that $c_1 n^5 \le \sum_{i=1}^{n} i^4 \le c_2 n^5$ for $n \ge n_0$. Therefore $\sum_{i=1}^{n} i^4$ is $\Theta(n^5)$.

2. (8 points)

   Consider the following recurrence:

   - $T(1) = T(2) = T(3) = 1000$
   - For $n > 3$, $T(n) = T(\lceil \frac{2n}{3} \rceil) + T(\lceil \frac{3n}{4} \rceil) - 2n - 1$

   (a) Find a constant c such that $T(n) \ge cn^2$ for $n \ge 1$. (Hint: if you're not sure what constant to pick, try starting part b first, and come back to this part)

   (b) Prove by induction that your chosen c is correct (i.e. that $T(n) \ge cn^2$ for $n \ge 1$).

(c) Conclude that $T(n)$ is $\Omega(n^2)$.

Answer:

(a) We'll use $c = 100$, but other values work too.

(b) We proceed by induction on n. We have three base cases:

- $T(1) = 1000 \geq 100$
- $T(2) = 1000 \geq 400$
- $T(3) = 1000 \geq 900$

Our inductive hypothesis is that $T(k) \geq 100k^2$ for all $1 \leq k < n$. In the inductive step, we seek to prove $T(n) \geq 100n^2$ where $n \geq 4$. Starting with the recurrence and replacing terms using the IH,

$$\begin{aligned}
T(n) &= T(\lceil \frac{2n}{3} \rceil) + T(\lceil \frac{3n}{4} \rceil) - 2n - 1 && (6)\\
&\geq 100\lceil \frac{2n}{3} \rceil^2 + 100\lceil \frac{3n}{4} \rceil^2 - 2n - 1 && (7)\\
&\geq \frac{6400}{144}n^2 + \frac{8100}{144}n^2 - 2n - 1 && (8)\\
&\geq 100n^2 + (\frac{100}{144}n^2 - 2n - 1) && (9)\\
&\geq 100n^2 && (10)\\
& && (11)
\end{aligned}$$

Where the last inequality is because $n \geq 4$, $\frac{100}{144}4^2 - 2 \cdot 4 - 1$, and $\frac{d}{dn}\frac{100}{144}n^2 - 2n - 1 = \frac{200}{144}n - 2 > 0$ for $n \geq 4$. This concludes the induction; therefore $T(n) \geq 100n^2$ for all $n \geq 1$.

(c) Invoking the prior result, we have $T(n) \geq cn^2$ for all $n \geq n_0$ where $c = 100$ and $n_0 = 1$. Therefore $T(n)$ is $\Omega(n^2)$.

3. (5 + 5 = 10 points) Consider the following pseudo code for finding the length of the longest common sub sequence (LCS) of two strings X and Y.

```
LCS-LENGTH(X,Y){
    m = X.length
    n = Y.length
    for i = 1 to m{
        c[i][0] = 0;
    }
    for j = 1 to n{
        c[0][j] = 0;
    }
    for i = 1 to m{
        for j= 1 to n{
            if x[i] == y[j]{
                c[i][j] = c[i-1][j-1] + 1
            }
            else{
                c[i][j] = max(c[i-1][j],c[i][j-1]);
            }
        }
    }
    return c;
}
```

- Given an input $X = AGGTAB$ and $Y = GXTXAYB$, show the contents of the 2-Dimensional array c upon calling LCS-LENGTH(X,Y). Note that you only need to show the final contents i.e., after the pseudo code stops running.
- Given an input $X = ABCDE$ and $Y = BCE$, show the contents of the 2-Dimensional array c upon calling LCS-LENGTH(X,Y).

Answer:

(a)

Index	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1
2	0	1	1	1	1	1	1	1
3	0	1	1	1	1	1	1	1
4	0	1	1	2	2	2	2	2
5	0	1	1	2	2	3	3	3
6	0	1	1	2	2	3	3	4

(b)

Index	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	1	1	1
3	0	1	2	2
4	0	1	2	2
5	0	1	2	3

4. (12 + 3 = 15 points) Gotham city is in trouble again! The Riddler has placed bombs in every major part of the city and they will explode tonight unless Batman stops him. Now, the Riddler wants to challenge Batman to a battle of wits and thus he presents a way for the Batman to defuse the bombs. To do so, all Batman has to do is solve an algorithmic problem.

   Given two strings X and Y, Batman has to find the length of the longest common substring among X and Y. Batman had written a naive solution earlier which had worked for small inputs. But, the Riddler drastically increased the input size such that no algorithm that runs slower than $O(m \cdot n)$ - where m, n are the lengths of strings X, Y respectively - can find the longest common substring before the bomb explodes. Batman stands thoroughly beaten by this challenge. Can you help him to obtain an $O(m \cdot n)$ algorithm to find the length of the longest common substring?

   Note: A substring is a contiguous sequence of characters in a string. For instance, "Computer Science" is a substring of "Computer Science is fun", while "Computer fun" is not. You can assume that $m, n \geq 1$.

   Example: Given X = ABABC and Y = BABCA, the longest common substring is "BABC" and its length is 4.

   - Find, describe, and prove the correctness of an algorithm to find the length of the longest common substring. It must run in $O(m \cdot n)$ time or better; faster is possible but not at all required. (12 points)
   - Analyze the algorithm and show it runs in $O(m \cdot n)$. (3 points)

   Hint: try using $OPT(i, j)$ = the length of the longest common substring of the first i characters of X and the first j characters of Y.

   Answer: Here is a Dynamic Programming solution that runs in $O(m \cdot n)$. Note that better algorithms are possible, but not required.

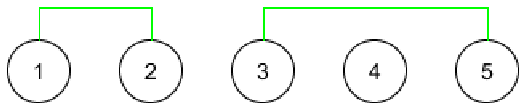
```
LONGEST-COMMON-SUBSTRING(X,Y){
    m = X.length;
    n = Y.length;
    result = 0;
    for i = 1 to m{
        LCSub[i][0] = 0;
    }
    for j = 1 to n{
        LCSub[0][j] = 0;
    }
    for i = 1 to m{
        for j= 1 to n{
            if x[i - 1] == y[j - 1]{
                LCSub[i][j] = LCSub[i - 1][j - 1] + 1;
                result = max(result,LCSub[i,j]);
            }
            else{
                LCSub[i][j] = 0;
            }
        }
    }
    return result;
}
```

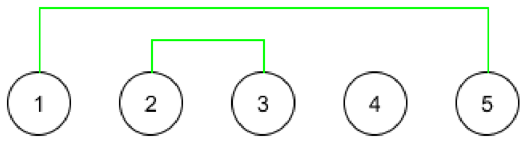
Note that the run time would be

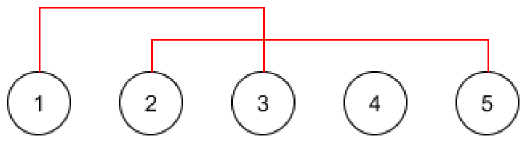
$$T(m,n) = O(m) + O(n) + O(m \cdot n) = O(m \cdot n)$$

For correctness, observe that at any step the entry in the LCSub matrix holds the correct length of the longest sub sequence including the characters at position i and j assuming that all the entries in the matrix until that point are correct. From there, we can use induction to see that the entire algorithm is correct.

5. (30 points) We say that two pairs of distinct natural numbers (a,b) and (c,d) with $a < b$ and $c < d$ are "entangled" if either $a < c < b < d$ or $c < a < d < b$. For example, the pairs $(1,2)$ and $(3,5)$ are not entangled, nor are $(1,5)$ and $(2,3)$, but $(1,3)$ and $(2,5)$ are entangled.







The order in a pair doesn't matter, so $(3,1)$ and $(2,5)$ are still entangled. You are given an $n \times n$ matrix V of non-negative "pair values"; i.e. $V_{i,j}$ is the value of the pair (i,j). You cannot pair a number with itself, so $V_{i,i} = 0$ for all $1 \leq i \leq n$. The pair (i,j) is the same as the pair (j,i), so $V_{i,j} = V_{j,i}$ for all $1 \leq i < j \leq n$. Your goal is to determine the maximum value possible from forming pairs from the numbers $1,2,\ldots,n$ such that no two pairs are entangled. Each number may be paired at most once. It is not necessary to pair every number.

For example, suppose we had $n = 5$ with

$$V = \begin{bmatrix} 0 & 2 & 3 & 5 & 1 \\ 2 & 0 & 4 & 12 & 3 \\ 3 & 4 & 0 & 8 & 2 \\ 5 & 12 & 8 & 0 & 6 \\ 1 & 3 & 2 & 6 & 0 \end{bmatrix} \tag{12}$$

$(1,2),(3,4)$ is a valid pairing, and has value $2+8=10$. $(1,5),(2,3)$ is a valid pairing, and has value $1+4=5$. $(2,4)$ is a valid pairing, and has value 12. It is also the optimal pairing. $(1,3),(2,4)$ is not a valid pairing, because $(1,3)$ and $(2,4)$ are entangled. $(1,2),(2,4)$ is not a valid pairing, because 2 appears in multiple pairs.

Find an algorithm with fastest asymptotic time complexity for this problem. Then describe it, prove its correctness, and prove its asymptotic time complexity. You do not need to return the optimal pairing, only its value. You do not need to prove that your time complexity is the fastest possible. Solutions which require exponential time are too slow, and will not receive credit. You do not need to prove anything regarding space complexity.

Answer: Define $OPT(i,j)$ to be the best value possible using only numbers $i, i+1, \ldots, j$. I.e., our goal is to compute $OPT(1,n)$. If $j - i < 1$, then no pairs are possible and $OPT(i,j) = 0$. Otherwise, we have a decision to make: what, if anything, will i be paired with? If we don't pair i with anything, then the best we can do is $OPT(i+1,j)$. Otherwise, let i's pair be (i,k). Then numbers in the range $[i+1,k-1]$ can only be paired with each other, and similarly numbers in the range $[k+1,j]$ can only be paired with each other, lest a pair be entangled with (i,k). Therefore the best we can do is $V_{i,k} + OPT(i+1,k-1) + OPT(k+1,j)$. This yields the recurrence $OPT(i,j) = \max(OPT(i+1,j), \max_{k\in[i+1,j]}(V_{i,k} + OPT(i+1,k-1) + OPT(k+1,j)))$.

Note that the recurrence for $OPT(i,j)$ only relies on values of OPT for strictly smaller regions; i.e. $OPT(a,b)$ where $b-a < j-i$. Because of this, we can build up a matrix of values of OPT from the bottom up by first considering cases where $j-i=1$, then $j-i=2$, as so on until $j-i=n-1$. (cases where $j-i<1$ can be filled in first with 0). Afterwards we return $OPT(1,n)$.

To prove this algorithm is correct, we claim, and will prove by strong induction, that $OPT(i,j)$ really is the largest value possible by pairing only numbers from i to j. In the base case, we consider $OPT(i,j)$ where $j-i<1$. These values are computed to be 0 by definition. This is the correct value, since if $j-i<1$ then there is at most 1 number in $[i,j]$, so no pairs are possible.

Our inductive hypothesis is that the recorded values of OPT are correct where $j-i<m$. We seek to prove it is correct where $j-i=m$. All of $j-(i-1)$, $(k-1)-(i+1)$, and $j-(k+1)$ are less than $j-i$ for $k \in [i+1,j]$, so the inductive hypothesis tells us that the values of OPT used in the recurrence are correct. Therefore $OPT(i,j)$ is also correct. This concludes the induction. Therefore, $OPT(1,n)$ is correct, which is also the final answer. Thus the algorithm is correct.

Computing $OPT(i,j)$ given the previous values of OPT can be done via the recurrence in $O(n)$ time, since it's the max of n things each of which takes $O(1)$ time to compute. The full matrix has less than n^2 entries, so in total this takes $O(n^3)$ time to fill in. Afterwards, we get the answer in $O(1)$ time by returning $OPT(1,n)$. Therefore the whole algorithm takes $O(n^3)$ time.

6. (10 + 10 + 5 points) Consider the problem of giving change "optimally" in American denominations. You are given a value V and are asked how much of each American coin and bill do you give back to have the total add up to V and such that you return the fewest bills and coins possible. Assume for this problem American Denominations are $\{\$20.00, \$10.00, \$5.00, \$1.00, \$0.10, \$0.05, \$0.01\}$ (note: there are No Quarters). If no combination exists for a given value (such as $V = \$10.005$) then the algorithm should return that there is no combination possible for this value.

   For example, if the input to the algorithm was V = \$31.27 the "optimal" answer would be: one 20, one 10, one 1, one quarter and two pennies.

   - Design an efficient polynomial time greedy algorithm that provides the "optimal" number of each denomination for an input value V.
   - Prove why a greedy approach will always find a correct and optimal solution. (Hint: use the fact that every denomination in question is a multiple of the next smallest).
   - Analyze and prove the algorithm's runtime (You may assume that $V < 2^n$ is the only input passed to the algorithm and that modulus operation can be computed in constant time).

   Answer: The psuedo code for the greedy algorithm is:

```
change(V){
    solution = ""
    for each ad in American Denominations{
        if(V / ad >= 1){
            solution.concatenate( (V / ad) + " " + ad + ", " )
            V = V % ad
        }
    }
    if( V != 0 ){
        return "No combination possible"
    }
    return solution
}
```

   Note: This algorithm returns a string but any return that describes how much of each denomination (such as an array) is acceptable.

   Proof of correctness and optimality, required, doesn't need to be as rigorous:

   In the American denominations (without the quarter) there exists a property that each denomination can be represented as multiple of the next smallest denomination, (ex. \$10.00 = 2*\$5.00). This property lets us prove two things:

   First, that if a method of making change exists (the problem is solvable) a greedy algorithm like the one above always returns some combination of denominations that result in the total value. That is to say, the greedy algorithm will only return that there is no combination of the denominations that add up to V if none exist.

Second, the optimal solution is found by using as many as the largest denominations possible. For the sake of contradiction, assume that a greedy solution was found but it is not the optimal solution. In the optimal solution there exists the first denomination, D_i, where the greedy algorithm took k more than the optimal solution for this denomination. Because both solutions add up to V, that means the value of denominations less than i in the optimal solution must add up to the the value for the same denominations in the greedy solution plus $k * D_i$

Lemma: Due to the above property, $k * D_i$ must be represented by at least $M_{i-1} * k$ other denominations of lesser value (where $D_i = M_{i-1} * D_{i-1}$ in the property). Because D_i is the first denomination where the greedy algorithm took more than the optimal solution, all the denominations that make up this difference must be smaller. This alone doesn't prove that $M_{i-1} * D_{i-1}$ denominations must be used, but consider representing $k * D_i$ solely in D_{i-1}: $k * D_i = k * M_{i-1} * D_{i-1}$ this equation shows that for an arbitrary denomination D_i and arbitrary k, representing $k * D_i$ in smaller denominations always uses at least $k * M_{i-1}$ denominations (one could choose to represent D_{i-1} in terms of D_{i-2}, but the problem of adding more denominations exists).

This means that solution is less optimal than the greedy solution because $k < k * M_{i-1}$. By contradiction, this proves that the optimal solution cannot choose a number of denominations less that the greedy solution for every denomination. Because the optimal solution cannot choose less than the greedy solution and the greedy solution always choose the maximum, the greedy solution is the optimal solution.

This algorithm's runtime is constant in the input, as it loops based on the number of denominations. The work done in each iteration is 1 divide and 1 modulus operation, so the work is constant in each iteration.

Further reading: the property that makes this work with the quarter as well is that for "canonical" coin systems (where canonical means that greedy works on change giving) is that for every denomination is either a multiple of the next smallest or the smallest multiple of the next smallest that is greater than the current denomination is not a counterexample to the greedy algorithm. Example: the coin system including the quarter is canonical because 30 (the smallest multiple of 10 greater than 25) is not a counter example (25 + 5 beats 10 + 10 + 10).

7. (5 points) Consider the problem of giving change "optimally" in n arbitrary denominations. You are given a value $V < 2^n$ and a set of denominations D_n and are asked how much of each denomination do you give back to have the total add up to the value and such that you return the fewest denominations possible, or return that there is no combination of the denominations that add up to V

   For example, if the input to the algorithm was $V = 3.32$ and $D_n = \{1.27, 0.67, 0.13\}$ the "optimal" answer would be: two 1.27s and six 0.13s.

   Either design/modify an efficient polynomial time greedy algorithm to solve this problem and prove it's correctness and runtime, or justify why no polynomial time greedy algorithm can correctly solve this problem for all cases. (You may assume that $V < 2^n$ is the only input passed to the algorithm and that modulus operation can be computed in constant time).

   Answer: Consider a modification of the above algorithm but now to work for arbitrary denominations:

```
change(V, D){
    solution = ""
    for each ad in D{
        if(V / ad >= 1){
            solution.concatenate( (V / ad) + " " + ad + ", " )
            V = V % ad
        }
    }
    if(V != 0){
        return "No combination possible"
    }
    return solution
}
```

   This algorithm is no longer correct. If the example was inputted into this algorithm it would return "No combination possible" as this greedy algorithm would try to fit the maximum number of 0.67s into V as possible, and therefore not find the correct answer.

   A sufficient answer to this question talks about how greedy algorithms cannot be used to solve this problem due to fact that arbitrary denominations do not have the same property as the one used to prove the correctness of the American denomination algorithm, namely that taking the maximum amount of the largest denominations possible does not always lead to a correct answer, so a greedy algorithm cannot be used.

   More generally, no polynomial time algorithm (yet known) exists to solve this problem because this problem is in NP. This is a variation of the Subset Sum problem.