Comp 251: Practice problems

Instructor: Jérôme Waldispühl

This is a collection of problems that you can use to prepare the COMP 251 final exam. This selection of problems covers the second part of the class (i.e. after the mid-term exam). However, the final exam will also cover the material of the first part of the class. Please, refer to the practice problems for the mid-term, and solution of the midterm (last lecture) to fully prepare the exam.

Dynamic programming

1. The Coin Row Problem: Suppose you have a row of coins with values that are positive integers $c_1, \cdots, c_n$. These values might not be distinct. Your task is to pick up coins have as much total value as possible, subject to the constraint that you don't ever pick up two coins that lie beside each other. How would you solve this using dynamic programming?
   Solve the problem for coins with values c_1 to c_6 as follows: $(5, 1, 2, 10, 6, 2)$.

   Solution: The idea for the recurrence is as follows. Start with the last coin. You either pick it up or you don't. If you pick it up, then you cannot pick up the second to last coin but you are free to pick up any others. If you don't pick up the last coin, then you are free to pick up any of the others (subject to the problem's constraints). The recurrence that describes this is $f(n) = \max(c_n + f(n-2), f(n-1))$, with base case $f(1) = c_1$, $f(0) = 0$.
   You can solve this either iteratively or recursively using dynamic programming. For the example given, the maximum value is 17 and uses coins $\{c_1 = 5, c_4 = 10, c_6 = 2\}$.

2. The Coin Change Problem: Suppose we have m types of coins with values $c_1 < c_2 < \cdots c_m$ (e.g. in the case of pennies, nickels, dimes, ... we would have $c_1 = 1$, $c_2 = 5$, $c_3 = 10$, $\cdots$). Let $f(n)$ be the minimum number of coins whose values add up to exactly n. Write a recurrence for $f(n)$ in terms of the values of the coins. You may use as many of each type of coin as you wish.
   As an example, suppose the coin values c_1, c_2, and c_3 are 1, 3, 4. Solve the problem for $n = 6$ using dynamic programming.

Solution: To write the recurrence, we consider the case that a coin of type j was used in the optimal solution. Then, if a coin of type j was used, we need to solve the subproblem of finding the minimum number of coins whose values sum up to $n - c_j$. Thus, $f(n) = min_{j \in \{1 \cdots m\}} 1 + f(n - c_j)$, $f(0) = 0$.
For the example given, the solution is two coins of value 3 each.

3. What is the optimal substructure of the Neddleman-Wunch algorithm (i.e. optimal pairwise sequence alignment)?

Solution: Recall the definition of the optimal substructure: "The optimal solution for one problem instance is formed from optimal solutions for smaller problems".
The optimal substructure is usually materialized in the dynamic programming equations. In the Neddleman-Wunch algorithm these equations are:

$$NW(i,j) = \max \begin{cases} NW(i-1,j) + \delta(a_i, -) & \text{(deletion)} \\ NW(i-1,j-1) + \delta(a_i, b_j) & \text{(match or mismatch)} \\ NW(i,j-1) + \delta(-, b_j) & \text{(insertion)} \end{cases}$$

Where a and b are the sequences to align, δ the edit cost function, and $NW(i,j)$ is the dynamic programming table that stores the score of the optimal alignment of the prefix $a_{1 \cdots i}$ and $b_{1 \cdots j}$.

The last column of any alignment is either a deletion, match/mismatch or insertion. Then, the optimal alignment of two string $a_{1 \cdots i}$ and $b_{1 \cdots j}$ is made from the concatenation of (i) an optimal alignment of $a_{1 \cdots i-1}$ and $b_{1 \cdots j}$ if the alignment ends with a deletion, (ii) an optimal alignment of $a_{1 \cdots i-1}$ and $b_{1 \cdots j-1}$ if the alignment ends with a match or mismatch, or (iii) an optimal alignment of $a_{1 \cdots i}$ and $b_{1 \cdots j-1}$ if the alignment ends with a insertion. The optimal alignment is this made from the best option among those three.

We note that the sub-problems are strictly smaller since the length of at least one of the two sequences has been reduce by 1.

Divide-and-Conquer

4. In Karatsuba multiplication, when you do the multiplication $(x_1 + x_0) \cdot (y_1 + y_0)$, the two values you are multiplying might be $n/2 + 1$ digits each, rather than $n/2$ digits, since the addition might have led to a carry e.g. $53 + 52 = 105$. Does this create a problem for the argument that the recurrence is $t(n) = 3t(n/2) + c_n$?

Solution: The recurrence would need to be written $t(n) = 2t(n/2) + t(n/2+1) + c_n$. We know that $t(n)$ is $O(n^2)$ and we are trying to prove a better bound, but the fact that it is $O(n^2)$ allows us to say that $t(n) \leq c \cdot n^2$ for some constant c and for sufficiently large n (Recall the formal definition from COMP 250). Thus, $t(n/2+1) \leq c \cdot (n/2+1)^2 = c(n/2)^2 + c_1 n/2 + c$ for some constant c and for sufficiently large n. Thus, $t(n/2+1) = t(n/2) + O(n)$, that is, $t(n/2+1)$ is bigger than $t(n/2)$ but only by an amount that grows linearly with n. Thus, we can write:

$$t(n) = 2t(n/2) + t(n/2+1) + cn = 3t(n/2) + O(n).$$

In case the above went too fast, here is the basic idea: there is no problem that the Karatsuba trick requires multiplying two numbers of size $n/2+1$ instead of $n/2$. The reason it doesn't matter is that the extra work you need to do is bounded by some O(n) term. You are already doing a bunch of $O(n)$ work at each node (of the call tree). So one extra $O(n)$ term won't make any difference.

5. Apply the master method to determine the asymptotic behavior of the function $T(n)$.

   1. $T(n) = 2 \cdot T(n/4) + n^{0.51}$
   2. $T(n) = 0.5 \cdot T(n/2) + 1/n$
   3. $T(n) = 64 \cdot T(n/8) - n^2 \cdot logn$
   4. $T(n) = \sqrt{2} \cdot T(n/2) + \log n$
   5. $T(n) = 6 \cdot T(n/3) + n^2 \cdot \log n$
   6. $T(n) = 3 \cdot T(n/3) + n/2$

Solution:

1. Case 3: $T(n) = \Theta(n^{0.51})$
2. Does not apply: $a < 1$
3. Does not apply: $f(n)$ not positive
4. Case 1: $T(n) = \Theta(\sqrt{n})$
5. Case 3: $T(n) = \Theta(n^2 \cdot \log n)$
6. Case 2: $T(n) = \Theta(n \cdot \log n)$

6. Write a recurrence that describes its worst-case running time of the quicksort algorithm.

Solution: $T(n) = T(n-1) + T(0) + \Theta(n)$

Amortized analysis

7. Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Solution: Charge \$2 for each PUSH and POP operation and \$0 for each COPY. When we call PUSH, we use \$1 to pay for the operation, and we store the other \$1 on the item pushed. When we call POP, we again use \$1 to pay for the operation, and we store the other \$1 in the stack itself. Because the stack size never exceeds k, the actual cost of a COPY operation is at most \$k, which is paid by the \$k found in the items in the stack and the stack itself. Since there are k PUSH and POP operations between two consecutive COPY operations, there are \$k of credit stored, either on individual items (from PUSH operations) or in the stack itself (from POP operations) by the time a COPY occurs. Since the amortized cost of each operation is $O(1)$ and the amount of credit never goes negative, the total cost of n operations is $O(n)$.

8. Suppose we perform a sequence of n operations on a data structure in which the i^{th} operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis or accounting method to determine the amortized cost per operation.

Solution:

Aggregate analysis:

Let c_i be the cost of i^{th} operation.

$$c_i = \begin{cases} i & \text{if i is an exact power of 2} \\ 1 & otherwise \end{cases}$$

n operations cost: $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log n} 2^j = n + (2n-1) < 3n$. (Note: We ignoring floor in upper bound of $\sum 2^j$).

Thus the Average cost of operation = Total cost < 3 number of operations. And by aggregate analysis, the amortized cost per operation = O(1).

Accounting method:

Charge each operation \$3 (amortized cost $\hat{c}_i$).

- If i is not an exact power of 2, pay \$1, and store \$2 as credit.
- If i is an exact power of 2, pay \$i, using stored credit.

Operation	Cost	Actual cost	Credit remaining
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Since the amortized cost is \$3 per operation, we have $\sum_{i=1}^{n} \hat{c}_i = 3n$. Moreover, from aggregate analysis, we know that $\sum_{i=1}^{n} c_i < 3n$. Thus $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \Rightarrow$ credit never goes negative.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of n operations is $O(n)$.