Illinois Institute of Technology
Department of Computer Science

Solutions to Homework Assignment 4

CS 430 Introduction to Algorithms
Spring Semester, 2016

Solution:

1. (a) Let $S_{i,j}$ represents the state that India still need i more victories to win, whereas Pakistan needs j more victories for the championship. There are only two possible states $S_{i-1,j}$ and $S_{i,j-1}$ after one game from $S_{i,j}$. Given that $S_{i-1,j}$ happens, there are two possible previous states. The possibility that the previous state is $S_{i,j}$ is p which indicates India win the game. Given that $S_{i,j-1}$ happens, the possibility that the previous state is $S_{i,j}$ is q, which indicates Pakistan win the game. Together, the possibility that we have state $S_{i,j}$ is $P_{i,j} = pP_{i-1,j} + qP_{i,j-1}$.

   (b) P_{00} is the possibility that Indian will win the series given that they still need 0 more victories, whereas Pakistan needs 0 more victories. This indicates that both India and Pakistan have won the game, which is impossible. Thus $P_{0,0} = 0$.

   (c) The recursive formation of the dynamic programming algorithm is given by

$$P_{i,j} = \begin{cases} 1, i = 0 \wedge 1 \leq j \leq n \\ 0, j = 0 \wedge 1 \leq i \leq n \\ pP_{i-1,j} + qP_{i,j-1}, otherwise \end{cases}$$

   (d) The memoized dynamic programming algorithm is given by algorithm 1. The running time of this algorithm is $O(n^2)$ as it takes constant time to calculate each entry of Matrix P.

Algorithm 1 India Win: Dynamic Programming

```
Let $P$ be matrix of size $(n+1)*(n+1)$
$P_{0,0} = 0$
for $j \leftarrow 1$ to n do
    $P_{0,j} = 1$
end for
for $i \leftarrow 1$ to n do
    $P_{i,0} = 0$
end for
for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
        $P_{i,j} = pP_{i-1,j} + qP_{i,j-1}$
    end for
end for
return $P_{n,n}$
```

2. (a) The greedy algorithm: each time, we pick the coin with the largest denomination that is smaller or equal to the change. The remaining amount of change is reduced by the value of the coin. We do the previous two steps repeatedly until the remaining amount of change is zero.

For a coin system consisting of quarters, dimes, nickels and pennies, this algorithm indicates that the number of quarters $a_q = \frac{n}{25}$, the number of dimes $a_d = \frac{n\%25}{10}$, the number of nickels $a_n = \frac{n\%10}{5}$, the number of pennies $a_p = n\%5$.

Proof of Correctness: To prove the correctness of the greedy algorithm for the coin system with consisting of quarters, dimes, nickels and pennies. We need the support of the following lemma:

Lemma 1 *In the optimal solution, we can have at most two dimes, one nickel and four cents; we also cannot have two dimes and one nickel at the same time.*

The correctness of this lemma is easy to prove. We can always replace three dimes with a quarter and a nickel, replace two nickels with one dime, replace five cents with one nickel, and replace two dimes and one nickel with a quarter. All these replacements will result in smaller number of coins. This lemma indicates that in the optimal solution, the total value dimes, nickels and pennies is no greater than 24 cents; the total value of nickels and cents is no greater than 9 cents; the total value of pennies is no greater than 4 cents.

Then we can prove the correctness of our algorithm for this coin system. Assume that there is an optimal solution OPT that use smaller number of coins to make the change. Suppose OPT uses $a_q^{'}$ quarters, $a_d^{'}$ dimes, $a_n^{'}$ nickels and $a_p^{'}$ pennies. Since our greedy algorithm use as many quarters as possible, $a_q^{'} \leq a_q$. If $a_q^{'} < a_q$, the total value of dimes, nickels and cents in OPT will be greater than 24 cents since both our greedy algorithm and OPT make change for the same amount. This goes against our lemma. Then $a_q^{'} = a_q$. Following the same procedure, we can prove $a_d^{'} = a_d$, $a_n^{'} = a_n$ and $a_p^{'} = a_p$. This means OPT will use the same number of coins as our algorithm, which goes against the assumption that OPT uses smaller number of coins.

(b) Assume that there is an optimal solution OPT that use smaller number of coins to make the change. Suppose the greedy algorithm uses a_0 coins with denomination c^0, a_1 coins with denomination c^1, ..., a_k coins with denomination c^k. Suppose the corresponding number in OPT are $a_0^{'}$, $a_1^{'}$, ..., $a_k^{'}$ respectively. Since our greedy algorithm use as many coin with largest denomination as possible, $a_k^{'} \leq a_k$. If $a_k^{'} < a_k$, the total value of the rest of the coins in OPT will be larger than c^k. In this coin system, this means that there are some coins with total value equal to c^k in the remaining coins. Then we can always replace these coins with one coin with value c^k. Thus $a_k^{'} = a_k$. Following the same procedure, we can prove $a_i = a_i^{'}$ for $i = k-1, k-2, ..., 0$. This means OPT will use the same number of coins as our greedy algorithm, which goes against the assumption.

(c) Suppose the coin denominations are $\{1, 7, 10\}$. If we want to make a change of 15 cents. The solution given by the greedy algorithm is one coin of 10 and five coins of 1. The optimal solution uses two coins of 7 and one coin of 1.

(d) Recursive form: let $f(n)$ denote the minimum number of coins we use to make change for n coins.Let d_i denote the i-th denomination. Then it will have the following form of recursion:

$$f(n) = \begin{cases} \infty, n < 0 \\ 0, n = 0 \\ \min\limits_{1 \leq i \leq k} f(n - d_i) + 1, n > 0 \end{cases}$$

The iterative form will be represented in algorithm 2. The array M stores the minimum number of coins to make change for n cents. D stores the coins used. This algorithm runs in $O(nk)$ time. There are n steps in the outer loop and each inner loops contains k steps.

(e) Let A be the set of denominations. Let $g(n, A)$ be the function that returns 1 if we can make change for n using denominations in A, and returns 0 otherwise. The recursive formation is

Algorithm 2 Make Change: Dynamic Programming

```
Let $M$ be an array of size $n+1$
Let $D$ be an array of size $n+1$
$M[0] \leftarrow 0$
for $i \leftarrow 1$ to n do
    $M[i] \leftarrow \infty$
    for $j \leftarrow 1$ to k do
        if $1 + M[i - d_j] < M[i]$ then
            $M[i] \leftarrow 1 + M[i - d_j]$
            $D[i] = d_j$
        end if
    end for
end for
```

$$g(n, A) = \begin{cases} 1, n = 0 \\ 0, n < 0 \vee (n \neq 0 \wedge A = \Phi) \\ \bigvee\limits_{1 \le i \le k} g(n - d_i, A - d_i), otherwise \end{cases}$$

3. For a TABLE-DELETE operation that does not incur contraction, the amortized cost is:

$$\begin{aligned} \widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) &\le 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\ &= \begin{cases} 1 + 2(num_i - num_{i-1}) = -1 & 2num_i \ge size_i \\ 1 - 2(num_i - num_{i-1}) = 3 & 2num_{i-1} \le size_i \\ 1 + 2size_{i-1} - 2(num_i + num_{i-1}) & 2num_i < size_i < 2num_{i-1} \end{cases} \end{aligned}$$

The term in the third case is equal to

$$1 + 2size_{i-1} - 2(num_{i-1} - 1 + num_{i-1}) = 3 + 2(size_{i-1} - 2num_{i-1}) < 3$$

because $size_{i-1} = size_i < 2num_{i-1}$ (case 3 condition). Therefore, the upper bound in this case is 3, which is $O(1)$.

For a TABLE-DELETE operation that incurs contraction, the amortized cost is:

$$\begin{aligned} \widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) &\le num_i + 1 + |2 \cdot num_i - size_i| - |2 \cdot num_{i-1} - size_{i-1}| \\ &= num_i + 1 + |2 \cdot num_i - \frac{2}{3}size_{i-1}| - |2 \cdot num_{i-1} - size_{i-1}| \\ &= num_i + 1 + 0 - |2 \cdot num_{i-1} - size_{i-1}| \qquad (num_i = \frac{1}{3}size_{i-1}) \\ &= num_i + 1 + 2 \cdot num_{i-1} - size_{i-1} \qquad (2num_{i-1} < size_{i-1}) \\ &= 3 \in O(1) \end{aligned}$$

4. (a) When inserting an element at the head, push the element to the *Head.* When inserting an element at the tail, push the element to the *Tail.*

(b) Without the loss of generality, I assume it is the *Tail* stack which is empty. For the case where *Head* is empty, it is trivial to get the similar algorithm based on the following one.

   i. Iteratively POP each element from *Head* and PUSH it into *Temp* until the last element.
   ii. POP the last element in *Head* and discard it.
   iii. In the remaining $n-1$ elements in *Temp*, do the following to the first $(n-1)/2$ elements:
   · Iteratively POP and PUSH each of them to *Head*.
   · Iteratively POP and PUSH every element from *Head* to *Tail*.
   iv. For the remaining $(n-1)/2$ elements in *Temp*, do the following: iteratively POP and PUSH each of them to *Head*.

(c) **Insertion**

In any case (either best, average or worst), the complexity of insertion is $O(1)$.

Deletion

In fact, the worst case is when we need to delete an element from the tail but *Tail* is empty or we need to delete an element from the head but *Head* is empty. Then, we can simply look at the above algorithm to find out the worst-case cost.

Suppose the costs of POP and PUSH are both 1, then the step 1's cost is $2(n-1)$. The cost of step 2 is 1. The cost of 3.(a) is $(n-1)/2 \times 2 = n-1$. The cost of 3.(b) is $(n-1)/2 \times 2 = n-1$. The cost of 4 is $(n-1)/2 \times 2 = n-1$. Sum of the cost above is $5(n-1)+1 = 5n-4$.

(d) **Wost-case deletion** Similarly, I assume it is *Tail* stack who is empty. Suppose the potential function is $\Phi(D_i) = k\Big||Head_i| - |Tail_i|\Big|$, where $Xxxx_i$ refers to the corresponding stack after the i-th operation. Then, the amortized cost is:

$$\begin{aligned}\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) =& 5n-4+k\Big||Head_i| - |Tail_i|\Big| - k\Big||Head_{i-1}| - |Tail_{i-1}|\Big| \\ &= 5n-4+k\Big||(n-1)/2| - |(n-1)/2|\Big| - k\Big|n-0\Big| \\ &= 5n-4-kn\end{aligned}$$

which should be $O(1)$. Therefore, $k=5$, and the potential function is $\Phi(D_i) = 5\Big||Head_i| - |Tail_i|\Big|$.

Normal deletion and insertions

In any case, it is easy to derive that the amortized time of deletion in the normal case is between $[1-k, 1+k]$, which is always constant. The amortized time of both insertions is same. Therefore, k can be any number for normal deletion or insertions.

Conclusion

In conclusion, k has to be 5 to have constant amortized time for all four operations.