

CSE 201

Homework Assignment 5

Solutions

1. Define the sequence $s(n)$ by the recurrence $s(n) = (n - 1) + \frac{n-1}{n^2} \cdot \sum_{k=1}^{n-1} s(k)$. Use induction to prove $s(n) \leq 2n$ for all $n \geq 1$. Also prove $s(n) = \Theta(n)$.

Proof:

I. Observe $s(1) = (1 - 1) + \frac{1-1}{1^2} \cdot (\text{empty sum}) = 0 \leq 2 = 2 \cdot 1$, establishing the base case.

II. Let $n > 1$, and assume for all k in the range $1 \leq k < n$ that $s(k) \leq 2k$. We must show that $s(n) \leq 2n$. We have

$$\begin{aligned} s(n) &= (n - 1) + \frac{n-1}{n^2} \cdot \sum_{k=1}^{n-1} s(k) \\ &\leq (n - 1) + \frac{n-1}{n^2} \cdot \sum_{k=1}^{n-1} 2k && \text{by the induction hypothesis} \\ &= (n - 1) + \frac{n-1}{n^2} \cdot 2 \cdot \frac{n(n-1)}{2} \\ &= (n - 1) + \frac{(n-1)^2}{n} \\ &= (n - 1) \left(1 + \frac{n-1}{n} \right) \\ &= (n - 1) \left(1 + 1 - \frac{1}{n} \right) \\ &= (n - 1) \left(2 - \frac{1}{n} \right) \\ &= 2n - 2 - 1 + \frac{1}{n} \\ &= 2n - 3 + \frac{1}{n} \\ &\leq 2n && \text{since } n > 1 \Rightarrow \frac{1}{n} \leq 1 \Rightarrow -3 + \frac{1}{n} \leq 0 \end{aligned}$$

as required. It follows that $s(n) \leq 2n$ for all $n \geq 1$.

We have $s(n) \leq 2n = O(n)$, and obviously $s(n) = (n - 1) + \frac{n-1}{n^2} \cdot \sum_{k=1}^{n-1} s(k) \geq (n - 1) = \Omega(n)$. Therefore $s(n) = \Theta(n)$, as required. ■

2. (This is a continuation of problem 6 on hw assignment 4, which was itself based on problem 8-5 on page 207 of CLRS 3rd edition.) Recall an array $A[1 \cdots n]$ is k -sorted if and only if the $(n - k + 1)$ consecutive k -fold averages of $A[1 \cdots n]$ are sorted:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

for $1 \leq i \leq n - k$. Observe that if $n = k$, then any array of length n is k -sorted, since there is only one k -fold average in this case. We may also *define* any $A[1 \cdots n]$ to be k -sorted if $n < k$, since in this case there are no k -fold averages. Modify Quicksort to produce an algorithm that k -sorts an array of any length. Write your algorithm in pseudo-code and call it $\text{Quick}[k]\text{sort}()$. Specifically, the call $\text{Quick}[k]\text{sort}(A, 1, n)$ will k -sort $A[1 \cdots n]$, but will not necessarily $(k - 1)$ -sort the same array. Prove the correctness of your algorithm, and analyze its worst case run time.

Solution:

The key to this problem is the observation that any array $A[1 \cdots n]$ is k -sorted, if $n \leq k$. We simply run the Quicksort algorithm, but only recur down to array size k , or smaller.

$\text{Quick}[k]\text{sort}(A, p, r)$

1. if $k < r - p + 1$
2. $q = \text{Partition}(A, p, r)$
3. $\text{Quick}[k]\text{sort}(A, p, q - 1)$
4. $\text{Quick}[k]\text{sort}(A, q + 1, r)$

Proof of correctness:

We show by induction by induction on $m = r - p + 1$, that $\text{Quick}[k]\text{sort}(A, p, r)$ correctly k -sorts any subarray $A[p \cdots r]$ of length $m \geq 1$.

- a. The base cases $m = 1, 2, \dots, k$ follow from the observations made in the problem statement, i.e. there is at most one k -fold average, and hence the consecutive k -fold averages cannot be out of order.
- b. Let $m > k$ and assume that $\text{Quick}[k]\text{sort}$ correctly k -sorts any subarray of length less than m . We must show that $\text{Quick}[k]\text{sort}(A, p, r)$ correctly k -sorts the subarray $A[p \cdots r]$ of length $m = r - p + 1$. Since $m > k$ the test on line 1 is true, and lines 2, 3 and 4 are executed. The call to $\text{Partition}(A, p, r)$ on line 2 rearranges $A[p \cdots r]$ and returns an integer q such that $p \leq q \leq r$ and

$$(*) \quad A[p \cdots (q - 1)] \leq A[q] < A[(q + 1) \cdots r]$$

holds. Since both

$$\text{length}(A[p \cdots (q - 1)]) = (q - 1) - p + 1 = q - p \leq r - p < r - p + 1 = m$$

and

$$\text{length}(A[(q + 1) \cdots r]) = r - (q + 1) + 1 = r - q \leq r - p < r - p + 1 = m,$$

the induction hypothesis guarantees that $A[p \cdots (q - 1)]$ and $A[(q + 1) \cdots r]$ are k -sorted after the recursive calls on lines 3 and 4 are executed. We show that this fact, together with (*), implies the subarray $A[p \cdots r]$ is now k -sorted.

The result of problem 6c on hw4 says that $A[p \cdots r]$ is k -sorted if and only if $A[i] \leq A[i + k]$ for all i in the range $p \leq i \leq r - k$. We have 3 cases to consider.

$$(1) \ p \leq i < i + k \leq q - 1$$

In this case, the fact that $A[p \cdots (q - 1)]$ is k -sorted implies $A[i] \leq A[i + k]$.

$$(2) \ q + 1 \leq i < i + k \leq r$$

Then $A[(q + 1) \cdots r]$ being k -sorted implies $A[i] \leq A[i + k]$.

$$(3) \ p \leq i < q \leq i + k \leq r$$

In this case, the inequality (*) itself implies $A[i] \leq A[i + k]$.

In all cases, $A[i] \leq A[i + k]$ for all i in the range $p \leq i \leq r - k$, and hence the full subarray $A[p \cdots r]$ is k -sorted, as required.

The result follows for all $m \geq 1$ by the 2nd Principle of Mathematical Induction. ■

Analysis of worst case runtime

The recurrence for worst case number of array comparisons is very similar to that for ordinary Quicksort, the only difference being where (i.e. for which n) the recurrence halts. As before, the worst case behavior is elicited when one of the two subarrays on lines 3 or 4 is empty.

$$T(n) = \begin{cases} 0 & 1 \leq n \leq k \\ T(n - 1) + (n - 1) & n > k \end{cases}$$

The iteration method yields $T(n) = \sum_{i=1}^s (n - i) + T(n - s)$, stopping at the smallest s for which $1 \leq n - s \leq k$, that is $s = n - k$. Thus

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-k} (n - i) \\ &= \sum_{i=1}^{n-k} n - \sum_{i=1}^{n-k} i \\ &= n(n - k) - \frac{(n - k)(n - k + 1)}{2} \\ &= \Theta(n^2). \end{aligned}$$

■

Remark

The average case runtime of Quick[k]sort can be shown to be $\Theta(n \log(n))$, as one would expect. We leave this as a further exercise for the reader. ■

3. Read the Coin Changing Problem in the handout on Dynamic Programming. Its solution is presented below for reference. Recall this algorithm assumes that an unlimited supply of coins in each denomination $d = (d_1, d_2, \dots, d_n)$ are available.

CoinChange(d, N)

1. $n = \text{length}[d]$
2. for $i = 1$ to n
3. $C[i, 0] = 0$
4. for $i = 1$ to n
5. for $j = 1$ to N
6. if $i = 1$ and $j < d[1]$
7. $C[1, j] = \infty$
8. else if $i = 1$
9. $C[1, j] = 1 + C[1, j - d[1]]$
10. else if $j < d[i]$
11. $C[i, j] = C[i - 1, j]$
12. else
13. $C[i, j] = \min(C[i - 1, j], 1 + C[i, j - d[i]])$
14. return $C[n, N]$

Write a recursive algorithm that, given the filled table $C[1 \dots n; 0 \dots N]$ generated by the above algorithm, prints out a sequence of $C[n, N]$ coin types which disburse N monetary units. In the case $C[n, N] = \infty$, print a message to the effect that no such coin disbursal is possible.

Solution:

Note that array $d = (d_1, d_2, \dots, d_n)$ is needed as input in order to navigate the table C .

PrintCoins(C, d, i, j) Pre: $C[1 \dots n; 0 \dots N]$ was filled by CoinChange(d, N)

1. if $j > 0$
2. if $C[i, j] == \infty$
3. print("cannot pay amount ", j)
4. return
5. if $i == 1$
6. print("pay one coin of denomination ", d_1)
7. PrintCoins($C, d, 1, j - d_1$)
8. else if $j < d_i$
9. PrintCoins($C, d, i - 1, j$)
10. else // both $i > 1$ and $j \geq d_i$
11. if $C[i, j] == C[i - 1, j]$
12. PrintCoins($C, d, i - 1, j$)
13. else // $C[i, j] == 1 + C[i, j - d_i]$
14. print("pay one coin of denomination ", d_i)
15. PrintCoins($C, d, i, j - d_i$)

4. Read the Discrete Knapsack Problem in the handout on Dynamic Programming. A thief wishes to steal n objects having values $v_i > 0$ and weights $w_i > 0$ (for $1 \leq i \leq n$). His knapsack, which will carry the stolen goods, holds at most a total weight $W > 0$. Let $x_i = 1$ if object i is to be taken, and $x_i = 0$ if object i is not taken ($1 \leq i \leq n$). The thief's goal is to maximize the total value $\sum_{i=1}^n x_i v_i$ of the goods stolen, subject to the constraint $\sum_{i=1}^n x_i w_i \leq W$.
- a. Write pseudo-code for a dynamic programming algorithm that solves this problem. Your algorithm should take as input the value and weight arrays $v[]$ and $w[]$, and the weight limit W . It should generate a table $V[1 \cdots n; 0 \cdots W]$ of intermediate results. Each entry $V[i, j]$ will be the maximum value of the objects that can be stolen if the weight limit is j , and if we only include objects in the set $\{1, \dots, i\}$. Your algorithm should return the maximum possible value of the goods which can be stolen from the full set of objects, i.e. the value $V[n, W]$. (Alternatively you may write your algorithm to return the whole table.)

Solution:

Knapsack(v, w, W) (pre: $v[1 \cdots n]$ and $w[1 \cdots n]$ contain positive numbers)

```

1.   $n = \text{length}[v]$ 
2.  for  $j = 0$  to  $W$  // fill in first row
3.      if  $j < w_1$ 
4.           $V[1, j] = 0$ 
5.      else
6.           $V[1, j] = v_1$ 
7.  for  $i = 2$  to  $n$  // fill remaining rows
8.      for  $j = 0$  to  $W$ 
9.          if  $j < w_i$ 
10.              $V[i, j] = V[i - 1, j]$ 
11.          else
12.              $V[i, j] = \max(V[i - 1, j], v_i + V[i - 1, j - w_i])$ 
13. return  $V[n, W]$ 
```

- b. Write an algorithm that, given the filled table generated in part (a), prints out a list of exactly which objects are to be stolen.

Solution:

PrintObjects(V, w, i, j) (pre: $V[1 \cdots n; 0 \cdots W]$ was filled by Knapsack(v, w, W))

```

1.  if  $i == 1$ 
2.      if  $j < w_i$ 
3.          print("do not include object ", 1)
4.      else
5.          print("include object ", 1)
6.  else //  $i > 1$ 
7.      if  $V[i, j] == V[i - 1, j]$ 
8.          PrintObjects( $V, w, i - 1, j$ )
9.          print("do not include object ",  $i$ )
10.     else // both  $j \geq w_i$  and  $V[i, j] == v_i + V[i - 1, j - w_i]$ 
11.         PrintObjects( $V, w, i - 1, j - w_i$ )
12.         print("include object ",  $i$ )
```

5. Canoe Rental Problem.

There are n trading posts numbered 1 to n as you travel downstream. At any trading post i you can rent a canoe to be returned at any of the downstream trading posts j , where $j \geq i$. You are given an array $R[i, j]$ defining the cost of a canoe that is picked up at post i and dropped off at post j , for i and j in the range $1 \leq i \leq j \leq n$. Assume that $R[i, i] = 0$, and that you can't take a canoe upriver (so perhaps $R[i, j] = \infty$ when $i > j$). Your problem is to determine a sequence of canoe rentals that start at post 1, end at post n , and which has a minimum total cost. As usual there are really two problems: determine the cost of a cheapest sequence, and determine the sequence itself.

Design a dynamic programming algorithm for this problem. First, define a 1-dimensional table $C[1 \cdots n]$, where $C[i]$ is the cost of an optimal (i.e. cheapest) sequence of canoe rentals that starting at post 1 and ending at post i . Show that this problem, with subproblems defined in this manner, satisfies the principle of optimality, i.e. state and prove a theorem that establishes the necessary optimal substructure. Second, write a recurrence formula that characterizes $C[i]$ in terms of earlier table entries. Third, write an iterative algorithm that fills in the above table. Fourth, alter your algorithm slightly so as to build a parallel array $P[1 \cdots n]$ such that $P[i]$ is the trading post preceding i along an optimal sequence from 1 to i . In other words, the last canoe to be rented in an optimal sequence from 1 to i was picked up at post $P[i]$. Write a recursive algorithm that, given the filled table P , prints out the optimal sequence itself. Determine the asymptotic runtimes of your algorithms.

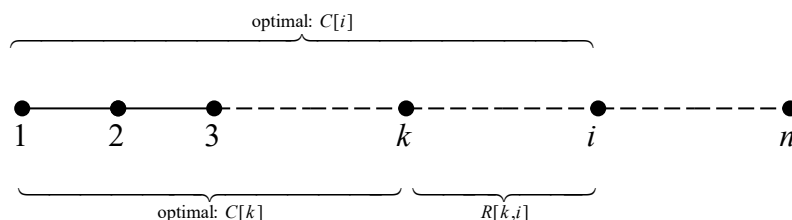
Solution:

One way to solve this problem would be to construct a 2-dimensional table whose i^{th} row and j^{th} column is the cost of an optimal sequence of canoe rentals starting at post i and ending at post j . This approach works, but one soon discovers that a 2-dimensional table is not really necessary. The entries in each row depend only on other entries in the same row, and since we are seeking an optimal sequence from 1 to n , only the first row is needed. Accordingly we define a 1-dimensional table $C[1 \cdots n]$ where $C[i]$ is the cost of an optimal sequence of canoe rentals starting at post 1 and ending at post i , for $1 \leq i \leq n$. When this table is filled, we simply return the value $C[n]$.

Clearly $C[1] = 0$ since one need not rent any canoes to get from 1 to 1. Let $i > 1$, and suppose we have found an optimal sequence taking us from 1 to i . In this sequence, there is some post k at which the final canoe was rented, where $1 \leq k < i$. In other words, our optimal sequence ends with a single canoe ride from post k to post i , whose cost is $R[k, i]$.

Claim: The subsequence of canoe rentals starting at 1 and ending at k is also optimal.

Proof: We prove this by contradiction. Assume that the above mentioned subsequence is not optimal. Then it must be possible to find a less costly sequence which takes us from 1 to k . Following that sequence by a single canoe ride from k to i , again of cost $R[k, i]$, yields a sequence taking us from 1 to i costing less than our original optimal one, a contradiction. Therefore the subsequence of canoe rides from 1 to k , obtained by deleting the final canoe ride in our optimal sequence from 1 to i , is itself optimal. ■



The above argument shows that this problem exhibits the required optimal substructure necessary for dynamic programming. It also shows how to define $C[i]$ in terms of earlier table entries. Indeed it's clear that $C[i] = C[k] + R[k, i]$. Since we do not know the post k beforehand, we take the minimum of this expression over all k in the range $1 \leq k < i$. Define

$$C[i] = \begin{cases} 0 & i = 1 \\ \min_{1 \leq k < i} (C[k] + R[k, i]) & 1 < i \leq n \end{cases}$$

With this formula, the algorithm for filling in the table is straightforward.

CanoeCost(R)

1. $n = \text{\#rows}[R]$
2. $C[1] = 0$
3. for $i = 2$ to n
4. $\text{min} = R[1, i]$
5. for $k = 2$ to $i - 1$
6. if $C[k] + R[k, i] < \text{min}$
7. $\text{min} = C[k] + R[k, i]$
8. $C[i] = \text{min}$
9. return $C[n]$

There are two equally valid approaches to determining the actual sequence of canoe rentals which minimizes cost. One approach would be to back-track through the array $C[1 \cdots n]$. The other method is to alter the CanoeCost() algorithm so as to construct the optimal sequence while $C[1 \cdots n]$ is being filled. We take the second approach in the algorithm below, where we maintain an array $P[1 \cdots n]$ whose i^{th} entry, $P[i]$ is defined to be the post k at which the final canoe is rented, in an optimal sequence from 1 to i . Note that the definition of $P[1]$ can be arbitrary, since it is never used. Array P is then used to recursively print out the sequence.

CanoeCost(R)

1. $n = \text{\#rows}[R]$
2. $C[1] = 0$
3. $P[1] = 0$
4. for $i = 2$ to n
5. $\text{min} = R[1, i]$
6. $P[i] = 1$
7. for $k = 2$ to $i - 1$
8. if $C[k] + R[k, i] < \text{min}$
9. $\text{min} = C[k] + R[k, i]$
10. $P[i] = k$
11. $C[i] = \text{min}$
12. return P

PrintSequence(P, i) (pre: $1 \leq i \leq \text{length}[P]$)

1. if $i > 1$
2. PrintSequence($P, P[i]$)
3. print("rent a canoe at ", $P[i]$, " and drop it off at ", i)

Both CanoeCost() and CanoeSequence() run in time $\Theta(n^2)$, since the inner for loop performs $i - 2$ comparisons to determine $C[i]$, and

$$\sum_{i=2}^n (i - 2) = \sum_{i=1}^{n-2} i = \frac{(n-1)(n-2)}{2} = \Theta(n^2)$$

The cost of the top level call PrintSequence(P, n) is the depth of the recursion, which is in turn, the number of canoes rented in an optimal sequence from post 1 to n . Thus PrintSequence() has worst case runtime in $\Theta(n)$. ■