Exercises - Dynamic Programming 1 - Algoritmiek

Tutorial February 13, 2024

1. **Basic knowledge:**

   (a) Explain in your own words what memoization is.

   (b) What is the difference between memoization and a "classical DP"?

   Solution: Answers are in the slides.

2. **Exact Change: Alternate and Constructive:** In the previous tutorial, you developed an alternate recurrence for the Exact Change problem. Let's revisit this problem.

   (a) Give pseudo-code for a dynamic program using this recurrence, and analyze its running time.

   (b) Your algorithm probably uses $O(nb)$ space. Explain how you would save space so that you only use $O(b)$ space.

   (c) Consider again the original algorithm, which uses $O(nb)$ space. Explain how to find an optimal solution (an optimal set of coins to pay).

   (d) Hard question: Can you combine saving space and finding the solution in a single algorithm? Explain your answer.

   Solution:

   (a) Array P[1…r,0…b]. All elements initialized to ∞.
   for i = 1 to r {
   P[i,0] = 0 }
   for c = 1 to b {
   if a_1 is a divisor of c then P[1,c] = c/a_1 else P[1,c] = ∞ }
   for i = 2 to r {
   for c = 1 to b {
   for k = 0 to c/a_i {
   P[i,c] = min{ P[i,c], k+P[i-1,$c - ka_i$] }
   } } }
   return P[r,b]
   Do not forget the base cases and the return value!
   Running time. The first for-loop takes $O(r)$ time. The second for-loop takes $O(b)$ time. The third, nested, for-loops, by using conservative estimates, take $O(rb \max_{i=1}^{r}\{b/a_i\}) = O(rb^2)$ time. Hence, the total running time is $O(rb^2)$.
   Compare this to the $O(rb)$ time algorithm discussed in class!

   (b) Observe that the algorithm only considers elements from the previous row. Hence, it suffices to maintain two rows: the current one and the previous one. Each row has b elements, so $O(b)$ space.

   (c) Starting from $P[r, b]$, consider how the minimum is achieved. Suppose you consider $P[i, c]$ for some i and c. Determine the value of the integer k ($0 \le k \le c/a_i$) for which $P[i, c] = k + P[i-1, c-ka_i]$. Then pay out k coins of value a_i and continue by inspecting $P[i-1, c-ka_i]$. Stop when i reaches 0 or c reaches 0. This approach can be implemented using a recursive algorithm or (preferably) using a simple while-loop (write down pseudocode to test yourself!)

(d) Using just the two rows of the table, as in the solution for part (a), and the order of filling the table (enumerate all i first, then all b), this seems difficult. However, the solution explained during the lecture uses $O(b)$ space and can be modified to yield a solution. So we can take a hint from there and change the order of filling the table, fixing c first. Consider the following implementation:

```
Array Q[0...b]. All elements initialized to ∞.
Array P[0...b]. All elements initialized to ∞.
Array R[0...b]. All elements initialized to ∞.
Q[0] = 0
for c = 1 to b {
for i = i to r {
for k = 0 to c/a_i {
P[c] = Math.min{ P[c], k+R[c − ka_i] }
}
Copy P into R
}
Q[c] = P[c]
}
return Q[b]
```

Observe that we fill the table for a particular value of c first. Again, note that we only need the previous row, as in (a), and that the table Q gives us enough information to retrieve the solution. Observe that the separate arrays P and R are superfluous, and it suffices to just use Q.

```
Array Q[0...b]. All elements initialized to ∞.
Q[0] = 0
for c = 1 to b {
for i = 1 to r {
for k = 0 to c/a_i {
Q[c] = Math.min{ Q[c], k+Q[c − ka_i] }
} }
}
return Q[b]
```

3. **Windmills: constructive:** In the previous tutorial, you developed a recurrence to solve the windmills problem.

   (a) Give a dynamic program for your recurrence and analyze its running time.

   (b) Explain how to find an optimal solution (an optimal set of positions for the windmills, such that they are at least K apart).

Solution:

(a) Array P[1...n]
P[1] = e_1
for i=2 to n {
P[i] = max { P[i-1], e_i + P[i-K] }
}
return $P[n]$
Don't forget the return value.
Running time. Clearly $O(n)$.

Alternative
Array P[1...n]
for i = 1 to K {
P[i] = e_i }
for i=K+1 to n {
for j=1 to i-K {
P[i] = max { P[i], e_i + P[j] }
} }
ret = $-\infty$
for i=n to n-K+1 {
ret = max { ret, P[i] } }
return ret
Don't forget the return value.
Running time. The first for-loop requires $O(K) = O(n)$ time. The second, nested for-loop takes $O(n^2)$ time by a conservative estimate. The last for-loop takes $O(K) = O(n)$ time. The total running time is $O(n^2)$. By being more precise in the implementation and estimates, the running time can be brought down to $O(nK)$.

(b) Starting from $P[n]$, consider how the maximum is achieved. Suppose you consider $P[i]$. If $P[i] = e_i + P[i-K]$, then a windmill is placed at i, no windmills are placed at $i-1, \ldots, i-K+1$, and you continue by inspecting $P[i-K]$. Otherwise, $P[i] = P[i-1]$, then no windmill is placed at i and you continue by inspecting $P[i-1]$. Stop when i reaches below 1. This can be implemented using a recursive algorithm or (preferably) using a simple while-loop (write down pseudocode to test yourself!).

4. **Mister Animal (revisited):** Consider Mister Animal and the recurrence that you developed for this problem in the previous tutorial.

(a) Give a dynamic program for your recurrence.
(b) Analyze the running time and memory space usage of your algorithm.
(c) Explain how to find a solution (a way to spend exactly B dollars).
(d) Can you save memory space? If so, explain how.

Solution:

(a) i. Array $S[0 \ldots i, 0 \ldots D]$, initialized to false
S[0,0] = true
for D=1 to B {
S[0,D] = false
}
for i=1 to n {
for D=1 to B {
if $v_i > D$ then S[i,D] = S[i-1,D]
else S[i,D] = S[i-1,$D - v_i$] or S[i-1,D]
} }
return S[n,D]

(b) The running time and space usage are both $O(nB)$.

(c) Trace back from $S[n, B]$. Suppose you consider $S[i, D]$. If $v_i \leq D$ and $S[i-1, D-v_i]$ is true, then take item i and continue by inspecting $S[i-1, D-v_i]$. Otherwise, leave item i and continue by inspecting $S[i-1, D]$. Stop when i reaches 0. This can be implemented using a recursive algorithm or (preferably) using a simple while-loop (write down pseudocode to test yourself!).

(d) Yes. Note that we only use $S[i-1, \cdot]$ to compute $S[i, \cdot]$. Hence, it suffices to maintain only the current and the previous row of the table. This yields $O(B)$ space.

5. **A2B revisited:** Consider the following recurrence for the A2B problem: for $a \leq c \leq b$, $K(c)$ is the smallest number of operations to get from c to b. Then:
$K(c) = \min\{1 + K(c+1), 1 + K(2c)\}$ if $2c \leq b$
$K(c) = 1 + K(c+1)$ otherwise.
Note that this solves the problem just as well, but from the 'other direction'.

(a) What is the base case?

(b) Give pseudocode for a memoization algorithm for this recurrence.

(c) Give pseudocode for a dynamic program for this recurrence.

Solution:

(a) $K(b) = 0$

(b) Array K[a . . . b], initialized to -1.
Method computeK(int c, int b).
{ if K[c] != -1 then return K[c].
if c == b then set K[c] to 0.
elseif $2c \leq b$ then set K[c] to 1+ minimum of computeK(2c,b) and computeK(c+1,b).
else set K[c] to 1+computeK(c+1,b).
return K[c]. }
Important to note that the value to compute is computeK(a,b).

(c) Array K[a . . . b]. Set K[b] = 0.
for c = b-1 to a do {
if $2c \leq b$ then set K[c] to 1+ minimum of computeK(2c,b) and computeK(c+1,b).
else set K[c] to 1+computeK(c+1,b)
}
return K[a]
Don't forget the return value!

6. **Splitting the inheritance:** You are executing a will and need to split an inheritance of n items for value $v_1, \ldots, v_n$ for two brothers (the items themselves are indivisible). To avoid any issues, the split must be done as fairly as possible. How can you find a fairest split of the items, that is, a split of the items such that the total value of items given to brother 1 differs as less as possible from the items given to brother 2? We will design a dynamic programming algorithm for this task.

   (a) Consider as a top-choice which brother gets item i. From this, you formulate the following subproblem: what is the fairest split of the first i items. Explain, by way of an example, that this is not a good subproblem (i.e., give a counterexample to the optimality principle).

   (b) Someone suggest as a subproblem: is there a split of the first i items such that brother 1 gets exactly c more in total value than brother 2. Prove the optimality principle using this subproblem.

   (c) Give a dynamic programming algorithm for this problem. Use the steps as described in class!

   (d) Explain to find an optimal solution (a fairest way to split up the inheritance).

   (e) Can you save memory space in your algorithm. If so, explain how.

   Solution:

   (a) Consider items of value 1, 4, 5. The optimal way to split items 1 and 2 is to give one item to brother 1 and another to brother 2. When considering items 1, 2, and 3, one can give item 3 to brother 1 (say), but then one cannot use the solution for items 1 and 2. Indeed, an optimal way to split items 1, 2, and 3 is to give brother 1 items 1 and 2, and brother 2 item 3; they both get an equal share. You cannot discover this solution using the solution to the subproblem.

   (b) Consider an optimal split of the first i items with difference exactly c; that is $B_1 \uplus B_2 = \{1, \ldots, i\}$ and $(\sum_{j \in B_1} v_j) - (\sum_{j \in B_2} v_j) = c$. Suppose $i \in B_1$. Then consider a solution $B_1' \uplus B_2'$ for the subproblem with parameters $i-1$ and $c - v_i$. This solution must exist, because $(B_1 \setminus \{i\}) \uplus B_2$ is such a solution. Note that $(B_1' \cup \{i\}) \uplus B_2$ is a solution for the subproblem with parameters i and c. Hence, the solution consists of solutions for the subproblem. The case that $i \in B_2$ is almost the same.

   (c) i. Let $V = \sum_{i=1}^{n} v_i$. Let $S[i, c]$ is true if and only if there is a way to split the first i items such that the brother 1 gets exactly c more in total value than brother 2, for $i = 1, \ldots, n$ and $c = -V, \ldots, 0, \ldots, V$.
   S[0,0] = true
   S[0,c] = false for all $c \neq 0$
   $S[i, c] = S[i-1, c - v_i] \vee S[i-1, c + v_i]$.

   ii. Array S[0...n, -V...0...V], initialized to false
   for c = -V to V {
   S[0,c] = false
   }
   S[0,0] = true
   for i=1 to n {
   for c = $-V + v_i$ to $V - v_i$ {
   S[i,c] = $S[i-1, c - v_i]$ or $S[i-1, c + v_i]$
   } }

```
ret = V
for c = -V to V {
if S[n,c] is true and |c| < ret then ret = |c|
}
return ret
```

(d) Suppose the algorithm returns a value o. Starting from $S[n, o]$, consider how it is achieved. Suppose you consider $S[i, c]$. If $S[i-1, c-v_i]$ is true, give item i to brother 1 and continue by inspecting $S[i-1, c-v_i]$. Otherwise, if $S[i-1, c+v_i]$ is true, give item i to brother 2 and continue by inspecting $S[i-1, c+v_i]$.

(e) Observe that the algorithm only considers elements from the previous row. Hence, it suffices to maintain two rows: the current one and the previous one. Each row has $2V+1$ elements, so $O(V)$ space is needed.

7. **Multiplication target:** Consider a set Σ of symbols on which an operator $* : \Sigma \times \Sigma \to \Sigma$ is given. This operator is neither associative nor commutative; so if $\Sigma = \{a, b\}$, then it is possible that $a * a = b$, $a * b = b$, $b * a = a$ and $b * b = a$. Suppose you are given n symbols $x_1, \ldots, x_n \in \Sigma$, possibly with duplications. You have to place parenthesis such that applying $*$ leads to a target value $d \in \Sigma$. For example, if $\Sigma = \{a, b\}$ and $*$ as before, and given is $x_1x_2x_3 = aba$ en $d = a$, then $(a * b) * a = b * a = a$ leads to a correct solution, but $a * (b * a) = a * a = b$ does not.
Careful: you are not allowed to change the order of the symbols, only place parenthesis.

(a) Give a dynamic programming algorithm for this problem. Use the steps discussed in class!

(b) Analyze the running time and memory space usage of your algorithm.

(c) Explain how to find a solution (a way to place parenthesis such that applying the operator gets you to d).

(d) Can you save memory space? If so, explain how.

No solution provided.