Intro Algo Chapter 15

Mengxi Wu (mw4355@nyu.edu)

September 27, 2020

Dynamic programming applies when sunproblems overlap (subproblems share subsubproblems). It sholves each subsubproblem just once and then saves its answer in a table.
1. Characterizethestructureofanoptimalsolution.
2. Recursivelydefinethevalueofanoptimalsolution.
3. Computethevalueofanoptimalsolution,typicallyinabottom-upfashion.
4. Constructanoptimalsolutionfromcomputedinformation.

1 Rod cutting

Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprise wants to know the best way to cut up the rods. For $i = 1, 2, ...$, the price p_i in dollars that Serling Enterprises charges for rod of length i inches. Given a rod of length n inches and a table of prices p_i for $i = 1, 2...n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Instead of solving the whole problem at once, we can solve smaller problem of the same type

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

More generally, we can express r_n =max(p_n,$r_1 + r_{n-1}$, $r_2 + r_{n-2}$,...$r_{n-1} + r_1$)
p_n corresponds to making no cut. The other $n-1$ arguments to max correspond to the maximum revenue obtained by making a cut of the rod into two pieces (size i and $n - i$). We can view the length n rod cutting in this way: a first piece followed by some decomposition of the remainder. r_n =max($p_i + r_{n-i}$) for $1 \leq i \leq n$.

Recursion Method

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

The running time for the recursion solution is exponential.
Cut-Rod(p, n) calls Cut-Rod($p, n - i$) for $i = 1, 2, 3..n$. In other expression, Cut-Rod(p, j) for $j = 0, 2, 3..n - 1$. Let $T(n)$ denote the total number of calls made to Cut-Rod when called with its second parameter equal to n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. $T(0) = 1$ and $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$.

How many ways are there to cut up a rod of length n?
There are $n - 1$ places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut. The total number ways is 2^{n-1}.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + 1 + T(1) + T(2)... + T(n-1) = 1 + 2 + 2^2 + 2^{n-1} = 2^n$$

Dynamic Programming Method

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

The procedure solves subproblems of sizes $j = 0, 1...n$. Line 6 directly references array entry $r[j-i]$ which is solved before instead of making a recursive call to salve the subproblem of size $j-i$. Line 7 saves in $r[j]$ the solution to size j. The running time of this method is $O(n^2)$.

2 Matrix Multiplication

Given sequence of matrices $A_1, A_2, ...A_n$ of n matrices to be multiplied. The multiplication of two matrices is shown as follow.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr. However, different costs can be incurred by different parenthesization. Suppose we have three matrics with different dimension 10×100, 100×5, and 5×50, respectively. If we multiply according to $((A_1A_2)A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ and $10 \cdot 5 \cdot 50 = 2500$. There are total of 7500 scalar multiplications. If we multiply according to $(A_1(A_2A_3))$, we perform $100 \cdot 5 \cdot 50 = 25000$ and $10 \cdot 100 \cdot 50 = 50000$. The total scalar multiplications is 75000/ The first way is 10 times faster.

Maxtrix-chain multiplication problem: given a chain $(A_1, A_2, ...A_n)$ of n matrices, where for $i = 1, 2, 3..n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2...A_n$ in a way that minimizes the number of scalar multiplications. Our goal is to determine an order for multiplying matrices that has the lowest cost.

For any multiplication $A_{i..j}$, we break it into $A_{i..k} \cdot A_{k+1..j}$, where $A_{i..k}$ and $A_{k+1..j}$ are optimal solutions for subproblem. We let $m[i, j]$ be the minimum number of scaler multiplication needed to compute $A_{i..j}$. Then, $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$. We need to figure which k make it minimum. We define $s[i, j] = k$ at which we split the product $A_{i..j}$.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Notice that we have relatively few distinct subproblems. Choose different $A_{i..j}$. When $i < j$, there are $\binom{n}{2}$. When $i = j$, there are n. Thus, $\binom{n}{2} + n = \Theta(n^2)$ in total.

Let A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, ..n$. Its input is a sequence $p = (p_0, p_1, ...p_n)$. Two tables store the results. $[1..n, 1..n]$ store the minimum scalar multiplications. $s[1..n-1, 2..n]$ store where to separate the chain. Separation happens when there are at least two matrics, so the first index i is from 1 to $n-1$, and the second index j is from 2 to n ($s[i, j]$ records the separation position for product chain $A_i..A_j$).

MATRIX-CHAIN-ORDER(p)

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                  // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

The algorithm first computes $m[i, i]$=0 for $i = 1, 2, ..n$ (the minimum costs for chains of length 1) in lines 3–4. It then computes $m[i, i + 1]$ for $i = 1, 2, ...n - 1$ (the minimum costs for chains of length l = 2) during the first execution of the for loop in lines 5–13. The second time through the loop,it computes $m[i, i + 2]$ for $i = 1, 2, ...n - 2$(the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 10–13 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed. The nested loop structure of this algorithm yields a running time of $O(n^3)$.

From table s, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..k}A_{k+1..n}$ where $k = s[1, n]$. $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$. $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$.

3 Optimal Binary Search Tree

We are given a sequence $K = (k_1, k_2, ...k_n)$ of n distinct keys in sorted order ($k_1 < k_2 < ... < k_n$). For each key k_i, we have probability p_i that a search will be for k_i. Some searches may be for value not in K, so we have $n + 1$ dummy keys $(d_0, d_1..., d_n)$. d_o represents all values less that less than k_1 and d_n represents all values larger than k_n. For each dummy key d_i we have a probability q_i that a search will correspond to d_i. $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.

Assume that the actual cost of a search equals the number of nodes examined. Since it is a binary search tree, at each level, only one node will be examined (smaller go to left child; larger go to right child). Thus, the number of node examined will be depth + 1 (The depth of a node is the number of edges from the root to the node).

$$\begin{aligned} E[searchcost] &= \sum_{i=1}^{n} (depth(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} depth(k_i) \cdot p_i + \sum_{i=0}^{n} depth(d_i) \cdot q_i \end{aligned}$$

We want to construct a tree whose expected search cost is smallest. We call such tree an **optimal binary search tree**. If an optimal binary search tree T has a subtree T' containing keys $k_i, ...k_j$, then this subtree T' must be optimal. Let k_r be the root. As long as we examine all candidate roots k_r, where $i \le r \le j$ and we determine all optimal binary search trees containing $k_i, ...k_{r-1}$ and those containing $k_{r+1}, ...k_j$, we are guaranteed that we will find an optimal binary search tree.

Let us define $e[i,j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, ...k_j$ $(1 \le i, i-1 \le j \le n)$. When $j = i-1$, we only has dummy key d_{i-1}, so $e[i,i-1] = q_i$ since the tree contains only one node. For $j \ge i$, we need to select a root k_r from among $k_i, ...k_j$ and make an optimal binary search tree with keys $k_i, ..., k_{r-1}$ as its left subtree and optimal binary search tree with keys $k_{r+1}, ..., k_j$ as its right subtree. When a tree becomes a subtree, the expected search cost of each node in this tree increases by 1. Thus, the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For subtree with keys $k_i, ..., k_j$, $w(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$. $e[i,j] = costfork_r + leftsubtreecost + rightsubtreecost.$

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

$$w(i,r-1) + p_r + w(r+1,j) = \sum_{k=i}^{r-1} p_k + p_r + \sum_{k=r+1}^{j} p_k + \sum_{k=i-1}^{r-1} q_k + \sum_{k=r}^{j} q_k = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k = w(i,j)$$

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

We use root$[i,j]$ track the index of the root of an optimal binary search tree containing keys $k_i, ..., k_j$. There are some similarities between matrix-chain multiplication and this problem. The subproblems consist of contiguous index subranges. We store the $e[i,j]$ values in a table $e[1..n+1, 0..n]$ (dummy key d_n is stored in $e[n+1,n]$ and d_0 is stored in $e[1,0]$). We compute $w[i,j] = w[i,j-1] + p_j + q_j$.

```
OPTIMAL-BST(p, q, n)
 1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
 2  for i = 1 to n + 1
 3      e[i, i-1] = q_{i-1}
 4      w[i, i-1] = q_{i-1}
 5  for l = 1 to n
 6      for i = 1 to n - l + 1
 7          j = i + l - 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j-1] + p_j + q_j
10          for r = i to j
11              t = e[i, r-1] + e[r+1, j] + w[i, j]
12              if t < e[i, j]
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

The innermost for loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, ...k_j$. The algorithm take $\Theta(n^3)$ time, since its for loops are nested three deep and each loop index takes on at most n values.

4 Longest common subsequence

Given two sequences $X = \{x_1, x_2, ..., x_m\}$ and $Y = \{y_1, y_2, ..., y_n\}$, our task is to find a maximum length common subsequence of X and Y. Let $Z = \{z_1, z_2, ..., z_k\}$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \ne y_n$, there are two cases:
   $z_k \ne x_m$ and Z is an LCS of X_{m-1} and Y.
   $z_k \ne y_n$ and Z is an LCS of X and Y_{n-1}.

In case 1, we can prove by contradiction. Given $x_m = y_n$, suppose $z_k \ne x_m$. If we add the x_m to Z, we obtain a common sequence with length $k+1$ which contradicts the supposition that Z is LCS. Thus, $z_k = x_m$. The next we need to prove is Z_{k-1} is LCS of X_{m-1} and Y_{n-1}. Suppose there exits a common sequence W of X_{m-1} and Y_{n-1} and the length of W is longer than $k-1$. Then, if we add x_m to W we

obtain a sequence that is longer than Z which contradicts Z is LCS of X and Y. Thus, Z_{k-1} is LCS of X_{m-1} and Y_{n-1}.
In case 2, given $z_k \neq x_m$, suppose we have a common sequence W of X_{m-1} and Y which have length greater than k. Then, W is also the LCS of X and Y and it is longer than Z, which contradicts that Z is LCS of X and Y. Proof for case $z_k \neq y_n$ is the same.

Then, we come up the idea that if $x_m = y_n$, we add x_m to the LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, we find a LCS of X_{m-1} and Y, and a LCS of X and Y_{n-1}; then we keep the longer one. Let us define $c[i, j]$ to be the length of an LCS of X_i and Y_j.

$$c[i,j] = \begin{cases} 0 & \text{if } j = 0 \text{ or } i = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

While $c[m, n]$ tracks the length of the LCS, $b[m, n]$ is used to construct the LCS. The diagonal arrow at $b[i, j]$ represents that $x_i = y_j$. Thus, $x_i = y_j$ is an element of the LCS. We start the tracing from $b[m, n]$ and record every $x_i = y_i$, and finally reverse the tracing result.

5 Elememts of dynamic programming

When we should use dynamic programming? There are two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. When we can tell a problem contains optimal substructure? A optimal solution to the problem contains within it optimal solutions to subproblems.

How to find optimal structure?
1. You find that a solution to the problem consists of making a choice. Making this choice leaves one ore more subproblems to be solved.
2. Given this choice, you determine which subproblems ensure and how to best characterize the resulting space of subproblems.
3. Then, we need to show that the solutions to the subproblems used within an optimal solution to the problem must themselves to be optimal. We do it by supposing that each subproblem solutions is not optimal and the deriving a contradiction. Suppose the optimal solution is constructed with some nonoptimal solution. We can just remove the nonoptimal solution to each subproblem and use the optimal solution for each subproblem. Then, we find we gain a better solution to the original problem, thus contradicting in the supposition that we already had an optimal solution.

The running time of a dynamic-programming algorithm depends on two factors: the number of subproblems overall and how many choices we look at for each subproblem. Dynamic programming often uses optimal substructure in bottom-up fashion. We first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem.