6 Design of Algorithms: By Dynamic Programming

6.1 Introduction

Reading: CLRS 15.3

- Divide-and-conquer algorithms are implemented by recursion. Its design is top-down, and it is efficient when the subproblems don't overlap. However, when subproblems do overlap (share sub-subproblems), recursion does redundant work. In this case, a tabular method is often used. It is nonrecursive and bottom-up. It is called dynamic programming.

- An example: Fibonacci numbers:

  $fib1(n)$

  if $n < 2$ return n

  else return $fib1(n-1) + fib1(n-2)$

  We can see that this recursive (divide-and-conquer) algorithm is not efficient. To compute $fib1(n)$, the algorithm computes $fib1(n-1)$ and $fib1(n-2)$ separately. To compute $fib1(n-1)$, the values of $fib1(n-2)$ and $fib1(n-3)$ are needed. To compute $fib1(n-2)$, the values of $fib1(n-3)$ and $fib1(n-4)$ are needed. We observe that subproblems $fib1(n-1)$ and $fib(n-2)$ share sub-subproblem.

  The time complexity $T(n) \geq T(n-1) + T(n-2)$. So $T(n)$ is larger than the nth Fibonacci number. So $T(n) \geq \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (-\frac{1+\sqrt{5}}{2})^{-n}) = \Theta(1.618^n)$.

  We can use the dynamic programming method by building a 1-D table as below and returning the nth entry of the table.

k	0	1	2	3	4	$\cdots$	n
f_k	0	1	1	2	3	$\cdots$	f_n

  $fib2(n)$

  if $n < 2$ return n

  else $i \leftarrow 0$

  $j \leftarrow 1$

  for $k \leftarrow 2$ to n

  $f \leftarrow i + j$

  $i \leftarrow j$

  $j \leftarrow f$

  return f

  The time complexity is obviously $O(n)$.

  To summarize how to use dynamic programming, first define a function F recursively (so that the solution information is embedded in $F(n)$): $F(n) = G(F(n_1), F(n_2), \ldots, F(n_k))$ for $n_1, n_2, \ldots, n_k < n$. Construct a table to compute nonrecursively $F(n_1), F(n_2), \ldots, F(n_k)$, hence $F(n)$.

6.2 Making change

- Let n, a positive integer, be the number of different types of coin in a country. Let $coin[1..n]$, an array of positive integers, be the values of these n types of coin. Let m, a positive integer, be the amount of change that one wishes to make. Design a dynamic programming algorithm that determines whether m can be made with the coins, and if so, computes the minimum number of coins needed.

- Define $count(i)$ to be the minimum number of coins to make i (> 0). That $count(i) = \infty$ implies that no solution exists. The recursive definition of $count(i)$ is as follows.

  $count(1) = \infty$ if $1 \notin coin[\,]$.

  $count(coin[j]) = 1$ for $j = 1, \ldots, n$.

  $count(i) = 1 + \min_{1 \leq j \leq n, coin[j] < i}\{count(i - coin[j])\}$

- The table is a 1-D table and its entries are filled from left to right until $count[m]$ is reached.

i	1	2	$\cdots$	m
$count[i]$	$\rightarrow$	$\rightarrow$	$\cdots$	$*$

- Algorithm:

  for $i \leftarrow 1$ to m $count[i] \leftarrow -1$
  $count[1] \leftarrow \infty$
  for $j \leftarrow 1$ to n
      $count[coin[j]] \leftarrow 1$
  for $i \leftarrow 1$ to m
      if $count[i] = -1$
          $min \leftarrow \infty$
          for $j \leftarrow 1$ to n
              if $coin[j] < i$
                  if $min > count[i - coin[j]]$
                      $min \leftarrow count[i - coin[j]]$
          $count[i] \leftarrow 1 + min$

- Time complexity: $\Theta(mn)$. (pseudo-polynomial)

6.3 Chained matrix multiplication

Reading: CLRS 15.2

- We wish to compute $A_1 \times A_2 \times \cdots \times A_n$, where A_i is a $p_{i-1} \times p_i$ matrix. Which order of computation should we use to achieve the highest efficiency of the algorithm?
- The number of basic operations needed to compute $A_i \times A_{i+1}$ is $p_{i-1}p_ip_{i+1}$.
- Order of computation determines the time efficiency. For example, $A_1 : 10 \times 20$, $A_2 : 20 \times 50$, $A_3 : 50 \times 1$, and $A_4 : 1 \times 100$. If we use the order in $A_1 \times (A_2 \times (A_3 \times A_4)$, the number of basic operations is $(50 \times 1 \times 100) + (20 \times 50 \times 100) + (10 \times 20 \times 100) = 125,000$. However, if we use the order in $((A_1 \times A_2) \times A_3) \times A_4$, the number of basic operations is $(10 \times 20 \times 50) + (10 \times 50 \times 1) + (10 \times 1 \times 100) = 11,500$.
- Question: What is the minimum number of basic operations in computing $A_1 \times A_2 \times \cdots \times A_n$?
- Let $m(i, j)$ be the minimum number of basic operations in computing $A_i \times A_{i+1} \times \cdots \times A_j$ for $1 \leq i \leq j \leq n$. Assume in general that k is used to indicate the position of the last multiplication to be performed among all: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$. Then

  $m(i, j) = 0$ if $i = j$.

  $m(i, j) = \min_{i \leq k \leq j-1}\{m(i,k) + m(k+1, j) + p_{i-1}p_kp_j\}$ if $i \neq j$.
- We can use a dynamic programming algorithm to compute $m(1,n)$, the minimum number of basic operations in computing $A_1 \times A_2 \times \cdots \times A_n$. Entries are filled left to right and bottom to top. Note that those in the lower left triangle are undefined.

$i \backslash j$	1	2	$\cdots$	n
1	0	$\uparrow$	$\cdots$	$*$
2	$--$	0	$\cdots$	$\uparrow$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
n	$--$	$--$	$\cdots$	0

- The algorithm:

  for $i \leftarrow 1$ to n

  $\quad m[i,i] \leftarrow 0$

  for $j \leftarrow 2$ to n

  $\quad$ for $i \leftarrow j-1$ to 1

  $\quad\quad m[i,j] \leftarrow \min_{i \le k \le j-1}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$

- Time complexity: $O(n^3)$.

6.4 Longest common subsequence

Reading: CLRS 15.4

- Subsequence: If $X =< A,B,C,B,D,A,B >$ and $Z =< B,C,D,B >$, then Z is a subsequence of X.

  Common subsequence: Let $Y =< B,D,C,A,B,A >$. Then $< B,C,A >$ is a common subsequence of X and Y.

  Longest common subsequence (LCS): For X and Y, there is no common subsequence with length longer than 4. $< B,C,B,A >$ and $< B,D,A,B >$ are both LCS's of X and Y.

  Question: Given two sequences, what is the length of their LCS? (What is the LCS of the sequences?)

- A brute-force method:

  Assume $X =< x_1,\ldots,x_m >$ and $Y =< y_1,\ldots,y_n >$. For each subsequence of X, check if it is also a subsequence of Y, keeping track of the longest found.

  How many possible subsequences are there for X? $\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{m} = 2^m$.

- A recursive approach:

  Define $X_i =< x_1,\ldots,x_i >$ and $Y_j =< y_1,\ldots,y_j >$. Define $C(i,j)$ to be the length of the LCS of X_i and Y_j.

  $C(i,j) = 0$ if $i = 0$ or $j = 0$;

  $C(i,j) = C(i-1,j-1) + 1$ if $i,j > 0$ and $x_i = y_j$;

  $C(i,j) = \max\{C(i,j-1),C(i-1,j)\}$ if $i,j > 0$ and $x_i \neq y_j$.

  When $x_i = y_j$, $X_i = X_{i-1} < x_i >$ and $Y_j = Y_{j-1} < y_j >$. So $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1})x_i$. Hence, $C(i,j) = C(i-1,j-1)+1$.

  When $x_i \neq y_j$, x_i and y_j cannot both appear in $LCS(X_i,Y_j)$. So $LCS(X_i,Y_j) = LCS(X_i,Y_{j-1})$ or $LCS(X_{i-1},Y_j)$. Hence, $C[i,j] = \max\{C(i,j-1),C(i-1,j)\}$.

- A nonrecursive implementation: Dynamic programming:

  A 2-D table is constructed where each entry is filled left to right and top to bottom. The initialization handles the first row and the first column of the table. Entry $C[m,n]$ is the length of the LCS of X and Y.

$i\backslash j$	0	1	2	$\cdots$	n
0	0	0	0	$\cdots$	0
1	0	$\rightarrow$	$\rightarrow$	$\cdots$	$\rightarrow$
2	0	$\rightarrow$	$\rightarrow$	$\cdots$	$\rightarrow$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
m	0	$\rightarrow$	$\rightarrow$	$\cdots$	$*$

- The algorithm:

  for $i \leftarrow 0$ to m $C[i,0] \leftarrow 0$

  for $j \leftarrow 0$ to n $C[0,j] \leftarrow 0$

  for $i \leftarrow 1$ to m

  $\quad$ for $j \leftarrow 1$ to n

  $\quad\quad$ if $x_i = y_j$ $C[i,j] \leftarrow C[i-1,j-1]+1$

  $\quad\quad$ else $C[i,j] = \max\{C[i,j-1],C[i-1,j]\}$

- Time complexity: $\Theta(mn)$.
- How to compute the LCS in addition to the length of the LCS: Maintain an array $S[i,j]$ of special characters. Set $S[i,0] = S[0,j] = \sqcup$ (single space) for $0 \leq i \leq m$ and $0 \leq j \leq n$. In the nested for loop, if $x_i = y_j$, set $S[i,j]$ to be $\nwarrow$, else if $C[i,j-1] > C[i-1,j]$, set $S[i,j]$ to be $\leftarrow$, and if $C[i,j-1] \leq C[i-1,j]$, set $S[i,j]$ to be $\uparrow$. The following additional code generates the LCS of two sequences.

```
i ← m
j ← n
while S[i, j] ≠ ⊔
    if S[i, j] =← j ← j − 1
    else if S[i, j] =↑ i ← i − 1
    else push x_i to a stack
        i ← i − 1
        j ← j − 1
output the content in the stack
```

6.5 Optimal binary search tree

Reading: CLRS 15.5

- Given a set of keys (numbers) and the probability that each key is located. How can one organize the set in a binary search tree so that the average time to locate a key in the tree is minimized?
- For each node (key) in a binary search tree, the time needed to locate the node is its level number.
- Let the keys be $a_1, a_2, \ldots, a_n$ (in increasing order). Let l_i be the level number of the node corresponding to key a_i in a given binary search tree. Let p_i be the probability that a_i is to be located. Then the average search time for that tree is $\sum_{i=1}^{n} p_i l_i$. We wish to build an optimal binary search tree, where this cost is minimized.
- An example: $n = 3$ and $p_1 = 0.7$, $p_2 = 0.2$ and $p_3 = 0.1$. The following figure contains all five possible binary search trees for $n = 3$.
  1. 3(0.7)+2(0.2)+1(0.1)=2.6
  2. 2(0.7)+3(0.2)+1(0.1)=2.1
  3. 2(0.7)+1(0.2)+2(0.1)=1.8
  4. 1(0.7)+3(0.2)+2(0.1)=1.5
  5. 1(0.7)+2(0.2)+3(0.1)=1.4 $\Leftarrow$ optimal!
- A recursive approach: Let $c(i,j)$ be the average search time in a tree with only $a_i, \ldots, a_j$, where $1 \leq i \leq j \leq n$. If a_k happens to be the root of the tree containing $a_i, \ldots, a_j$, then in the left subtree are $a_i, \ldots, a_{k-1}$ and in the right subtree are $a_{k+1}, \ldots, a_j$.

  $c(i,i) = p_i$ for $1 \leq i \leq n$

  $c(i,j) = \min_{i \leq k \leq j}\{c(i,k-1) + c(k+1,j) + \sum_{l=i}^{j} p_l\}$ for $i < j$

  $c(i,j) = 0$ for $i = j+1$ (Why needed?)
- A dynamic programming algorithm with $O(n^3)$:

$i\backslash j$	1	2	$\cdots$	$n-1$	n
1	p_1	$\uparrow$	$\cdots$	$\uparrow$	$*$
2	0	p_2	$\cdots$	$\uparrow$	$\uparrow$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
n	$--$	$--$	$\cdots$	0	p_n

6.6 Comparing Two Sequences

- Interested in finding the best alignment of two sequences for the purpose of comparison.
- Given two sequences, s and t, over the same alphabet. For any alignment A of the two sequences, define its score $score_A(s,t)$ to be the sum of the scores of all columns in the alignment, where the score of a column containing characters a and b, denoted as $p(a,b)$, may be defined to be, for example
  - -1 if a and b are nonspaces and $a = b$ (a match)
  - 1 if a and b are nonspaces $a \neq b$ (a mismatch)
  - 2 if one of a and b is a space (one space)
- Problem: Given two sequences, s and t, over the same alphabet, determine the optimal alignment with the minimum score, i.e., $score^*(s,t) = \min_{\forall A}\{score_A(s,t)\}$.
- Example: s =GACGGATTAG and t =GATCGGAATAG are two DNA sequences. For the following (optimal) alignment,

```
G A - C G G A T T A G
G A T C G G A A T A G
```

  its score is $-9+1+2 = -6$.

- Let $score^*(s[1..i],t[1..j])$ be the minimum score of any alignment for sequences $s[1..i]$ and $t[1..j]$. Note $i = 0,\ldots,|s|$ and $j = 0,\ldots,|t|$. If $i = 0$ (or $j = 0$) then $s[1..i]$ (or $t[1..j]$) becomes the empty string.
- Three possibilities to align $s[1..i]$ and $t[1..j]$:
  - Align $s[1..i]$ with $t[1..j-1]$ and match a space with $t[j]$,or
  - Align $s[1..i-1]$ with $t[1..j-1]$ and match $s[i]$ with $t[j]$, or
  - Align $s[1..i-1]$ with $t[1..j]$ and match $s[i]$ with a space.
- A recursive definition of $score^*(s[1..i],t[1..j])$:
  - $score^*(\varepsilon,\varepsilon) = 0$
  - $score^*(s[1..i],\varepsilon) = sspace * i$ for $i = 1,\ldots,|s|$
  - $score^*(\varepsilon,t[1..j]) = sspace * j$ for $j = 1,\ldots,|t|$
  - $score^*(s[1..i],t[1..j]) = \min\{score^*(s[1..i],t[1..j-1])+sspace, score^*(s[1..i-1],t[1..j-1])+cij, score^*(s[1..i-1],t[1..j])+sspace\}$

  where *sspace* is the score of a space opposite a nonspace, *smatch* is the score of a match, and *smiss* is the score of a mismatch. cij is *smatch* if $s[i] = t[j]$ and is *smiss* if $s[i] \neq t[j]$. Note that a space is never aligned against another space in the alignment of two sequences.

- A dynamic programming algorithm:

```
Algorithm: Optimal Pairwise Alignment
   input: sequence s and t
   output: score*(s,t) //Use table cell a[i,j] for score*(s[1..i],t[1..j])
   m <- |s|
   n <- |t|
   a[0,0] <- 0
   for i <- 1 to m do
       a[i,0] <- sspace * i
   for j <- 1 to n do
       a[0,j] <- sspace * j
   for i <- 1 to m do
       for j <- 1 to n do
           if s[i] = t[j] then cij <- smatch
                          else cij <- smiss
           a[i,j] <- min {a[i,j-1] + sspace, a[i-1,j-1] + cij, a[i-1,j] + sspace}
   return a[m,n]
```

- Example: $s =$AAAC and $t =$AGC. Let $sspace = 2$, $smatch = -1$, and $smiss = 1$. What is $score^*(s,t)$ and what is the optimal alignment for s and t (maybe more than one)?
- Time complexity: Computing $score^*(s,t)$ (constructing the table) takes $O(|s||t|)$ and constructing the optimal alignment once the table is given takes $O(|s|+|t|)$.
- A similar dynamic programming algorithm exists for aligning three or more DNA sequences.

6.7 Memory functions

Reading: CLRS 15.3

- Divide and conquer: Only needed entries are computed but some entries are computed more than once. Dynamic programming: All entries in the table are computed once, whether needed or not.
- A compromise: Only compute needed entries exactly once. To do so, we combine the recursive implementation with a table. Before we enter a recursion, we check the table to see whether the entry has been computed before. This method is called the memory function method.
- Example: Chained matrix multiplication revisited.

  We first initialize all entries in table $m[1..n, 1..n]$ to be -1, and then call $mf(1,n)$.

  $mf(i, j)$

      if $i = j$ return 0

      if $m[i, j] \neq -1$ return $m[i, j]$

      $c \leftarrow \infty$

      for $k \leftarrow i$ to $j-1$

          $c \leftarrow \min\{c, mf(i,k) + mf(k+1, j) + p_{i-1}p_kp_j\}$

      $m[i, j] \leftarrow c$

      return c

- The time complexity is no larger than that in the corresponding dynamic programming algorithm, but the space complexity will be more since recursion requires more space to implement.