Algorithms

授課老師：張景堯

Introduction to Dynamic Programming

Chapter.10-Dynamic Programming

Dynamic Programming(動態規劃)

- Dynamic programming is a very powerful, general tool for solving optimization problems on *left-right-ordered* among items such as character strings, rooted trees, polygons, and integer sequences.
- Once understood it is relatively easy to apply, it looks like magic until you have seen enough examples.
- Floyd's all-pairs shortest-path algorithm was an example of dynamic programming.

(動態規劃是將問題拆小，依序解決，每個小問題階段解，又是下個小問題藉以找出當前最優解的根據~聞到遞迴的味道了嗎?)

Greedy vs. Exhaustive Search

- *Greedy* algorithms focus on making the best local choice at each decision point. In the absence of a correctness proof such greedy algorithms are very likely to fail.
- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid re-computing (thus providing efficiency).

(貪婪法只看目前最佳解，前面做的不再往前看，可能找不到整體最佳解，而動態規畫會利用前面拆小的問題有系統地得到所有可能的各階段解，最後確保能得到整體最優解，還會儲存階段解避免重複計算，增進計算效率)

Recurrence Relations(遞迴關係式)

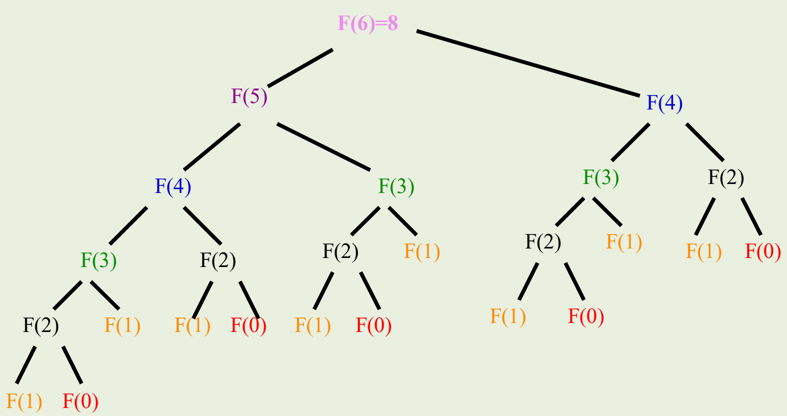
- A recurrence relation is an equation which is defined in terms of itself. They are useful because many natural functions are easily expressed as recurrences:
- Polynomials: $a_n = a_{n-1} + 1,\ \ a_1 = 1 \rightarrow a_n = n$
- Exponentials: $a_n = 2a_{n-1},\ a_1 = 2 \rightarrow a_n = 2^n$
- Weird: $a_n = n \cdot a_{n-1},\ a_1 = 1 \rightarrow a_n = n!$
- Computer programs can easily evaluate the value of a given recurrence even without the existence of a nice closed form. (使用遞迴呼叫的程式就能輕易計算這類關係式)

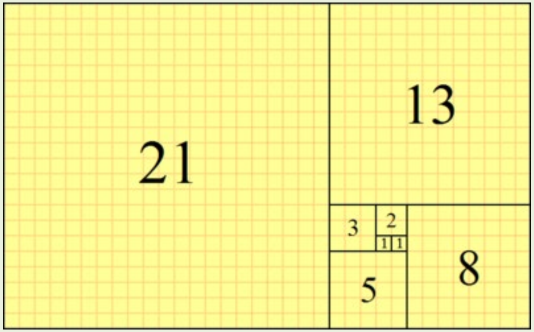
Computing Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2},\ F_0 = 0, F_1 = 1$$

- Implementing this as a recursive procedure is easy, but slow because we keep calculating the same value over and over.

(找得到哪裡重複計算嗎?)

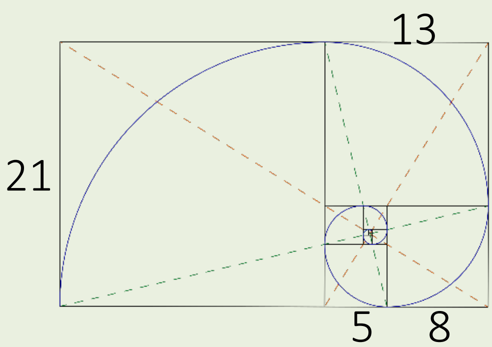




How Slow?

$$\frac{F_{n+1}}{F_n} \approx \varphi = \frac{\left(1+\sqrt{5}\right)}{2} \approx 1.61803$$

- Thus $F_n \approx 1.6^n$.
- Since our recursion tree has 0 and 1 as leaves, computing F_n requires $\approx$ 1.6^n calls!



$$\frac{a+b}{a} = \frac{a}{b}$$

黃金比例 在各國國會的實例...

What about Memorization?

- We can <u>explicitly</u> cache calls after computing results to avoid re-computation:

```
def fib_c(n:int)->int:
    F = [None]*(n+1)
    F[0] = 0
    if n > 0:
        F[1] = 1

    def fib(n:int)->int:
        if F[n] == None:
            F[n] = fib(n-1) + fib(n-2)
        return F[n]

    return fib(n)
```

What about Dynamic Programming?

- We can calculate F_n in linear time by <u>storing small values</u>:

$F_0 = 0$

$F_1 = 1$

For $i = 2$ to n

$\quad F_i = F_{i-1} + F_{i-2}$

F

- Moral: we traded space for time.

(以空間換取時間)

Why I Love Dynamic Programming

- Dynamic programming is a technique for efficiently computing recurrences by storing partial results.

(動態規畫以儲存部分解來有效率地計算遞迴關係式)

- Once you understand dynamic programming, it is usually easier to reinvent certain algorithms than try to look them up! I have found dynamic programming to be one of the most useful algorithmic techniques in practice:
  - Morphing in computer graphics. (影像變形)
  - Data compression for high density bar codes. (條碼資料壓縮)
  - Designing genes to avoid or contain specified patterns.(基因設計)

Avoiding Recomputation by Storing Results

- The trick to dynamic programming is to see that the naïve recursive algorithm repeatedly computes the same subproblems over again, so storing the answers in a table instead of recomputing leads to an efficient algorithm.
- We first hunt for a correct recursive algorithm, then we try to speed it up by using a results matrix.

(我們可以先用正確的遞迴演算法計算後，再改成用如：矩陣等資料結構存放部分解的動態規劃法)

Binomial Coefficients(二項式係數)

- The most important class of counting numbers are the binomial coefficients, where $\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities. (n取k)
    - *Committees* – How many ways are there to form a k-member committee from n people? By definition, $\binom{n}{k}$.
    - *Paths Across a Grid* – How many ways are there to travel from the upper-left corner of an $n \times m$ grid to the lower-right corner by walking only down and to the right? Every path must consist of $n + m$ steps, n downward and m to the right, so there are $\binom{n+m}{n}$ such sets/paths.

$$\binom{3+3}{3}$$

1	1	1	1
1			
1			
1			

Computing Binomial Coefficients

- Since

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

,in principle you can compute them straight from factorials.

- However, intermediate calculations can *easily* cause arithmetic overflow even when the final coefficient fits comfortably within an integer.

Pascal's Triangle

- No doubt you played with this arrangement of numbers in high school. Each number is the sum of the two numbers directly above it:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

$$\binom{5}{0}\ \binom{5}{1}\ \binom{5}{2}\ \binom{5}{3}\ \binom{5}{4}\ \binom{5}{5}$$

Pascal’s Recurrence

- A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal’s triangle, namely, that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- It works because the nth element either appears or does not appear in one of the $\binom{n}{k}$ subsets of k elements.
  - If so, We can complete the subset by picking $k-1$ other items from the other $n-1$. (若第n個在k之中就是從$n-1$取$k-1$)
  - If not, we must pick all k items from the remaining $n-1$. (若第n個不在k之中就全部從 $n-1$取k)

Basis Case

- No recurrence is complete without basis cases.
- How many ways are there to choose 0 things from a set? Exactly **one**, the empty set.
- The right term of the sum drives us up to $\binom{k}{k}$. How many ways are there to choose k things from a k-element set? Exactly **one**, the complete set.

Binomial Coefficients Recursive Version

```python
def binomial_coefficient(n:int, k:int)->int:
    #Compute n choose k
    if n == k or k == 0:
        return 1
    return binomial_coefficient(n-1, k-1) + \
        binomial_coefficient(n-1, k)
```

Binomial Coefficients DP Implementation

```
def binomial_coefficient(n:int, k:int)->int:
    #Table of binomial coefficients
    bc = [x[:] for x in [[1]*(n+1)]*(n+1)]
    for i in range(2, n+1):
        for j in range(1, i):
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j]
    return bc[n][k]
```

i	j:0	1	2	3	4	5
0	1					
1	1	1				
2	1		1			
3	1			1		
4	1				1	
5	1					1

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm. (找出遞迴關係式)
2. Show that the number of different instances of your recurrence is bounded by a polynomial. (確認遞迴關係式不是指數型的)
3. Specify an order of evaluation for the recurrence so you always have what you need. (確定遞迴的計算先後順序)

Edit Distance

Chapter.10-Dynamic Programming

Edit Distance(編輯距離)

- Mispellings make *approximate pattern matching* an important problem
- If we are to deal with inexact string matching, we must first define a cost function telling us how far apart two strings are, i.e., a distance measure between pairs of strings.
- A reasonable distance measure minimizes the cost of the *changes* which have to be made to convert one string to another. (編輯距離就是計算從某個字詞改到另一字詞的最少字元操作步驟)
- Which word in this slide is misspelled?

String Edit Operations

- There are three natural types of changes from s to t :
  - *Substitution(替換)* – Change a single character from pattern s to a different character in text t, such as changing “shot” to “spot”.
  - *Insertion(插入)* – Insert a single character into pattern s to help it match text t, such as changing “at” to “eat”.
  - *Deletion(刪除)* – Delete a single character from pattern s to help it match text t, such as changing “hour” to “our”.

Recursive Algorithm

- We can compute the edit distance with recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.
- *If* we knew the cost of editing the three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly.

(取前一個字元的最短編輯距離經過三種操作後距離最小值)

- We *can* learn this cost, through the magic of recursion:

Recurrence Relation

- Let $D[i, j]$ be the minimum number of changes to convert the first i characters of string S into the first j characters of string T .
- Then $D[i, j]$ is the minimum of:
  - $D[i-1, j-1]$ if $S[i] = T[j]$
  - $D[i-1, j-1] + 1$ if $S[i] \neq T[j]$
  - $D[i, j-1] + 1$ for an insertion into S
  - $D[i-1, j] + 1$ for a deletion from S

Recursive Edit Distance Code

```
def string_compare_r(s:str, t:str, i:int, j:int)->int:
    opt = {}
    if i < 0:
        return j+1
    if j < 0:
        return i+1
    opt['MATCH'] = string_compare_r(s,t,i-1,j-1) + (0 if s[i] == t[j] else 1)
    opt['INSERT'] = string_compare_r(s,t,i,j-1) + 1
    opt['DELETE'] = string_compare_r(s,t,i-1,j) + 1
    return min(opt.values())
```

(i-1,j-1)	(i-1,j)
(i,j-1)	(i,j)

Speeding it Up

- This program is absolutely correct but takes exponential time because it recomputes values again and again and again!
- But there can only be $|s| \cdot |t|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the parameters of recursive calls.
- By storing the values for each of these (i, j) pairs in a table, we can avoid recomputing them and just look them up as needed.

(因為只有$|s| \cdot |t|$個不同遞迴呼叫每個回傳一個中間數值，所以我們就維護一個$|s| \cdot |t|$大小的表格來存放即可)

The Dynamic Programming Table

- The table is a two-dimensional matrix m where each of the $|s| \cdot |t|$ cells contains the cost of the optimal solution of this subproblem, as well as a parent pointer explaining how we got to this location:

```
def Cell:
    def __int__(self):
        self.cost = 0 #Cost of reaching this cell
        self.parent = -1 #Parent opt
    def __str__(self):
        return f"{self.cost}[{str(self.parent)[0]}]"
```

Differences with Dynamic Programming

- The dynamic programming version has three differences from the recursive version:
  - First, it gets its intermediate values using table lookup instead of recursive calls.
  - Second, it updates the parent field of each cell, which will enable us to reconstruct the edit-sequence later.
  - Third, it can be instrumented using a more general goal_cell() function instead of just returning m[|s|][|t|].cost. This will enable us to apply this routine to a wider class of problems.
- We assume that each string has been padded with an initial blank character, so the first real character of string s sits in $s[1]$.

Evaluation Order

- To determine the value of cell (i, j) we need three values sitting and waiting for us, namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order used in this program.
- Think of the cells as vertices, where there is an edge (i, j) if cell i's value is needed to compute cell j. Any topological sort of this DAG provides a proper evaluation order.

Dynamic Programming Edit Distance

```python
m = [x[:] for x in [[None]*(len(t)+1)]*(len(s)+1)]

def string_compare(s:str, t:str)->int:
    opt = {}
    init_matrix(m)
    for i in range(1, len(s)+1):
        for j in range(1, len(t)+1):
            opt['MATCH'] = m[i-1][j-1].cost + (0 if s[i-1] == t[j-1] else 1)
            opt['INSERT'] = m[i][j-1].cost + 1
            opt['DELETE'] = m[i-1][j].cost + 1
            m[i][j].cost = min(opt.values())
            m[i][j].parent = min(opt, key=opt.get)
    return m[len(s)][len(t)].cost
```

s \ t	0	s	p	o	r	t
0						
s						
h						
o						
t						

0[-]	1[I]	2[I]	3[I]	4[I]	5[I]
1[D]	0[M]	1[I]	2[I]	3[I]	4[I]
2[D]	1[D]	1[M]	2[M]	3[M]	4[M]
3[D]	2[D]	2[M]	1[M]	2[I]	3[I]
4[D]	3[D]	3[M]	2[D]	2[M]	2[M]

Initialize Matrix

```
def init_matrix(m:list):
    for i in range(len(m)):
        for j in range(len(m[0])):
            m[i][j] = Cell()
            if i == 0: #Row init
                m[0][j].cost = j
                if j > 0:
                    m[0][j].parent = 'INSERT'
        m[i][0].cost = i #Column init
        if i > 0:
            m[i][0].parent = 'DELETE'
```

0[-]	1[I]	2[I]	3[I]	4[I]	5[I]
1[D]	0[-]	0[-]	0[-]	0[-]	0[-]
2[D]	0[-]	0[-]	0[-]	0[-]	0[-]
3[D]	0[-]	0[-]	0[-]	0[-]	0[-]
4[D]	0[-]	0[-]	0[-]	0[-]	0[-]

Example

- Below is an example run, showing the cost and parent values turning “thou shalt” to “you should” in five moves:

	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
:		0	1	2	3	4	5	6	7	8	9	10
t:	1	1	1	2	3	4	5	6	7	8	9	10
h:	2	2	2	2	3	4	5	5	6	7	8	9
o:	3	3	3	2	3	4	5	6	5	6	7	8
u:	4	4	4	3	2	3	4	5	6	5	6	7
-:	5	5	5	4	3	2	3	4	5	6	6	7
s:	6	6	6	5	4	3	2	3	4	5	6	7
h:	7	7	7	6	5	4	3	2	3	4	5	6
a:	8	8	8	7	6	5	4	3	3	4	5	6
l:	9	9	9	8	7	6	5	4	4	4	4	5
t:	10	10	10	9	8	7	6	5	5	5	5	5

- The edit sequence from “thou-shalt” to “you-should” is DSMMMMMISMS

Reconstructing the Path

- Dynamic programming solutions are described by paths through the dynamic programming matrix, starting from the initial configuration (the empty strings $(0,0)$) down to the final goal state (the full strings $(|s|,|t|)$).
- Reconstructing these decisions is done by walking backward from the goal state, following the parent pointer to an earlier cell. The parent field for $m[i,j]$ tells us whether the transform at (i,j) was MATCH, INSERT, or DELETE.

(透過parent欄位就能回推可經過什麼操作使兩字串相同)

Reconstruct Path Code

- Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
def reconstruct_path(m:list, s:str, t:str, i:int, j:int):
    if m[i][j].parent == -1:
        return
    elif m[i][j].parent == 'MATCH':
        yield from reconstruct_path(m, s, t, i-1, j-1)
        yield ('M' if s[i-1] == t[j-1] else 'S')
    elif m[i][j].parent == 'INSERT':
        yield from reconstruct_path(m, s, t, i, j-1)
        yield 'I'
    elif m[i][j].parent == 'DELETE':
        yield from reconstruct_path(m, s, t, i-1, j)
        yield 'D'
```

Exercises

Problem of the Day

- Q1: What is the minimum cost of editing from “shot” to “sport” ? and what is the edit sequence (operation: M for Match, S for Substitution, I for Insert, D for Deletion; S,I,D costs 1; M costs 0) ?

- Q2: What is the minimum cost of editing from “shall” to “should” ? and what is the edit sequence (operation: M for Match, S for Substitution, I for Insert, D for Deletion; S,I,D costs 1; M costs 0)?

Problem of the Day (Cont.)

- Suppose you are given three strings of characters: X, Y , and Z, where $|X| = n$, $|Y| = m$, and $|Z| = n + m$. Z is said to be a shuffle of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.
  - For example, **cchocohilaptes** is a shuffle of **chocolate** and **chips**, but **chocochilatspe** is not.
  - Recurrence idea:
    - F *is the Boolean function of returning whether* Z *is a shuffle of* X *and* Y
    - $F(X_0, Y_j, Z_k) = (Y_{1\ldots j} == Z_{1\ldots k})$
    - $F(X_i, Y_0, Z_k) = (X_{1\ldots i} == Z_{1\ldots k})$
    - $F(X_i, Y_j, Z_k) = F(X_{i-1}, Y_j, Z_{k-1})\ and\ X_i == Z_k\ or$
      $F(X_i, Y_{j-1}, Z_{k-1})\ and\ Y_j == Z_k$

Problem of the Day (Cont.)

- Q3: Finish the follow table of Boolean shows the string "aaxaby" is a shuffle of X="aab" and Y="axy"

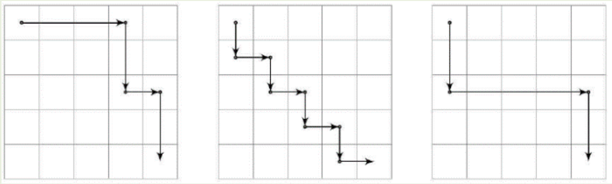
X \ Y	0	a	x	y
0	T	T	F	F
a	T			
a	T			
b	F			

- Program Exercise : Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y .

Program Exercises (Moodle CodeRunner)

■Exercise 09 (close at 5/13 23:59)

- Count The Number Of Ways To Traverse A 2D Array:
  - ➢ *In this problem you are to count the number of ways of starting at the top-left corner of a 2D array and getting to the bottom-right corner. All moves must either go right or down.*
  - ➢ *For example, the following picture shows three ways in a 5x5 2D array.*



- Interleaving String:
  - ➢ *Given three strings of characters: X, Y, and Z, where |X|=n, |Y|=m, and |Z|=n+m, find whether Z is formed by the interleaving of X and Y.*
  - ➢ *For example, **cchocohilaptes** is a shuffle of **chocolate** and **chips**, but **chocochilatspe** is not.*