Paradigmas de Resolución de Problemas (Parte 2)

folivares13@alumnos.utalca.cl

@Jestter

Lectura complementaria

- El contenido de esta presentación se encuentra en el **capítulo 3** del libro **Competitive Programming 3**

Dynamic Programming (DP)

	59	33	74	87	39	8	91
71	7	100	4	17	55	82	95
97	44	80	86	88	62	26	41
96	94	14	1	12	56	15	52
5	37	2	10	21	23	61	85
9	29	24	77	89	90	68	18
60	30	64	57	78	50	99	98
70	83	66	93	16	73	22	

Dynamic Programming (DP)

- La programación dinámica es un paradigma para abordar problemas que se basa en transformar un problema complejo en una secuencia de sub-problemas, teniendo la posibilidad de separar éstos últimos en otros sub-problemas y así sucesivamente
- Prerrequisitos:
  - **El problema tiene sub-estructuras óptimas:** La solución del sub-problema es parte de la solución del problema original
  - **El problema tiene sub-problemas solapados:** Los mismos sub-problemas se repiten en las distintas ramas de la búsqueda de soluciones

- Para que la programación dinámica sea efectiva, es necesario identificar:
  - **Estados:** corresponde al conjunto de parámetros que definen de manera única un sub-problema
  - **Transiciones:** define el cambio que lleva de un estado al otro
- Variantes:
  - **Bottom-Up:** Esta es la forma clásica de DP. Se construyen las soluciones a los sub-problemas desde los casos base según las reglas del problema.
  - **Top-Down:** Esta es una variante recursiva llamada **Memoización**. Se realiza un búsqueda guardando los resultados de los estados ya recorridos en una tabla (arreglo, hashtable, etc.) para que sean consultados en las soluciones de otros estados.

Dynamic Programming (DP)

Top-Down	Bottom-Up
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Table 3.2: DP Decision Table

Problemas para discusión

- Jill Rides Again https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=448
- **Maximum Sum** **https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=44**
- **Is Bigger Smarter** https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1072

Kadane - Max 1D Range Sum

```
// inside int main()
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5 };   // a sample array A
  int sum = 0, ans = 0;           // important, ans must be initialized to 0
  for (int i = 0; i < n; i++) {                          // linear scan, O(n)
    sum += A[i];                       // we greedily extend this running sum
    ans = max(ans, sum);                  // we keep the maximum RSQ overall
    if (sum < 0) sum = 0;                   // but we reset the running sum
  }                                              // if it ever dips below 0
  printf("Max 1D Range Sum = %d\n", ans);
```

2D Range Sum

```
scanf("%d", &n);                              // the dimension of input square matrix
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];                   // if possible, add from top
  if (j > 0) A[i][j] += A[i][j - 1];                  // if possible, add from left
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];     // avoid double count
}                                                 // inclusion-exclusion principle
```

Longest Increasing Subsequence (LIS)

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

Figure 3.9: Longest Increasing Subsequence

1. LIS(0) = 1 // the base case
2. LIS(i) = max(LIS(j) + 1), $\forall j \in [0..i-1]$ y $A[j] < A[i]$