

# Dynamic Programming

*Topics:*

What is the dynamic programming?

Applications

- Fibonacci Numbers
- Longest Increasing Sequence
- Minimum Weight Triangulation
- The Partition Problem
- Approximate String Matching

Conclusions

*Presented by: Zhihui Xue*

# Dynamic Programming

In optimization problems (maximizes or minimizes some function), we need always get the best possible solution.

- *Greedy* algorithms occasionally happen to produce a global optimum for certain problems. These are typically efficient.
- *Exhaustive search* algorithms must always produce the optimum result, but usually at a prohibitive cost.

*Dynamic programming* combines the best of both worlds. It typically removes one element from the problem, solves the smaller problem, and then uses the solution to this smaller problem to add back the element in the proper way.

- This technique systematically considers all possible decisions and always selects the one that proves to be the best. By storing the consequences of all possible decisions to date, the total amount of work is minimized.

## Fibonacci Numbers

The *Fibonacci numbers* were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. The recurrence relation is:

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

The numbers are {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...}

- *How to compute the  $n$ th Fibonacci number?*

```
Fibonacci[n]      /* recursive algorithm */
    if (n=0) then return (0)
    else if (n=1) then return (1)
    else return (Fibonacci[n-1]+Fibonacci[n-2])
```

- Time Complexity:  $F_{n+1}/F_n \approx 1.61803$ ,  $F_n > 1.6^n$ ,  $O(1.6^n)$

```
Fibonacci[n]      /* dynamic programming */
    F0=0          F1=1
    For i=2 to n   Fi=Fi-1+Fi-2
```

- Time and Space Complexity:  $O(n)$  and  $O(2)$ .
- Tradeoff between space and time.

## Longest Increasing Sequence

- Find the longest increasing sequence in a sequence of  $n$  numbers. The selected elements need not be neighbors of each other, but must be in sorted order from left to right.

For example  $S=(9, 5, 2, 8, 7, 3, 1, 6, 4)$ , the longest increasing subsequence has length 3 and is either  $(2,3,4)$  or  $(2,3,6)$ .

- Using *exhaustive search* for all subsets of the  $n$  numbers, *there are*  $2^n$  different subsets.
- Define  $h_i$  to be the length of the longest sequence ending with  $s_i$ . The longest increasing sequence containing the  $i$ th number will be formed by appending it to the longest increasing sequence to the left of  $i$  that ends on a number smaller than  $s_i$ .

$$h_i = \max_{0 \leq j \leq i-1} h_j + 1, \quad \text{where } (s_j < s_i)$$

$$h_0=0, \quad s_0 \text{ be a very small number}$$

*The length of the longest increasing subsequence is*  $\max_{1 \leq i \leq n} h_i$ .

- Each one of the  $n$  values of  $h_i$  is computed by comparing  $s_i$  against up to  $i-1$   $n$  values, so time complexity is  $O(n^2)$ . The space complexity is  $O(n)$ .
- By using some data structures (e.g. binary tree), time complexity can be reduced to  $O(n \lg n)$ . The space complexity remains the same.
- By storing the index  $p_i$  of the element that appears immediately before  $s_i$  in the longest increasing sequence ending at  $s_i$ , we can reconstruct the actual sequence starting from the last points.

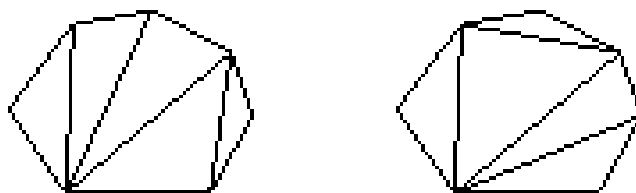
Example:  $S=(9, 5, 2, 8, 7, 3, 1, 6, 4)$

$i$	1	2	3	4	5	6	7	8	9
sequence $s_i$	9	5	2	8	7	3	1	6	4
length $h_i$	1	1	1	2	2	2	1	3	3
predecessor $p_i$				2	2	3		6	6

## Minimum Weight Triangulation

A *triangulation* of a polygon  $P = \{v_1, v_2, \dots, v_n, v_1\}$  is a set of non-intersecting diagonals that partitions the polygon into triangles.

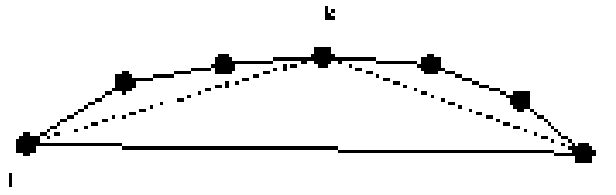
The *weight* of a triangulation is the sum of the lengths of its diagonals.



- There are  $\frac{1}{n-1} \binom{2n-4}{n-2}$  different triangulations. How to find the minimum weight triangulation?
- Observe that every edge of the polygon must be in exactly one triangle. Turning this edge into a triangle means identifying the third vertex. The third vertex will partition the polygon into two smaller pieces, both of which need to be triangulated optimally.

- Let  $T[i, j]$  be the cost of triangulating from vertex  $v_i$  to  $v_j$ , ignoring the length of the chord  $d_{ij}$  (avoid double counting).

$$T[i, j] = \min_{k=i+1}^{j-1} (T[i, k] + T[k, j] + d_{ik} + d_{kj}) \quad T[i, i+1] = 0$$



Selecting the vertex  $k$  to pair with an edge  $(i, j)$  of the polygon

Minimum-Weight-Triangulation( $P$ )

for  $i=1$  to  $n-1$  do  $T[i, i+1]=0$

for  $gap=1$  to  $n-1$

for  $i=1$  to  $n-gap$  do

$j=i+gap$

$$T[i, j] = \min_{k=i+1}^{j-1} (T[i, k] + T[k, j] + d_{ik} + d_{kj})$$

return  $T[1, n]$

- $\binom{n}{2}$  of  $T$ , each of which takes  $O(j-i-1)$  time. Time and space complexities are  $O(n^3)$  and  $O(n^2)$ .

## The Partition Problem

Suppose a given arrangement  $S$  of non-negative numbers  $\{s_1, \dots, s_n\}$  and an integer  $k$ . How to cut  $S$  into  $k$  or fewer ranges, so as to minimize the maximum sum over all the ranges?

- *A simple heuristic*: compute the average size of a partition, then insert the dividers so as to come close to this average.
- *Exhaustive search*: there are  $\binom{n}{k-1}$  possible partitions.

Let  $M[n, k]$  be the minimum possible cost over all partitionings of  $\{s_1, \dots, s_n\}$  into  $k$  ranges, where the cost of a partition is the largest sum of elements in one of its parts.

$$M[n, k] = \min_{i=1}^{n-1} \max(M[i, k-1], \sum_{j=i+1}^n s_j)$$

$$M[1, k] = s_1 \quad \text{for all } k > 0 \qquad M[n, 1] = \sum_{i=1}^n s_i$$

$k \times n$  cells, each  $M[n', k']$  finds the minimum of  $n'$  quantities, each of which is the maximum of the table lookup and a sum of at most  $n'$  elements. Time Complexity is  $O(kn^3)$ .

Partition[ $S, k$ ]

$p[0]=0$

for  $i=1$  to  $n$  do  $p[i]=p[i-1]+s_i$  /\* compute  $p[k]=\sum_{i=1}^k s_i$  \*/

for  $i=1$  to  $n$  do  $M[i,1] = p[i]$  /\* boundary conditions \*/

for  $i=1$  to  $k$  do  $M[i, 1] = s_1$

for  $i=2$  to  $n$  do /\* evaluate main recurrence \*/

for  $j = 2$  to  $k$  do

$M[i, j] = \infty$

for  $x = 1$  to  $i-1$  do

$s = \max(M[x, j-1], p[i]-p[x])$

if ( $M[i, j] > s$ ) then

$M[i, j] = s$

$D[i, j] = x$  /\* divider position \*/

ReconstructPartition( $S, D, n, k$ )

If ( $k = 1$ ) then print the first partition  $\{s_1, \dots, s_n\}$

else ReconstructPartition( $S, D, D[n, k], k-1$ )

Print the  $k$ th partition  $\{s_{D[n,k]+1}, \dots, s_n\}$

- Compute the small values first and store  $\sum_{i=1}^k s_i$ , evaluate the recurrence in linear time per cell. The time complexity is reduced to  $O(kn^2)$ . The space complexity is  $O(kn)$ .
- Using matrix  $D$  to record the divider position, we can reconstruct the partitions by working backward from  $D[n,k]$ .

*Example: Dynamic programming partitioning in  $\{1,2,3,4,5,6,7,8,9\}$*

$M$	$k$				$D$	$k$		
$n$	1	2	3		$n$	1	2	3
1	1	1	1		1	-	-	-
2	3	2	2		2	-	1	1
3	6	3	3		3	-	2	2
4	10	6	4		4	-	3	3
5	15	9	6		5	-	3	4
6	21	11	9		6	-	4	5
7	28	15	11		7	-	5	6
8	36	21	15		8	-	5	6
9	45	24	17		9	-	6	7

$$M[9,3] = \min \{$$

$$\max(1, 44) = 44, \quad \max(2, 42) = 42, \quad \max(3, 39) = 39$$

$$\max(6, 35) = 35, \quad \max(9, 30) = 30, \quad \max(11, 24) = 24$$

$$\max(15, 17) = 17, \quad \max(21, 9) = 21 \} = 17$$

## Approximate String Matching

Let  $P$  be a pattern string and  $T$  a text string over the same alphabet. How to find the smallest number of changes to transform a substring of  $T$  into  $P$ , where the changes may be:

1. *Substitution*: KAT -- CAT
2. *Insertion*: CT -- CAT
3. *Deletion*: CAAT -- CAT

Let  $D[i,j]$  be the minimum number of differences between  $P_1, \dots, P_i$  and the part of  $T$  ending at  $j$ .  $D[i,j]$  is the minimum of:

1. If  $(P_i=T_j)$  then  $D[i-1, j-1]$  else  $D[i-1, j-1]+1$ . Either match or substitute the  $i$ th and  $j$ th characters.
  2.  $D[i-1, j]+1$ . Extra character in  $P$ , cost of an insertion.
  3.  $D[i, j-1]+1$ . Extra character in  $T$ , cost of a deletion.
- Let  $n=|P|$ ,  $m=|T|$ ,  $D[n,m]$  is the cost of comparing the spelling of two words.

- $D[i,0]=i$ . Cost of matching the first  $i$  characters of the pattern with none of the text.
- $D[0,i]$  Cost of matching the first  $i$  characters of the text with none of the pattern.
  - $D[0,i]=i$ . if match the entire pattern against the entire text, pay the cost of the deletions.
  - $D[0,i]=0$ . if find the pattern occurs in a text, pay no cost for deleting the first  $i$  characters of the text.

MinimumDifference( $P,T$ )    /\* full pattern matching \*/

For  $i = 0$  to  $n$  do  $D[i,0]=i$     /\* initialization \*/

For  $i = 0$  to  $m$  do  $D[0,i]=i$

For  $i = 1$  to  $n$  do    /\* recurrence \*/

    For  $j = 1$  to  $m$  do  $D[i,j]=\min(D[i-1,j-1]+matchcost(s_i,t_j),$

$D[i-1,j]+1, D[i,j-1]+1)$  /\* if  $s_i=t_j$  matchcost=0 else 1 \*/

- Each cell  $D[i,j]$  compares two characters and looks at three other cells. It requires constant time, so the total time is  $O(mn)$ .

## Reconstructing the actual alignment:

From the cell with the lowest cost, walk upwards and backwards through the matrix. The direction of each backwards step (to the left, up, or diagonal to the upper left) identifies whether it was an insertion, deletion, or match/substitution. It takes  $O(n+m)$  time.

*Example: Compare  $P=abcdefghijkl$  with  $T=bcdeffghixkl$*

		b	c	d	e	f	f	g	h	i	x	k	l
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	1	1	2	3	4	5	6	7	8	9	10	11	12
b	2	1	2	3	4	5	6	7	8	9	10	11	12
c	3	2	1	2	3	4	5	6	7	8	9	10	11
d	4	3	2	1	2	3	4	5	6	7	8	9	10
e	5	4	3	2	1	2	3	4	5	6	7	8	9
f	6	5	4	3	2	1	2	3	4	5	6	7	8
g	7	6	5	4	3	2	2	2	3	4	5	6	7
h	8	7	6	5	4	3	3	3	2	3	4	5	6
i	9	8	7	6	5	4	4	4	3	2	3	4	5
j	10	9	8	7	6	5	5	5	4	3	3	4	5
k	11	10	9	8	7	6	6	6	5	4	4	3	4
l	12	11	10	9	8	7	7	7	6	5	5	4	3

- By divide and conquer, find the best paths from  $D[1,1]$  to  $D[x,m/2]$  and from  $D[x,m/2]$  to  $D[n,m]$ , both of which can be solved recursively. The space complexity is  $O(m)$ .

## Conclusion

- Every dynamic programming (DP) solution has three parts:
  1. Formulate as a recurrence relation or recursive algorithm.
  2. Show that the number of different values of the recurrence is bounded by a (hopefully small) polynomial.
  3. Specify an order of evaluation for the recurrence, so always have the partial results available when need them.
- DP can be applied to any problem that observes the *principle of optimality*. That is, the partial solutions can be optimally extended with regard to the state after the partial solution instead of the partial solution itself.
- DP must keep track of the partial solutions. For any optimization problem having relatively few possible stopping states, DP will likely be efficient.