Dynamic Programming

Introduction

Dynamic programming is a confusing name for a programming technique that dramatically reduces the runtime of algorithms: from exponential to polynomial. The basic idea is to try to avoid solving the same problem or subproblem twice. Here is a problem to demonstrate its power:

Given a sequence of as many as 10,000 integers (0 < integer < 100,000), what is the maximum decreasing subsequence? Note that the subsequence does not have to be consecutive.

Recursive Descent Solution

The obvious approach to solving the problem is recursive descent. One need only find the recurrence and a terminal condition. Consider the following solution:

```
#include <stdio.h>

 long n, sequence[10000];
 main () {
          FILE *in, *out;
          int i;
          in = fopen ("input.txt", "r");
          out = fopen ("output.txt", "w");
          fscanf(in, "%ld", &n);
          for (i = 0; i < n; i++) fscanf(in, "%ld", &sequence[i]);
          fprintf (out, "%d\n", check (0, 0, 99999));
          exit (0);
}

check (start, nmatches, smallest) {
        int better, i, best=nmatches;
        for (i = start; i < n; i++) {
                if (sequence[i] < smallest) {
                        better = check (i, nmatches+1, sequence[i]);
                        if (better > best) best = better;
                }
20          }
21          return  best;
22    }
```

Lines 1-9 and and 11-12 are arguably boilerplate. They set up some standard variables and grab the input. The magic is in line 10 and the recursive routine `check'. The `check' routine knows where it should start searching for smaller integers, the length of the longest sequence so far, and the smallest integer so far. At the cost of an extra call, it terminates `automatically' when `start' is no longer within proper range. The `check' routine is simplicity itself. It traverses along the list looking for a smaller integer than the `smallest' so far. If found, `check' calls itself recursively to find more.

The problem with the recursive solution is the runtime:

N	Seconds
60	0.130
70	0.360
80	2.390
90	13.190

Since the particular problem of interest suggests that the maximum length of the sequence might approach six digits, this solution is of limited interest.

Starting At The End

When solutions don't work by approaching them `forwards' or `from the front', it is often fruitful to approach the problem backward. In this case, that means looking at the end of the sequence first.

Additionally, it is often fruitful to trade a bit of storage for execution efficiency. Another program might work from the end of the sequence, keeping track of the longest descending (sub-)sequence so far in an auxiliary variable.

Consider the sequence starting at the end, of length 1. Any sequence of length 1 meets all the criteria for a longest sequence. Notate the `bestsofar' array as `1' for this case.

Consider the last two elements of the sequence. If the penultimate number is larger than the last one, then the `bestsofar' is 2 (which is 1 + `bestsofar' for the last number). Otherwise, it's `1'.

Consider any element prior to the last two. Any time it's larger than an element closer to the end, its `bestsofar' element becomes at least one larger than that of the smaller element that was found. Upon termination, the largest of the `bestsofar's is the length of the longest descending subsequence.

This is fairly clearly an O(N^2) algorithm. Check out its code:

```
 1   #include <stdio.h>
 2   #define MAXN 10000
 3   main () {
 4           long num[MAXN], bestsofar[MAXN];
 5           FILE *in, *out;
 6           long n, i, j, longest = 0;
 7           in = fopen ("input.txt", "r");
 8           out = fopen ("output.txt", "w");
 9           fscanf(in, "%ld", &n);
10          for (i = 0; i < n; i++) fscanf(in, "%ld", &num[i]);
11          bestsofar[n-1] = 1;
12          for (i = n-1-1; i >= 0; i--) {
13                  bestsofar[i] = 1;
14                  for (j = i+1; j < n; j++) {
15                          if (num[j] < num[i] && bestsofar[j] >= bestsofar[i]) {
16                                  bestsofar[i] = bestsofar[j] + 1;
17                                  if (bestsofar[i] > longest) longest = bestsofar[i];
18                          }
19                  }
20          }
21          fprintf(out, "bestsofar is %d\n", longest);
22          exit(0);
23  }
```

Again, lines 1-10 are boilerplate. Line 11 sets up the end condition. Lines 12-20 run the O(N^2) algorithm in a fairly straightforward way with the `i' loop counting backwards and the `j' loop counting forwards. One line longer then before, the runtime figures show better performance:

N	Secs
1000	0.080
2000	0.240
3000	0.550
4000	0.950

5000	1.450
6000	2.080
7000	2.990
8000	3.700
9000	4.700
10000	6.330
11000	7.350

The algorithm still runs too slow (for competitions) at N=9000.

That inner loop (``search for any smaller number'') begs to have some storage traded for it.

A different set of values might best be stored in the auxiliary array. Implement an array `bestrun' whose index is the length of a long subsequence and whose value is the first (and, as it turns out, `best') integer that heads that subsequence. Encountering an integer larger than one of the values in this array means that a new, longer sequence can potentially be created. The new integer might be a new `best head of sequence', or it might not. Consider this sequence:

```
    10  8  9  4  6  3
```

Scanning from right to left (backward to front), the `bestrun' array has but a single element after encountering the 3:

```
0:3
```

Continuing the scan, the `6' is larger than the `3', to the `bestrun' array grows:

```
0:3
1:6
```

The `4' is not larger than the `6', though it is larger than the `3', so the `bestrun' array changes:

```
0:3
1:4
```

The `9' extends the array:

```
0:3
1:4
2:9
```

The `8' changes the array similar to the earlier case with the `4':

```
0:3
1:4
2:8
```

The `10' extends the array again:

```
0:3
1:4
2:8
3:10
```

and yields the answer: 4 (four elements in the array).

Because the `bestrun' array probably grows much less quickly than the length of the processed sequence, this algorithm probabalistically runs much faster than the previous one. In practice, the speedup is large. Here's a coding of this algorithm:

```
#include <stdio.h>
#define MAXN 200000
main () {
        FILE *in, *out;
        long num[MAXN], bestrun[MAXN];
        long n, i, j, highestrun = 0;
        in = fopen ("input.txt", "r");
        out = fopen ("output.txt", "w");
        fscanf(in, "%ld", &n);
       for (i = 0; i < n; i++) fscanf(in, "%ld", &num[i]);
       bestrun[0] = num[n-1];
       highestrun = 1;
       for (i = n-1-1; i >= 0; i--) {
               if (num[i] < bestrun[0]) {
                   bestrun[0] = num[i];
                   continue;
           }
           for (j = highestrun - 1; j >= 0; j--) {
                   if (num[i] > bestrun[j]) {
                           if (j == highestrun - 1 || num[i] < best
run[j+1]){
                                   bestrun[++j] = num[i];
                                   if (j == highestrun) highestrun++;
                                   break;
                           }
                   }
           }
       }
       printf("best is %d\n", highestrun);
       exit(0);
   }
```

Again, lines 1-10 are boilerplate. Lines 11-12 are initialization. Lines 14-17 are an optimization for a new `smallest' element. They could have been moved after line 26. Mostly, these lines only effect the `worst' case of the algorithm when the input is sorted `badly'.

Lines 18-26 are the meat that searches the bestrun list and contain all the exceptions and tricky cases (bigger than first element? insert in middle? extend the array?). You should try to code this right now - without memorizing it.

The speeds are impressive. The table below compares this algorithm with the previous one, showing this algorithm worked for N well into five digits:

N	orig	Improved
1000	0.080	0.030
2000	0.240	0.030
3000	0.550	0.050
4000	0.950	0.060
5000	1.450	0.080
6000	2.080	0.090
7000	2.990	0.110
8000	3.700	0.130
9000	4.700	0.140
10000	6.330	0.160
11000	7.350	0.170
20000		0.290
40000		0.570
60000		0.910
80000		1.290
100000		2.220

Marcin Mika points out that you can simplify this algorithm to this tiny little solution:

```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int     best[SIZE];        // best[] holds values of the optimal
sub-sequence

int
main (void) {
    FILE *in  = fopen ("input.txt", "r");
    FILE *out = fopen ("output.txt", "w");
    int     i, n, k, x, sol = -1;

    fscanf (in, "%d", &n);     // N = how many integers to read in
    for (i = 0; i < n; i++) {
       best[i] = -1;
       fscanf (in, "%d", &x);
       for (k = 0; best[k] > x; k++)
           ;
       best[k] = x;
       sol = MAX (sol, k + 1);
    }
    printf ("best is %d\n", sol);
    return 0;
}
```

Not to be outdone, Tyler Lu points out the program below. The solutions above use a linear search to find the appropriate location in the 'bestrun' array to insert an integer. However, because the auxiliary array is sorted, using binary search will make it run even faster, decreasing the runtime to O(N log N).

```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int     best[SIZE];        // best[] holds values of the optimal
sub-sequence

int
main (void) {
    FILE *in  = fopen ("input.txt", "r");
    int i, n, k, x, sol;
    int low, high;

    fscanf (in, "%d", &n);     // N = how many integers to read in
    // read in the first integer
    fscanf (in, "%d", &best[0]);
    sol = 1;
    for (i = 1; i < n; i++) {
        best[i] = -1;
          fscanf (in, "%d", &x);

        if(x >= best[0]) {
          k = 0;
          best[0] = x;
        }
        else {
          // use binary search instead
          low = 0;
          high = sol-1;
          for(;;) {
            k = (int) (low + high) / 2;
            // go lower in the array
            if(x > best[k] && x > best[k-1]) {
              high = k - 1;
              continue;
            }
            // go higher in the array
            if(x < best[k] && x < best[k+1]) {
              low = k + 1;
              continue;
            }
            // check if right spot
            if(x > best[k] && x < best[k-1])
              best[k] = x;
            if(x < best[k] && x > best[k+1])
              best[++k] = x;
            break;
          }
        }
               sol = MAX (sol, k + 1);
    }
    printf ("best is %d\n", sol);
    fclose(in);
    return 0;
}
```

Summary

These programs demonstrate the main concept behind dynamic programming: build larger solutions based on previously found solutions. This building-up of solutions often yields programs that run very quickly.

For the previous programming challenge, the main subproblem was: Find the largest decreasing subsequence (and its first value) for numbers to the `right' of a given element.

Note that this sort of approach solves a class of problems that might be denoted ``one-dimensional''.

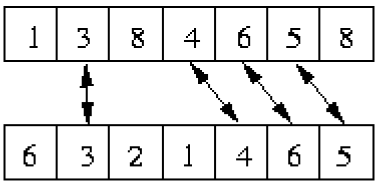
Two Dimensional DP

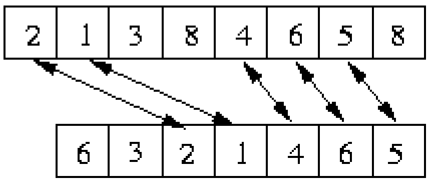
It is possible to create higher dimension problems such as:

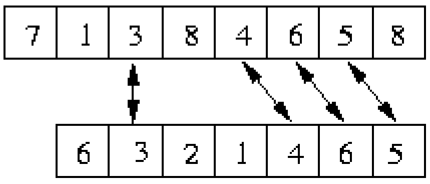
Given two sequences of integers, what is the longest sequence which is a subsequence of both sequences?

Here, the subproblems are the longest common subsequence of smaller sequences (where the sequences are the tails of the original subsequences). First, if one of the sequences contains only one element, the solution is trivial (either the element is in the other sequence or it isn't).

Look at the problem of finding the longest common subsequence of the last i elements of the first sequence and the last j elements of the second sequences. There are only two possibilities. The first element of the first tail might be in the longest common subsequence or it might not. The longest common sequence not containing the first element of the first tail is merely the longest common subsequence of the last i-1 elements of the first sequence and the last j elements of the second subsequence. The other possibility results from some element in the tail of the second sequence matching the first element in tail of the first, and finding the longest common subsequence of the elements after those matched elements.







Pseudocode

Here's the pseudocode for this algorithm:

```
      # the tail of the second sequence is empty
1      for element = 1 to length1
```

```
        length[element,  length2+1]  =  0


 #  the  tail  of  the  first  sequence  has  one  element
    matchelem  =  0
    for  element  =  length2  to  1
        if  list1[length1]  =  list2[element]
            matchelem  =  1
        length[length1,element] = nmatchlen

 #  loop  over  the  beginning  of  the  tail  of  the  first  sequence
    for  loc  =  length1-1  to  1
        maxlen  =  0
        for  element  =  length2  to  1
 #  longest  common  subsequence  doesn't  include  first  element
            if  length[loc+1,element]  >  maxlen
                maxlen  =  length[loc+1,element]
 #  longest  common  subsequence  includes  first  element
            if  list1[loc]  =  list2[element]  &&
                                          length[loc+1,element+1]+1  >  maxlen
                    maxlen  =  length[loc,element+1]  +  1
            length[loc,element]  =  maxlen
```

This program runs in O(*N* x *M*) time, where *N* and *M* are the respectively lengths of the sequences.

Note that this algorithm does not directly calculate the longest common subsequence. However, given the length matrix, you can determine the subsequence fairly quickly:

```
    location1  =  1
    location2  =  1



    while  (length[location1,location2]  !=  0)
        flag  =  False
        for  element  =  location2  to  length2
            if  (list1[location1]  =  list2[element]  AND
                  length[location1+1,element+1]+1  =  length[location1,location2]
                output  (list1[location1],list2[element])
                location2  =  element  +  1
                flag  =  True
               break  for
           location1  =  location1  +  1
```

The trick to dynamic programming is finding the subproblems to solve. Sometimes it involves multiple parameters:

A bitonic sequence is a sequence which increases for the first part and decreases for the second part. Find the longest bitonic sequence of a sequence of integers (technically, a bitonic can either increase-then-decrease or decrease-then-increase, but for this problem, only increase and then decrease will be considered).

In this case, the subproblems are the longest bitonic sequence and the longest decreasing sequence of prefixes of the sequence (basically, what's longest sequence assuming the turn has not occurred yet, and what's longest sequence starting here assuming the turn has already occurred).

Sometimes the subproblems are well hidden:

You have just won a contest where the prize is a free vacation in Canada. You must travel via air, and the cities are ordered from east to west. In addition, according to the rules, you must start at the further city west, travel only east until you reach the furthest city east, and then fly only west until you reach your starting location. In addition, you may visit no city more than once (except the starting city, of course).

Given the order of the cities, with the flights that can be done (you can only fly between certain cities, and just because you can fly from city A to city B does not mean you can fly the other direction), calculate the maximum number of cities you can visit.

The obvious item to try to do dynamic programming on is your location and your direction, but it's important what path you've taken (since you can't revisit cities on the return trip), and the number of paths is too large to be able to solve (and store) the result of doing all of those subproblems.

However, if, instead of trying to find the path as described, it is found a different manner, then the number of states greatly decreases. Imagine having two travelers who start in the western most city. The travelers take turns traveling east, where the next traveler to move is always the western-most, but the travelers may never be at the same city, unless it is either the first or the last city. However, one of the traveler is only allowed to make "reverse flights," where he can travel from city A to city B if and only if there is a flight from city **B** to city **A**.

It's not too difficult to see that the paths of the two travelers can be combined to create a round-trip, by taking the normal traveler's path to the eastern-most city, and then taking the reverse of the other traveler's path back to the western-most city. Also, when traveler x is moved, you know that the traveler y has not yet visited any city east of traveler x except the city traveler y is current at, as otherwise traveler y must have moved once while x was west of y. Thus, the two traveler's paths are disjoint. Why this algorithm might yield the maximum number of cities is left as an exercise.

Recognizing Problems solvable by dynamic programming

Generally, dynamic programming solutions are applied to those solutions which would otherwise be exponential in time, so if the bounds of the problem are too large to be able to be done in exponential time for any but a very small set of the input cases, look for a dynamic programming solution. Basically, any program you were thinking about doing recursive descent on, you should check to see if a dynamic programming solution exists if the inputs limits are over 30.

Finding the Subproblems

As mentioned before, finding the subproblems to do dynamic programming over is the key. Your goal is to completely describe the state of a solution in a small amount of data, such as an integer, a pair of integers, a boolean and an integer, etc.

Almost without fail, the subproblem will be the `tail-end' of a problem. That is, there is a way to do the recursive descent such that at each step, you only pass a small amount of data. For example, in the air travel one, you could do recursive descent to find the complete path, but that means you'd have to pass not only your location, but the cities you've visited already (either as a list or as a boolean array). That's too much state for dynamic programming to work on. However, recursing on the pair of cities as you travel east subject to the constant given is a very small amount of data to recurse on.

If the path is important, you will not be able to do dynamic programming unless the paths are **very** short. However, as in the air travel problem, depending on how you look at it, the path may not be important.

Sample Problems

Polygon Game [1998 IOI]

Imagine a regular N-gon. Put numbers on nodes, either and the operators `+' or `*' on the edges. The first move is to remove an edge. After that, combine (e.g., evaluate the simple term) across edges, replacing the edge and end points with node with value equal to value of end point combined by operations, for example:

```
...-- 3 --+-- 5 --*-- 7 --...
...----- 8 ----*----- 7 --...
...-------- 56 -----------...
```

Given a labelled N-gon, maximize the final value computed.

Subset Sums [Spring 98 USACO]

For many sets of consecutive integers from 1 through N (1 <= N <= 39), one can partition the set into two sets whose sums are identical. For example, if N=3, one can partition the set {1, 2, 3} in one way so that the sums of both subsets are identical:

{3} and {1,2}

This counts as a single partitioning (i.e., reversing the order counts as the same partitioning and thus does not increase the count of partitions).

If N=7, there are 4 ways to partition the set {1, 2, 3, ... 7} so that each partition has the same sum:

{1,6,7} and {2,3,4,5}
{2,5,7} and {1,3,4,6}
{3,4,7} and {1,2,5,6}
{1,2,4,7} and {3,5,6}

Given N, your program should print the number of ways a set containing the integers from 1 through N can be partitioned into two sets whose sums are identical. Print 0 if there are no such ways.

Number Game [IOI 96, maybe]

Given a sequence of no more than 100 integers (-32000..32000), two opponents alternate turns removing the leftmost or rightmost number from a sequence. Each player's score at the end of the game is the sum of those numbers he or she removed. Given a sequence, determine the maximum winning score for the first player, assuming the second player plays optimally.

动态规划

引言

译 by 铭(**特别感谢,这是她在N次死机中译出来的**)

作为显著减少算法运行时间（从指数的到多项式的）的一种编程技巧，动态规划是一个令人困惑的名字。它的基本思想是努力避免重复解决相同问题或子问题。如下面的问题证明了它的威力：

已知10，000个整数的序列（整数大于0小于100，000），求最长递减子序列。注意，子序列不一定非得连续。

递归下降解法

解决此问题显然的方法是递归下降。只需找到循环和一个终止条件。看看下面的解法：

```
#include <stdio.h>
long n, sequence[10000];
main () {
FILE *in, *out;
int i;
in = fopen ("input.txt", "r");
out = fopen ("output.txt", "w");
fscanf(in, "%ld", &n); &a mp;n bsp;
for (i = 0; i < n; i++) fscanf(in, "%ld", &sequence[i]);
fprintf (out, "%d\n", check (0, 0, 99999));
exit (0);
}

check (start, nmatches, smallest) {
int better, i, best=nmatches;
for (i = start; i < n; i++) {
if (sequence[i] < smallest) {
better = check (i, nmatches+1, sequence[i]);
& amp;n bsp; if (better > best) best = better;
}
}
return best;
22 }
```

1-9 行和 11-12 行有疑义。它们创建了一些标准变量控制输入。秘密就在于第 10 行的递归程序‘check’。‘check’程序知道它应该从哪开始查寻较小的整数，当前最长序列的长度及最小整数。由于额外调用的开销，当“开始”不超过其合适的范围，它就会“自动地”终止。‘check’程序本身很简单。它沿着列表寻找比当前‘smallest’更小的整数。如果找到了，‘check’递归调用它本身来寻找更多的整数。
同递归解法随之而来的问题是运行时间：

N	Seconds
60	0.130
70	0.360
80	2.390
90	13.190

由于序列的最大长度可能会接近 6 位数，这个解法是受限的。

从末端开始

当利用向前或从前面开始的方法无法运作时，从相反的方面开始处理问题却可以收获良多。像这个例子可以先从序列的末端开始。
另外，用一点儿存贮空间来交换执行效率也会获利很多。另一种程序可能会从序列的末尾开始，用一个辅助变量来记录当前最长递减（子）序列。
考虑长度为 1 的序列从末端开始的情况。任意长度为 1 的序列都满足最长序列的所有标准。本例将‘bestsofar’数组标记为‘1’。
考虑序列的最后两个元素。如果倒数第二个数字比最后那个大，那么‘bestsofar’为 2（1+最后那个数字的‘bestsofar’）。否则为 1。
考虑在最后两个元素之前的任一元素。它总是比接近末尾的那个元素大。它的‘bestsofar’元素至少比找到的较小元素的大 1。最后‘bestsofar’的最大值就是最长递减子序列的长度。
这很明显是一个 O(N 2)算法。代码如下：

```
1 #include <stdio.h>
2 #define MAXN 10000
3 main () {
4 long num[MAXN], bestsofar[MAXN];
5 FILE *in, *out;
6 long n, i, j, longest = 0;
7 in = fopen ("input.txt", "r");
8 out = fopen ("output.txt", "w");
9 fscanf(in, "%ld", &n);
10 for (i = 0; i < n; i++) fscanf(in, "%ld", &num[i]);
11 bestsofar[n-1] = 1;
12 for (i = n-1-1;&am p;nb sp;i >= 0; i--) {
13 bestsofar[i] = 1;
14 for (j = i+1; j < n; j++) {
15 if (num[j] < num[i] && bestsofar[j] >= bestsofar[i]) {
16 bestsofar[i] = bestsofar[j] + 1;
17 if (bestsofar[i] > longest) longest = bestsofar[i];
18 }
19 }
20 }
21 fprintf(out, "bestsofar is %d\n", longest);
22 exit(0);
23 }
```

同样地，11 行建立了末尾条件。12-20 行以‘i’循环从后计数和‘j’循环从前计数的巧妙直接的方法运行 O（N2）算法。尽管比前面的那个算法多 1 行，运行时间却更佳：

```
N Secs
1000 0.080
2000 0.240
3000 0.550
4000 0.950
5000 1.450
6000 2.080
7000 2.990
8000 3.700
9000 4.700
10000 6.330
11000 7.350
```

当 N=9000 时由于竞争，算法依就很慢。
内部循环（“寻找任意较小的数”）乞求用一些存储空间来交换时间。
一组不同的值最好存在辅助数组中。数组‘bestrun’的指针是最长子序列的长度，其值是率领子序列的第一个（也被证明是最佳的）整数。执行该数组。当遇到一个整数比在这个数组中的值大时就意味着可能会产生一个新的更长的序列。新整数可能是一个新的“序列最佳首部”，也可能不是。看下面这个序列：
10 8 9 4 6 3
从右向左浏览（从后向前），遇到 3 之后‘bestrun’数组有且只有一个元素：
0：3
继续浏览，6 比 3 大，‘bestrun’数组变成：
0:3
1:6

4 不比 6 大，但它比 3 大，所以数组变为：

0:3
1:4

9 使数组增大为：

0:3
1:4
2:9

8 同 4 使数组变成：

0:3
1:4
2:8

10 再次扩增数组为：

0:3
1:4
2:8
3:10

于是结果出来了：4（数组中有 4 个元素）。

因为‘bestrun’数组可能比处理过的序列长度增长得慢，这个算法可能比上一个运行得要快。实际上是增速很多。代码如下：

```
#include <stdio.h>
#define MAXN 200000
main () {
FILE *in, *out;
long num[MAXN], bestrun[MAXN];
long n, i, j, highestrun = 0;
in = fopen (″input.txt″, ″r″);
out = fopen (″output.txt″, ″w″);
fscanf(in, ″%ld″, &n);
for (i = 0; i < n; i++) fscanf(in, ″%ld″, &num[i]);
bestrun[0] = num[n-1];
highestrun = 1;
for (i = n-1-1; i >= 0; i--) {
if (num[i] < bestrun[0]) {
bestrun[0] = num[i];
continue;
}
for (j = highestrun - 1; j >= 0; j--) {
if (num[i] > bestrun[j]) {
if (j == highestrun - 1 || num[i] < bestrun[j+1]){
```

```
21 bestrun[++j] = num[i];
22 if (j == highestrun) highestrun++;
23 break;
24 }
25 }
26 }
27 }
28 printf("best is %d\n", highestrun);
29 exit(0);
30 }
```

同样地，11-12 行是初始化。14-17 行是对一个新的最小的元素优化。它们可以放在 26 行之后。通常，这些行只能影响输入以最坏方式分类的这种最糟糕情况的算法。

18-26 行查找‘bestrun’列表并含括了所有异常和棘手的情况（大于第一个元素？在中间插入？数组溢出？）。不用背诵，你应该可以马上将该代码写出来。速度很重要。下表比较了本算法和上一个算法，加看出当 N 变成 5 位数时本算法仍工作得不错：

N	orig	Improved
1000	0.080	0.030
2000	0.240	0.030
3000	0.550	0.050
4000	0.950	0.060
5000	1.450	0.080
6000	2.080	0.090
7000	2.990	0.110
8000	3.700	0.130
9000	4.700	0.140
10000	6.330	0.160
11000	7.350	0.170
20000		0.290
40000		0.570
60000	& ;nbs p;	0.910
80000		1.290
100000		2.220

Marcin Mika 指出你可以将此算法如下化简：

```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int best[SIZE]; // best[] holds values of the optimal sub-sequence
```

```
int
main (void) {
FILE *in = fopen ("input.txt", "r");
FILE *out = fopen ("output.txt", "w");
int i, n, k, x, sol = -1;

fscanf (in, "%d", &n); // N = how many integers to read in
for (i = 0; i < n; i++) {
best[i] = -1;
fscanf (in, "%d", &x);
for (k = 0; best[k] > x; k++)
;
best[k] = x;
sol = MAX (sol, k + 1);
}
printf ("best is %d\n", sol);
return 0;
}
```

为了不溢出，Tyler Lu 指出：
目前存在的算法使用线性查找来寻找合适的位置在‘bestrun’数组中插入一个整数。但是，因为辅助数组已分好类，我们可以用二叉查找（O（logN））。这样对于大型序列我们的运行时间就降下来了。下面是将上面代码修改后的解法：

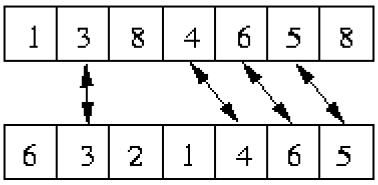
```
#include <stdio.h>
#define SIZE 200000
#define MAX(x,y) ((x)>(y)?(x):(y))

int best[SIZE]; // best[] holds values of the optimal sub-sequence

int
main (void) {
FILE *in = fopen ("input.txt", "r");
int i, n, k, x, sol;
int low, high;

fscanf (in, "%d", &n); // N = how many integers to read in
// read in the first integer
fscanf (in, "%d", &best[0]);
sol = 1;
for (i = 1; i < n; i++) {
best[i] = -1;
fscanf (in, "%d", &x);
if(x >= best[0]) {
k = 0;
best[0] = x;
}
else {
// use binary search instead
low = 0;
high = sol-1;
for(;;) {
k = (int) (low + high) / 2;
// go lower in the array
if(x > best[k] && x > best[k-1]) {
high = k - 1;
continue;
}
// go higher in the array
if(x < best[k] && x < best[k+1]) {
low = k + 1;
continue;
}
// check if right spot
if(x > best[k] && x < best[k-1])
best[k] = x;
if(x < best[k] && x > best[k+1])
best[++k] = x;
break;
}
}
sol = MAX (sol, k + 1);
}
printf ("best is %d\n", sol);
fclose(in);
return 0;
}
```

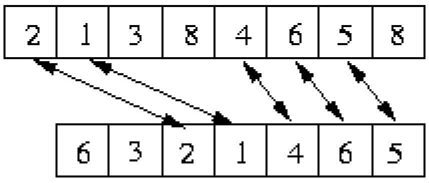
摘要

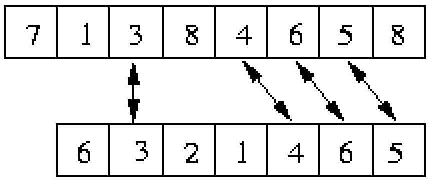
这些程序证明了动态规划背后的主要构想：在先前找到的算法的基础上建立更大的算法。这种逐步建立的解法通常能产生速度非常快的程序。
由于前面编程的挑战，主要的子问题是：在已知元素的右侧数字中找到最大的递减子序列（及其第一个值）。
这种方法可解决被称为“一维的”一类问题。

二维动态规划

有时可能会创建如此的多维问题：已知两个整数序列，求二者的最长公共子序列。这里，子问题就是较小序列（原来子序列末端部分）的最长公共子序列。首先，如果两序列之中的一个序列只含有一个元素，那解法就不重要了（或者所求元素在另一个序列中或者不在）。
考虑在第一个序列最后 i 个元素和在第二个元素最后 j 个元素中找到最长公共子序列的问题。有两种可能。第一个末尾部分的首元素可能在最大公共子序列中也可能不在。不包含第一个末尾部分首元素的最长公共序列就是第一个序列最后 i-1 个元素和第二个序列最后 j 个无素的最长公共子序列。第二个序列的末尾部分中某元素同第一个序列末尾部分的首元素一样且能找到在那些匹配元素之后元素的最长公共子序列，这导致了第二种可能。







伪代码

下面是本算法的伪代码：

```
# the tail of the second sequence is empty
1 for element = 1 to length1
2 length[element, length2+1] = 0


# the tail of the first sequence has one element
3 matchelem = 0
4 for element = length2 to 1
5 if list1[length1] = list2[element]
6 matchelem = 1
7 if matchelem = 0 then
8 length[length1,element] = 0
9 else
10 length[length1,e lement] = 1


# loop over the beginning of the tail of the first sequence
11 for loc = length1-1 to 1
```

```
maxlen = 0
for element = length2 to 1
# longest common subsequence doesn't include first element
if length[loc+1,element] > maxlen
maxlen = length[loc+1,element]
# longest common subsequence includes first element
if list1[loc] = list2[element]
&nbs p; ; length[loc+1,element+1]+1 > maxlen
maxlen = length[loc,element+1] + 1
length[loc,element] = maxlen
```

这个程序的运行时间是 O（NxM），N 和 M 分别为序列的长度。
本算法并不直接计算最长公共子序列。但是，如果给出长度矩阵，你就可以很快地确定子序列了：

```
location1 = 1
location2 = 1


while (length[location1,location2] != 0)
flag = False
for element = location2 to length2
if (list1[location1] = list2[element] AND
length[location1+1,element+1]+1 = length[location1,location2]
output (list1[location1],list2[element])
location2 = element + 1
flag = True
break for
location1 = location1 + 1
```

动态规划的技巧就是找到子问题来解决它。有时它需要多个参数：

振荡（？？？）序列是指第一部分递增且第二部分递减的序列。求一整数列的最长振荡序列（振荡可以是先增再减也可以是先减再增，但本问题只考虑先增加再减少的情况）。
这个问题的子问题是最长振荡序列和序列前缀的最长递减序列。

有时子问题被很好的隐藏起来了：
你刚赢得一场比赛，奖励是一份免费到加拿大的旅游。你必须乘飞机游玩，城市从东到西依次排列。另外，根据规则，你必须从最西侧的城市开始一直向东游览，直到你到达最东边的城市，再一直向西飞回到起点。另外每个城市参观一次（当然起点城市除外）。
已知城市顺序和乘坐的航班（你只能在某些城市间飞，因为你可以从 A 城飞到 B 城不意味着你可以向其它方向下），求你最多能参观多少个城市。
若使用动态规划应注意的是你的位置和你的方向，但重要的是你采用的路径（因

为在回程时你不能再次参观某个已参观过的城市），而且路径数不能太多，否则将无法解出（并存贮）所有子问题的答案。
但是，如果不是寻找上面描述的路径，而是用另一不同的方式，那么状态数将大大减少。想像一下，有两个旅行者从最西侧的城市出发。两旅行者轮流向东走，而旅行者走的下一站总是最西侧的城市，但是两旅行者不能在同一个城市，除非他们在第一个或最后一个城市。但是，其中一个旅行者只允许使用“相反的班次”，即他可以从 A 城飞到 B 城当且仅当有一架从 B 城飞往 A 城的航班飞机。从两个旅行者的路径中找到可以组合成一个环程旅行(用一个旅行者的正常路径飞到最东边的城市，然后再用另一个旅行者相反的路径返回最西侧的城市)并不太困难。而当旅行者 x 走了，你知道旅行者 y 除了他现在所在的城市外没参观过任何一个旅行者 x 东边的城市，否则 x 在 y 西侧时旅行者 y 已经走过一次了。于是两旅行者的路径就无法接在一起了。此算法能产生参观城市最大值的原因留作练习。

通过动态规划将问题求解

一般地，动态规划解法应用在那些可能会花费指数级时间的算法上，因此如果有界限问题在任意但输入较少的情况下太大以至于不能以指数时间级运行时，可考虑使用动态规划法。基本上，任何问题你只要考虑到用递归下降的方法，如果输入不超过 30 就可以试试是否有动态规划的解法存在。

寻找子问题

如上所述，找到子问题是进行动态规划的重要一步。你的目的是用少量的数据，如一个整数、一对整数、一个布尔数和一个整数等等，完整地描述出算法的状态。若不失败，子问题会是一个问题的“尾端”。也就是说，有一个进行递归下降的方法这样每一步你只能处理一小部分数据。举个例子，在飞行旅游的那道题中，你可从使用递归下降的方法找到完整的路径，那也意味着你得记住你的位置和你已参观过的城市(记城列表或布尔数组的形式)。这样动态规划要做的就太多了。但是在你向东飞的一对城市间使用递归给已知常数是需要递归的非常小的一部分数据。
如果路线很重要，除非路径很短，你不要使用动态规划。但是正如飞行旅行问题，路线也许不重要，就看你怎么看了。

示例题目

多边形游戏《1998 IOI》

想像一个均匀的 N 边形。结点处放入数字，边放入操作符‘+’或‘x’。第一次去掉一条边。然后，合并边，即计算出操作符两边结点的值，将其代替原来的边和结点，举例如下：

```
...-- 3 --+-- 5 --*-- 7 --...
...----- 8 ----*----- 7 --...
```

...-------- 56 -----------...

已知一作好标记的 N 边形，求最后计算出的最大值。

子集和《春季 98 USACO》

从 1 到 N(N 在 1 到 39 之间)的连续整数的集合,可将其分成和相等的两个集合,例如，若 N=3，将集合《1，2，3》分成《3》和《1，2》两个子集合且其和相等。这计为单独划分（如，反序计为同样的划分，并不增加划分数）。
若 N=7，有 4 种方法将集合{1, 2, 3, ... 7} 分成和相等的两部分：
{1,6,7} - {2,3,4,5}
{2,5,7} - {1,3,4,6}
{3,4,7} - {1,2,5,6}
{1,2,4,7} - {3,5,6}

已知 N，打印将从 1 到 N 个整数的集合分成和相等的两个集合的各种方法。若没有则打印 0。

数字游戏《IOI96 ，maybe》

已知不超过 100 个整数（-32000_32000）列，两个对手依次轮流从数列的最左边或最后边去掉一个数。游戏到最后每个玩家的分是他去掉的数的和。已知一数列，假设第二位玩家玩得非常好，求第一位玩家赢时的最高分。