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:

```
 1 #include <stdio.h>


 2  long n, sequence[10000];
 3  main () {
 4       FILE *in, *out;
 5       int i;
 6       in = fopen ("input.txt", "r");
 7       out = fopen ("output.txt", "w");
 8       fscanf(in, "%ld", &n);
 9       for (i = 0; i < n; i++) fscanf(in, "%ld", &sequence[i]);
10       fprintf (out, "%d\n", check (0, 0, 999999));
11       exit (0);
12  }


13  check (start, nmatches, smallest) {
14      int better, i, best=nmatches;
15      for (i = start; i < n; i++) {
16          if (sequence[i] < smallest) {
17              better = check (i, nmatches+1, sequence[i]);
18              if (better > best) best = better;
19          }
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', so 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, since 9>4 and now there's a sequence of length 3 whose 'first' element is 9:

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

The '8' changes the array similar to the earlier case with the '4'; since 8<9, the 8 replaces the 9 for a subsequence of length 3 whose 'first' element is (now) 8:

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

The '10' extends the array again since 10 > 8 and now there's a longest subsequence of length 4 (10, 8, 4, 3 -- among others):

```
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:

```
 1  #include <stdio.h>
 2  #define MAXN 200000
 3  main () {
 4      FILE *in, *out;
 5      long num[MAXN], bestrun[MAXN];
 6      long n, i, j, highestrun = 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      bestrun[0] = num[n-1];
12      highestrun = 1;
13      for (i = n-1-1; i >= 0; i--) {
```

```
14        if (num[i] < bestrun[0]) {
15            bestrun[0] = num[i];
16            continue;
17        }
18        for (j = highestrun - 1; j >= 0; j--) {
19            if (num[i] > bestrun[j]) {
20                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  }
```

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 important ones that search the bestrun list; these lines 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".

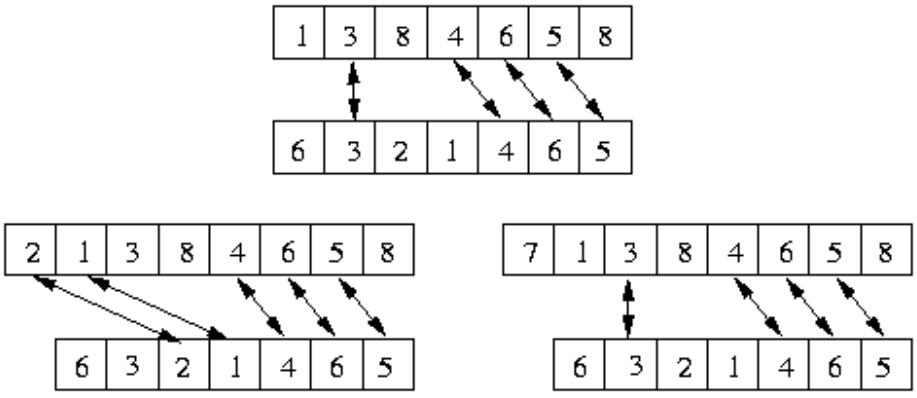
Two Dimensional DP

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
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     length[length1,element] = nmatchlen

    # loop over the beginning of the tail of the first sequence
 8   for loc = length1-1 to 1
 9     maxlen = 0
10     for element = length2 to 1
    # longest common subsequence doesn't include first element
11       if length[loc+1,element] > maxlen
12         maxlen = length[loc+1,element]
    # longest common subsequence includes first element
13       if list1[loc] = list2[element] &&
14                       length[loc+1,element+1]+1 > maxlen
15           maxlen = length[loc,element+1] + 1
16       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:

```
 1    location1 = 1
 2    location2 = 1


 3    while (length[location1,location2] != 0)
 4      flag = False
 5      for element = location2 to length2
 6        if (list1[location1] = list2[element] AND
 7           length[location1+1,element+1]+1 = length[location1,location2]
 8          output (list1[location1],list2[element])
 9          location2 = element + 1
10          flag = True
11          break for
12        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 travelers 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.

Back to USACO Training Gateway | Comment or Question