Indian National Olympiad in Informatics

Online Programming Contest, 2-3 October 2004

Related:
Overview
Current
Study Material
Online Judge
Archives

Advanced Division

Solution to Problem 1: Dividing Sequences

This problem illustrates a very general method than can be used solve a vast number of computational problems. We examine a solution to this problem first and then describe the principles behind this technique.

Let the input be a_1, a_2, ..., a_N. Let us define Best(i) to be the length of longest dividing sequence in a_1,a_2,...a_i that includes a_i.

For example in the sequence 2 3 4 8 16 9 14 18 with i = 6, Best(i) is 2 corresponding to the sequence 3 9 (Even though 2 4 8 16 is a dividing sequence within the first 6 elements, that does not count as Best(6) refers to the length of the longest dividing sequence among 2 3 4 8 16 9 that uses the the 6th element, in this case 9.)

Next, we express Best(i) in terms of Best(j) with j <i.

Suppose none of the elements a_1, a_2, ... a_{i-1} divide a_i then clearly Best(i) = 1.

Otherwise, Best(i) >= 2. Consider the longest dividing sequence among a_1 a_2 ... a_i that uses ai. Suppose, the previous element in this sequence is aj (with j <i) then clearly Best(i) = Best(j)+1. (Why?)

Further, for each k, with k <i, either a_k does not divide a_i or Best(k) + 1 ≤ Best(j) + 1 = Best(i). (Otherwise, a_k divides a_i and Best(k) + 1 > Best(j) + 1 = Best(i). So the longest dividing sequence ending at a_k extended with a_i has length Best(k) + 1 > Best(i) contradicting the maximality of Best(i).)

Thus, we see that

$$Best(1) = 1$$

and

$$Best(i) = \max \{ Best(j) + 1 \mid j < i \text{ and } a_j \text{ divides } a_i \}$$

(where we assume that MAX {} = 1)

This indicates a rather direct method to compute Best(i):

```
Best(i) {
       if (i=1) return (1)
       else {
            max=1;
            for (j=1 to i-1) {
                 if (a_j divides a_i) {
                      max = Maximum(max,Best(j)+1)
                      }
                 }
            return(max)
            }
      }
```

Once we have all the Best(i)'s, the required answer is the maximum among Best(i) for all 1 ≤ i ≤ n.

The problem with our description above is that it is very sllloowwwww. That is because, it makes too many recursive calls

```
to Best. To see this let us compute Best(i) for say i=1,2,3,4,5,6
(in that order) on the sequence 2 3 4 8 16 9

Best(1) --- returns 1 immediately.

Best(2) --- returns 1 after some computation, but does not make
            any recursive calls.

Best(3) --- Calls Best(1).

Best(4) --- Calls Best(1).
            Calls Best(3) which in turn calls Best(1).

Best(5) --- Calls Best(1)
            Calls Best(3) which in turn calls Best(1)
            Calls Best(4) which in turn calls Best(1)
                          and then calls Best(3) which in turn
                          calls Best(1).

Best(6) --- Calls Best(2).

We make repeated calls to Best on inputs on which the value has
already been calculated! (For instance Best(4) calls Best(3)
though Best(3) was already execcuted --- to compute the value of
Best(3). Much worse, Best(5) calls Best(3) and then calls Best(4)
which in turn calls Best(3). The fate of Best(1) is much
worse. When Best(5) is running, Best(1) is called once directly,
then once when Best(3) is called by Best(5) then once via
Best(4), and once when Best(4) calls Best(3) and Best(3) calls
Best(1)!. Thus Best(1) is called 4 times by Best(5).

This can get much worse: consider the sequence  2 4 8 ... 2^i.
Here
      Best(2) calls Best(1) once.
      Best(3) calls Best(1) twice (once directly and once via Best(2))
      Best(4) calls Best(1) four times (once directly, once via Best(2)
                                        and twice via Best(3))
      Best(5) calls Best(1) eight times
      Best(6) calls Best(1) sixteen times
      .
      .
      .
      Best(j) calls Best(1) 2^{j-2} times!!

And if j is 50 we make 2^48 calls to Best(1) and it will take a
few billion years for this computation to end.

The solution to this wasteful repetitive computation is
obvious. When we compute Best(j) we store it somewhere and pick
it up from there when it is needed later. A direct implementation
of this idea is as follows:

for i=1 to n
      StoredBest[i]=0;    /* whenever Best(j) is computed, store the value
                          /*   in StoredBest[j].

Best(i) {
       if (StoredBest[i] != 0)  /* Not 0, so Best(i) has already been computed.
            return(StoredBest[i]) /* so just lookup the array and return value.

       if (i=1) {
                 StoredBest[1]=1
                 return (1)
                }
       else {
            max=1;
            for (j=1 to i-1) {
                 if (a_j divides a_i) {
                      if (StoredBest[j] != 0)  /* Best(j) already computed
                           temp=StoredBest[j]
                      else                     /* Best(j) has to be computed
                           temp=Best(j);
                      max = Maximum(max,temp+1)
                      }
                 }
            StoredBest[i]=max      /* Now that we have computed the value
            return(max)            /* store it so that it can be reused.
            }
      }


Now, if we decide to use this program to directly compute Best(5)
then this is what happens:

Best(5) calls Best(1) which in turn returns 1 and sets StoredBest[1]=1.
Best(5) calls Best(3) which in turn gets the value of Best(1) from
                  StoredBest[1] and returns 2 and sets StoredBest[3]=2.
Best(5) calls Best(4) which in turn gets the values of Best(1) and
                  Best(3) from the array StoredBest and returns 3 and
                  sets StoredBest[4]=3.
Best(5) returns  4 and sets StoredBest[5]=4.
```

```
Thus, Best(1), Best(3), Best(4) are called exactly once each by Best(5).

Now, suppose we called Best(1), Best(2), ... Best(6) in that order:

1) Best(1) returns 1 and sets StoredBest[1]=1
2) Best(2) returns 1 and sets StoredBest[2]=1
3) Best(3) returns 2 and sets StoredBest[3]=2 (it uses StoredBest[1])
4) Best(4) returns 3 and sets StoredBest[4]=3 (it uses StoredBest[1] and
5) Best(5) returns 4 and sets StoredBest[5]=4 (it uses StoredBest[1],
                                                  StoredBest[3] and
                                                  StoredBest[4])
6) Best(6) returns 2 and sets StoredBest[6]=2 (it uses StoredBest[2]).

Thus, Best(j) is called only once for each j.

A final observation gives us a nonrecursive program --- in our
program to compute the longest dividing sequence we are going to
call Best(1), ...  Best(n) in that order and Best(i) only depends
on Best(j) where j <i.  Thus when Best(i) is called, the values
of all the Best(j) it needs are all available in the array
StoredBest!!! So the "else" clause that makes the recursive call
in the above routine will never be evoked.  So we can simply
eliminate the recursive calls and compute StoredBest[i] for all i
as follows:

              StoredBest[1]=1;

              for (i = 1 to n) {
                   max = 1
                   for (j=1 to i-1)  {
                       if (a_j divides a_i){
                            max=Maximum(max,StoredBest[j]+1)
                       }
                   }
                   StoredBest[i]=max
              }

Clearly Best(i) uses roughly i steps and thus computing Best(1)
through Best(n) takes roughly n*n steps and the maximum of all
the entries in StoredBest can be computed in roughly n
steps. Thus overall we have a solution that takes about n*n + n,
which is roughly n*n steps.

The above technique is called dynamic programming and the key
steps involved in the use of this technique are:

1) Identify the right quantity to compute.

   We have to be careful about deciding what needs to be
   computed.  For instance, if we chose Best(i) to denote the
   length of the longest dividing sequence among a_1, ... a_i,
   then it is not easy to describe Best(i) in terms of Best(j)
   for j <i. The key to our success above was the choice of
   Best(i) --- to be the length of the longest dividing sequence
   among a1 a2 ... ai THAT USES ai.

2) Write out that recursive definition.  (This depends on the
   choice made in step 1 and in fact the choice of step 1 is to a
   great extent guided by the need to carry out step 2)

3) Determine the order of evaluation so that computation can be
   carried out without recursive calls.

   In the above example we realised that Best(i) depends only on
   Best(j) for j<i and thus by evaluating Best(i) in the order 1,
   2, ...n we managed to completely eliminate recursive calls.

In order to illustrate the ideas behind dynamic programming
better, we present a solution to the Siruseri Sports Stadium
problem (September 2004, Advanced Problem 1) using this
technique.

Recall that you are given (s_1,d_1), (s_2,d_2) ... (s_n,d_n)
where s_i is the starting date and d_i is the length of event i
and the task is to find the length of the longest "schedule" (
where a schedule is a sequence of the form i_1 i_2, ...  i_k
where s_i_{j+1} > s_i_j + d_i_j (i.e.  the j+1 event begins after
the jth event ends and thus there is no overlap)).

In the first step we compute the ending time e_i for each event i
and sort the events in that order. Let (S_1,E_1) (S_2,E_2)
... (S_n,E_n) be the sequence obtained when we order the events
by ending time E_i.  We use "event i" to denote the ith event in
this order.

Let Sched(i) be the length of the longest schedule whose last
event is (S_i, E_i).  Then,

        Sched(1)   = 1

        Sched(i)   =  Max { Sched(j) + 1 | j < i and E_j <S_i }
```

Any schedule ending with event i cannot use any event from i+1 onwards. So, in any schedule ending with i, the previous event is some j with j < i. Further, in any such schedule E_j <S_i. So Sched(i) >= Sched(j) + 1. And if the previous event in the longest schedule ending with i is j then Sched(i) = Sched(j) + 1 (Why?)

Once again we notice that Sched(i) only depends on Sched(j) for j < i and these can be computed in the order Sched(1), ... , Sched(n) without recursion as follows:

(Where we assume that (S_1,E_1) ... (S_n,E_n) are in ascending order of E_i's).

```
StoredSched[1]=1;
for (i = 2 to n ) {
       max=1
       for (j=1 to i-1) {
           if (E_j <S_i)  max=Maximum(max,StoredSched[j]+1)
          }
       StoredSched[i]=max
     }
```

Thus, computing StoredSched for all i takes time n*n and and sorting the events by E_i takes time n*logn or n*n depending on the sorting algorithm used and finally finding the maximum takes time proportional to n. So overall we have a algorithm that takes roughly n*n steps to solve this problem.

We will encounter more examples of dynamic programming as we go along.