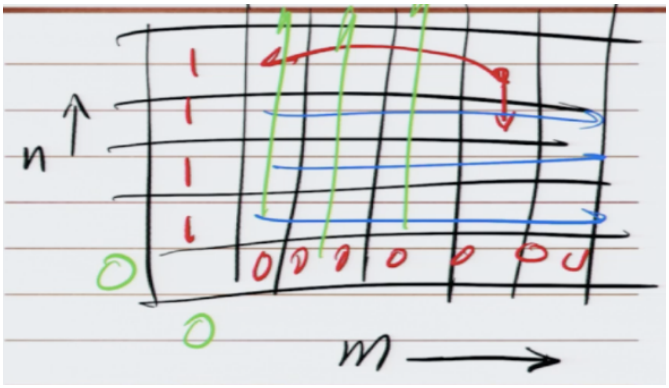
Discussion 6

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order: 1 = d1 < d2 < … < dn. Formulate the solution to this problem as a dynamic programming problem.

   COUNT(n,m) = number of ways to pay for an amount m using coins 1...n.

   Either you are using a coin denomination or not using it at all.
   COUNT(n,m) = COUNT(n-1, m) + COUNT(n, m-dn)
   + because we want to consider both cases to find the total number of ways.



   You need 1. n - 1, m and 2. same row, some m before it.
   row by row or col by col.
   Values.
   - We want to pay for the amount m, but we have no coins!!! There are 0 ways to pay for m.
   - We have n coins and want to pay for an amount of 0. There is 1 way (using 0 of each coin)

   Takes O(mn). Is this efficient? No! Because of the m term. m is a numerical value of the input. n is okay because it's an array of coin denominations (d1 through dn).
   Pseudo-polynomial.

   Solution:
   We will define COUNT(n, m) as the total number of ways to make change for amount m using coin denominations 1 to n.

   The recurrence formula will be:
   COUNT(n,m) = COUNT(n-1, m) + COUNT(n, m-dn)

   Note: COUNT(n-1, m) is the number of ways to make change for amount m without using coin denomination n. And COUNT(n, m-dn) represents the number of ways to make change for amount m using at least one coin of denomination n.

   Initialization:
   COUNT (i,0) = 1 for 0 < i ≤ n since there is a way to pay the amount 0 (by paying 0 of all coin types)
   COUNT(0,j) = 0 for 0 < j ≤ m since there is no way to pay a non-zero amount without any coins

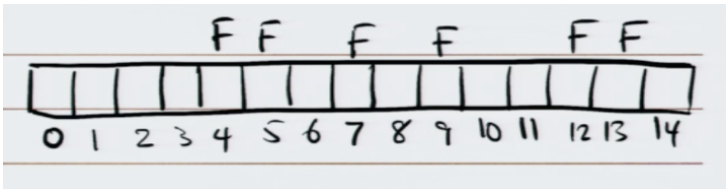
   Bottom up pass:

```
For i= 1 to n
    For j= 1 to m
        COUNT(n,m) = COUNT(n-1, m)
        If (m - dn ≥ 0 )
            COUNT(n,m) = COUNT(n,m) + COUNT(n, m-dn)
    Endfor
Endfor
```

The total count will be at COUNT(n,m)

This will take O(mn) which is pseudo polynomial since m is the numerical value of an input term.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next n days marked with those days when you get free dinner, and those days on which you must acquire dinner on your own. On any given day, you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.

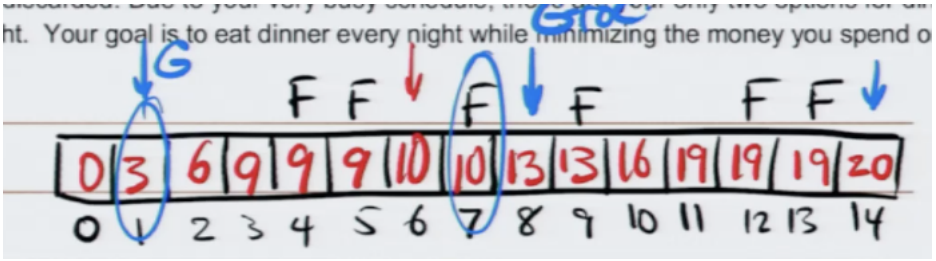


Subproblems - plan for dinner for day 1 to some day i that gives us the minimum total cost.
Define recurrence

$$OPT(i) = \begin{cases} OPT(i-1) & \text{if there is free food on day } i \\ \text{Otherwise, } \min(OPT(i-1) + 3,\ OPT(i-7) + 10) & \end{cases}$$

Bottom up:

Top down:



Start at the end, n. How did I get this 20? Where is it coming from?
Is there free food on this day (in which case it'd be from the day before)? No…
Is it OPT(i - 1) + 3 or **OPT(i - 7) + 10**?
14 - 7 = index 7 = 10. Then you find you went grocery shopping cause 10 + 10 = 20.
Mark calendar at index 7 as "grocery shop" and from i - 6 to i as "eat at home"

Then ask, how did I get to this 10?
Is there **free food** on this day (in which case it'd be from the day before)? Yes!
Mark as "free food"
Go to previous day at index 6.

Then ask, how did I get to this 10?
Is there free food on this day (in which case it'd be from the day before)? No…
Is it OPT(i - 1) + 3 or **OPT(i - 7) + 10**?
6 - 7 = index -1.. ~~~Careful here~~~ the same way you have to initialize the first few indices of the array so that you don't get into array indices that are negative, when you are going back, you have to make sure that you don't overshoot and go grocery shopping on a negative day:
The first possible day you could go grocery shopping is on day 1. So make that as "grocery shop" and from i + 1 to i as "eat from home"

Takes linear time.

Solution:
We will define OPT(i) as the minimum cost of dinner for days 1 to i.
The recurrence formula will be:

$$OPT(i) = \begin{cases} OPT(i-1) & \text{if there is free food on day } i \\ \text{Otherwise, } \min(OPT(i-1) + 3,\ OPT(i-7) + 10) \end{cases}$$

Initialization:
We need to initialize OPT(0..6). These are trivial problems to solve

Bottom up pass:

```
For i= 7 to n
    If Free(i) then
        OPT(i) = OPT(i-1)
    Else
        OPT(i) = Min( OPT(i-1) + 3 , OPT(i-7) + 10 )
    Endif
Endfor
```

The minimum cost of dinner will be at OPT(n).

This will take O(n) which is polynomial if we assume that the input consists of an array of size n called Free() where Free(i) is true when there is free food on day i, and false otherwise

To be able to determine the dinner schedule, we will go top down:

```
i = n
While i > 0
    If Free(i) then
        Mark up the calendar with “free food” on day i
        i = i - 1
    Else if OPT(i - 1) + 3 < OPT(i - 7) + 10 then
        Mark up the calendar with “cafeteria” on day i
        i = i - 1
    Else:
        Mark up the calendar with “go grocery shopping” on day i - 6
        Mark up the calendar with “eat from home” on days i - 5 to i
        i = i - 7
    Endif
EndWhile
```

Note: on the day we “go grocery shopping”, we also “eat at home”.
The top down pass will also take O(n) time.So the whole solution runs in O(n) time

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.
    a. In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))? Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define the notations you use.

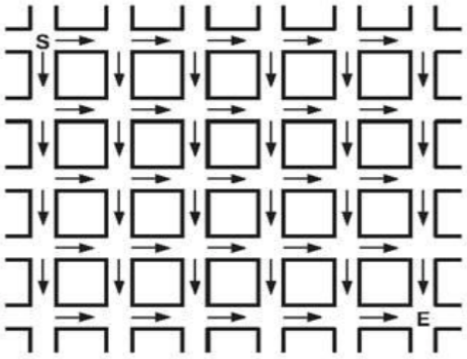


Figure A.

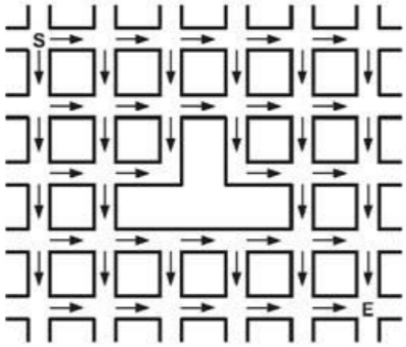


Figure B.

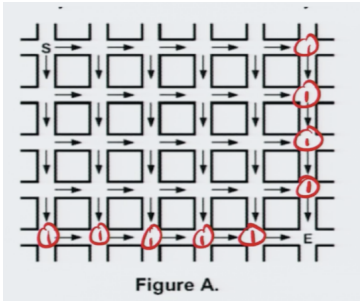


Figure A.

From any of these points, you can just go one direction.

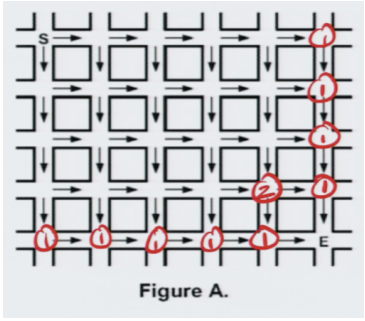


Figure A.

From here you can go two directions (right or down)

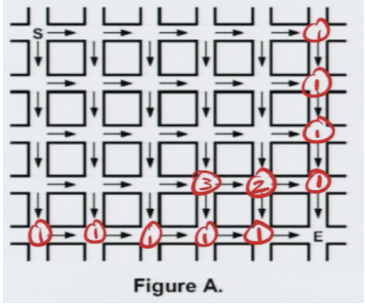


Figure A.

3. If you go to the right you have 2 ways, if you go down you have 1 way.

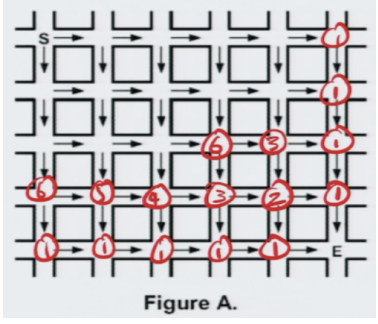


Figure A.

Following this pattern...the recurrence formula becomes clear, and also what we need to initialise.

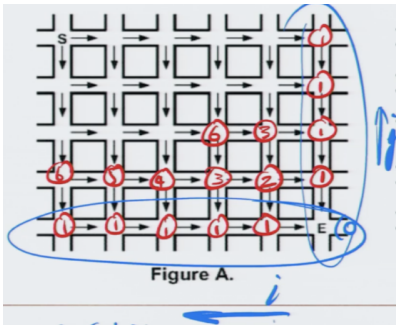


Figure A.

Need to initialize these!

OPT(i, j) = number of ways to go from (i, j) to E.
OPT(i, j) = OPT(i - 1, j) + OPT(i, j - 1)
Do we need a top down pass? No, because we are only interested in the value, the number of ways.
Takes O(nm). Is this efficient? It depends on how the input is presented to us. A graph with streets and connection, or a list of all the intersections (e.g. m arrays of length n)? Then yes. Or just given two numbers n and m? Then no.

Solution:
Let's say E is at coordinates (0,0) and s is at coordinates (n,m).
We will define COUNT(n, m) as the total number of ways to go from coordinates (n,m) to (0,0).

The recurrence formula will be:
COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)

Initialization:
COUNT (i,0) = 1 for $0<i\leq n$ since there is only one way to go (horizontally) from (i,0) to (0,0)
COUNT(0,j) = 1 for $0<j\leq n$ since there is only one way to go (vertically) from (0,j) to (0,0)

Bottom up pass:
For i= 1 to n
    For j= 1 to m
        COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)
    Endfor
Endfor

The total count will be at COUNT(n,m)
This will take O(mn) which is pseudo-polynomial if we assume that the input only consists of the coordinates n and m. If the input consists of the 2D array of all intersections, then the run time will be polynomial.

b. Repeat this process with Figure B; be wary of dead ends.

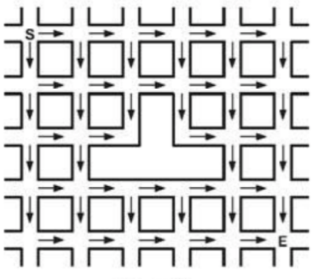


Figure B.

The recurrence formula is NOT affected at any of these intersections in blue. ONLY those in red.

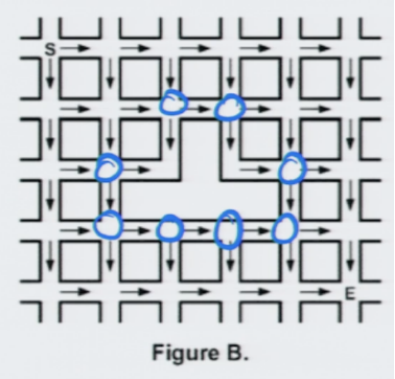


Figure B.

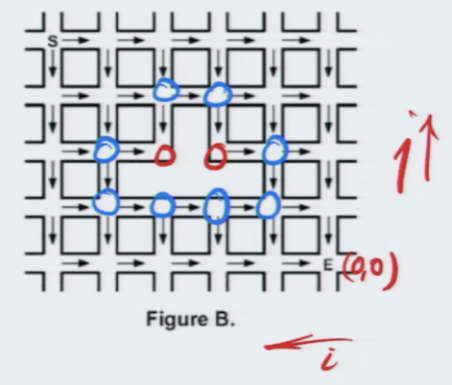


Figure B.

(2,2) and (3,2) ← recurrence is diff

Can you just initialize these then jump into a clean loop? Nope, then the loop will overwrite these terms! Deal with the following special cases in your recurrence:

(2,2) → OPT(i, j) = OPT(i - 1, j) (can only go to the right)

(3,2) → OPT(i, j) = 0 (can't go right or down)

We can use the same approach and recurrence formula as in part a, except that we need to apply special recurrence formulae to the intersections that are affected namely (2, 2) and (3, 2). So, the implementation will be modified like this:

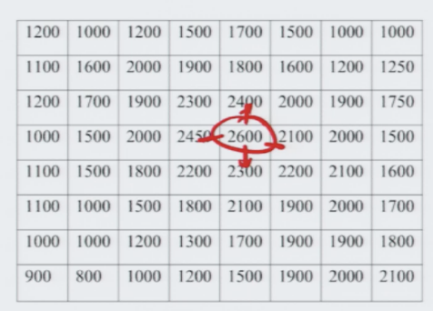
```
Bottom up pass:
For i = 1 to n
    For j = 1 to m
        If (n=2 and m=2) then
            COUNT(i,j) = COUNT(i-1, j)
        Elseif (n=3 and m=2) then
            COUNT(i,j) = 0
         Else
            COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)
        Endif
    Endfor
Endfor
```

4. Assume you want to ski down the mountain. You want the total length of your run to be as long as possible, but you can only go down, i.e. you can only ski from a higher position to a lower position. The height of the mountain is represented by an n X n matrix A. A[i][j] is the height of the mountain at position (i,j). At position (i,j), you can potentially ski to four adjacent positions (i-1,j) (i,j-1), (i,j+1), and (i+1,j) (only if the adjacent position is lower than current position). Movements in any of the four directions will add 1 unit to the length of your run. Provide a dynamic programming solution to find the longest possible downhill ski path starting at any location within the given n by n grid.

n

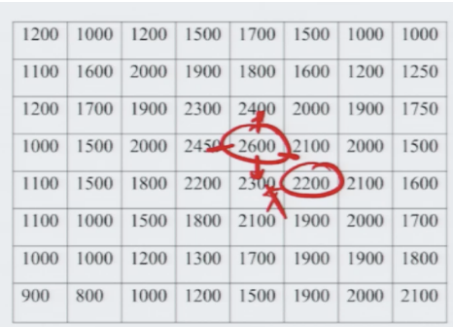
1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

OPT(i,j) = the length of the longest path starting from (i, j). (Or you could do length of the longest path that ends at (i,j))



1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

Can go any way at a peak.



1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

Can't go left because it's uphill.

For which nodes do we actually KNOW the longest path from? The local minimums. Look for all local minimums and set OPT(i, j) = 0. From there you want to go in the order of elevation. Sort the points in the order of elevation.

Sort takes n^2 logn^2 = O(n^2 logn). Can you get rid of the logn?

Substitute topological order for the sort. It's a directed graph with no cycles - you can't go downhill in a cycle - it's a DAG. Find topological ordering in linear time (n^2 nodes and n^2 edges then n^2 time).

5. Imagine starting with the given decimal number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence
   32492 → **3249** → **324** → 24 → **4**
   since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.
   32492 → **3249** → 249 → **49** → **9**
   also gives you 3 perfect squares, viz. 3249, 49, and 9. Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number a1, a2,.... ad. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

Chapter 6

Solved Exercise 1
Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily traveled stretch of road that runs west-east for M miles. The possible sites for billboards are given by numbers x1, x2, ... , xn, each in the interval [0,M] (specifying their position along the - highway, measured in miles from its western end). If you place a billboard at location xi, you receive a revenue of ri > 0.

Regulations imposed by the county's Highway Department require that no two of the billboards be within less than or equal to 5 miles of each other. You'd like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.
Example. Suppose M = 20, n = 4,

{x1, x2, x3, x4} ={6, 7, 12, 14},
and
{r1, r2, r3, r4} = {5, 6, 5, 1}.

Then the optimal solution would be to place billboards at x1 and x3, for a total revenue of 10.

Give an algorithm that takes an instance of this problem as input and returns the maximum total revenue that can be obtained from any valid subset of sites. The running time of the algorithm should be polynomial in n.

Solution

We can naturally apply dynamic programming to this problem if we reason as follows. Consider an optimal solution for a given input instance; in this solution, we either place a billboard at site xn or not. If we don't, the optimal solution on sites x1, ... ,xn is really the same as the optimal solution on sites x1, . . . , xn-1; if we do, then we should eliminate xn and all other sites that are within 5 miles of it, and find an optimal solution on what's left. The same reasoning applies when we're looking at the problem defined by just the first j sites, x1, . . . ,xj we either include xj in the optimal solution or we don't, with the same consequences.

Let's define some notation to help express this. For a site xj, we let e(j) denote the easternmost site xi that is more than 5 miles from xj. Since sites are numbered west to east, this means that the sites x1, x2….xe(j) are still valid options once we've chosen to place a billboard at xi, but the sites xe(j)+1, . . . ,xj-1 are not.

Now, our reasoning above justifies the following recurrence. If we let OPT(j) denote the revenue from the optimal subset of sites among x1……xj, then we have

OPT(j) = max(rj + OPT(e(j)), OPT(j - 1)).

We now have most of the ingredients we need for a dynamic programming algorithm. First, we have a set of n subproblems, consisting of the first j sites for j = 0, 1, 2, ... , n. Second, we have a recurrence that lets us build up the solutions to subproblems, given by OPT(j) = max(rj + OPT(e(j)), OPT(j - 1)). To turn this into an algorithm, we just need to define an array M that will store the OPT values and throw a loop around the recurrence that builds up the values M[j] in order of increasing j.

```
Initialize M[0] = 0 and M[1] = r1
For j = 2,3,...,n:
    Compute M[j] using the recurrence
Endfor
Return M[n]
```

As with all the dynamic programming algorithms we've seen in this chapter, an optimal set of billboards can be found by tracing back through the values in array M.

Given the values e(j) for all j, the running time of the algorithm is O(n), since each iteration of the loop takes constant time. We can also compute all e(j) values in O(n) time as follows. For each site location xi, we define xi’ = xi - 5. We then merge the sorted list x1, . . . , xn with the sorted list x1’, ... , xn’ in linear time, as we saw how to do in Chapter 2. We now scan through this merged list; when we get to the entry xj’, we know that anything from this point onward to xj cannot be chosen together with xj (since it's within 5 miles), and so we simply define e(j) to be the largest value of i for which we've seen xi in our scan.

Here's a final observation on this problem. Clearly, the solution looks very much like that of the Weighted Interval Scheduling Problem, and there's a fundamental reason for that. In fact, our billboard placement problem can be directly encoded as an instance of Weighted Interval Scheduling, as follows. Suppose that for each site xi, we define an interval with endpoints [xi - 5, xi] and weight ri. Then, given any non overlapping set of intervals, the corresponding set of sites has the property that no two lie within 5 miles of each other. Conversely, given any such set of sites (no two within 5 miles), the intervals associated with them will be nonoverlapping. Thus the collections of nonoverlapping intervals correspond precisely to the set of valid billboard placements, and so dropping the set of intervals we've just defined (with their weights) into an algorithm for Weighted Interval Scheduling will yield the desired solution.

Solved Exercise 2

Through some friends of friends, you end up on a consulting visit to the cutting-edge biotech firm Clones 'R' Us (CRU). At first you're not sure how your algorithmic background will be of any help to them, but you soon find yourself called upon to help two identical-looking software engineers tackle a perplexing problem.

The problem they are currently working on is based on the concatenation of sequences of genetic material. If X and Y are each strings over a fixed alphabet S, then XY denotes the string obtained by *concatenating* them- writing X followed by Y. CRU has identified a *target sequence* A of genetic material, consisting of m symbols, and they want to produce a sequence that is as similar to A as possible. For this purpose, they have a library £, consisting of k (shorter) sequences, each of length at most n. They can cheaply produce any sequence consisting of copies of the strings in £ concatenated together (with repetitions allowed).

Thus we say that a *concatenation* over £ is any sequence of the form B1B2….Be, where each Bi belongs the set £. (Again, repetitions are allowed, so Bi and Bj could be the same string in £, for different values of i and j.) The problem is to find a concatenation over {Bi} for which the sequence alignment cost is as small as possible. (For the purpose of computing the sequence alignment cost, you may assume that you are given a gap cost δand a mismatch cost α_{pq} for each pair p,q ∈ S.) Give a polynomial-time algorithm for this problem.

Solution

This problem is vaguely reminiscent of Segmented Least Squares: we have a long sequence of "data" (the string A) that we want to "fit" with shorter segments (the strings in £).

If we wanted to pursue this analogy, we could search for a solution as follows. Let B = B1B2...Be denote a concatenation over £ that aligns as well as possible with the given string A. (That is, B is an optimal solution to the input instance.) Consider an optimal alignment M of A with B, let t be the first position in A that is matched with some symbol in Be. and let Ae denote the substring of A from position t to the end. (See Figure 6.27 for an illustration of this with e= 3.)

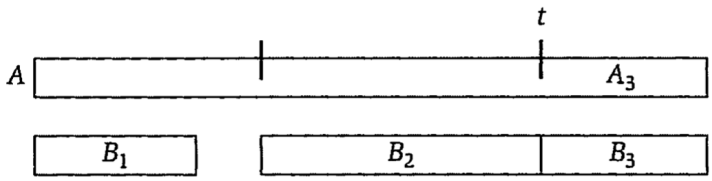


Figure 6.27 In the optimal concatentation of strings to align with A, there is a final string (B_3 in the figure) that aligns with a substring of A (A_3 in the figure) that extends from some position t to the end.

Now, the point is that in this optimal alignment M, the substring Ae is optimally aligned with Be: indeed, if there were a way to better align Ae with Be. we could substitute it for the portion of M that aligns Ae with Be and obtain a better overall alignment of A with B.

This tells us that we can look at the optimal solution as follows. There's some final piece of Ae that is aligned with one of the strings in £, and for this piece all we're doing is finding the string in £ that aligns with it as well as possible. Having found this optimal alignment for Ae, we can break it off and continue to find the optimal solution for the remainder of A.

Thinking about the problem this way doesn't tell us exactly how to proceed-we don't know how long Ae is supposed to be, or which string in £ it should be aligned with. But this is the kind of thing we can search over in a dynamic programming algorithm. Essentially, we're in about the same spot we were in with the Segmented Least Squares Problem: there we knew that we had to break off some final subsequence of the input points, fit them as well as possible with one line, and then iterate on the remaining input points.

So let's set up things to make the search for Ae possible. First, let A[x : y] denote the substring of A consisting of its symbols from position x to position y, inclusive. Let c(x, y) denote the cost of the optimal alignment of A [x : y] with any string in £. (That is, we search over each string in £ and find the one that aligns best with A[x : y].) Let OPT(j) denote the alignment cost of the optimal solution on the string A[1 : j]. The argument above says that an optimal solution on A[1 : j] consists of identifying a final "segment boundary" t < j, finding the optimal alignment of A[t : j] with a single string in £, and iterating on A[1: t - 1]. The cost of this alignment of A[t : j] is just c(t, j), and the cost of aligning with what's left is just OPT(t - 1). This suggests that our subproblems fit together very nicely, and it justifies the following recurrence.

(6.37) $OPT(j) = min_{t<j} c(t, j) + OPT(t - 1)\ for\ j \geq 1,\ and\ OPT(0) = 0.$

The full algorithm consists of first computing the quantities c(t, j), for t < j, and then building up the values OPT(j) in order of increasing j. We hold these values in an array M.

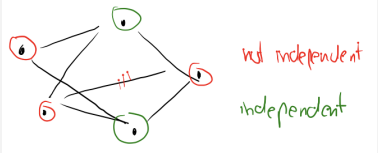
Set M[0] = 0
For all pairs $1 \leq t \leq j \leq m$
    Compute the cost c(t, j) as follows:
    For each string B ∈ £
        Compute the optimal alignment of B with A[t : j]
    Endfor
    Choose the B that achieves the best alignment, and use this alignment cost as c(t, j)
Endfor
For j =1, 2, ... , n
    Use the recurrence (6.37) to compute M[j]
Endfor
Return M[n]

As usual, we can get a concatenation that achieves it by tracing back over the array of OPT values.

Let's consider the running time of this algorithm. First, there are O(m^2) values c(t, j) that need to be computed. For each, we try each string of the k strings B ∈ £, and compute the optimal alignment of B with A[t : j] in time O(n*(j - t)) = O(mn). Thus the total time to compute all c(t, j) values is O(k m^3 n).

Thls dominates the time to compute all OPT values: Computing OPT(j) uses the recurrence in (6.37), and this takes O(m) time to compute the minimum. Summing this over all choices of j = 1, 2, ... , m, we get O(m^2) time for this portion of the algorithm.

Exercise 1
Let G = (V, E) be an undirected graph with n nodes. Recall that a subset of the nodes is called an independent set if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.



Call a graph G = (V, E) a path if its nodes can be written as v1, v2,....,vn with an edge between vi and vj if and only if the numbers i and j differ by exactly 1. With each node vi, we associate a positive integer weight wi.

Consider, for example, the five-node path drawn in Figure 6.28.

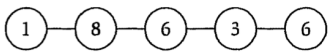


Figure 6.28 A paths with weights on the nodes. The maximum weight of an independent set is 14.

The weights are the numbers drawn inside the nodes. The goal in this question is to solve the following problem:
Find an independent set in a path G whose total weight is as large as possible.

a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.
The "heaviest-first" greedy algorithm
Start with S equal to the empty set
While some node remains in G
    Pick a node vi of maximum weight
    Add vi to S
    Delete vi and its neighbors from G
Endwhile
Return S

Contradiction: 10 -- 11 -- 10 -- 1. This would choose 11 and 1 = 12, rather than 10 and 10 = 20.

b) Give an example to show that the following algorithm also does not always find an independent set of maximum total weight..
Let S1 be the set of all vi where i is an odd number
Let S2 be the set of all vi where i is an even number
(Note that S1 and S2 are both independent sets)
Determine which of S1 or S2 has greater total weight, and return this one

Contradiction: 10 -- 1 -- 0 -- 50. The odd set contains 10 and 0. The even set contains 1 and 50. However, the optimal set is neither, it would contain one from the odd, and one from the even set, to get 50 + 10 = 60.

c) Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

If we have nodes N1 …. Nn, and weights w1 …. wn, then let's look at the last node, Nn. Either this node will belong to the independent set of maximum total weight, or it will not. If node Nn **does not belong** to the optimal solution Sn, then we can look for the optimal solution containing nodes 1….n - 1, or OPT(n - 1). If node Nn **belongs** to the optimal solution Sn, then we know its immediate neighbor will not belong. Thus, we'd look for: wn + OPT(n - 2).

OPT(j) = the maximum total weight of the independent set with nodes 1….j.

OPT(j) = max(OPT(j - 1), wj + OPT(j - 2))

To turn this into an algorithm, we can define an array M that will store the OPT values.

```
M[0] = 0
M[1] = w1
For j in 2 to n:
    M[j] = max(OPT(j - 1), wj + OPT(j - 2))
Endfor
return M[n]
```

This bottom up approach gives us the maximum total weight of the independent set.

To find the independent set of Sn, we can use a top down approach to trace back through the computations of the max operator.

Time complexity: Since we spend constant time per iteration, over n iterations, the total running time is O(n).

Exercise 2

Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are low-stress (e.g., setting up a Website for a class at the local elementary school) and those that are high-stress (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of ei > 0 dollars; if you select a high-stress job, you get a revenue of hi > 0 dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week i - 1; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week i - 1.

So, given a sequence of n weeks, a plan is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week i > 1, then "none" has to be chosen for week i - 1. (It's okay to choose a high-stress job in week 1.) The value of the plan is determined in the natural way: for each i, you add ei to the value if

you choose "low-stress" in week i, and you add h1 to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values l1, l2, ... , ln and h1, h2, ..., hm find a plan of maximum value. (Such a plan will be called optimal.)

Example. Suppose n = 4, and the values of l1 and hi are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be 0 + 50 + 10 + 10 = 70.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

For iterations i = 1 to n
  If h(i+1) > ℓi + ℓi+1 then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i + 1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+ 1
  Endif
End

To avoid problems with overflowing array bounds, we define hi= ℓi =0 when i > n.

In your example, say what the correct answer is and also what the above algorithm finds.

This algorithm wouldn't work in the following example:

ℓ 10 1 4 20

h 10 50 100 5

This would choose the high stress job of 50 of week 2, since 50 > 10 + 1. But then it'd have to skip the high-stress job of higher value in week 3!

Our solution would give us: 50 + 20 = 70

The optimal solution gives: 10 + 100 + 20 = 130

(b) Give an efficient algorithm that takes values for ℓ1, ℓ2 , . . . , ℓn and h1, h2 , . . . , hn and returns the value of an optimal plan.

Let's look at the last job, jn. The choices are either that ℓn belongs to the optimal solution, or hn belongs to the optimal solution. If ℓn belongs to the optimal solution, then we could look at ℓn + OPT(n - 1). If hn belongs to the optimal solution, then we could look at hn + OPT(n - 2).

Thus, based on this logic, let's define:

OPT(i) = the maximum value revenue achievable in the input instance restricted to weeks 1 through i.

Recurrence:

OPT(i) = max(ℓi + OPT(i - 1), hi + OPT(i - 2))

To turn this into an algorithm, we can define an array M that will store the OPT values.

M[0] = 0

M[1] =max(ℓ1, h1)

For i in 2 to n:

M[i] = max(ℓi + OPT(i - 1), hi + OPT(i - 2))

Endfor

Return M[n]

This takes O(n) time.

The actual sequence of jobs can be reconstructed by tracking back (top down) through the set of OPT values.

An alternate, but essentially equivalent, solution is as follows. We define the following sub-problems. Let $L(i)$ be the maximum revenue achievable in weeks 1 through i, given that you select a low-stress job in week i, and let $H(i)$ be the maximum revenue achievable in weeks 1 through i, given that you select a high-stress job in week i.

Again, the optimal solution for the input instance restricted to weeks 1 through i will select some job in week i. Now, if it selects a low-stress job in week i, it can select anything it wants in week $i-1$; and if it selects a high-stress job in week i, it has to sit out week $i-1$ but can select anything it wants in week $i-2$. Thus we have

$$L(i) = \ell_i + \max(L(i-1), H(i-1)),$$

$$H(i) = h_i + \max(L(i-2), H(i-2)).$$

The L and H values can be built up by invoking these recurrences for $i = 1, 2, \ldots, n$, with the initializations $L(1) = \ell_1$ and $H_1 = h_1$.

3. Let G= (V, E) be a directed graph with nodes v1, ... , vn. We say that G is an ordered graph if it has the following properties.

1) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (vi, vj) with i < j.
2) Each node except vn has at least one edge leaving it. That is, for every node vi, i = 1, 2, ... , n-1, there is at least one edge of the form (vi, vj). The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

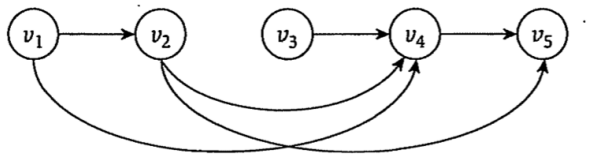


Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

Given an ordered graph G, find the length of the longest path that begins at v1 and ends at vn.

a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

Set w=v1

Set L=0

While there is an edge out of the node w

  Choose the edge (w, vj) for which j is as small as possible

  Set w=vj

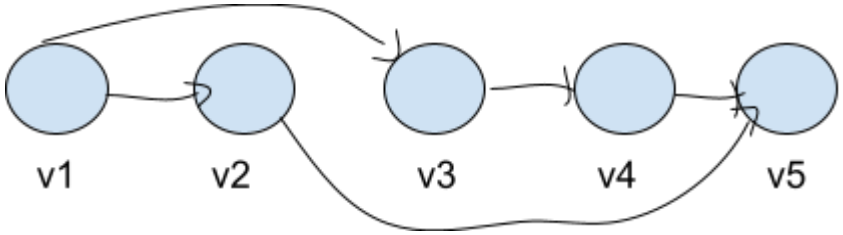
  Increase L by 1

end while

Return L as the length of the longest path

In your example, say what the correct answer is and also what the algorithm above finds.

We are looking for the longest path. In the following example, we would choose v2 (because 2 < 3), then v5. However, the longest path is from v1->v3->v4->v5.



b) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v1 and ends at vn. (Again, the length of a path is the number of edges in the path.)

Subproblem:

OPT(i) = the length of the longest path from v1 to vi.

*Not all nodes have a path from v1 to vi, thus we can use -∞ for that case. '-inf'

OPT(1) = 0 (the longest path from v1 to v1 is 0)

```
Long_path(n):
    Array M[1…..n]
    M[1] = 0
    For i = 2….n
        M = -∞     #if no incoming edges to reach v1, then set to -inf after skipping loop
        For all edges (j, i) in G then
            if M[j]  !=  -∞ and M < M[j] + 1 then    #i = 2 (node 2), j will be 1, the incoming edge
                M = M[j] + 1    #M = 1
            endif
        endfor
        M[i] = M
   endfor
   Return M[n] as the length of the longest path
```

The running time is O(n^2) if you assume that all edges entering a node i can be listed in O(n) time.

4. Suppose you're running a lightweight consulting business-just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an operating cost of Ni if you run the business out of NY; you'll incur an operating cost of Si if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month i + 1, then you incur a-fixed moving cost of M to switch base offices. Given a sequence of n months, a plan is a sequence of n locations- each one equal to either NY or SF-such that the ith location indicates the city in which you will be based in the ith month. The cost of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M, and sequences of operating costs N1, …., Nn and S1, …., Sn, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose n = 4, M = 10, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of 1+3+2+4+10=20, where the final term of 10 arises because you change locations once.

a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For i = 1 to n
    If Ni < Si then
        Output "NY in Month i"
    Else
        Output "SF in Month i"
End
```

In your example, say what the correct answer is and also what the algorithm above finds.

See the following example:
NY 3 4 3
SF 4 3 4
Here, our algorithm would ignore moving cost and move twice, NY → SF → NY, for a total cost of 3+3+3+10*2 = 29.
The optimal solution would be to stay in NY the whole time with a cost of 3+4+3 = 10.

b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times. Provide a brief explanation, saying why your example has this property.

NY 3 1000 3 1000
SF 1000 3 1000 3

Above is an example, where in the optimal solution you would move 3 times. This is because the cost of staying in the same location is higher than the moving cost.

c) Give an efficient algorithm that takes values for n, M, and sequences of operating costs N1, ... , Nn and S1, ... , Sn, and returns the cost of an optimal plan.

Subproblem:
OPT(i) = the minimum cost plan for months 1 to i

At month i, the company is either in the same location as the previous month (no moving cost incurred). Or the company moves location, because the moving cost + current month cost is less than the cost of staying at the same location.
This can happen for **either location** (it'll end up in one of them). So compute both then see which gives you the minimum.

OPTN(n) = Nn + min(OPTN(n - 1), M + OPTS(n - 1))
OPTS(n) = Sn + min(OPTS(n - 1), M + OPTN(n - 1))

OPTN(0) = 0
OPTS(0) = 0
For i = 1…...n:
    OPTN(i) = Ni + min(OPTN(i - 1), M + OPTS(i - 1))
    OPTS(i) = Si + min(OPTS(i - 1), M + OPTN(i - 1))
End
Return **min**(OPTN(n), OPTS(n))

This algorithm has n iterations, and each takes constant time. Thus the running time is O(n).

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the word segmentation problem-inferring likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight," or "meet ate ight," or any of a huge number of even less plausible alternatives). How could we automate this process?

A simple approach that is at least reasonably effective is to find a segmentation that simply maximizes the cumulative "quality" of its individual constituent words. Thus, suppose you are given a black box that, for any string of letters x = x1x2….xk, will return a number quality(x). This number can be either positive or negative; larger numbers correspond to more plausible English words. (So quality("me") would be positive, while quality("ght") would be negative.)

Given a long string of letters y = y1y2...yn, a segmentation of y is a partition of its letters into contiguous blocks of letters; each block corresponds to a word in the segmentation. The *total quality* of a segmentation is determined by adding up the qualities of each of its blocks. (So we'd get the right answer above provided that quality("meet") +quality("at")+ quality("eight") was greater than the total quality of any other segmentation of the string.)

Give an efficient algorithm that takes a string y and computes a segmentation of maximum total quality. (You can treat a single call to the black box computing quality(x) as a single computational step.)

(*A final note, not necessary for solving the problem*: To achieve better performance, word segmentation software in practice works with a more complex formulation of the problem-for example, incorporating the notion that solutions should not only be reasonable at the word level, but also form coherent phrases and sentences. If we consider the example "theyouthevent," there are at least three valid ways to segment this into common English words, but one constitutes a much more coherent phrase than the other two. If we think of this in the terminology of formal languages, this broader problem is like searching for a segmentation that also can be parsed well according to a grammar for the underlying language. But even with these additional criteria and constraints, dynamic programming approaches lie at the heart of a number of successful segmentation systems.)

Subproblem:
OPT(i) = the value the total maximum quality found in the string from 1….i

We can segment the string (y = y1y2...yi) at a certain index, j, where the recurrence is as follows:
OPT(i) = min j<=i { OPT(j -1) + quality(j….n) }

We prove the correctness of the above formula by induction on the index i. The base case is trivial, since there is only one word with one letter.

For the inductive step, assume that we know that the Opt function as written above finds the optimum solution for the indices less than i, and we want to show that the value $Opt(i)$ is the optimum cost of any segmentation for the prefix of y up to the i-th character. We consider the last word in the optimal segmentation of this prefix. Let's assume it starts at index $j \leq i$. Then according to our key observation above, the prefix containing only the first $j-1$ characters must also be optimal. But according to our induction hypothesis, $Opt(j)$ will yield us the value of the aforementioned optimal segmentation. Therefore the optimal cost $Opt(i)$ would be equal to $Opt(j)$ plus the cost of the last word.

But notice that our above recurrence exactly does this calculation for each possibility of the last word. Therefore our recurrence will correctly find the cost of the optimal segmentation.

```
OPT(0) = 0
M = [i……n]
for i in 1…..n:
   for j in 1…..i:
      M[j] =  OPT(j - 1)  + quality(j….i)
return M[n]
```

A solution starting from index 1 until n, will yield a quadratic algorithm. O(n^2)?

6. In a word processor, the goal of "pretty-printing" is to take text with a ragged right margin, like this,

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

and turn it into text whose right margin is as "even" as possible, like this.

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even." So suppose our text consists of a sequence of words, W = {w1, w2, ... , wn}. where wi consists of ci characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A formatting of W consists of a partition of the words in W into lines. In the words assigned to a single line, there should be a space after each word except the last; and so if wj, wj+1……..wk are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L.$$

We will call an assignment of words to a line valid if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line-that is, the number of spaces left at the right margin.

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines (including the last line) is minimized.

7. As a solved exercise in Chapter 5, we gave an algorithm with O(n log n) running time for the following problem. We're looldng at the price of a given stock over n consecutive days, numbered i = 1, 2, ... , n. For each day i, we have a price p(i) per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) We'd like to know: How should we choose a day i on which to buy the stock and a later day j > i on which to sell it, if we want to maximize the profit per share, p(j) - p(i)? (If there is no way to make money during the n days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days i and j in time O(n log n). But, in fact, it's possible to do better than this. Show how to find the optimal numbers i and j in time O(n).

8. The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the ith second, xi robots arrive. Based on remote sensing data, you know this sequence x1, x2, ... , Xn in advance.
- You have at your disposal an electromagnetic pulse (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function f(-) so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to f(j) robots.
- So specifically, ifitisused in the k1hsecond,andithasbeenjseconds since it was previously used, then it will destroy min(xk, f(j)) robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the jth second, then it is capable of destroying up to f(j) robots.

The problem. Given the data on robot arrivals x1, x2 , . . . , xn, and given the recharging function f(-), choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example. Suppose n = 4, and the values of xi and f(i) are given by the following table.

i	1	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

The best solution would be to activate the EMP in the 3rd and the 4th seconds. In the 3rd second, the EMP has gotten to charge for 3 seconds, and so it destroys min(10, 4) = 4 robots; In the 4th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys min(1,1)= 1 robot. This is a total of 5.

a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
Schedule-EMP(x_1, ..., x_n)
  Let j be the smallest number for which f(j) ≥ x_n
    (If no such j exists then set j = n)
  Activate the EMP in the n^th second
  If n − j ≥ 1 then
    Continue recursively on the input x_1, ..., x_{n−j}
      (i.e., invoke Schedule-EMP(x_1, ..., x_{n−j}))
```

In your example, say what the correct answer is and also what the algorithm above finds.

b) Give an efficient algorithm that takes the data on robot arrivals x1, x2 ,...., xn, and the recharging function f(-), and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with xi terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s1 terabytes, on the second day after a reboot, you can process s2 terabytes, and so on, up to sn; we assume s1 > s2 > s3 > · · · > sn > 0. (Of course, on day i you can only process up to xi terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data x1, x2, ... , xn for the next n days, and given the profile of your system as expressed by s1, s2, . . . , sn (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose n = 4, and the values of xi and si are given by the following table.

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process 8+1+2+1= 12; and other rebooting strategies give you less than 19 as well.)

a) Give an example of an instance with the following properties.
   - There is a "surplus" of data in the sense that xi> s1 for every i.
   - The optimal solution reboots the system at least twice.

   In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

b) Give an efficient algorithm that takes values for x1, x2 , . . . , xn and s1, s2 , . . . , sn and returns the total *number* of terabytes processed by an optimal solution.

10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete steps as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high- priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available. Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run ai > D steps .of the simulation if your job is on machine A, and bi > D steps of the simulation i f your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job. So, given a sequence of n minutes, a plan is specified by a choice of A, B, or "move" for each minute, with the property that choices A and

29. Let G= (V, E) be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight wij on each edge (i,j); and we will assume these weights satisfy the *triangle inequality* wik <= wij + wik. For a subset V' ⊆ V, we will use G[V'] to denote the subgraph (with edge weights) induced on the nodes in V'.
We are given a set X ⊆ V of k tenninals that must be connected by edges. We say that a Steiner tree on X is a set Z so that X ⊆ Z ⊆V, together with a spanning subtree T of G[Z]. The *weight* of the Steiner tree is the weight of the tree T.

Show that there is function f(·) and a polynomial function p(·) so that the problem of finding a minimum-weight Steiner tree on X can be solved in time O(f(k) · p(n)).