Explain in simple terms :

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,..., Un with an edge between vi and vj if and only if the numbers i and j differ by exactly 1. With each node v, we associate a positive integer weight wi. Consider, for example, the five-node path 1->8->6->3->6. 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 v of maximum weight Add v to S  
   Delete v and its neighbors from G

End while

Return S

Answer :

﻿ Consider the sequence of weights 2, 3, 2. The greedy algorithm will pick the middle

node, while the maximum weight independent set consists of the first and third.

-The problem is about finding a set of nodes in a line-like graph (called a "path") where no two nodes are next to each other, and the sum of their weights is as large as possible. However, finding such a set is generally difficult, except when the graph has a simple structure.

-In this case, the graph is a path and we can use a greedy algorithm to find the solution. The algorithm works as follows: start with an empty set, then repeatedly add the heaviest remaining node to the set, and remove its neighbors (the nodes that are next to it) from consideration.

-Continue until no nodes remain. However, the algorithm may not always find the set of maximum total weight. As an example, consider a path with nodes of weight 2, 3, and 2. The algorithm would pick the middle node (weight 3), but the maximum weight independent set consists of the first and third nodes (total weight 4).

Explain in simple terms :

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,..., Un with an edge between vi and vj if and only if the numbers i and j differ by exactly 1. With each node v, we associate a positive integer weight wi. Consider, for example, the five-node path 1->8->6->3->6. 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.

(b) Give an example to show that the following algorithm also does not always find an independent set of maximum total weight.

Let S 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 S or S2 has greater total weight, and return this one.

Answer : Consider the sequence of weights 3, 1, 2, 3. The given algorithm will pick the first and third nodes, while the maximum weight independent set consists of the first and fourth.

The problem is about finding a set of nodes in a line-like graph (called a "path") where no two nodes are next to each other, and the sum of their weights is as large as possible.

In this case, we are given an algorithm that works by dividing the nodes into two sets, one consisting of all the odd-numbered nodes and the other consisting of all the even-numbered nodes. Both of these sets are independent sets, meaning that no two nodes in either set are next to each other.

The algorithm then determines which of these two sets has a greater total weight and returns that set as the solution.

However, this algorithm may not always find the set of maximum total weight. As an example, consider a path with nodes of weight 3, 1, 2, and 3. The algorithm would pick the first and third nodes (total weight 5) from the odd-numbered set, but the maximum weight independent set consists of the first and fourth nodes (total weight 6).

﻿

Explain in simple terms :

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,..., Un with an edge between vi and vj if and only if the numbers i and j differ by exactly 1. With each node v, we associate a positive integer weight wi. Consider, for example, the five-node path 1->8->6->3->6. 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.

(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.

Answer :

Let S denote an independent set on {v1,v2,...vi }, and let Xi, denote its weight. Define X0 = 0 and note that X1= w1. Now, for i>1, either vi belongs to Si or it doesn't. In the first case, we know that vi-1 cannot belong to S1, and so Xi = wi + Xi-2. In the second case, Xi = Xi-1. Thus we have the recurrence

Xi = max(Xi-1, Wi + Xi−2).

We thus can compute the values of Xi, in increasing order from i = 1 to n. Xn, is the value we want, and we can compute Sn by tracing back through the computations of the max operator. Since we spend constant time per iteration, over n iterations, the total running time is O(n).

explain in simple terms

question :

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 Web site 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 li; > 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 li to the value if you choose "low-stress" in week i, and you add hi to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

answer :  
  
Let OPT(i) denote the maximum value revenue achievable in the input instance restricted to weeks 1 through i. The optimal solution for the input instance restricted to weeks 1 through i will select some job in week i, since it's not worth skipping all jobs there are no future high-stress jobs to prepare for. If it selects a low-stress job, it can behave optimally up to week i - 1, followed by this job, while if it selects a high-stress job, it can behave optimally up to week i-2, followed by this job. Thus we have justified the following  
recurrence.  
OPT(i) = max(li + OPT(i − 1), h2 + OPT(i − 2)).  
We can compute all OPT values by invoking this recurrence for i = 1, 2, ..., n, with the initialization OPT(1) = max(l1, h1). This takes constant time for each value of i, for a total time of O(n). As usual, the actual sequence of jobs can be reconstructed by tracing back 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 weck i-2. Thus we have  
-  
L(i) = li+ max(L(i − 1), H (i − 1)),  
H(i) – hi + max(L(i − 2), H(i − 2)).  
-  
The L and H values can be built up by invoking these recurrences for i = 1, 2,..., n, with the initializations L(1) = 1 and H1 = h1.

Explain in simple terms

Question : 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 N; if you run the business out of NY; you'll incur an operating cost of S; 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.

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.

Answer :  Suppose that M = 10, {N1, N2, N3} = {1, 4, 1}, and {S1, S2, S3} = {20, 1, 20}. Then the optimal plan would be [NY, NY, NY], while this greedy algorithm would return [NY, SF, NY].

Explain in simple terms

Question : 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 N; if you run the business out of NY; you'll incur an operating cost of S; 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.

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.

Answer :  Suppose that M 10, {N1, N2, N3, N1} = {1, 100, 1, 100}, and {S1, S2, S3, S4} = {100, 1, 100, 1}.

Explanation: The plan [NY, SF, NY, SF] has cost 34, and it moves three times. Any other plan pays at least 100, and so is not optimal.

Explain in simple terms

Question : 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 N; if you run the business out of NY; you'll incur an operating cost of S; 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.

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.

Answer :

The basic observation is: The optimal plan either ends in NY, or in SF. If it ends in NY, it will pay Nn plus one of the following two quantities:  
• The cost of the optimal plan on n − 1 months, ending in NY, or  
• The cost of the optimal plan on n − 1 months, ending in SF, plus a moving cost of M. An analogous observation holds if the optimal plan ends in SF. Thus, if OPTN (j) denotes the minimum cost of a plan on months 1,...,j ending in NY, and OPTS(1) denotes the minimum cost of a plan on months 1,...,j ending in SF, then  
OPTN(n) =  Nn+ min(OPTÑ(n − 1), M + OPTs(n − 1))  
OPTs(n) = Sn + min(OPTs(n − 1), M + OPTn(n − 1))  
This can be translated directly into an algorithm:  
OPTn(0) = OPTs(0) = 0  
For 1,..., n  
OPTN(i) = Ni +min(OPTN(i − 1), M + OPTs(i − 1))

OPTS(i) =Si +min(OPTs(i − 1), M + OPTŃ (i − 1))  
End  
Return the smaller of OPT(n) and (PTs(n)  
The algorithm has n iterations, and each takes constant time. Thus the running time is O(n).

Rephrase the following :

The problem is analogous to the Subset Sum Problem. We will have subproblems analogous to the subproblems for that problem. The first idea is to consider subproblems using a subset of jobs {1,...,m}. As always we will order the jobs by increasing deadline, and we will assume that they are numbered this way, i.e., we have that d1<=d2<= ... < dn = D To solve the original problem we consider two cases: either the last job n is accepted or it is rejected. If job n is rejected, then the problem reduces to the subproblem using only the first n - 1 items. Now consider the case when job n is accepted. By part (a) we know that we may assume that job n will be scheduled last. In order to make sure that the machine can finish job n by its deadline D, all other jobs accepted by the schedule should be done by time D-tn. We will define subproblems so that this problem is one of our subproblems.

For a time 0 ≤ d ≤ D and m - 0,...,n let OPT(d, m) denote the maximum subset of requests in the set {1,..., m} that can be satisfied by the deadline d. What we mean is that in this subproblem the machine is no longer available after time d, so all requests either have to be scheduled to be done by deadline d, or should be rejected (even if the deadline di of the job is di > d). Now we have the following statement.

﻿(1)

If job m is not in the optimal solution OPT(d, m) then OPT(m, d) - OPT (m − 1, d).

If job m is in the optimal solution OPT(m, d) then OPT(m, d) = OPT(m - 1,d- tm)+1.

This suggests the following way to build up valucs for the subproblems.

Select-Jobs (n,D)

Array M[0 ...n, 0 D]

Array S[0...n, 0 ... D]

For d0,..., D

M[0, d] = 0

S[0, d] - phi

Endfor

For m=1,...,n

For d 0,..., D

If

Mm - 1,d> M [m − 1, d − tm]+1 then

M[m, d] = M[m — 1, d] S[m, d] = S[m- 1, d]

Else

M[m, d] = Mm - 1, d-tm] + 1

S[m, d] = S[m1, d − tm] U {m}

Endif

Endfor

Endfor

Return M[n, D] and S[n, D]

The correctness follows immediately from the statement (1). The running time of O(n2D) is also immediate from the for loops in the problem, there are two nested for loops for m and one for d. This means that the internal part of the loop gets invoked O(nD) time. The internal part of this for loop takes O(n) time, as we explicitly maintain the optimal solutions. The running time can be improved to O(nD) by not maintaining the S array, and only recovering the solution later, once the values are known.

Explain in simple words :

Question : ﻿ Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a rising trend, as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted P[1], P[2], . . ., P[n]. A rising trend in these prices is a subsequence of the prices P[i], P[i2],..., P[ik], for days i1 < i2 < ... < ik, so that

i1 = 1, and

• P[ij] < P[ij+1] for each j = 1, 2, ..., k-1.

Thus a rising trend is a subsequence of the days-beginning on the first day and not necessarily contiguous-so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose n = 7, and the sequence of prices is

10, 1, 2, 11, 3, 4, 12.

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

﻿Show that the following algorithm does not correctly return the length of the longest rising trend, by giving an instance on which it fails to return the correct answer.

Define i=1

L=1

For j = 2 to n

If P[j] > P[i] then

Set i=j.

Add 1 to L

Endif

Endfor

In your example, give the actual length of the longest rising trend, and say what the algorithm above returns.

Answer :

Consider the sequence 1,4,2,3. The greedy algorithm produces the rising trend 1,4 while the optimal solution is 1,2,3

Explain in simple words :

Question : ﻿ Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a rising trend, as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted P[1], P[2], . . ., P[n]. A rising trend in these prices is a subsequence of the prices P[i], P[i2],..., P[ik], for days i1 < i2 < ... < ik, so that

i1 = 1, and

• P[ij] < P[ij+1] for each j = 1, 2, ..., k-1.

Thus a rising trend is a subsequence of the days-beginning on the first day and not necessarily contiguous-so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose n = 7, and the sequence of prices is

10, 1, 2, 11, 3, 4, 12.

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend. Give an efficient algorithm that takes a sequence of prices P[1], P[2],..., P[n] and returns the length of the longest rising trend.

Answer :

﻿Let OPT(j) be the length of the longest increasing subsequence on the set P[j], P[j+ 1], Idots, P[n], including the element P[j]. Note that we can initialize OPT(n)=1 OPT(1) is the length of the longest rising trend, as desired.

Now, consider a solution achieving OPT(j). Its first element is P[j], and its next element is P[k] for some k > j for which P[k] > P[j]. From k onward, it is simply the longest increasing subsequence that starts at P[k]; in other words, this part of the sequence has length OPT(k), so including P[j], the full sequence has length 1+ OPT(k). We have thus justified the following recurrence.

OPT(j) = 1+ max k>j:P[k]>P[j] OPT(k).

The values of OPT can be built up in order of decreasing j, in time O(n−j) for iteration j, leading to a total running time of O(n2). The value we want is OPT(1), and the subsequence itself can be found by tracing back through the array of OPT values.