1 Dynamic Programming

1.1 Max independent set in a vertex-weighted path

Let $G = (V, E)$ be an undirected graph with n nodes. A subset S of V is called an *independent set* if no two vertices of S are joined by an edge. Finding large independent sets is difficult in general but not for paths. A graph is a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if i and j differ by exactly 1. With each node v_i, we associate a positive integer weight w_i. The goal in this question is to find an independent set in a path G whose total weight is as large as possible.

Solution

(a) The "heaviest-first" greedy algorithm: Not correct.
$1, 3, 5, 4$ — output $5 + 1 = 6$ but opt $= 3 + 4 = 7$.

(b) Max between odd set and even set: Not correct.
$1, 5, 3, 3, 5$ — output $1 + 3 + 5 = 9$ but opt $= 5 + 5 = 10$.

(c) For $1 \leqslant i \leqslant n$, let OPT$[i]$ denote the optimum value for houses from 1 to i.
Our goal is to find OPT$[n]$.

For DP recurrence, the base bases are OPT$[1] = c_1$, OPT$[2] = \max(c_1, c_2)$.
For $i \geqslant 3$, two possible cases:

(i) v_i does not contribute to OPT$[i] \implies$ OPT$[i]$ = OPT$[i-1]$.

(ii) v_i contributes to OPT$[i] \implies$ OPT$[i] = c_i +$ OPT$[i-2]$.

Combining the above two cases for $i \geqslant 3$, we get OPT$[i] = \max($OPT$[i-1], c_i +$ OPT$[i-2])$.

Time to compute each OPT$[i] = O(1) \implies$ total time complexity $= O(n)$.

Algorithm 1: Max-Indep-Set

```
1 OPT[1] ← c_1, OPT_2 ← max(c_1, c_2)
2 for i ← 3, 4, ..., n do
3 |  OPT[i] ← max(OPT[i − 1], c_i + OPT[i − 2]) // constant time
4 return OPT[n]
```

1.2 Optimum plan

Given a sequence of n weeks containing the value ℓ_i for "low-stress" job and h_i for "high-stress" job for every week i. A *plan* is specified by a choice of "low-stress", "high-stress", or "none" for each week, with the constraint 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 week i, you add ℓ_i or h_i or 0 to the value if you choose low-stress or high-stress or none in that week, respectively. Your task is to find a plan of maximum value.

Example: Suppose $n = 4$, and the values are as follows.

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

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 $0 + 50 + 10 + 10 = 70$.

Solution

For $1 \leqslant i \leqslant n$, let OPT[$i$] denote the value of an optimal plan for the first i weeks. Our goal is to find OPT[n].

Base cases: $\text{OPT}[1] = \max(\ell_1, h_1),\ \text{OPT}[2] = \max(h_1 + \ell_2, h_2)$.
Recurrence: In an optimal plan of the first $i \geqslant 3$ weeks, there are 3 possibilities:

(i) No job in week $i \implies \text{OPT}[i] = \text{OPT}[i-1]$.

(ii) Low-stress job in week $i \implies \text{OPT}[i] = \ell_i + \text{OPT}[i-1]$.

(iii) High-stress job in week $i \implies \text{OPT}[i] = h_i + \text{OPT}[i-2]$.

Combining cases (i), (ii), (iii), we get

$$\text{OPT}[i] = \max(\text{OPT}[i-1], \ell_i + \text{OPT}[i-1], h_i + \text{OPT}[i-2])\ \forall i \geqslant 3.$$

Algorithm: Create a 1D array OPT[1..n] and fill it up using the above recurrence and base cases. Return OPT[n].

Time complexity: Time to compute $\text{OPT}[i] = O(1) \implies$ total time $= O(n)$.

1.3 Longest path in ordered graph

Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) 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 (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node $v_i,\ i = 1, \ldots, n-1$, there is at least one edge of the form (v_i, v_j) with $i < j \leqslant n$.

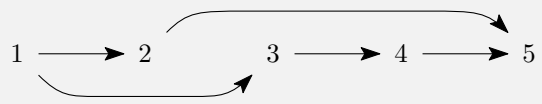
Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n. (The *length of a path* is the number of edges in it.)

(a) Show that the following greedy algorithm does not correctly solve this problem.

```
Set w = v_1
Set L = 0
While there is an edge out of the node w
  Choose the edge (w, v_j) for which j is as small as possible
  Set w = v_j
  Increase L by 1
end while
Return L as the length of the longest path
```

Solution

Following is a counterexample. The optimum is 3 but the algorithm returns 2.



(b) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v_1 and ends at v_n.

Solution

For $1 \leqslant i \leqslant n$, let OPT[$i$] denote the length of the longest path that begins at v_i and ends at v_n.
Note that OPT[i] is a well-defined number for $1 \leqslant i \leqslant n$ because G is an ordered graph.
Our goal is to find OPT[1].

Base cases: $\text{OPT}[n] = 0, \text{OPT}[n-1] = 1$.
Recurrence: $\text{OPT}[i] = \max_{(v_i, v_j) \in E} \{1 + \text{OPT}[j]\}$ for $1 \leqslant i \leqslant n-2$.

Algorithm: Create a 1D array OPT[1..n].
Fill up $\text{OPT}[n], \text{OPT}[n-1], \ldots, \text{OPT}[1]$ using the above recurrence and base cases.
Return OPT[1].

Time complexity: Time to compute $\text{OPT}[i] = |\{(v_i, v_j) \in E\}|$. So, total time is

$$\text{OPT}[1..n] = \sum_{v_i \in V} \left|\{(v_i, v_j) \in E\}\right| = O(E).$$

1.4 Minimum operating cost

Suppose you're running a lightweight consulting business. 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_i if you run the business out of NY or an operating cost of S_i if you run it out of SF. 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 i-th location indicates the city in which you will be based in the i-th 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.

Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost.

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 N_i < S_i then
     choose NY in Month i
   else
     choose SF in Month i
end for
```

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

Solution

Example of an instance with $M = 10$:

	Month 1	Month 2	Month 3	Month 4
NY	20	21	20	21
SF	21	20	21	20

The algorithm returns (NY, SF, NY, SF), which is not an optimal plan. An optimal plan is (NY, NY, NY, NY) or (SF, SF, SF, SF).

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

Solution

Example of an instance with $M = 2$:

	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6
NY	20	40	20	40	20	40
SF	40	20	21	20	40	20

The above example has a unique optimal plan (NY,SF,NY,SF,NY,SF), which changes locations three times.

(c) Give an efficient algorithm that finds an optimal plan.

Solution

For $1 \leqslant i \leqslant n$, define:
$A[i]$ = minimum cost of a plan for the first i months ending with NY;
$B[i]$ = minimum cost of a plan for the first i months ending with SF.

Our goal is to find $\min(A[n], B[n])$.

Base cases: $A[1] = N_1, B[1] = S_1$.
Recurrences: For $i \geqslant 2$,
$A[i] = \min(A[i-1] + N_i, B[i-1] + M + N_i)$,
$B[i] = \min(A[i-1] + M + S_i, B[i-1] + S_i)$.

Algorithm: Create 1D arrays $A[1..n]$ and $B[1..n]$, fill them up, using the above recurrence and base cases, and return $\min(A[n], B[n])$.

Time complexity: Time to compute $A[i]$ and $B[i] = O(1) \implies$ total time $= O(n)$.

1.5 Maximum-quality segmentation

Given a string of letters $y = y_1 y_2 \cdots y_n$, its segmentation means a partition into blocks of contiguous letters. If a block is a dictionary word, then its *quality* is $+1$, else -1. The quality for any block is available from a black box in constant time.

The *total quality* of a segmentation is determined by adding up the qualities of its blocks. For example, for `meetateight`, the total quality for the segmentation `meet + ate + ight` is $1 + 1 - 1 = 1$, whereas that for `meet + at + eight` is $1 + 1 + 1 = 3$.

Give an efficient algorithm that takes a string and computes a segmentation of maximum total quality.

Solution

For $1 \leqslant i \leqslant n$, let OPT[$i$] denote the maximum total quality of a segmentation of $y_1 \cdots y_i$.
Our goal is to find OPT[n].

Base cases: OPT[1] = quality(y_1).

Recurrence: Consider the last contiguous block in an optimal segmentation of $y_1 \cdots y_i$.
Let the last block be $y_j \cdots y_i$, where $1 \leqslant j \leqslant i$.
$j = 1 \implies$ OPT[i] = quality($y_1 y_2 \cdots y_i$).
$2 \leqslant j \leqslant i \implies$ OPT[i] = OPT[$j-1$] + quality($y_j y_{j+1} \cdots y_{i-1} y_i$).
So, for $2 \leqslant i \leqslant n$,

$$\text{OPT}[i] = \max\left(\max_{2 \leqslant j \leqslant i}\left\{\text{OPT}[j-1] + \text{quality}(y_j y_{j+1} \cdots y_{i-1} y_i)\right\}, \text{quality}(y_1 y_2 \cdots y_{i-1} y_i)\right).$$

Algorithm: Create a 1D arrays OPT[1..n], fill it up using the above recurrence and base cases, and return OPT[n].

Time complexity analysis: Time to compute OPT[i] = $O(i) \implies$ total time = $O(n^2)$.

1.6 Minimum-slack text formatting

Suppose our text consists of a sequence of words, $W = \{w_1, \ldots, w_n\}$, where w_i consists of c_i characters. Assume that all characters have same fixed-width font and ignore issues of punctuation or hyphenation. We have a maximum line-length of L. 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 $w_j, \ldots, w_k$ are assigned to one line, then we should have

$$k - j + \sum_{i=j}^{k} c_i \leqslant L.$$

An assignment of words to a line is valid if it satisfies the above constraint. The amount by which the sum (left-hand side of the above equation) falls short of L, is 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.

The goal of "minimum-slack text formatting" 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.
```

Solution

For $1 \leqslant i \leqslant n$, let OPT[$i$] denote the minimum possible value of the sum of the squares of the slacks of all lines in a partition of $\{w_1, w_2, \ldots, w_i\}$. Our goal is to find OPT[n].

Base cases: $\text{OPT}[0] = 0,\ \text{OPT}[1] = (L - c_1)^2$.

Recurrence: Consider the last line in an optimal partition of $\{w_1, w_2, \ldots, w_i\}$. Let it be $\{w_h, w_{h+1}, \ldots, w_i\}$, where $1 \leqslant h \leqslant i$. Since it is a valid line, we have

$$i - h + \sum_{k=h}^{i} c_k \leqslant L \quad \Longrightarrow \quad \text{OPT}[i] = \text{OPT}[h-1] + \left(L - i + h - \sum_{k=h}^{i} c_k \right)^2 .$$

So, for $2 \leqslant i \leqslant n$, we have

$$\text{OPT}[i] \quad = \quad \min_{\substack{1 \leqslant h \leqslant i \\ i - h + \sum_{k=h}^{i} c_k \leqslant L}} \left\{ \text{OPT}[h-1] + \left(L - i + h - \sum_{k=h}^{i} c_k \right)^2 \right\} .$$

Algorithm 2: Text formatting

```
1  OPT[0] ← 0, OPT[1] ← (L − c_1)^2
2  for h ← 1, ..., n do
3  |  c[h][h] ← c_h
4  |  for i ← h + 1, ..., n do
5  |  |  c[h][i] ← c[h][i − 1] + c_i // c[h][i] = c_h + ··· + c_i
6  for i ← 2, ..., n do
7  |  OPT[i] ← ∞
8  |  for h ← 1, ..., i do
9  |  |  if i − h + c[h][i] ≤ L then
10 |  |  |  OPT[i] ← min(OPT[i], OPT[h − 1] + (L − i + h − c[h][i])^2) // constant time
11 return OPT[n]
```

Time complexity: $c[1..n][1..n]$ is computed in $O(n^2)$ time. OPT[i] takes $O(i)$ time to compute, so total time = $O(n^2)$.