Dynamic programming example: high- and low-stress jobs

CSCI 382, Algorithms

October 13, 2017

Consider the following table composed of n weeks where each week i has low stress job that pays L_i and a high stress job that pays H_i.

WEEK	1	2	3	$\cdots$	n
low stress	L_1	L_2	L_3	$\cdots$	L_n
high stress	H_1	H_2	H_3	$\cdots$	H_n

Each week you are allowed to pick either a low stress job or a high stress job; however, picking a high stress job at week i means that you must take the week before (*i.e.* week $i-1$) off. Your goal is to maximize your total income. Let's suppose that you *are* allowed to take the high-stress job in week 1 (it doesn't change the overall solution much either way, only the base case).

Step 1: A Recurrence

Consider different ways of splitting up or restricting the overall problem into subproblems or subcases, and come up with a recurrence.

The key to solving this problem is to consider subcases of the problem corresponding to working *only up to week i*. Define $OPT(i)$ as the maximum amount you can get for working weeks $1 \ldots i$ only. Then we can write down a recurrence for OPT:

$$
\begin{aligned}
OPT(0) &= 0 \\
OPT(1) &= \max(L_1, H_1) \\
OPT(i) &= \max \begin{cases} L_i + OPT(i-1) \\ H_i + OPT(i-2). \end{cases}
\end{aligned}
$$

Step 2: Induction

Prove the recurrence correct by induction.

Let's prove that $OPT(i)$ is indeed the most money we could possibly making by working only weeks $1 \ldots i$.

Proof. By (strong) induction on i.

- Base cases:

  - $OPT(0) = 0$ since we don't make any money for working 0 weeks.
  - $OPT(1) = \max(L_1, H_1)$, since if we're only working 1 week, the best we can do is to just take the better-paying job.

  Note that we need *two* base cases because in the recursive case, we define $OPT(i)$ in terms of both $OPT(i-1)$ and $OPT(i-2)$.

- In the inductive case, let $i \geq 2$ and suppose that for all $k < i$, $OPT(k)$ is indeed the most money we can possibly make working weeks $1 \ldots k$. Then we must show that $OPT(i)$ is also correct, that is, it is the optimal amount for working weeks $1 \ldots i$.

  Note first that if we didn't work week i, we could always increase our profit by just taking the low-stress job in week i—so taking the last week off is never the optimal schedule. Thus, we should definitely work the last week, and our only decision is between taking the high-stress job or low-stress job. If we work the low-stress job in week i, we get L_i for that job, and we should do the best we possibly can for the previous $i-1$ weeks, which by the IH is $OPT(i-1)$. If we work the high-stress job in week i for an income of H_i, we have to take week $i-1$ off, but we should then do the best we can in the previous $i-2$ weeks, which again by the IH is $OPT(i-2)$. Since these are our only two choices, the best we can possibly do is just the maximum of the total we would get in either case.

This is a typical pattern: (1) consider subproblems of the original problem; (2) write down a recurrence for computing the optimal value for each subproblem; (3) prove it is correct by induction.

We could have also considered a divide and conquer approach: *i.e.* split the weeks in half, find the best schedule for each half recursively, ... and then what? The problem is that the halves are not independent. The best schedule for each of the halves in isolation might not give us the best overall schedule—in particular it might not be allowed, if the best schedule for the second half starts with a high-stress job.

Step 3: Memoize

If there are overlapping subproblems, memoize.

If none of the recursive subproblems will overlap, rejoice! In that case you just have a divide-and-conquer algorithm. For example, the

recursive calls of mergesort will never overlap, becuase the recursive calls are always on completely disjoint parts of the input list.

In this case, however, we can see that naively computing $OPT(i)$ using recursion would lead to many values of OPT being redundantly computed over and over—in fact, it has the same recursion pattern as the Fibonacci numbers.

In this case, we can store the values of $OPT(i)$ in an array of length $n + 1$; since each $OPT(i)$ depends only on the previous two, we can fill in the array from index 0 up to index n. Here is how we might implement it in Python:

```
# low, high are lists of weekly salaries for low- and
# high-stress job, respectively.

def best_income(low, high):

    n = len(low)

    # Make low and high 1-indexed by adding dummy values to the
    # beginning.  This works much more nicely since we want opt[i] =
    # maximum for working i weeks, and opt[0] = 0 is a natural base
    # case, so we want to have the weeks be 1-indexed.  (Adding extra
    # "padding" elements to arrays to make room for base cases is a
    # common technique with DP problems.)

    low  = [0] + low
    high = [0] + high

    # Initialize opt[0..n] with all zeros.
    opt = [0] * (n+1)

    # The first base case, opt[0] = 0, is already taken care of.
    # For opt[1], just take the higher of the two jobs.
    opt[1] = max(low[1], high[1])

    # Now loop through remaining weeks.
    for i in range(2, n+1):

        # How much could we make taking the low or high stress job?
        low_total  = low[i]  + opt[i-1]
        high_total = high[i] + opt[i-2]

        # The optimal for weeks 1..i is the higher of the two
        opt[i] = max(low_total, high_total)
```

```
    return opt[n]

# Example:
# >>> best_income([2,2,1,7,5,20,3,19,10,13], [1,5,10,100,23,20,5,21,30,30])
# 182
```

Step 4: Remember Your Choices!

To compute the actual optimal solution instead of just the optimal value, save the choices made at each step.

This solution tells us *how much* money we can make, but it doesn't actually tell us which job we should take each week. However, it is not too hard to modify our solution to do this as well. The key observation is that during the algorithm, for each i, taking the max of two values corresponds to deciding which option would be better—to take the low- or high-stress job at week i. So we can just add another array, *JOBS*, to keep track of which decision leads to the best outcome at each week—that is, $JOBS[i]$ records whether we should pick the low- or high-stress job at week i in order to maximize our income if we only work weeks $1 \ldots i$.

Note that *JOBS* does *not* directly tell us which jobs we should actually take to get the most money overall. For example, suppose $JOBS[3]$ contains H. This means that *if we only worked the first three weeks*, then we should take the high-stress job in week 3. However, it may turn out that the best schedule *overall* has us working the low-stress job or taking week 3 off.

So, how do we actually reconstruct the optimal schedule for weeks $1 \ldots n$ from *JOBS*? We just start at week n and work backwards (because this mirrors the recursive structure of *OPT*):

- If $JOBS[n] = L$, then we take the low-stress job in week n, and continue by looking at $JOBS[n-1]$ to see what we should do the week before.
- If $JOBS[n] = H$, then we take the high-stree job in week n, take week $n-1$ off, and continue by looking at $JOBS[n-2]$.

We continue this process until reaching week 1.

Here is an enhanced version of the Python solution which computes the optimal schedule along with the optimal income:

```
def best_income(low, high):

    n = len(low)
```

```
    low  = [0] + low
    high = [0] + high

    # Initialize an array of booleans to keep track of whether the
    # optimal choice is to take the high-stress job at week i (if we
    # only work weeks 1..i).
    take_high_job = [False] * (n+1)
    opt = [0] * (n+1)

    opt[1] = max(low[1], high[1])

    # We should take the high-stress job at week 1 iff it pays more.
    take_high_job[1] = (high[1] > low[1])

    for i in range(2, n+1):

        low_total  = low[i]  + opt[i-1]
        high_total = high[i] + opt[i-2]
        opt[i] = max(low_total, high_total)

        # Record which choice produced the higher total
        take_high_job[i] = high_total > low_total

    # Now, to produce a work schedule, work backwards from the end
    # Start at week n
    w = n
    schedule = []
    while w > 0:

        # If we should take the high-stress job at week w, schedule it
        # and a week off, and proceed to look at week w-2 next
        if take_high_job[w]:
            schedule = ['off', 'HI'] + schedule
            w -= 2

        # Otherwise schedule the low-stress job and look at week w-1 next
        else:
            schedule = ['LO'] + schedule
            w -= 1

    return (opt[n], schedule)

# Example:
```

```
# >>> best_income([2,2,1,7,5,20,3,19,10,13], [1,5,10,100,23,20,5,21,30,30])
# (182, ['off', 'HI', 'off', 'HI', 'LO', 'LO', 'LO', 'LO', 'off', 'HI'])
```