CS561, HW5

Ryan Scherbarth, University of New Mexico

September 2024

1. A frog is jumping across a line of lily pads. It starts at lily pad 1. When the frog is at lily pad i, for any $i \geq 1$, it jumps to lily pad $i+1$ with probability $\frac{1}{2}$ and to lily pad $i+2$ with probability $\frac{1}{2}$.

   - (a) Let $p(i)$ be the probability that the frog ever visits lily pad i, for any $i \geq 1$. Write a recurrence relation for $p(i)$. Don't forget the base case(s)

   Case $p(1) = 1$.
   Case $p(2) = \frac{1}{2}$.

   We can calculate the probability each of the cases happen by just multiplying against the probability, so the probability of jumping 1 lilly pad is given by $\frac{1}{2}p_1 + \frac{1}{2}p_2$. We can now take into account the effect on the total number of remaining lily pads on each recursive step to come to our final recurrence;

$$
\begin{aligned}
p(1) &= 1 \\
p(2) &= \frac{1}{2} \\
p(i) &= \frac{1}{2}p(i-1) + \frac{1}{2}p(i-2) \text{ for } i \geq 3
\end{aligned}
$$

   - (b) Use annihilators to solve for a general solution to your recurrence relation

$$
\begin{aligned}
p(i) &= p(i) \\
Lp &=< p(i+1) > \\
L^2p &= \frac{1}{2}Lp(i) + \frac{1}{2}p(i) \\
&= L^2p - \frac{1}{2}Lp(i) - \frac{1}{2}p(i)
\end{aligned}
$$

   The annihilator $L^2 + \frac{1}{2}L + \frac{1}{2}$ can be factored to $(L-1)(L-\frac{1}{2})$. We know that sequences of $(L-a)(L-b)$ have solutions in the form $< c_1a^n + c_2b^n >$.

- (c) Use the base case(s) of your recurrence to solve for an exact solution

$$\begin{aligned} p(1) &= 1 \\ p(2) &= \frac{1}{2} \\ p(i) &= A + B(-\frac{1}{2})^i \\ A &= \frac{2}{3} \\ B &= -\frac{2}{3} \\ p(i) &= \frac{2}{3} - \frac{2}{3}(-\frac{1}{2})^i \end{aligned}$$

- (d) Now, let X be a random variable giving the number of lily pads between lily pad 1 and n that the frog visits, for some number n. Compute $E(x)$ by using: linearity of expectation, indicator random variables, and your solution to the recurrence $p(i)$ that you found above.

We will define I to be a discrete random indicator variable s.t. I_i is 1 if the frog visits lily pad i, and 0 otherwise.
The expected value of I; $E[I_i] = p(i)$. Using the linearity of expectation, the expected number of lily pads visited can be represented as $E[X] = E[\sum_{i=1}^{n} I_i] = \sum_{i=1}^{n} E[I_i]$.
Therefore, our new formula will be $\sum_{i=1}^{n} p(i)$.

$$\begin{aligned} E[I_i] &= \sum_{i=1}^{n} p(i) \\ &= \sum_{i=1}^{n} \frac{2}{3} - \frac{2}{3}(-\frac{1}{2})^i \\ &= \sum_{i=1}^{n} \frac{2}{3} \sum_{i=1}^{n} (-\frac{2}{3})(-\frac{1}{2})^i \\ &= (\frac{2}{3}n)(\frac{-\frac{1}{2}(1-(-\frac{1}{2})^n)}{1-(-\frac{1}{2})}) \\ &= \frac{2}{3}n + \frac{2}{9}(1-(-\frac{1}{2})^n) \end{aligned}$$

So the expected value representing the number of lily pads the frog jobs to for any set of n lilypads is given by $f(n) = \frac{2}{3}n + \frac{2}{9}(1-(-\frac{1}{2})^n)$.

2. As in last homework, a thief repeatedly robs the same bank. To avoid capture, he never robs the bank fewer than 10 days after the last robbery. He has obtained information, for the next n days, ont eh amount of money b_i that is held at the bank on day i.
But now, the thief is lazy: (1) for each day, there is an integer value giving the amount of work w_i that the thief must perform to rob the bank on that day (due to the amount of security on that day); and (2) there is an additional constraint that the sum of work the thief ever performs is less than some value W.
Let $r(i, j)$ be the maximum amount of revenue obtainable on days 1 through i, with at most j total work.

   - (a) Give a recurrence relation for $r(i, j)$

   Base case: $r(0, j) = 0$. The bank is empty if it has been 0 days since it was robbed.
   Recursive case: $r(i, j) = b_i + r(i - 10, j - w_i)$.

   We can define the final recursive case, then, as $r(i, j) = r(i - 1, j) + b_i + r(i - 10, j - w_i))$. Our recursive case is further restricted and therefore determined by the value of j. The robber will always choose the best option, given the amount of work is valid.

   if $j \geq w$, the robber will choose the larger case $r(i - 1, j) + b_i + r(i - 10, j - w_i)$.
   If $j < w_i$, the robber must choose $r(i - 1, j)$.

   - (b) Describe a dynamic program based on this recurrence. What is the runtime of your algorithm?

   Given our recurrence relation above, we will create a 2D table to visualize the relationship between work and total revenue. We will set rows to represent the days, i, where $i \exists [0..n]$. The columns represent the remaining work capacity j where $j \exists [0..W]$.
   We will fill in each index on the table by just using the recurrence relation from above. We can then iterate through the 2D plot.

   We begin by filling out the able of n rows and $W + 1$ columns. Each cell takes $O(1)$ time as we're just doing a single calculation. Overall the running time is bounded by the process of creating the table in the first place, which will end up taking us $O(n * W)$ time, which will ultimately be the bound for our algorithm.

3. You have n feet of cable to be cut into pieces for resale. On a given day, pieces of length 1, 3, and 7 can resell for values v_1, v_2, and v_3. You want to cut the cable into pieces with maximum total resell value. For example, if $n = 14$, and $v_1 = 1$, $v_2 = 4$, and $v_3 = 8$, then the optimal cutting is: 4 pieces of length 3, and 2 pieces of length 1, for a total resell value of $4 * 4 + 2 * 1 = 18$.
You decide to solve this problem with dynamic programming. For any number $i \exists [0, n]$, let $m(i)$ be the maximum resell value you can get from optimally cutting a cable of length i. Write a recurrence relation for $m(i)$. Don't forget the base case(s)

Base case: $m(0) = 0$
Recursive step: $v_1 + m(i-1) + v_2 + m(i-3) + v_3 + m(i-7)$

$$m(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\left(v_1 + m(i-1),\, v_2 + m(i-3),\, v_3 + m(i-7)\right) & \text{if } i \geq 7 \\ \max\left(v_1 + m(i-1),\, v_2 + m(i-3)\right) & \text{if } 3 \leq i < 7 \\ v_1 + m(i-1) & \text{if } 1 \leq i < 3 \end{cases}$$

In the recursive relation above:

- For $i = 0$: No cable length means no resell value, so $m(0) = 0$.
- For $i \geq 7$: The maximum resell value is obtained by considering the cuts of lengths 1, 3, or 7.
- For $3 \leq i < 7$: Only cuts of lengths 1 and 3 are possible.
- For $1 \leq i < 3$: Only cuts of length 1 are possible.

Initialize a table $m[0 \ldots n]$ where $m[i]$ stores the maximum resell value for cable length i.

Set $m(0) = 0$, for each $i = 1$ to n:
If $i \geq 1$: $m(i) = \max(m(i), v_1 + m(i-1))$
If $i \geq 3$: $m(i) = \max(m(i), v_2 + m(i-3))$
If $i \geq 7$: $m(i) = \max(m(i), v_3 + m(i-7))$
$m(n)$ will contain the maximum resell value for the cable of length n.

The time complexity is $O(n)$ due to filling the table once, with each entry requiring constant time to update. The space complexity is $O(n)$ to store the table of length $n + 1$.

4. Now consider a variant of the above problem where the maximum number of cuts you can make is some integer k. As before, pieces of length 1, 3, and 7 resell for values v_1, v_2, and v_3. Any pieces of other lengths have zero value. For example, if $n = 14$, $k = 1$, and $v_1 = 1$, $v_2 = 4$, and $v_3 = 8$, then the optimal cutting is: 2 pieces of length 7 for a total resell value of $2 * 8 = 16$.

   Write a recurrence relation for a dynamic program for this variant. In particular, for numbers $i \exists [0, n]$, and $j \exists [0, k]$, let $m(i, k)$ be the maximum resell value you can get from cutting a cable of length i with at most k cuts/ Write a recurrence relation for $m(i, k)$. Don't forget the base case(s).

   Let $m(i, j)$ represent the maximum resell value obtainable from the cable length i with a maximum of j cuts.

   Base case: $m(0, j) = 0$ for all $j \geq 0$
   Base case: $m(i, 0) = 0$ for all $i \geq 0$

   Recursive case: $m(i, j) = m(i - 1, j)$. When making a cut, we have three options;

$$m(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\left(m(i-1,j),\ v_1 + m(i-1,j-1)\right) & \text{if } 1 \leq i < 3 \text{ and } j \geq 1 \\ \max\left(m(i-1,j),\ v_1 + m(i-1,j-1),\ v_2 + m(i-3,j-1)\right) & \text{if } 3 \leq i < 7 \text{ and } j \geq 1 \\ \max\left(m(i-1,j),\ v_1 + m(i-1,j-1),\ v_2 + m(i-3,j-1),\ v_3 + m(i-7,j-1)\right) & \text{if } i \geq 7 \text{ and } j \geq 1 \end{cases}$$

   Now we would create a 2D table $m[i][j]$ where i represents the cable length ranging from $[0..n]$ and j represents the maximum number of cuts ranging from $[0..k]$.
   At this point, we simply iterate through the table filling out each value according to our recurrence relation, which will give a running time of $O(i * j)$ for the first iteration. We will then have a running time of $O(1)$ each subsequent time as we just need to check the corresponding values in the table.
   This implementation will use $O(i * j)$ space complexity as well, in order to store the entire table.

5. You have a chocolate bar consisting of n chunks aligned in a single row. Each chunk, i, for $1 \leq i \leq n$ has some positive value v_i (for example, chunks with high nougat content are more valuable than those without!).
You must break the bar into exactly k parts to share with your friends, where each part consists of some number of contiguous, unbroken chunks. The value of a part is the sum of the value of all chunks in that part. Your (greedy) friends chose their parts first, and you get the part remaining, i.e. the part of the smallest value. Thus, your goal is to break the bar in such a way that you maximize the value of the minimum value part.
Write a recurrence relation and give a dynamic program to solve this problem. What is the runtime of your algorithm?

We can solve this problem using a dynamic programming approach with a 1D array. Let $f[j]$ represent the maximum value of the minimum part achievable up to the current chunk, when split into exactly j parts. The sum of any sub-array can be computed in $O(1)$ time.

Base Case:

$$\begin{aligned} f[1] &= \text{prefix\_sum}[i], && \text{for } i = 1 \text{ to } n \\ f[j] &= 0, && \text{for } j > 1 \end{aligned}$$

Here, $f[1]$ is the cumulative sum of the chunks up to i when only one part is allowed.

Recursive Step:

$$f[j] = \max_{x=1}^{i-1} (\min(f[j-1], \text{prefix}[i] - \text{prefix}[x]))$$

This relation computes the maximum minimum part value by checking all possible splits x from 1 to $i-1$, updating $f[j]$ with the maximum of the minimum part values.

Initialize a prefix sum array $prefix$ such that $prefix[i] = \sum_{x=1}^{i} v_x$.
Initialize a 1D array f of length $k+1$ with all zeros.
For each number of parts j from 2 to k, and for each chunk position i from 1 to n:
Update $f[j]$ using the recursive relation above.
The final value $f[k]$ will be the maximum value.

Time Complexity:
The time complexity of this algorithm is $O(n \times k)$, as we iterate over each chunk and each number of parts, performing $O(1)$ operations using the prefix sum array.