CSE 421 Section 5

Dynamic Programming

Administrivia

Announcements & Reminders

- **HW3** regrade requests are open, answer keys on Ed

- **HW4** was due yesterday, 10/23
    - Late submissions open until tomorrow, 10/25 @ 11:59pm

- **HW5** is due Wednesday, 10/30 @ 11:59pm
    - Last homework before the midterm exam

- Your **midterm exam** is in about 1.5 weeks!
    - **Monday, 11/4 @ 6:00–7:30pm, Gates G04**
    - If you can’t make it, let us know and we will schedule a makeup exam

Ideas for dynamic programming

What is dynamic programming?

Warmup! Compare and contrast **divide and conquer** with **dynamic programming**. What the defining features of each? When might you want to use each?

This problem is not on your handout.

Feel free to work with the people around you!

Solution

What is dynamic programming?

Warmup! Compare and contrast **divide and conquer** with **dynamic programming**. What the defining features of each? When might you want to use each?

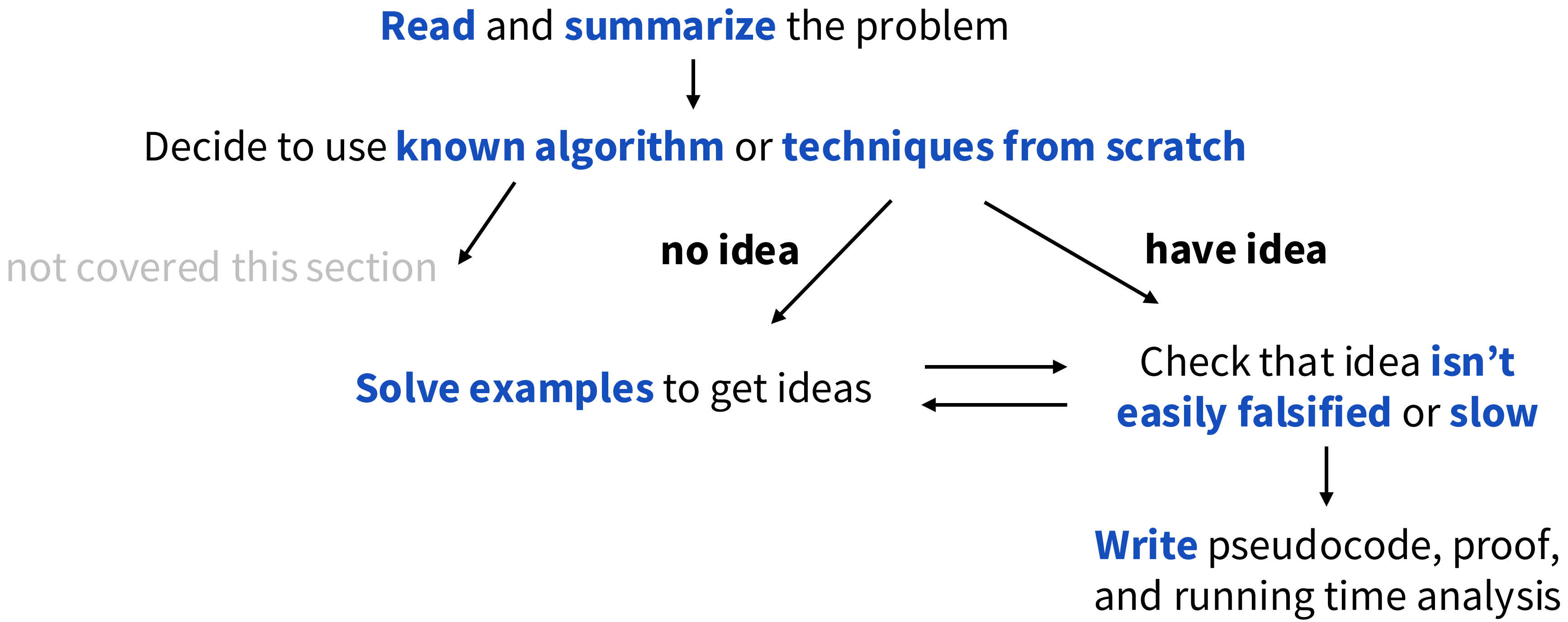
Divide and conquer	**Dynamic programming**
• Subproblems are significantly smaller, disjoint pieces (e.g. half)	• Subproblems can be as large as “one smaller” and overlap
• Memory not needed because every subproblem is used once	• Memory is useful because each subproblem is used many times

What is dynamic programming?

Unlike divide and conquer, where subproblems are typically obvious, subproblems in dynamic programming have many flavors:

Prefixes	Intervals	Other
• Fibonacci • Weighted interval scheduling • Longest increasing subsequence • Edit distance (two prefixes)	• RNA secondary structure	• Knapsack (prefix of items with capacity bound) • Bellman-Ford (tomorrow's lecture, vertex with path length bound)

Problem solving strategy overview


Read and **summarize** the problem
Decide to use **known algorithm** or **techniques from scratch**
not covered this section
no idea
have idea
Solve examples to get ideas
Check that idea **isn't easily falsified** or **slow**
Write pseudocode, proof, and running time analysis

Problem solving strategy overview

Read and **summarize** the

Decide to use **known algorithm** or **tec**

not covered this section

no idea

~~**Solve examples**~~ to get ideas

Check that idea **isn't easily falsified** or **slow**

Write pseudocode, proof, and running time analysis

Dynamic programming is difficult because **examples rarely help until you know your subproblems**. You need to abstractly analyze the problem to determine subproblems first.

We'll go over some strategies today.

Problem 1 – Going to parties

You are planning your calendar for n days, where every day, there is a party that you can go to. Every day, you can choose to go to that party or stay in and catch-up on sleep. If you party, you will enjoy yourself. But you can only party for two consecutive days – if you party three days in a row, you'll fall too far behind on sleep and miss class. Luckily, you have an excellent social sense, so you know exactly how much you will enjoy any of the parties, and you have assigned each day a positive integer happiness score in an array $H[1..n]$. You get 0 happiness if you do not go to the party. Maximize the sum of your happiness for these n days, while not going out for more than two consecutive days.

a) Write a summary for this problem.

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

a) Write a summary for this problem.

Input: An array of positive integers $H[1..n]$.
Expected output: The maximum value of $\sum_{i \in S} H[i]$ over all $S \subseteq \{1, \dots, n\}$ with no three consecutive indices.

When examples don’t help

Here’s an in-class exercise to show why examples are not so helpful before you know your subproblems.

Without trying any particular class of subproblems, just try to maximize your happiness given the following array, using your personal logic and heuristics:

[9, 2, 3, 8, 6, 6, 4, 7, 1]

Try this for a minute, we’ll see who can get the best!

Solution

When examples don’t help

Here’s an in-class exercise to show why examples are not so helpful before you know your subproblems.

Without trying any particular class of subproblems, just try to maximize your happiness given the following array, using your personal logic and heuristics:

[9, 2, 3, 8, 6, 6, 4, 7, 1]

The answer is 36, by taking [9, 2, 8, 6, 4, 7].

A common strategy

Here is a useful outline for what to try first when doing dynamic programming on a one-dimensional array. You've seen this strategy in lecture before.

1. Let $\text{OPT}(j) =$ the optimal solution on inputs up to j.
2. Divide $\text{OPT}(j)$ into **cases based on what to do with the jth element**.
3. For each case, can you use $\text{OPT}(j-1)$ to handle it? Why or why not?
4. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

A common strategy

We'll use knapsack as an example. Recall that the problem is to maximize the value of a knapsack with maximum capacity W, given items with weights w_i and values v_i.

1. Let $\mathrm{OPT}(j)$ = the optimal solution on inputs up to j.

Let $\mathrm{OPT}(j)$ = the most value you can pack using items up to j.

2. Divide $\mathrm{OPT}(j)$ into cases based on what to do with the jth element.

Either pack or discard item j.

A common strategy

3. For each case, can you use $\mathrm{OPT}(j-1)$ to handle it? Why or why not?

If you discard item j, the best you can pack is $\mathrm{OPT}(j-1)$. But if you pack item j, we cannot use $\mathrm{OPT}(j-1)$, because item j has some weight and $\mathrm{OPT}(j-1)$ might already be nearing capacity.

4. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

It would help to know the best we can do with various amounts of remaining capacity. Thus, we can try subproblems that are also parameterized by remaining capacity, i.e. $\mathrm{OPT}(j, w) =$ the most value you can pack using items up to j and at most w weight.

A common strategy

One more quick example: Recall that in weighted interval scheduling, you are given n jobs with start times s_i, finish times f_i, and weights w_i, and you want to maximize the weight of a valid schedule.

1. Let $\mathrm{OPT}(j) =$ the optimal solution on inputs up to j.

Let $\mathrm{OPT}(j) =$ the maximum weight of a valid schedule, using jobs up to jth largest finish time.

2. Divide $\mathrm{OPT}(j)$ into cases based on what to do with the jth element.

Either schedule or discard the job with jth largest finish time.

A common strategy

3. For each case, can you use $\mathrm{OPT}(j-1)$ to handle it? Why or why not?

If you discard the job, the best you can do is $\mathrm{OPT}(j-1)$. But if you schedule the job, we cannot use $\mathrm{OPT}(j-1)$, because the job with jth largest finish time may overlap with some jobs taken in $\mathrm{OPT}(j-1)$.

4. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

We don't need to introduce extra parameters this time, we just need to look back further. If we discard the overlapping jobs, we are left with a subproblem that is still a prefix of the original. In other words, we can continue to use our definition of $\mathrm{OPT}(j)$.

Problem 1 – Going to parties

b) Now you try. Say how to adapt each step to the party problem at hand.
   i. Let $\mathrm{OPT}(j) =$ the optimal solution on inputs up to j.
   ii. Divide $\mathrm{OPT}(j)$ into cases based on what to do with the jth element.
   iii. For each case, can you use $\mathrm{OPT}(j-1)$ to handle it? Why or why not?
   iv. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

Previous Solution

Problem 1 – Going to parties

b) Now you try. Say how to adapt each step to the party problem at hand.
   i. Let $\mathrm{OPT}(j) =$ the optimal solution on inputs up to j.
   ii. Divide $\mathrm{OPT}(j)$ into cases based on what to do with the jth element.
   iii. For each case, can you use $\mathrm{OPT}(j-1)$ to handle it? Why or why not?
   iv. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

Input: An array of positive integers $H[1..n]$.
Expected output: The maximum value of $\sum_{i\in S} H[i]$ over all $S \subseteq \{1, \dots, n\}$ with no three consecutive indices.

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

i. Let $\mathrm{OPT}(j)$ = the optimal solution on inputs up to j.

Let $\mathrm{OPT}(j)$ = the maximum happiness you can get on days $1..j$.

ii. Divide $\mathrm{OPT}(j)$ into cases based on what to do with the jth element.

On day j, you will either sleep or party.

iii. For each case, can you use $\mathrm{OPT}(j-1)$ to handle it? Why or why not?

If you sleep, your happiness will be $\mathrm{OPT}(j-1)$. However, if you party, there is no way to use $\mathrm{OPT}(j-1)$, because maybe $\mathrm{OPT}(j-1)$ called for partying both yesterday and the day before.

If you haven't gotten part (iv) yet, take a moment to think about it!

Solution

Problem 1 – Going to parties

iv. If you could not handle it, what additional information would help you? What kinds of subproblems are these?

Keep track of the last time you slept (equivalently, how many days you've been partying). This way, you could choose to only party if you slept within the last 2 days.

Two equivalent ways to implement this:

1. Let $\mathrm{OPT}(j, s) =$ maximum happiness from days $1..j$, assuming you last slept s days ago, for $s = 0, 1, 2$.
2. Look back to $\mathrm{OPT}(j - 2)$ if you last slept on day $j - 1$, or look back to $\mathrm{OPT}(j - 3)$ if you last slept on day $j - 2$.

Both ways involve only prefixes as subproblems.

Problem 1 – Going to parties

c) Now that you know what form you want your subproblems to take, retry this example to flesh out the details of your recurrence and convince yourself that it works.

[9, 2, 3, 8, 6, 6, 4, 7, 1]

d) Write the recurrence relation (for either of the two ideas from last slide). Don’t forget the base case(s).

Feel free to work with the people around you!

Previous Solution

Problem 1 – Going to parties

c) Now that you know what form you want your subproblems to take, retry this example to flesh out the details of your recurrence and convince yourself that it works. In other words, since our subproblems are prefixes, compute $\mathrm{OPT}(j, s)$ or $\mathrm{OPT}(j)$ (your choice) for every prefix of the following array:

[9, 2, 3, 8, 6, 6, 4, 7, 1]

d) Write the recurrence relation (for either of the two ideas from last slide). Don't forget the base case(s).

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

If using $\mathrm{OPT}(j, s)$:

$H[j]$	9	2	3	8	6	6	4	7	1
$\mathrm{OPT}(j, 0)$	0	9	11	12	20	25	26	30	36
$\mathrm{OPT}(j, 1)$	9	2	12	19	18	26	29	33	31
$\mathrm{OPT}(j, 2)$	9	11	5	20	25	24	30	36	34

If using $\mathrm{OPT}(j)$:

$H[j]$	9	2	3	8	6	6	4	7	1
$\mathrm{OPT}(j)$	9	11	12	20	25	26	30	36	36

Solution

Problem 1 – Going to parties

If using $\mathrm{OPT}(j, s)$:

$$\mathrm{OPT}(j,0) = \max\big(\mathrm{OPT}(j-1,0), \mathrm{OPT}(j-1,1), \mathrm{OPT}(j-1,2)\big)$$
$$\mathrm{OPT}(j,1) = \mathrm{OPT}(j-1,0) + H[j]$$
$$\mathrm{OPT}(j,2) = \mathrm{OPT}(j-2,0) + H[j-1] + H[j]$$

Nice to also notice $\mathrm{OPT}(j,2) = \mathrm{OPT}(j-1,1) + H[j]$, can avoid looking back 2 days.

If using $\mathrm{OPT}(j)$:

$$\mathrm{OPT}(j) = \max\begin{pmatrix} \mathrm{OPT}(j-1), \\ \mathrm{OPT}(j-2) + H[j], \\ \mathrm{OPT}(j-3) + H[j-1] + H[j] \end{pmatrix}$$

Solution

Problem 1 – Going to parties

If using $\mathrm{OPT}(j, s)$:

$$\mathrm{OPT}(j,0) = \max\big(\mathrm{OPT}(j-1,0), \mathrm{OPT}(j-1,1), \mathrm{OPT}(j-1,2)\big)$$
$$\mathrm{OPT}(j,1) = \mathrm{OPT}(j-1,0) + H[j]$$
$$\mathrm{OPT}(j,2) = \mathrm{OPT}(j-2,0) + H[j-1] + H[j]$$

Nice to als ck 2 days.

Bonus: To see that these are equivalent, just use substitution to prove that $\mathrm{OPT}(j) = \mathrm{OPT}(j+1, 0)$.

If using $\mathrm{OPT}(j)$:

$$\mathrm{OPT}(j) = \max\begin{pmatrix} \mathrm{OPT}(j-1), \\ \mathrm{OPT}(j-2) + H[j], \\ \mathrm{OPT}(j-3) + H[j-1] + H[j] \end{pmatrix}$$

Solution

Problem 1 – Going to parties

Now for base cases. If using $\mathrm{OPT}(j, s)$:

$$\mathrm{OPT}(j, 0) = \max\big(\mathrm{OPT}(j-1, 0), \mathrm{OPT}(j-1, 1), \mathrm{OPT}(j-1, 2)\big)$$
$$\mathrm{OPT}(j, 1) = \mathrm{OPT}(j-1, 0) + H[j]$$
$$\mathrm{OPT}(j, 2) = \mathrm{OPT}(j-2, 0) + H[j-1] + H[j]$$

We have many equivalent options.

$$\mathrm{OPT}(0, s) = 0$$
$$\mathrm{OPT}(1, 2) = H[1]$$

$$\mathrm{OPT}(0, s) = 0$$
$$\mathrm{OPT}(-1, 0) = 0$$

$$\mathrm{OPT}(1, 0) = 0$$
$$\mathrm{OPT}(1, 1) = H[1]$$
$$\mathrm{OPT}(1, 2) = H[1]$$
$$\mathrm{OPT}(2, 2) = H[1] + H[2]$$

Solution

Problem 1 – Going to parties

Now for base cases. If using $\mathrm{OPT}(j, s)$:

$$\mathrm{OPT}(j,0) = \max\big(\mathrm{OPT}(j-1,0), \mathrm{OPT}(j-1,1), \mathrm{OPT}(j-1,2)\big)$$

OPT

OPT

If you recognized $\mathrm{OPT}(j, 2) = \mathrm{OPT}(j-1, 1) + H[j]$, you wouldn't need the last case in each.

We have many equivalent options.

$\mathrm{OPT}(0, s) = 0$
~~$\mathrm{OPT}(1, 2) = H[1]$~~

$\mathrm{OPT}(0, s) = 0$
~~$\mathrm{OPT}(-1, 0) = 0$~~

$\mathrm{OPT}(1, 0) = 0$
$\mathrm{OPT}(1, 1) = H[1]$
$\mathrm{OPT}(1, 2) = H[1]$
~~$\mathrm{OPT}(2, 2) = H[1] + H[2]$~~

Solution

Problem 1 – Going to parties

Now for base cases. If using $\mathrm{OPT}(j, s)$:

O

O

O

Base cases involving 0 are usually easier.

Careful: Not always OPT(0) = 0! But it will usually be true whenever optimizing for a sum or total of something.

We have many equivalent options:

$\mathrm{OPT}(0, s) = 0$
~~$\mathrm{OPT}(1, 2) = H[1]$~~
✓

$\mathrm{OPT}(0, s) = 0$
~~$\mathrm{OPT}(-1, 0) = 0$~~
✓

$\mathrm{OPT}(1, 0) = 0$
$\mathrm{OPT}(1, 1) = H[1]$
$\mathrm{OPT}(1, 2) = H[1]$
~~$\mathrm{OPT}(2, 2) = H[1] + H[2]$~~
△

Solution

Problem 1 – Going to parties

Now for base cases. If using $\mathrm{OPT}(j, s)$:

$$\mathrm{OPT}(j,0) = \max\big(\mathrm{OPT}(j-1,0), \mathrm{OPT}(j-1,1), \mathrm{OPT}(j-1,2)\big)$$
$$\mathrm{OPT}(j,1) = \mathrm{OPT}(j-1,0) + H[j]$$
$$\mathrm{OPT}(j,2) = \mathrm{OPT}(j-2,0) + H[j-1] + H[j]$$

We have mar

It's nice to **1-index the input** for dynamic programming.

Reserve 0 for the empty input and easy base case, so that we can work with $H[j]$ on day j.

$\mathrm{OPT}(0, s$ $[1]$

~~$\mathrm{OPT}(1, 2$~~ $[1]$

~~$\mathrm{OPT}(2,2) = H[1] + H[2]$~~

✓ ✓ △

Solution

Problem 1 – Going to parties

Likewise, if using $\mathrm{OPT}(j)$:

$$\mathrm{OPT}(j) = \max\begin{pmatrix} \mathrm{OPT}(j-1), \\ \mathrm{OPT}(j-2) + H[j], \\ \mathrm{OPT}(j-3) + H[j-1] + H[j] \end{pmatrix}$$

We also have many equivalent options.

$$\begin{aligned} \mathrm{OPT}(0) &= 0 \\ \mathrm{OPT}(1) &= H[1] \\ \mathrm{OPT}(2) &= H[1] + H[2] \end{aligned}$$

✓

$$\begin{aligned} \mathrm{OPT}(-2) &= 0 \\ \mathrm{OPT}(-1) &= 0 \\ \mathrm{OPT}(0) &= 0 \end{aligned}$$

✓

$$\begin{aligned} \mathrm{OPT}(1) &= H[1] \\ \mathrm{OPT}(2) &= H[1] + H[2] \\ \mathrm{OPT}(3) &= \max\begin{pmatrix} H[1] + H[2], \\ H[1] + H[3], \\ H[2] + H[3] \end{pmatrix} \end{aligned}$$

△

Problem 1 – Going to parties

In dynamic programming, the pseudocode will end up being a fairly direct translation of the recurrence, so we'll do the proof first. In this class, we will **focus on just proving the recursive case**. A complete formal proof is, of course, induction.

e) Prove your recurrence to be correct.

Feel free to work with the people around you!

Previous Solution

Problem 1 – Going to parties

In dynamic programming, the pseudocode will end up being a fairly direct translation of the recurrence, so we'll do the proof first. In this class, we will **focus on just proving the recursive case**. A complete formal proof is, of course, induction.

e) Prove your recurrence to be correct.

For consistency, please try the $\text{OPT}(j)$ version.

$$\text{OPT}(j) = \max\begin{pmatrix} \text{OPT}(j-1), \\ \text{OPT}(j-2) + H[j], \\ \text{OPT}(j-3) + H[j-1] + H[j] \end{pmatrix}$$

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

Cases based on when we last slept. We must have slept within the last 2 days, otherwise the schedule violates the excessive partying condition.

Case 1: Sleep today. We get nothing today. All schedules on $1..j-1$ are compatible with sleeping today, so the maximum happiness we can get in this case is $\mathrm{OPT}(j-1)$.

Case 2: Slept yesterday. Then we have no reason not to party today, and all schedules on $1..j-2$ are compatible with sleeping yesterday, so $\mathrm{OPT}(j-2)+H[j]$.

Case 3: Slept 2 days ago. Then we had no reason not to party yesterday, nor today, and all schedules on $1..j-3$ are compatible, thus $\mathrm{OPT}(j-3)+H[j-1]+H[j]$.

Overall, the maximum of three cases is the maximum possible overall happiness.

Solution

Problem 1 – Going to parties

Cases based o st 2 days,
otherwise the

This is key! Mention this in your proofs!

This is what allows us to use $\mathrm{OPT}(j-2)$ when partying today, and fixes the problem we noticed initially when we attempted to use $\mathrm{OPT}(j-1)$.

Case 1: Sleep 1 are compatible
with sleeping ase is $\mathrm{OPT}(j-1)$.

Case 2: Slept yesterday. Then we have no reason not to party today, and all schedules on $1..j-2$ are compatible with sleeping yesterday, so $\mathrm{OPT}(j-2)+H[j]$.

Case 3: Slept 2 days ago. Then we had no reason not to party yesterday, nor today, and all schedules on $1..j-3$ are compatible, thus $\mathrm{OPT}(j-3)+H[j-1]+H[j]$.

Overall, the maximum of three cases is the maximum possible overall happiness.

Implementation details

Problem 1 – Going to parties

Even though we're use a recurrence relation, **do not call your function recursively**! Calling the function recursively can lead to blowing up the running time, so we need to consider how to remember the solutions to subproblems.

There are two steps:

- State the **parameters** for your subproblems and what kind of structure you will use to store them.
- Describe the **order** for evaluating your subproblems.

f) Give this a try.

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

- State the **parameters** for your subproblems and what kind of structure you will use to store them.

Parameters are j from 0 to n and $s = 0, 1, 2$. We will store OPT in an $(n + 1) \times 3$ array.

OR

Parameters are j from 0 to n. We will store OPT in an array of length $n + 1$.

OR

other options depending on choice of base cases (j from -2 to n, or -1 to n, or 1 to n)

Solution

Problem 1 – Going to parties

- Describe the **order** for evaluating your subproblems.

We will evaluate the base cases, then an outer loop with j from 1 to n, and in each iteration, we will evaluate the each $\mathrm{OPT}(j, s)$ for $s = 0, 1, 2$.

OR

We will evaluate the base cases, then each $\mathrm{OPT}(j)$ for j from 3 to n.

OR

other options depending on choice of base cases (j from 2 to n, etc.)

Problem 1 – Going to parties

g) Write the pseudocode for your iterative algorithm.

Feel free to work with the people around you!

Previous Solution

Problem 1 – Going to parties

g) Write the pseudocode for your iterative algorithm.

For consistency, please try the following version:

$$\mathrm{OPT}(j) = \max\begin{pmatrix} \mathrm{OPT}(j-1), \\ \mathrm{OPT}(j-2) + H[j], \\ \mathrm{OPT}(j-3) + H[j-1] + H[j] \end{pmatrix}$$

$$\mathrm{OPT}(0) = 0$$

$$\mathrm{OPT}(1) = H[1]$$

$$\mathrm{OPT}(2) = H[1] + H[2]$$

Feel free to work with the people around you!

Solution

Problem 1 – Going to parties

g) Write the pseudocode for your iterative algorithm.

```
1. Let OPT be a zero-indexed array of length $n + 1$ initialized to anything.
2. Let $\mathrm{OPT}[0] = 0$, $\mathrm{OPT}[1] = H[1]$, and $\mathrm{OPT}[2] = H[1] + H[2]$.
3. For $j = 3..n$,
    a. Let $\mathrm{OPT}[j] = \max\begin{pmatrix}\mathrm{OPT}[j-1], \\ \mathrm{OPT}[j-2] + H[j], \\ \mathrm{OPT}[j-3] + H[j-1] + H[j]\end{pmatrix}$.
4. Return $\mathrm{OPT}[n]$.
```

Extra question: If you use $\mathrm{OPT}[j, s]$, what should you return?

$$\max(\mathrm{OPT}[j,0], \mathrm{OPT}[j,1], \mathrm{OPT}[j,2])$$

Problem 1 – Going to parties

h) What is the running time of your algorithm?

Solution

Problem 1 – Going to parties

h) What is the running time of your algorithm?

It is one for loop with constant work per iteration, so $O(n)$.

Problem 1 – Going to parties

Sometimes, you will be asked to **return the optimal object**, rather than the optimal value. However, doing it naively can cost you.

- A first idea might be to track the optimal object at each j, instead of the optimal value. In today's problem, this will use $O(n^2)$ total time and space, very bad!
- Thus, we usually try to **backtrack to find the optimal object** after finding the optimal value.
    - You may need to leave some hints for yourself to know where to go.

i) How would you modify the algorithm if you were asked to return the optimal party schedule?

Feel free to work with the people around you!

Problem 1 – Going to parties

i) How would you modify the algorithm if you were asked to return the optimal party schedule?

While processing each day j, additionally remember an arrow pointing to which $\mathrm{OPT}(j-k)$ the optimal solution uses (for some $k = 1, 2, 3$). Note that an arrow pointing to $\mathrm{OPT}(j-k)$ means we slept $k-1$ days ago. At the end of the algorithm, follow the arrows back to collect the set of days we slept.

Summary

- Start by thinking about **what to do with the last element.**
  - Ask: Why is $\mathrm{OPT}(j-1)$ not the thing you need? Then fix it.
  - In your proof, mention the key features of your solution that allow you to call previous iterations of OPT.
- For implementation, **determine an execution order** that allows you to use memory instead of recursive calls.
- If asked to return the object, maintain arrows for backtracking.

Thanks for coming to section this week!