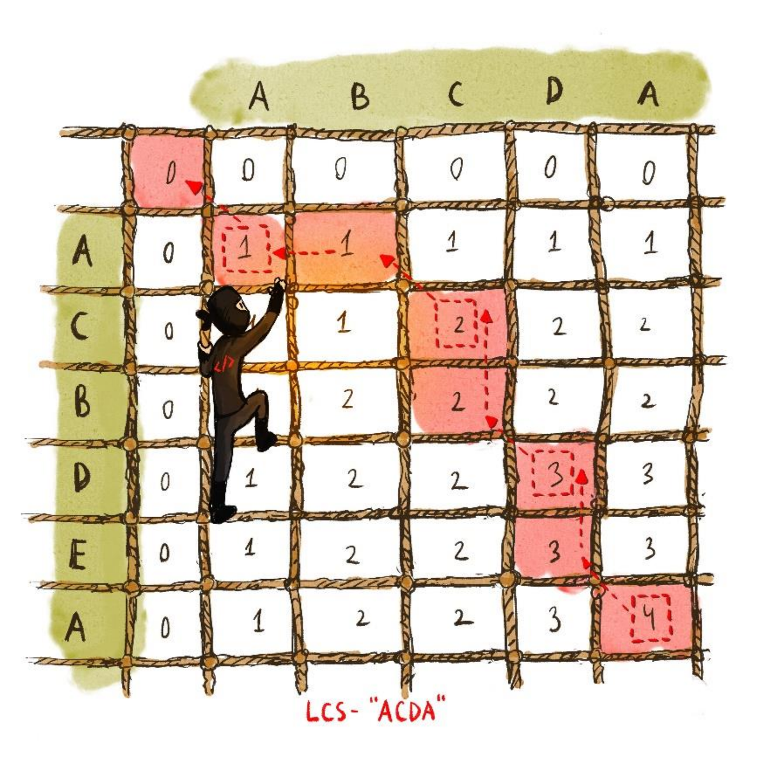
CS218: Design And Analysis Of Algorithms

Dynamic Programming III



Yan Gu

Midterm info

- Time: 11:05am-1:50pm (165 minutes) on Feb 13
- Location: WCH 205/206
- Preparation: 2-page double-sided letter-size handwritten cheat-sheet
- All the standard rules for in-person exams apply here

- Grading: The exam contains 22+3 points. You don't need to finish all of the problems (you could, of course). Your final score will be

min(your basic score, 20) + your bonus score

Problem info

- 22+3 points in total

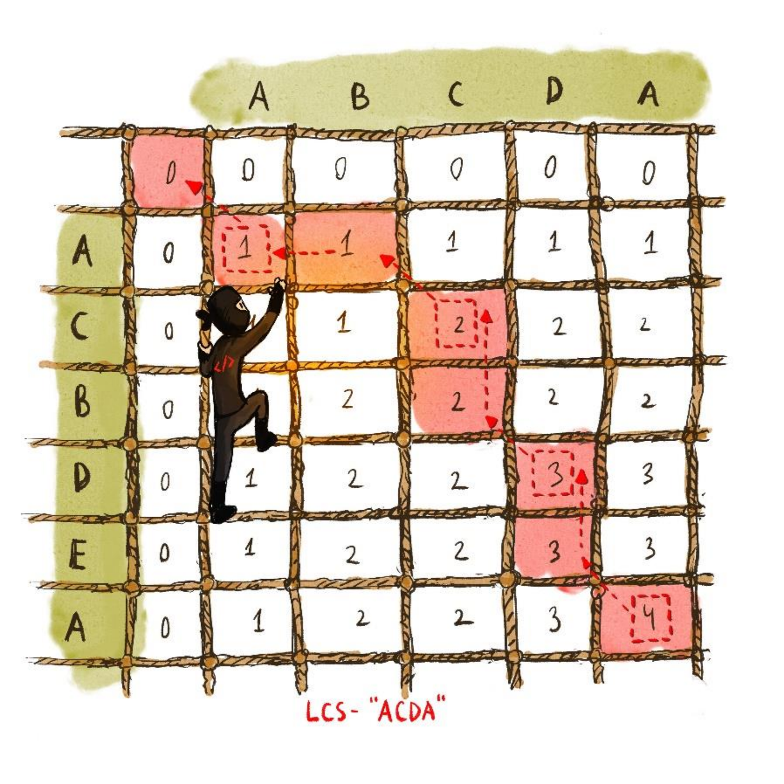
No.	Problem	Basic Pts	Bonus Pts	Candies
1	Multiple Choices	/6	0	0
2	Basic DP	/4	/1	/1
3		/4	/1	/1
4		/5	/1	/1
5		/3	0	0
	Total	/22	/3	/3

General suggestions

Useful topics include but are not limited to:

- **For greedy:**
  - Understand the proofs of the problems we covered in class
  - Read the proofs in the textbook, and those in CS141 lecture notes
  - Understand "greedy choice" and "optimal substructure"
- **For dynamic programming:**
  - Understand all DP recurrences for the problems we covered in class
  - Understand what the states in the DP recurrences are
  - Understand the boundary cases
  - Understand memorization
- **For data structures**
  - Understand range queries

CS218: Design And Analysis Of Algorithms



Dynamic Programming III

Yan Gu

Longest Increasing Subsequence (LIS) and Other Similar Problems

What is an increasing subsequence?

4	2	7	0	1	6	1	8	5	9

- Increasing subsequence:

	2				6		8		9

- Longest increasing subsequence (LIS):

			0	1	6		8		9

Why studying LIS?

- **The length of LIS reflect some intrinsic properties of the sequence**
  - Consider it as the “eigenvalue” of a sequence (LIS as the “eigenvector”)
  - Applications in many algorithms and quantum computing
- **Many similar DP algorithms are similar to the DP algorithm for LIS**
  - More examples are given later in this lecture

4	2	7	0	1	6	1	8	5	9

What are the states for LIS?

- Let l_i be the LIS for the first i element with i selected (as the last in LIS)
- What is the recurrence of LIS?

$$l_i = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + \mathbf{1}\} \end{cases}$$

- Why is it an optimal substructure?

4	2	7	0	1	6	1	8	5	9

Running the example input

$$l_i = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i,\, a_j<a_i} \{ l_j + \mathbf{1} \} \end{cases}$$

l_i:	1	1	2	1	2	3	2	4	3	5
	4	2	7	0	1	6	1	8	5	9

Running the example input

$$\boldsymbol{l_i} = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i,\, a_j<a_i} \{\boldsymbol{l_j} + \mathbf{1}\} \end{cases}$$

l_i:	**1**	**2**	**3**	**4**	**5**
	0	**1**	**6**	**8**	**9**

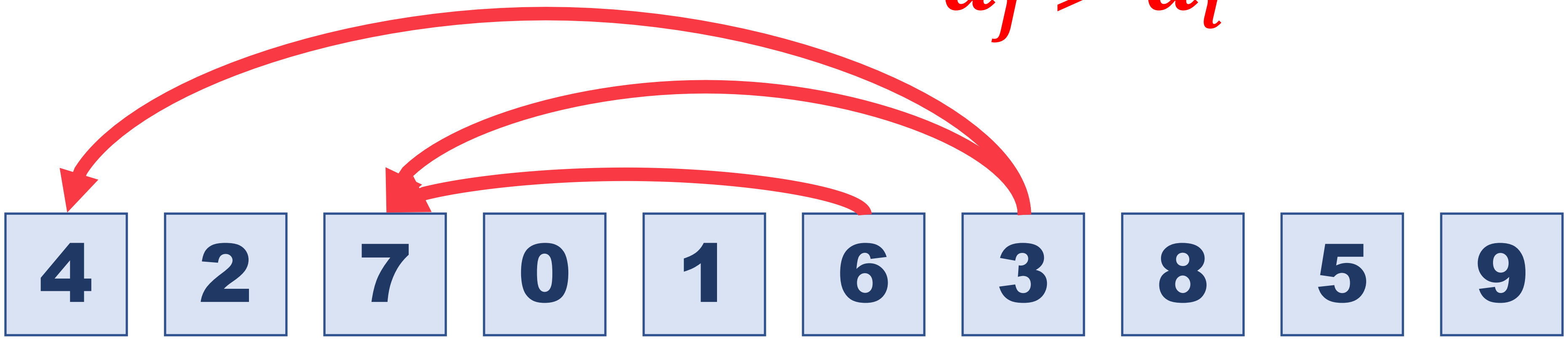
LIS and similar problems

$$\boldsymbol{l_i} = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i, a_j<a_i} \{\boldsymbol{l_j} + \mathbf{1}\} \end{cases}$$

Longest decreasing subsequence?

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i,\cancel{a_j<a_i}} \{l_j + 1\} \end{cases}$$

$a_j > a_i$



Longest increasing subsequence with gap ≥ 3?

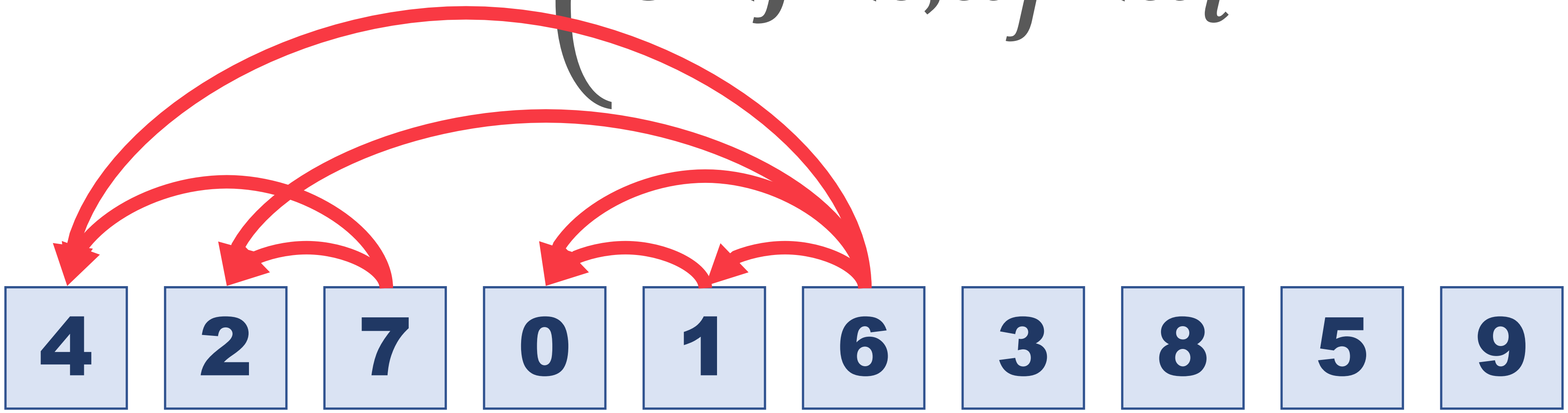
$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i,\cancel{a_j<a_i}} \{l_j + 1\} \end{cases}$$

$$a_j \leq a_i - 3$$

4	2	7	0	1	6	3	8	5	9

Increasing subsequence with MAX SUM?

$$l_i = \max \begin{cases} \cancel{1}\ \color{red}{a_i} \\ \max\limits_{0<j<i,a_j<a_i} \{l_j + \cancel{1}\} \quad \color{red}{a_i} \end{cases}$$



4 2 11 0 1 10

What is the time complexity of LIS?

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O\left(n^2\right)$ cost in total

Can we do better?

$$l_i = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i, a_j<a_i} \{ l_j + \mathbf{1} \} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total

Optimize LIS algorithm to $O(n \log n)$

LIS DP recurrence

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j + 1\} \end{cases}$$

- n element, each takes $O(n)$ time to compute, so $O(n^2)$ cost in total

$$l_i = \max \begin{cases} 1 \\ \max\limits_{0<j<i, a_j<a_i} \{l_j\} + 1 \end{cases}$$

- We need to find, for all l_j before l_i, with $a_j < a_i$, which is the largest value

l_i:	1	1	2	1	2	?				
a_i:	4	2	7	0	1	6	1	8	5	9

l_i:	1	1	2	1	2	3	?			
a_i:	4	2	7	0	1	6	1	8	5	9

l_i:	1	1	2	1	2	3	2	?		
a_i:	4	2	7	0	1	6	1	8	5	9

l_i:	1	1	2	1	2	3	2	4	?	
a_i:	4	2	7	0	1	6	1	8	5	9

l_i:	1	1	2	1	2	3	2	4	3	
a_i:	4	2	7	0	1	6	1	8	5	9

LIS – DP recurrence

- $\boldsymbol{l_i} = \max \begin{cases} \mathbf{1} \\ \max\limits_{\boldsymbol{0<j<i, a_j<a_i}} \{\boldsymbol{l_j}\} + \mathbf{1} \end{cases}$

- **When processing $\boldsymbol{i}$**
  - For all other elements that have been processed (all $j < i$)
  - We only consider when $a_j < a_i$
  - Find the largest l_j

- **A range-max query?**

LIS + range query

- **Assume we have an ADT D that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key < k, find the maximum value

Key	4	7	8	9	10	12	13	15	16	20	25
value	4	7	9	2	2	11	9	17	10	3	2

range_max(18) = 17

LIS + range query

$$\boldsymbol{l_i} = \max \begin{cases} \mathbf{1} \\ \max\limits_{0<j<i, a_j<a_i} \{\boldsymbol{l_j}\} + \mathbf{1} \end{cases}$$

- **When processing $\boldsymbol{i}$**
  - We need to look at all $\boldsymbol{j}$ before $\boldsymbol{i}$ (processed $\boldsymbol{j}$) and $\boldsymbol{a_j} < \boldsymbol{a_i}$
  - Find the largest $\boldsymbol{l_j}$
- **Assume we have an ADT $\boldsymbol{D}$ that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key < k, find the maximum value
  - Key: a_j, value: l_j
- **When processing $\boldsymbol{i}$**
  - Call range_max(a_i) on D, get v^*, let $l_i = v^* + 1$
  - Insert (a_i, l_i) to D
  - Go to $i + 1$

Example

- **Current stored in data structure:**
  - (0, 1), (1, 2), (2, 1), (4, 1), (9, 2)
- **Range_max query:**
  - What is the largest value for all keys smaller than 6??
- $l[6] = 2 + 1 = 3$
- **Insert (6, 3) to the data structure**
  - (0, 1), (1, 2), (2, 1), (4, 1), (6, 3), (9, 2)

l_i:	1	1	2	1	2	3				
a_i:	4	2	9	0	1	6	2	7	5	9

Example

- **Current stored in data structure:**
  - (0, 1), (1, 2), (2, 1), (4, 1), (6, 3), (9, 2)
- **Range_max query:**
  - What is the largest value for all keys smaller than 2??
- $l[7] = 2 + 1 = 3$
- **Insert (2, 3) to the data structure (can update (2,1))**
  - (0, 1), (1, 2), (2, 3), (4, 1), (6, 3), (9, 2)

l_i: 1 1 2 1 2 3 3

a_i: 4 2 9 0 1 6 2 7 5 9

Example

- **Current stored in data structure:**
  - (0, 1), (1, 2), (2, 3), (4, 1), (6, 3), (9, 2)
- **Range_max query:**
  - What is the largest value for all keys smaller than 7??
- $l[8] = 3 + 1 = 4$
- **Insert (7, 4) to the data structure**
  - (0, 1), (1, 2), (2, 2), (4, 1), (6, 3), (7, 4), (9, 2)

l_i:	1	1	2	1	2	3	3	4		
a_i:	4	2	9	0	1	6	2	7	5	9

LIS + range query

- **Assume we have an ADT D that can deal with range-max query**
  - Store key-value pairs
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key<k, find the maximum value
  - Key: a_i, value: l_i

- **To compute each value in $l[\cdot]$, we need:**
  - **A range_max query: $O(\log n)$ time**
  - **An insert operation: $O(\log n)$ time**
  - **Total running time $O(n \log n)$**

- **We can use an augmented tree and possibly other DS to implement D**

Another way for optimization

- Maintain a list B, where $B[i]$ is the smallest a_j that makes l_j to be i
- l_j can only be 1 to n

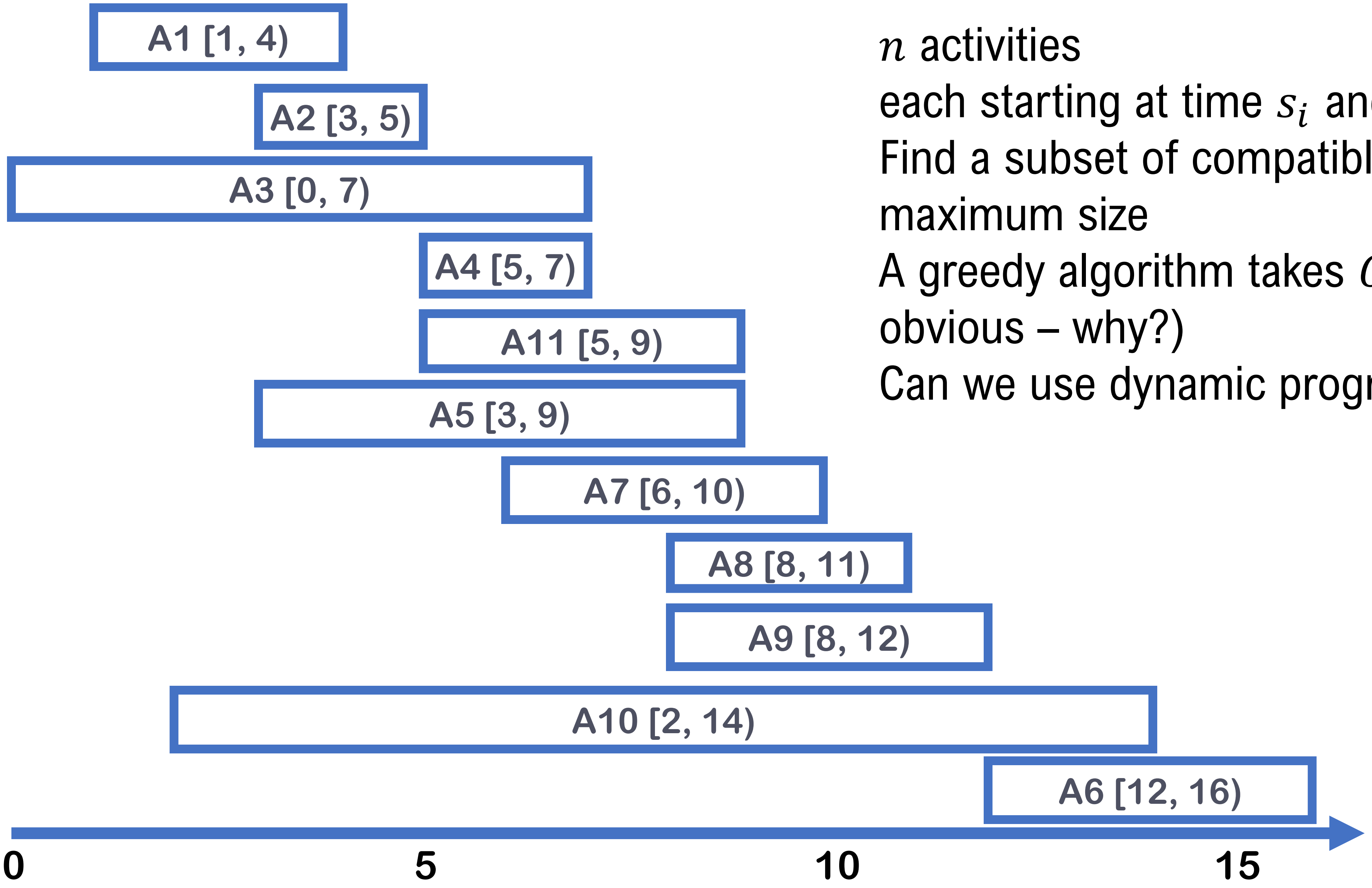
i:	1	2	3	4	5	6
$B[i]$:	0	1	5	6	9	$+\infty$

6

l_i:	1	1	2	1	2	3	2	4	3	5	4
a_i:	4	2	7	0	1	6	1	8	5	9	6

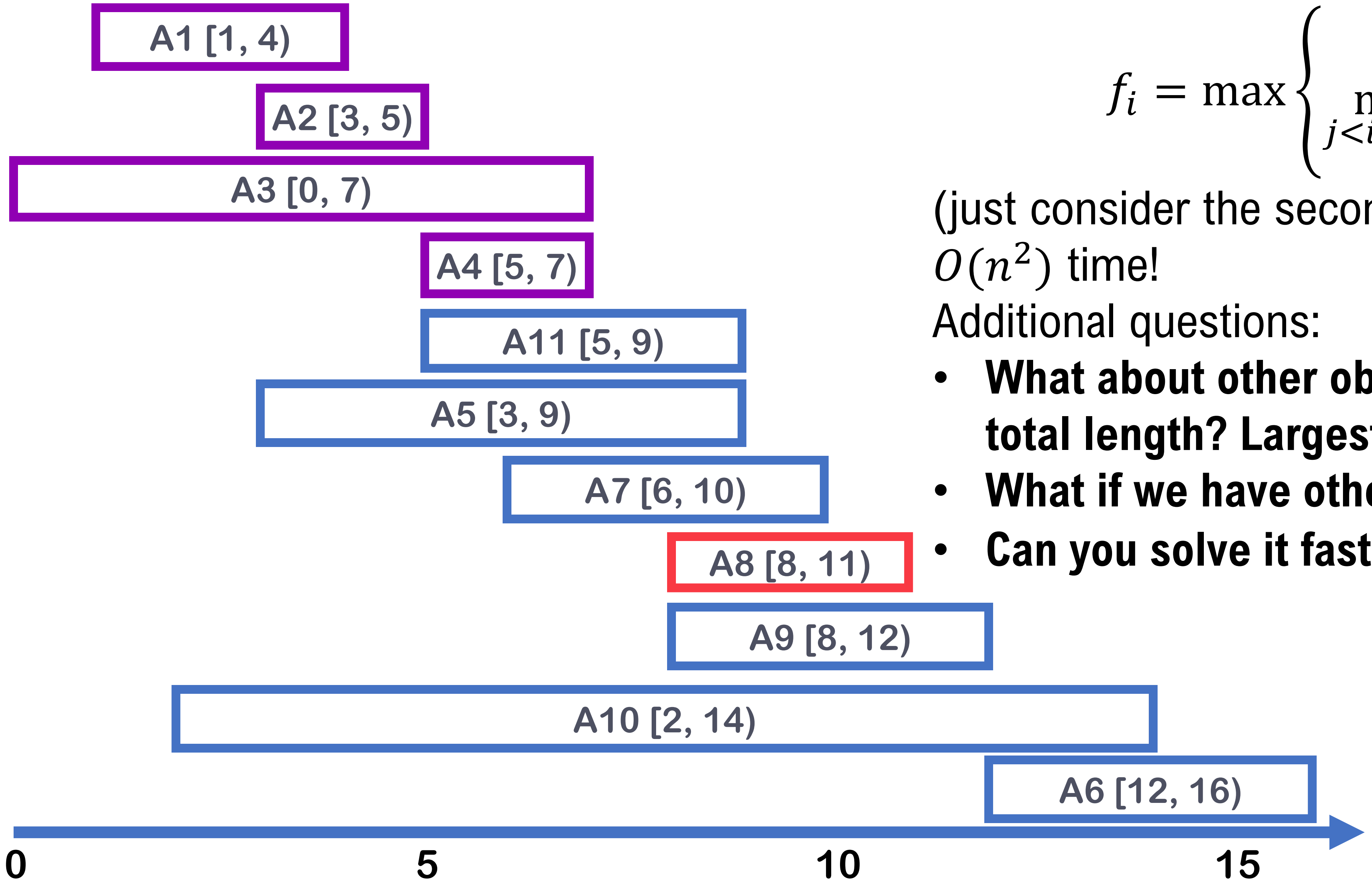
Activity selection and some variants

Revisit: activity selection



n activities
each starting at time s_i and ends at e_i
Find a subset of compatible activities with maximum size
A greedy algorithm takes $O(n)$ time (not very obvious – why?)
Can we use dynamic programming to solve it?

Activity selection



Let f_i be the longest length of the first i activities, **with the i-th activity selected**, then

$$f_i = \max \begin{cases} 1 \\ \max\limits_{j<i, e_j \le s_i} \{f_j + 1\} \end{cases}$$

(just consider the second last activity)

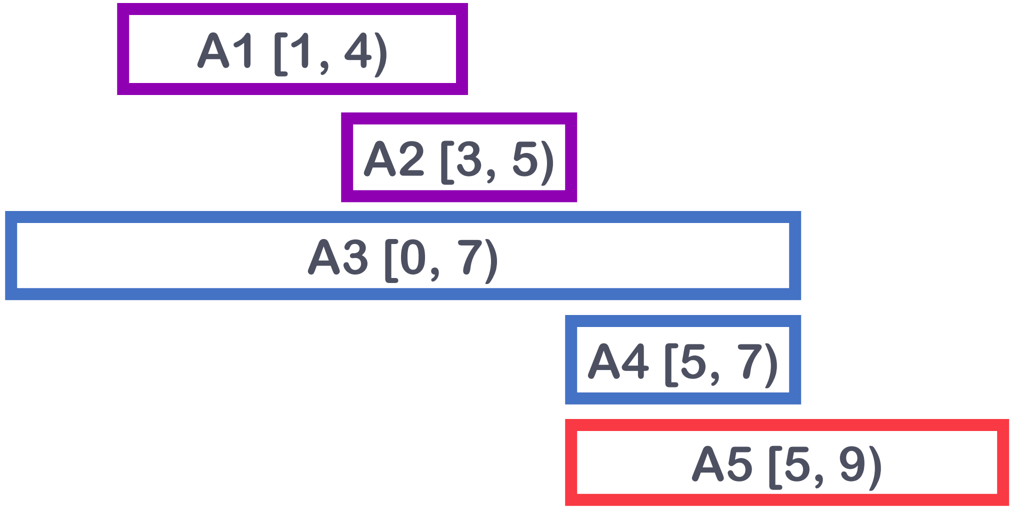
$O(n^2)$ time!

Additional questions:

- **What about other objectives? Maximize total length? Largest value? …**
- **What if we have other constraints?**
- **Can you solve it faster than $O(n^2)$?**

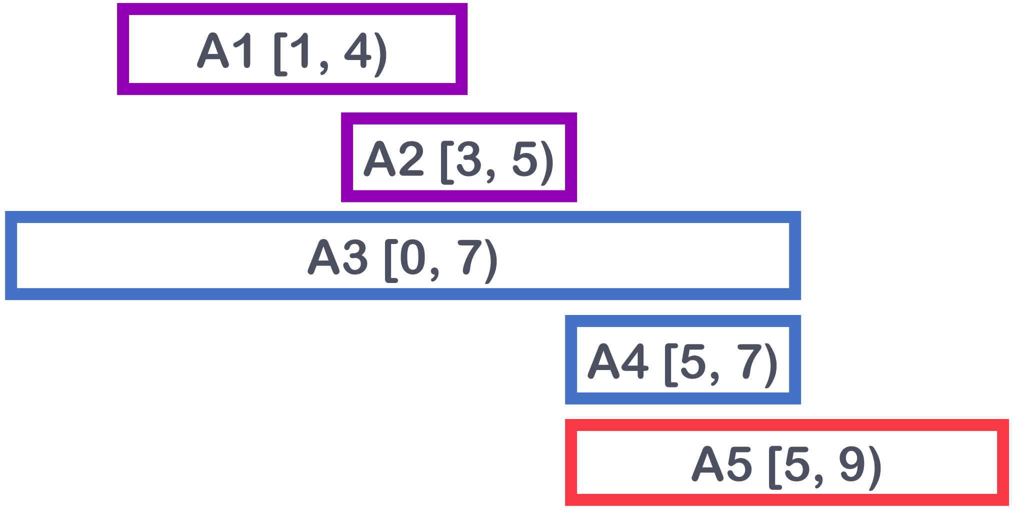
Activity selection: longest total length

- Let f_i be the longest length of the first i activities and with the i-th activity selected, then
- $f_i = \max\begin{cases} e_i - s_i \\ \max\limits_{j<i, e_j \le s_i} \{f_j + (e_i - s_i)\} \end{cases}$
- $\boldsymbol{O(n^2)}$



Activity selection: largest total value

- **Each activity has a value** v_i
- **Let** f_i **be the largest total value of the first** i **activities, with the** i**-th activity selected, then**
- $f_i = \max\begin{cases} v_i \\ \max\limits_{j<i, e_j \le s_i} \{f_j + v_i\} \end{cases}$
- $\boldsymbol{O(n^2)}$

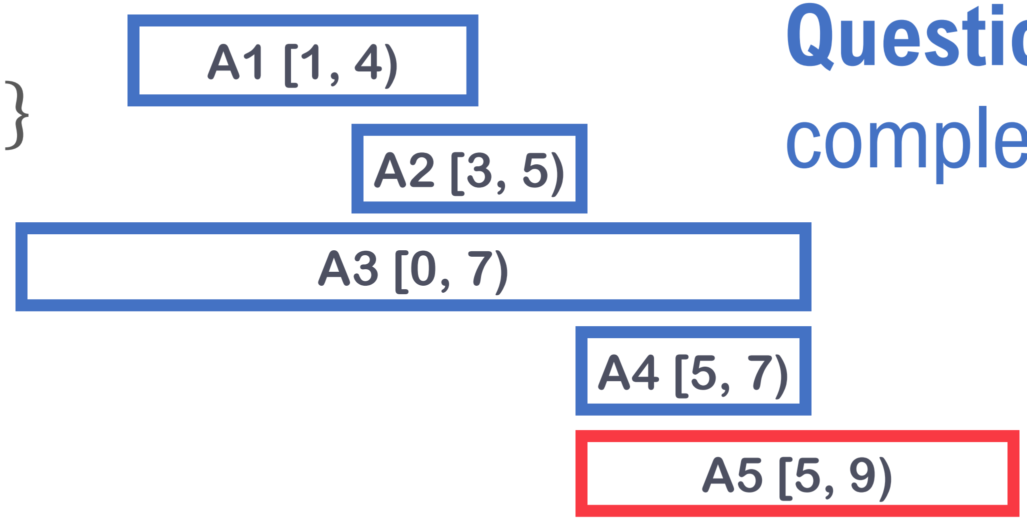


Conclusion:
For different **objectives**, we can just slightly modify DP recurrence

Question: More constraints? Faster than $O(n^2)$?

Activity selection: limited size, largest total value

- Each activity has a value v_i, can only choose s activities
- Let $f_{i,j}$ be the largest total value for first i activities, choosing j activities
- $f_{i,j} = \max \begin{cases} f_{i-1,j} \\ \max\limits_{k<i, e_k \le s_i} \{f_{k,j-1} + v_i\} \end{cases}$
- $\boldsymbol{f_{i,0} = 0, f_{0,j} = 0}$
- $\boldsymbol{O(n^2 s)}$ time



Question: Better complexity?

Conclusion:
By adding **another dimension**, we can deal with one more restriction, but more expensive in complexity

Activity selection: can we make it more efficient?

- **Let's assume we are dealing with the maximum total value**
- $f_i = \max \begin{cases} v_i \\ \max\limits_{j<i, e_j \le s_i} \{f_j + v_i\} = \max\limits_{j<i, e_j \le s_i} \{f_j\} + v_i \end{cases}$
- **For all activities before i, with $e_j < s_i$, we want the largest f_j**
- **A range max query again!**

Activity selection: can we make it more efficient?

- $f_i = \max \begin{cases} v_i \\ \max\limits_{j<i, e_j \le s_i} \{f_j + v_i\} = \max\limits_{j<i, e_j \le s_i} \{f_j\} + v_i \end{cases}$
- **For all activities before i, with $e_j < s_i$, we want the largest f_j**
- **Assume we have an ADT D that can do range-max query on k-v pairs**
  - insert(k,v): add a new key-value pair
  - range_max(k): for all key<k, find the maximum value
  - Key: e_j, value: f_j
- **When processing i**
  - Find in D range_max(s_i), get v^*, let $f_i = v^* + v_i$
  - Insert (e_i, f_i) into D
  - Continue to $i + 1$

What can we learn from activity selection problem?

- **When we have different objectives, we can use similar idea, but slightly modify the DP recurrence**
  - Max number, max total length, max total value
- **When we have more restrictions, we can add another dimension**
  - $f_{i,j}$ for the first i activities (select i) and select j of them
- **When we use different state in DP recurrence, we can have different algorithms, with different complexity**
  - Use f_i for max value for the first i activities (select i)
  - Use $f_{i,j}$ for max value for the first i activities up to time j
- **If we want to make the algorithm more efficient, sometimes range-max/range-min query can help!**

What can we learn today's class?

- **For many DP recurrence with high computation cost, usually it's because we need to loop over a large candidate set to make the decision**
- **Since we are usually taking min/max, they can be viewed as range searches, and we have talked about data structures to do so efficiently**
  - For data structures in higher dimension, we will cover them in (my) CS 219
- **Other ways to optimize DP**
- **1: Decision Monotonicity**
  - We can get extra information from the decisions we have made previously
  - Narrow down the search space for a decision
  - May also use data structures to maintain such information
- **2: Parameterization**
  - Run time proportional to problem properties

Dynamic Programming – Summary again

- **OK, yes, it is hard!**
- **It is hard in that you need a LONG TIME and A LOT OF PRACTICE to get used to it**
- **But there is an "Aha! moment" when you then get to the point of designing many DP algorithm**
- **"First you hate ~~them~~ DP, then you get used to ~~them~~ DP. Enough time passes, you get so you depend on ~~them~~ DP."**
  - From *The Shawshank Redemption*
  - Got this comment from another course, but it's true for many "hard" learning processes. (although it means totally different things in the movie)
- **But again, you should be happy if you are learning something hard from school. The harder/more creative the skill is, the more irreplaceable you are at work.**

Dynamic Programming – Summary again

- **80-min lectures (or 80min*4 lectures) won't be sufficient for you to have a good understanding of DP (even more lectures won't be enough)**
- **What you need**
  - Some background (CS218 assumes you have basic knowledge about DP from prerequisite courses)
  - More reading (textbook chapters will be provided)
  - More practice (I can provide more problems for practice if any of you want)
- **Welcome to come to any of our OHs for help**

Readings

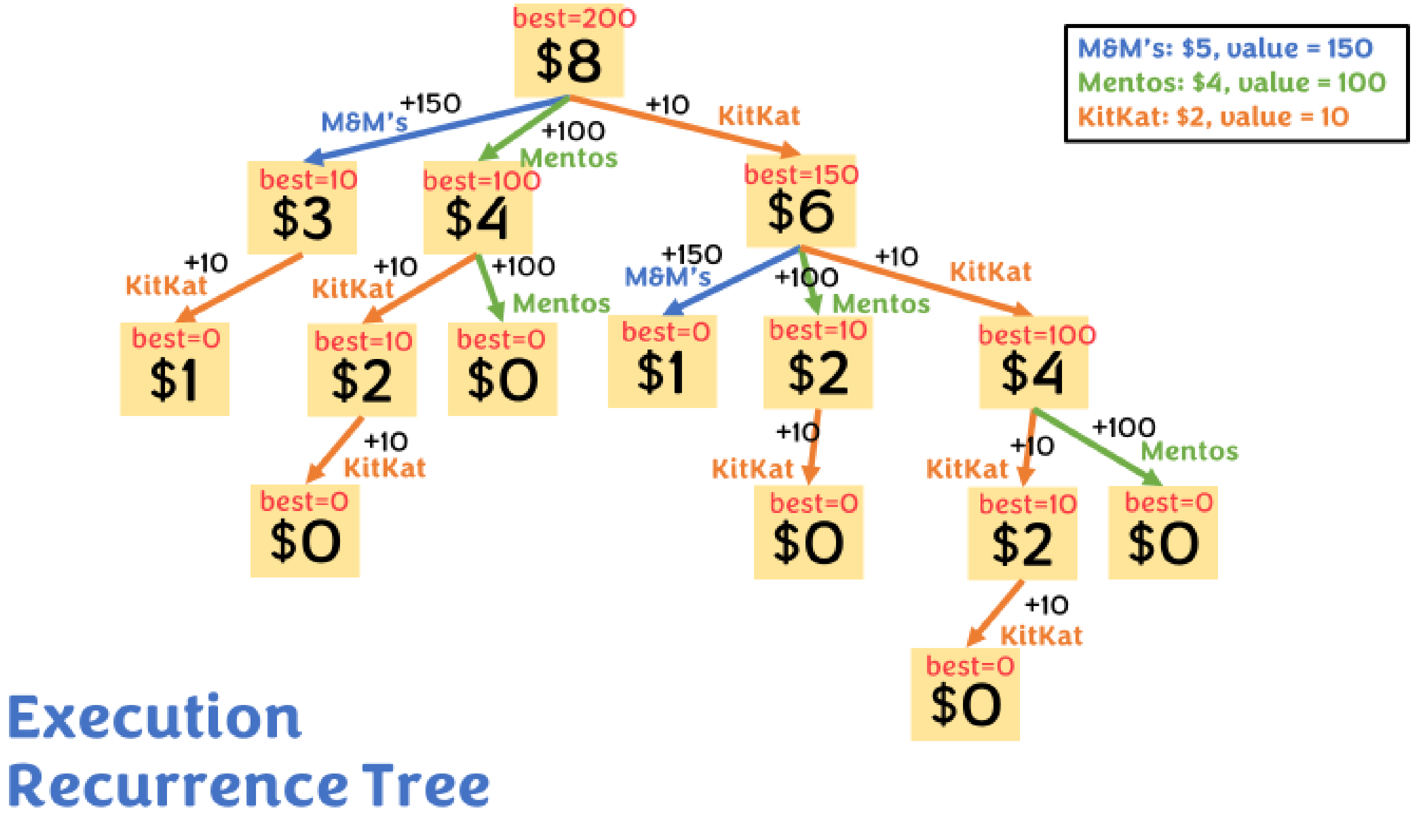
- **CLRS Sec. 15**
  - 15.1: Memorization (a similar walk-through example similar to our knapsack problem, about how to go from a “brute-force” algorithm to a “DP” algorithm)
  - 15.2 Matrix multiplication chain
  - 15.3 Elements of dynamic programming (you can read this part at last, as it is more “philosophic” and requires higher-level understanding in DP)
  - 15.4 Longest common subsequence
  - 15.5 Optimal binary search trees (one more example, not covered in class, but a good reading in general)
- **Another useful textbook: https://www.ics.uci.edu/~goodrich/teach/cs161/notes/dynamicp.pdf (chapter on DP)**
- **CS141 lecture notes (provided in previous classes. If you didn’t get it, you can come to my OH to get one copy)**
- **So far almost all topics about DP are covered in undergraduate classes**

More readings

- **CMU 15-451 and 15-210**
  - https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf
  - http://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/dp-notes.pdf (may be hard to understand if you are not familiar with functional languages)
  - https://www.cs.cmu.edu/~15451-s23/lectures/lec09-dp1.pdf
- **Stanford CS161**
  - https://stanford-cs161.github.io/winter2022/assets/files/lecture13-notes.pdf
  - https://stanford-cs161.github.io/winter2022/assets/files/lecture13-slides.pdf
- **MIT 6.006**
  - https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/pages/lecture-notes/ (Lecture notes are brief, may need to watch videos)
- **UCI CS161**
  - https://www.ics.uci.edu/~goodrich/teach/cs161/notes/dynamicp.pdf (another nice textbook to read: https://www.ics.uci.edu/~goodrich/teach/cs161/notes/)
- **CS141 materials are available in the Dropbox folder**
- **Many more (the examples of these course overlaps but generally differ from each other. DP is a broad topic.)**

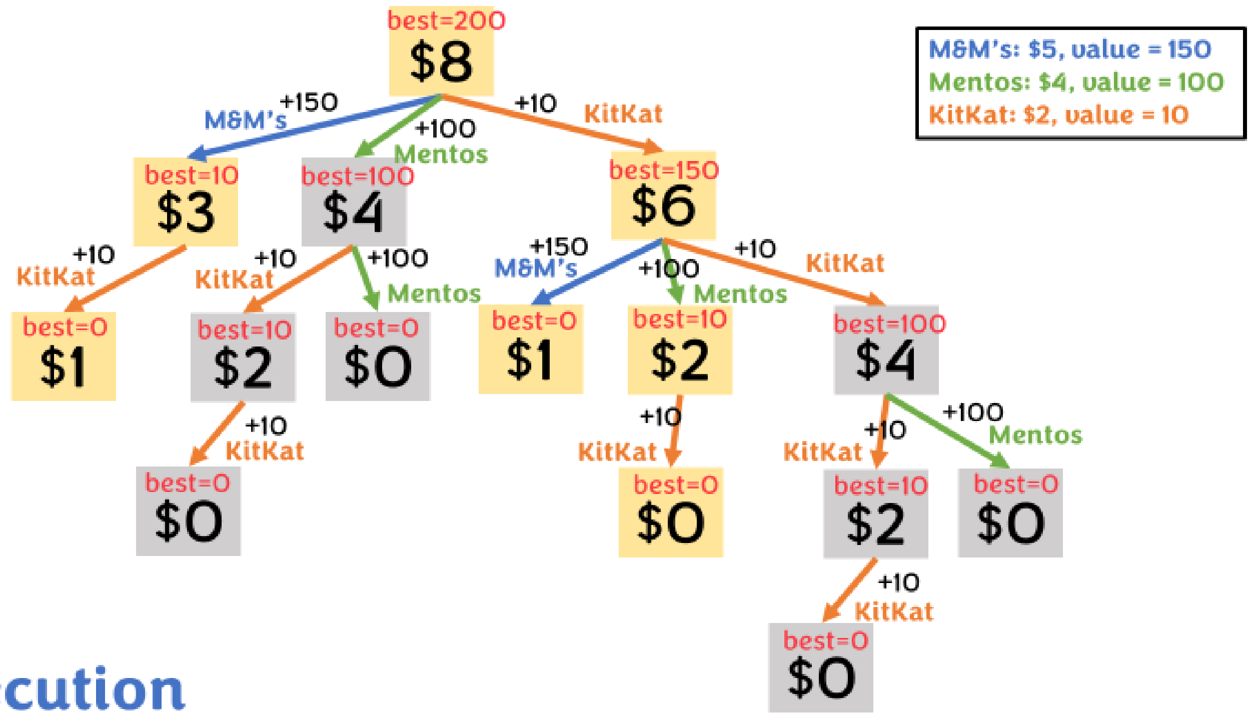
What is DP / what can be solve by DP / how to design DP algorithms?

- DP is an algorithmic “idea”, not a specific algorithm
- “optimal substructure” [need to solve subproblems]



What is DP / what can be solve by DP / how to design DP algorithms?

- "optimal substructure" [need to solve subproblems]
- Then DP is useful when there are "overlapping subproblems"



Execution Recurrence Tree

What is DP / what can be solve by DP / how to design DP algorithms?

- "optimal substructure" [need to solve subproblems]
- Then DP is useful when there are "overlapping subproblems"
- But there is no "formula" that works for all DP problems. There are common ideas that might be useful (Reading CLRS 15.3)

- In CLRS, "DP" comes before greedy
  - DP requires optimal substructure, greedy further requires greedy choice?
  - DP: find best decision
  - Greedy is a special DP, where you know your best decision is your greedy choice!

Buying gifts again

- Yihan is going to buy candies for 218 students
- Her budget is W dollars
- There are n candies in store, with price $p[i]$ each
- She doesn't want to buy the same candy twice
- She wants to buy **as many candies as possible**

Buying gifts again

- Yihan is going to buy candies for 218 students
- Her budget is W dollars
- There are n candies in store, with price $p[i]$ each
- She doesn't want to buy the same candy twice
- She wants to buy **as many candies as possible**
- **0/1 Knapsack problem! Weight = price. Value = 1 for each candy**

$$s[i,j] = \max \begin{cases} s[i-1,j] & \\ s[i-1,j-w_i]+1 & j \geq w_i \end{cases}$$

Don't buy the i-th item

Buy the i-th item

Buying gifts again

- Buy as many candies as possible with W dollars
- **0/1 Knapsack problem! Weight = price. Value = 1 for each candy**
- Since we can prove "greedy choice", we know we just need to buy the cheapest candies

If we sort candies by price:

$$s[i,j] = \max \begin{cases} \cancel{s[i-1,j] \qquad \text{Don't buy the i-th item}} \\ s[i-1, j-w_i] + 1 \quad j \geq w_i \qquad \text{Buy the i-th item} \end{cases}$$

- **But when value $\neq$ 1, "greedy choice" is not true any more. We have to use DP!**

Buying gifts again

- Yihan is going to buy candies for 218 students
- Her budget is s dollars
- There are n candies in store, with price $p[i]$ each
- She doesn't want to buy the same candy twice
- She wants to buy as many candies as possible
- 0/1 Knapsack problem! Weight = price. Value = 1 for each candy
- Since we can prove "greedy choice", we know we just need to buy the cheapest candies
- But when value $\neq 1$, "greedy choice" is not true any more. We have to use DP!

Buying gifts again

- Yihan is going to buy candies for 218 students
- Her budget is W dollars
- There are n candies in store, with price $p[i]$ each
- She can buy the same candy twice
- She wants to buy as many candies as possible

Buying gifts again

- Yihan is going to buy candies for 218 students
- Her budget is W dollars
- There are n candies in store, with price $p[i]$ each
- She can buy the same candy twice
- She wants to bu
- Unlimited knaps

as possible

The best value with $i - w_j$ weight

- $s[i] = \max \begin{cases} 0 \\ \max\limits_{\cancel{(w_j, v_j) \text{ is an item}}} \{s[i - w_j] + 1\} : \ i \geq w_j \end{cases}$

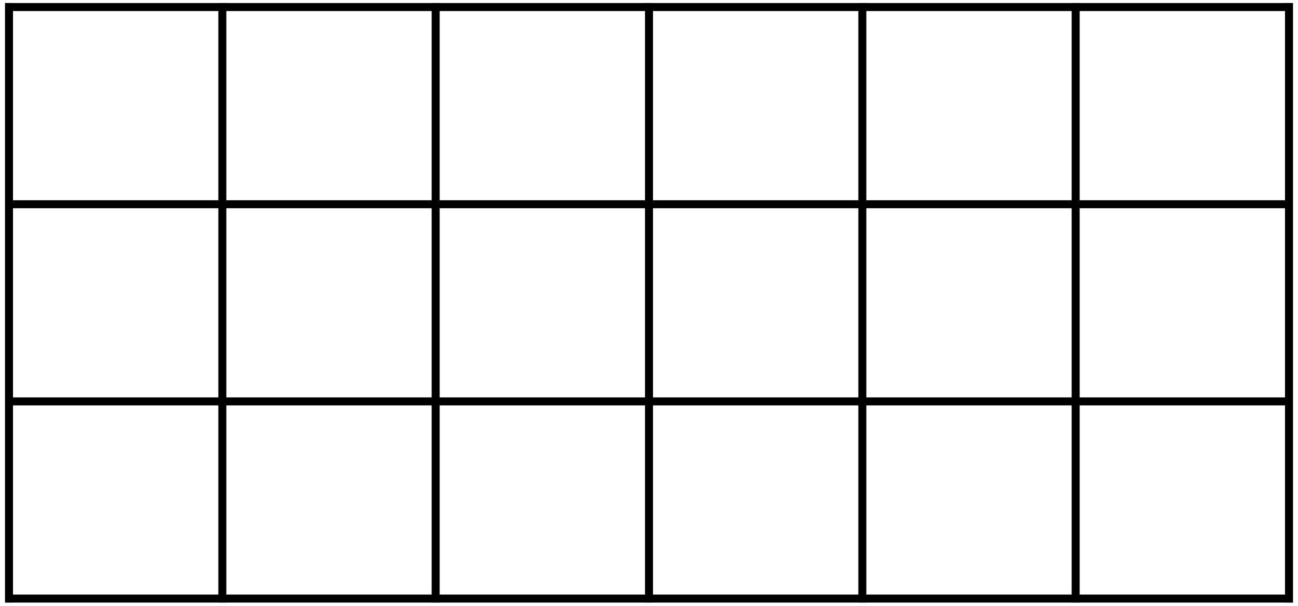
~~Trying all possible items~~

Only choose the one with the lowest price!

DP can be used in many more fun topics!

Bitmask DP

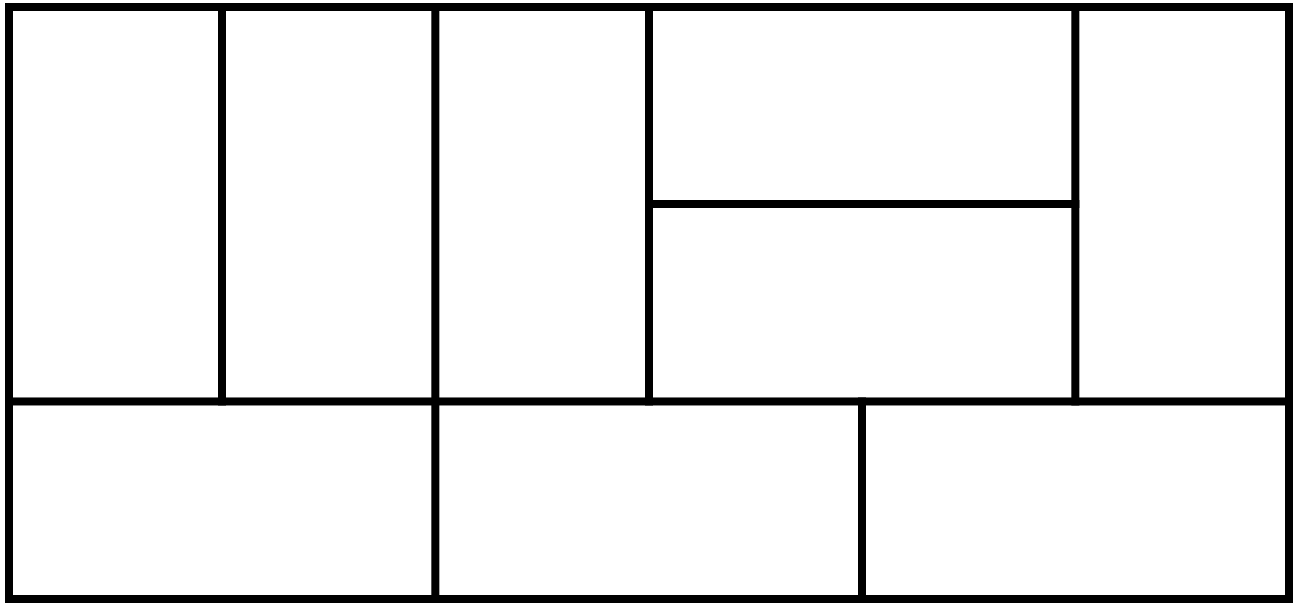
Counting 1*2 domino tiles on a 3*n grid



Counting 1*2 domino tiles on a 3*n grid

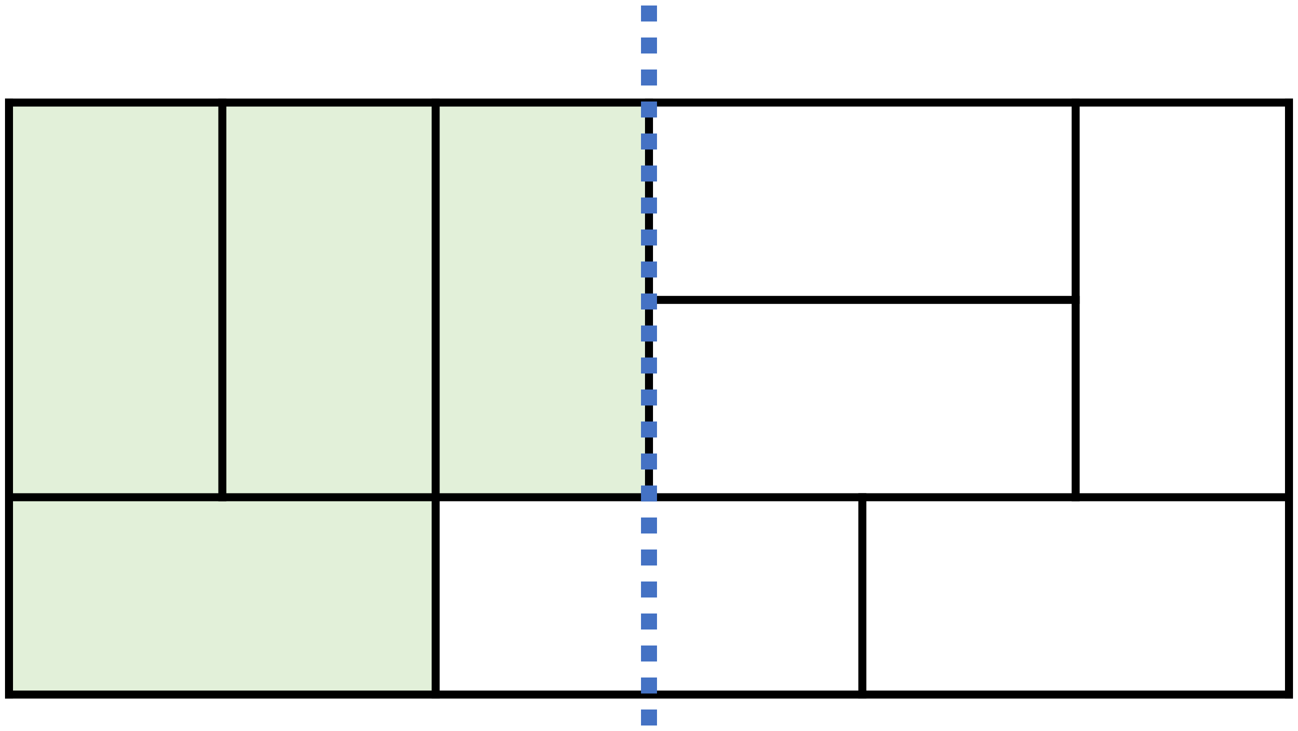
Counting 1*2 domino tiles on a 3*n grid

- You want to compute the total number of combinations that tessellate the entire grid, which can be pretty large



Counting 1*2 domino tiles on a 3*n grid

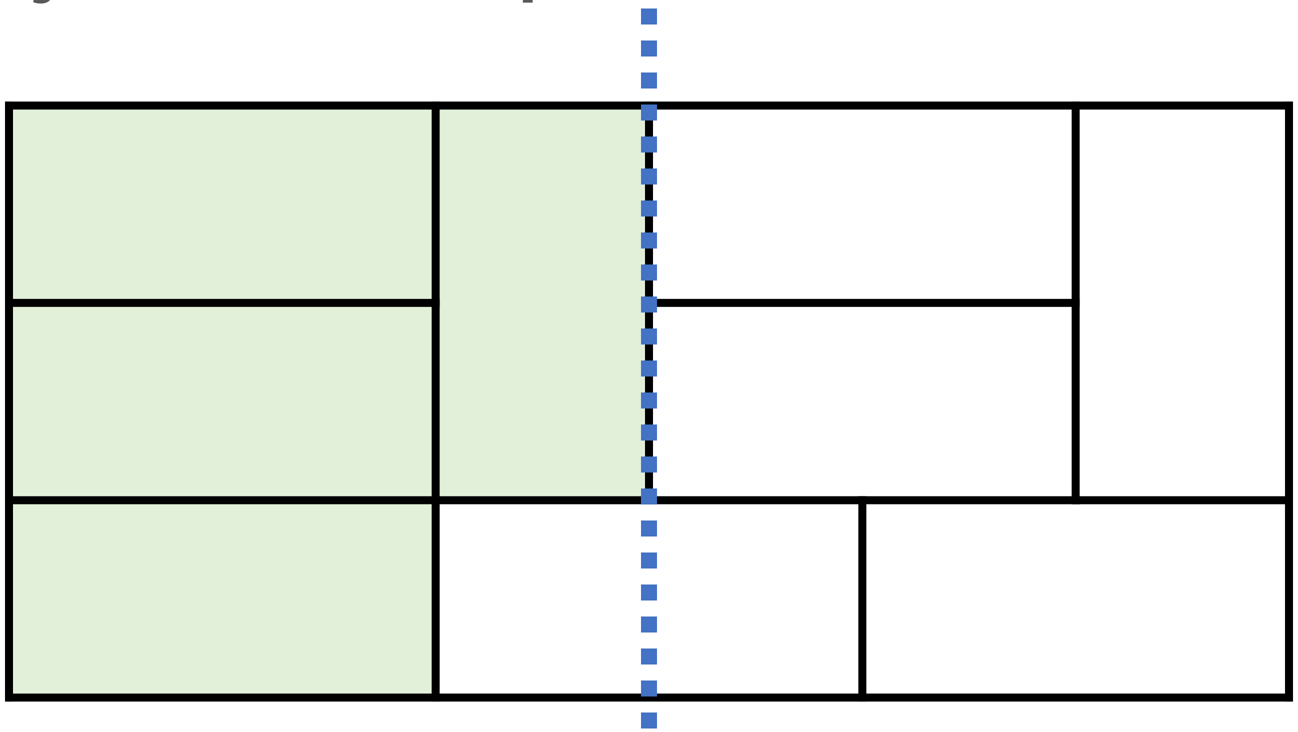
- Consider a snapshot here during a sweepline process



Counting 1*2 domino tiles on a 3*n grid

- Consider a snapshot here during a sweepline process

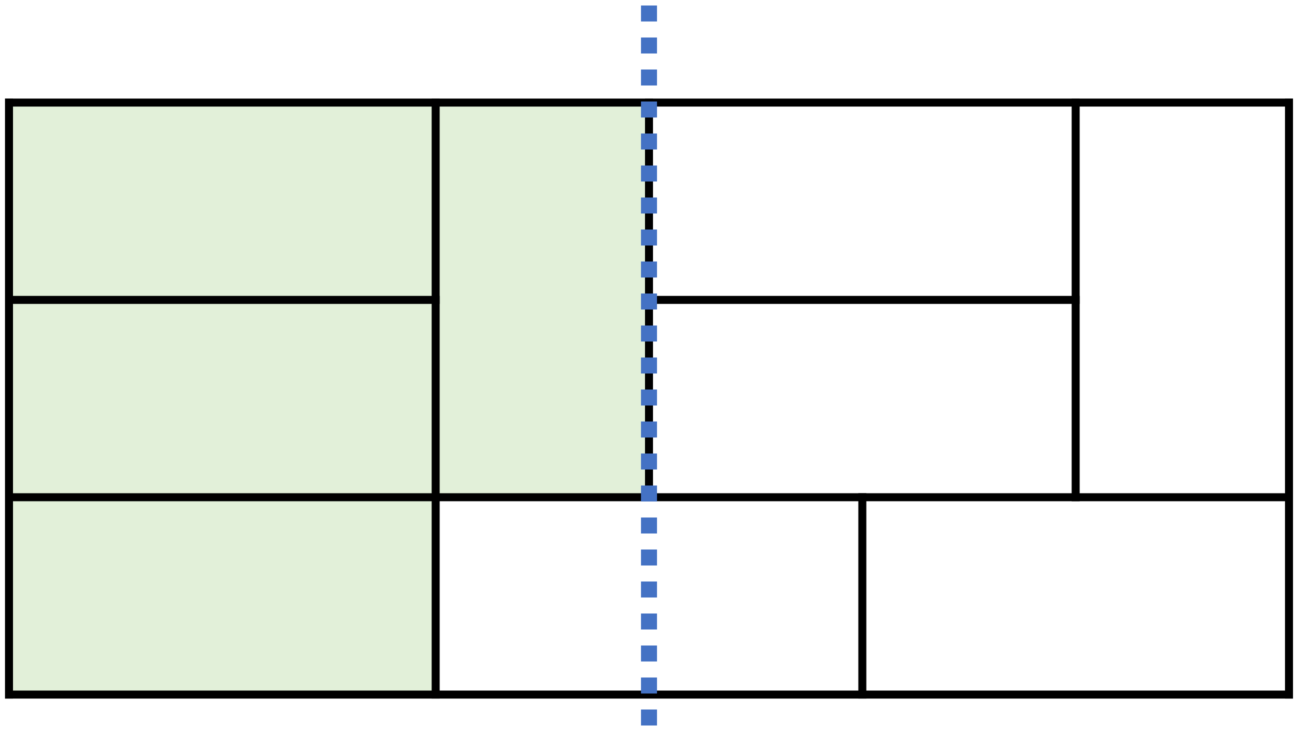
- Let's consider such a shape as our DP state: note that any cell with distance 1 away from the sweepline does not matter



Counting 1*2 domino tiles on a 3*n grid

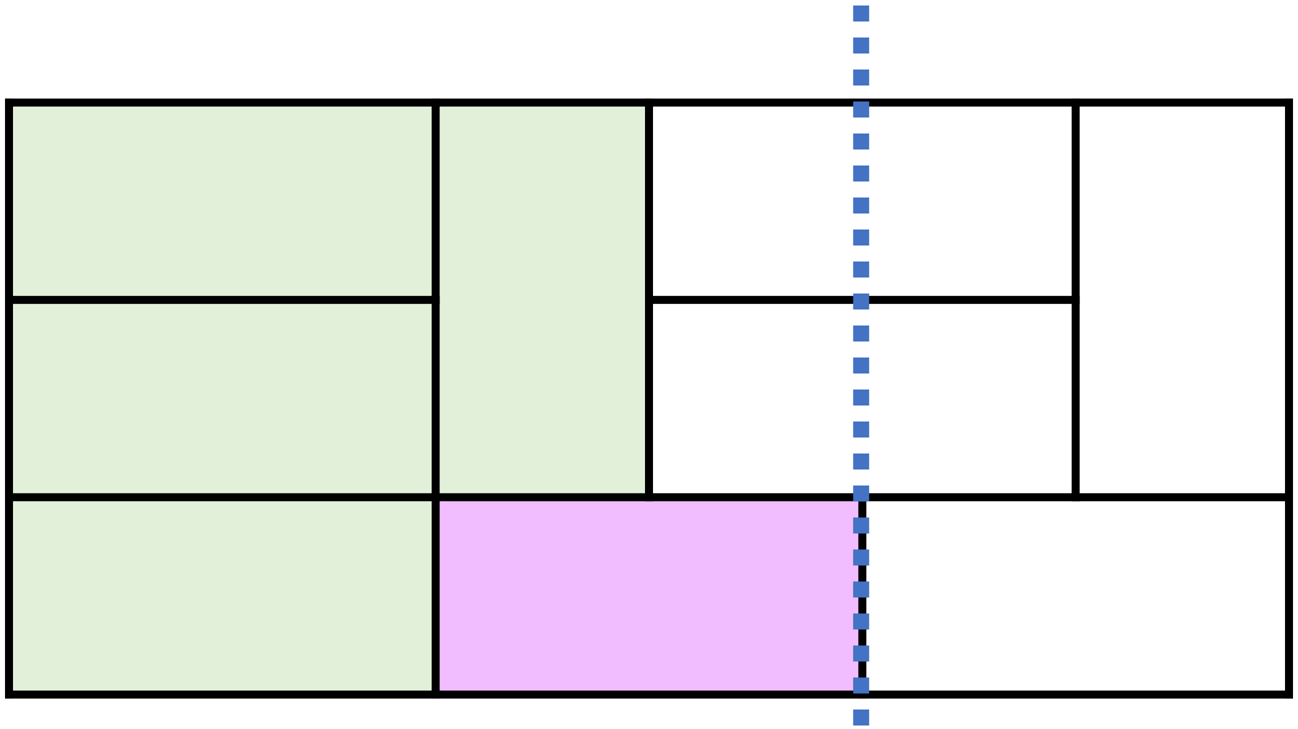
- Denote the state as DP[3, ‘110’]

- Generally, it will be DP[0..n, 0..7], and the final answer is DP[n,7]



Moving forward

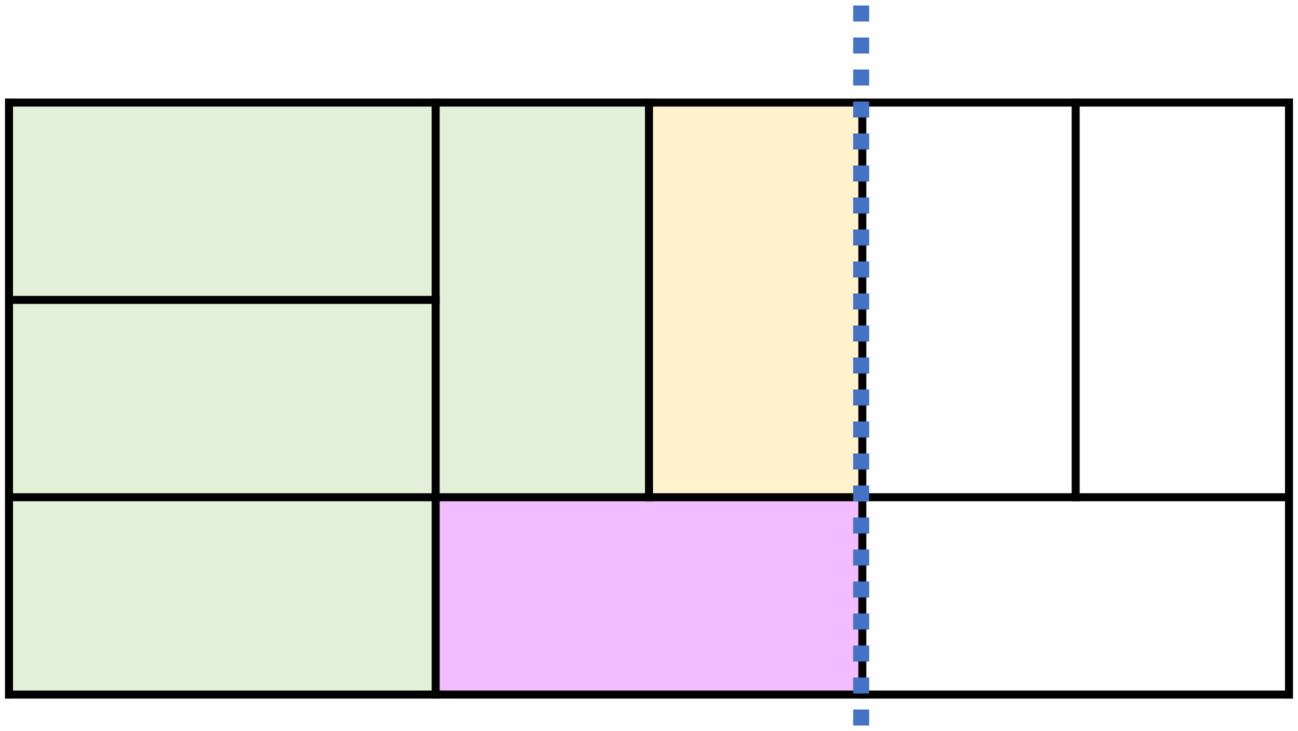
- All the empty slots have to be placed by a horizontal tile
- Can choose to put vertical tiles on the "current" tile



Moving forward

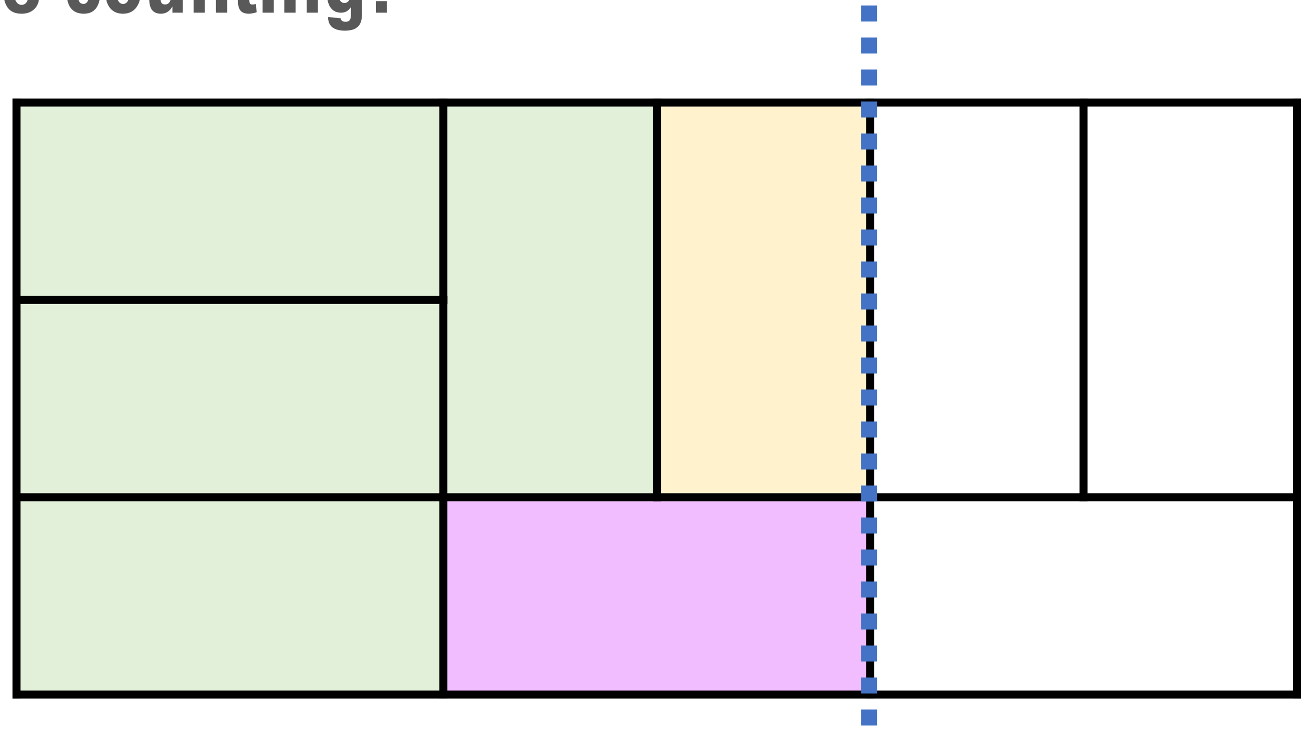
- All the empty slots have to be placed by a horizontal tile

- Can choose to put vertical tiles on the "current" tile



Moving forward

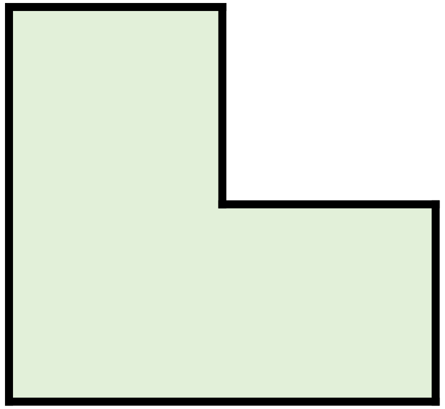
- Will need a mini-search process to extend a current state (e.g., DP[3, ‘110’]) to all possible future states (e.g., DP[4, ‘001’] and DP[4, ‘111’])
- Avoid double counting!



Summary for this simple tiling problem

- **Number of states:** $n \times 2^m$
- **Total work:** $n \times 2^m \times cost_{search} = n \times 2^{2m}$
- **Works fairly well for reasonable n and small m**
- **Details: how to apply the search**

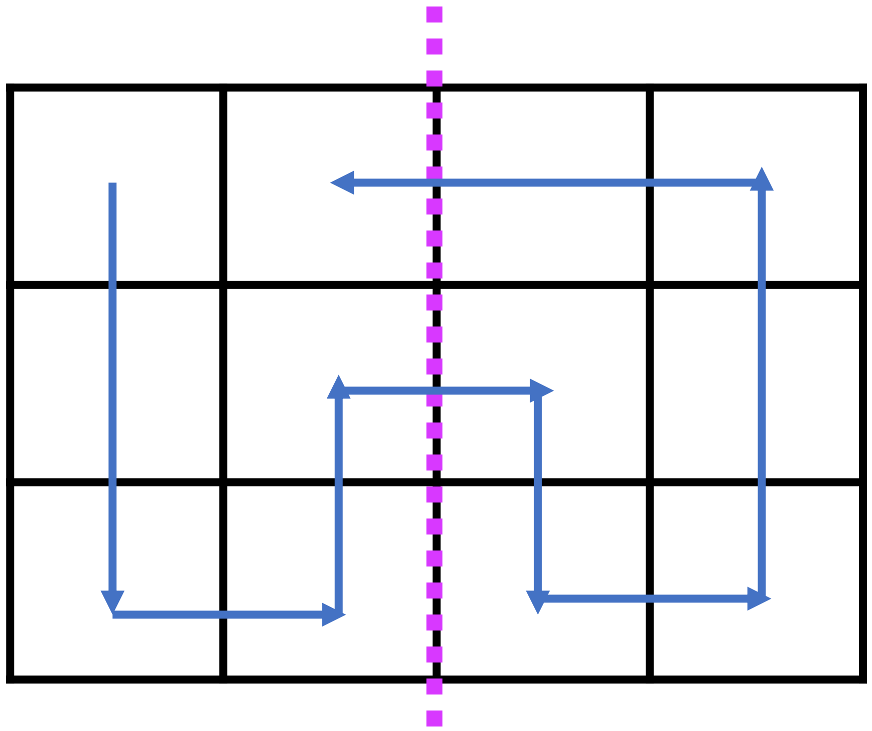
Interesting relevant questions



- **What if the grid size is not 1*2?**
    - What about 1*3?
    - What about L shape?
    - What about all Tetris shapes?
- **What if part of the grids are occupied?**
- **What if the shapes can have constraints?**
    - I.e., a 1*2 tile cannot be put “next to” a L shape?

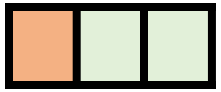
A more advanced version

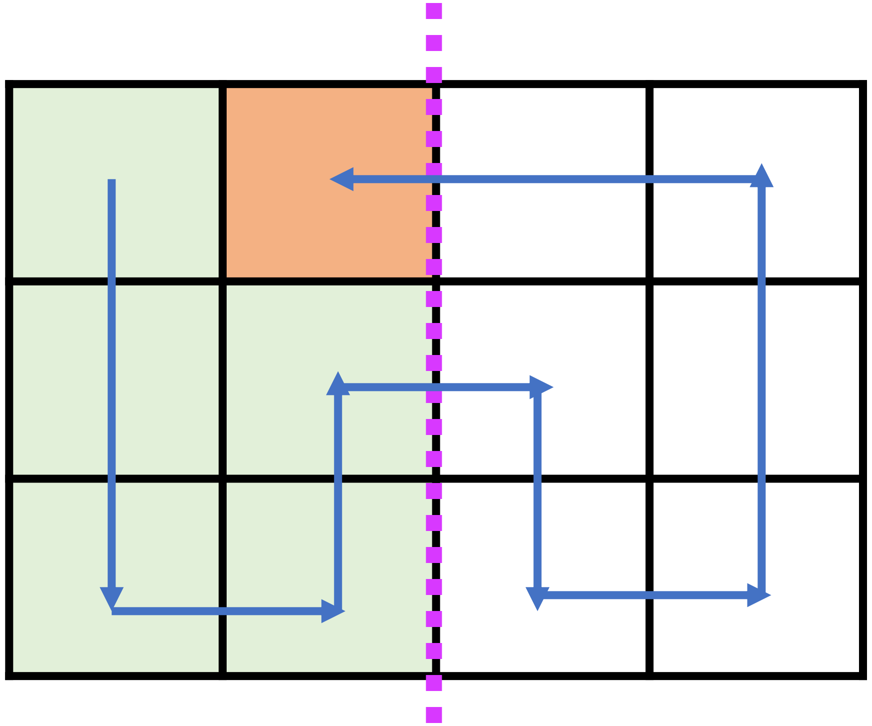
- Counting the paths in a grid
- Need to in addition maintaining the connectivity information



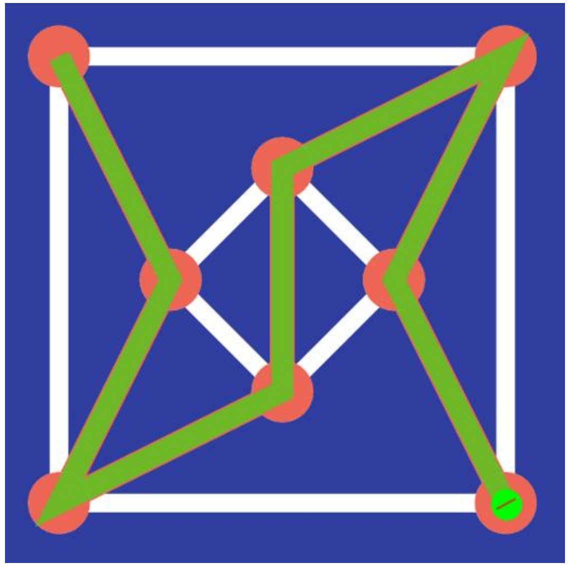
A more advanced version

- Need to in addition maintaining the connectivity information

- State: DP[2, ]

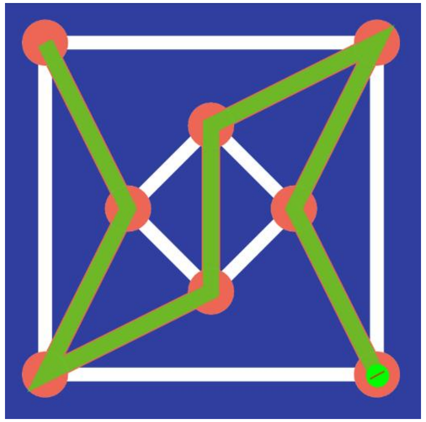


Another type of bitmask DP



- Consider the classic shortest Hamiltonian path (cycle)
- Given a graph $G = (V, E)$, find the shortest path/cycle that reaches all vertices in a graph
- One of Karp's 21 NP-complete problem

Another type of bitmask DP



- **One of Karp's 21 NP-complete problem**
- **Consider a DP solution:**
  - State: DP['10100101', 3] that represent the visited nodes and the last visited nodes
  - DP['10100101', 3]=min(DP['10000101', 6]+w(6,3), DP['10000101', 8]+w(8,3), DP['10000101', 6]+w(1,3))
  - Boundary: DP['10000000',1]=0, others with $+\infty$
  - Cost: $O(2^n \cdot n)$

Summary for this type of bitmask DP

- **Indicating used / unused “objects” as a bitmask**
- **When designing the DP recurrence, consider what is useful for your decision**
  - For the Hamiltonian path, the “last vertex” matters but not the others
- **When programming, consider the correct order to all states**

The next lecture…

- Algorithmic game theory based on dynamic programming