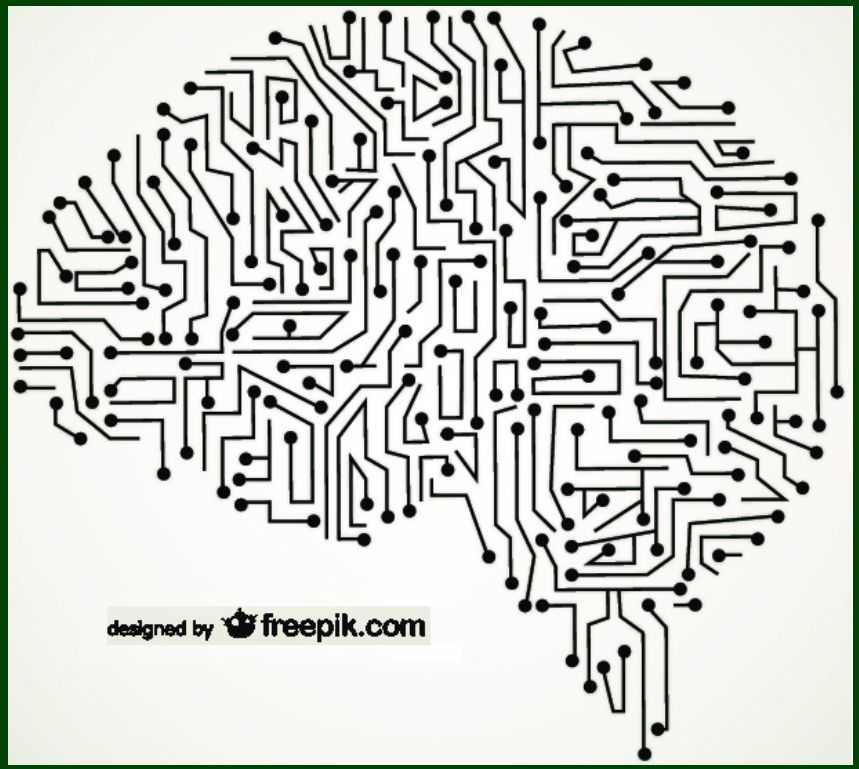
Introduction to Programming Methods

Dynamic Programming



Manhattan Walks

How many Manhattan Walks are there from $(0,0)$ to (n,m)?

As we're doing this, it's **crucial** to give a **combinatorial meaning** to every function, argument, etc. This is going to feel a lot like writing a recursive method, and there's a reason for that.

Combinatorial Meaning

```
walks(n, m)
```

- `walks` : # of manhattan walks
- `n` : x-coord end point
- `m` : y-coord end point

Writing a Recurrence

```
walks(n, m) = walks(n - 1, m) + walks(n, m - 1)
```

- `walks(n, m)` : # of walks to (n,m)
- `walks(n - 1, m)` : # of walks to (n,m) that end with "right"
- `walks(n, m - 1)` : # of walks to (n,m) that end with "up"

Writing a Recurrence

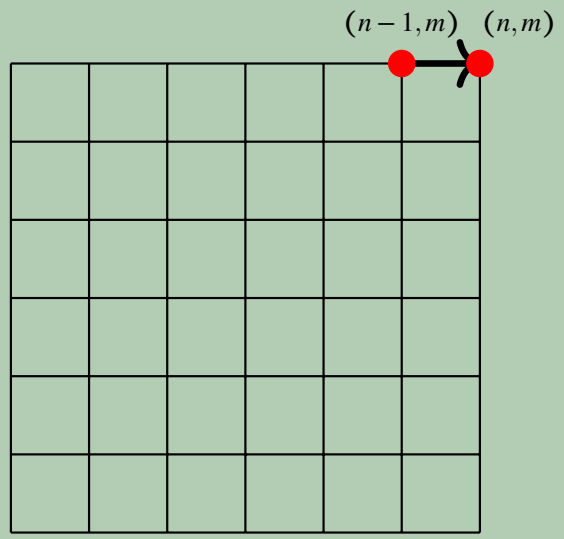
```
walks(n, m) = walks(n - 1, m) + walks(n, m - 1)
```

# of walks to (n,m)

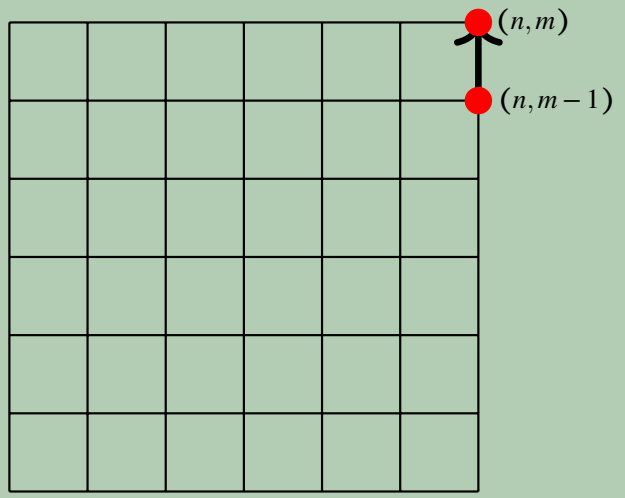
# of walks to (n,m) that end with "right"

# of walks to (n,m) that end with "up"

The Recurrence Pictorially



The last step is "right"

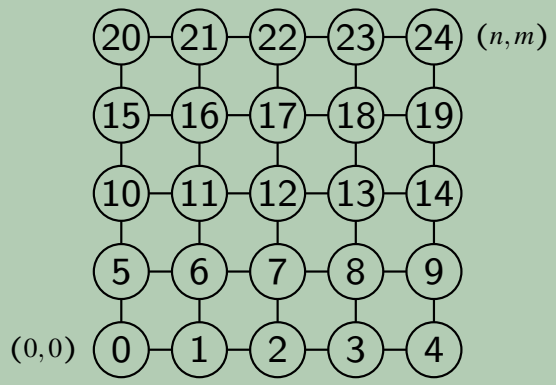


The last step is "up"

Memoization

```
1  walks(n, m) {
2     if ((n, m) == (0, 0)) {
3        return 1;
4     }
5     else if ((n, m) ∈ memo) {
6        return memo[(n, m)];
7     }
8     else {
9        memo[(n, m)] = walks(n − 1, m) + walks(n, m − 1);
10       return memo[(n, m)];
11    }
12 }
```

"Bottom-Up"



- Fill in all the base cases: $(0,0)$.
- Now, look for the next cell we can fill in: $(1,0)$ only depends on $(0,0)$
- Fill in the entire row
- Repeat.

Is This Worth It?

```
>> walks(3, 2)
>>     walks(2, 2)
>>         walks(1, 2)
>>             walks(0, 2)
>>                 walks(0, 1)
>>                     walks(0, 0)
>>             walks(1, 1)
>>                 walks(0, 1)
>>                     walks(0, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
>>         walks(2, 1)
>>             walks(1, 1)
>>                 walks(0, 1)
>>                     walks(0, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
>>             walks(2, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
>>     walks(3, 1)
>>         walks(2, 1)
>>             walks(1, 1)
>>                 walks(0, 1)
>>                     walks(0, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
>>             walks(2, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
>>         walks(3, 0)
>>             walks(2, 0)
>>                 walks(1, 0)
>>                     walks(0, 0)
```

Is This Worth It?

Function Call	# Of Unique Calls	# Of Total Calls
`walks(0, 0)`	0	1
`walks(1, 1)`	3	5
`walks(2, 2)`	8	19
`walks(3, 3)`	15	69
`walks(4, 4)`	24	251
`walks(5, 5)`	35	923
`walks(6, 6)`	48	3431
`walks(7, 7)`	63	12869
`walks(8, 8)`	80	48619
`walks(9, 9)`	99	184755
`walks(10, 10)`	120	705431

The number of unique calls grows **quadratically**

The number of total calls grows **exponentially**

Find The LCS

Given two strings s_1 and s_2, find the **longest common subsequence** that they share. Note that the characters do not have to be contiguous.

For example, if $s_1 =$ `abcdef` and $s_2 =$ `axxcxxdexxf`, then the LCS is `acdef`.

With DP, it's often useful to start with a simpler version and massage the answer into the original:

~~Find The~~ **Length of** LCS

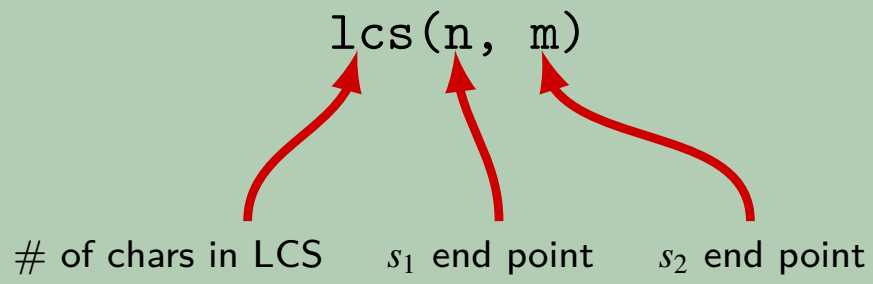
Given two strings s_1 and s_2, find the **length of the longest common subsequence** that they share. Note that the characters do not have to be contiguous.

For example, if $s_1 =$ `abcdef` and $s_2 =$ `axxcxxdexxf`, then the length of the LCS is 5.

~~Find The~~ Length of LCS

Given two strings s_1 and s_2, find the **length of the longest common subsequence** that they share. Note that the characters do not have to be contiguous.

Combinatorial Meaning



To write a recurrence, the question we ask is "what are the options to deal with the last character(s)?"

- Don't match the last character of s_1
- Don't match the last character of s_2
- Match the last character of s_1 with s_2 if possible

- Don't match the last character of s_1
- Don't match the last character of s_2
- Match the last character of s_1 with s_2 if possible

Writing a Recurrence

end with a match

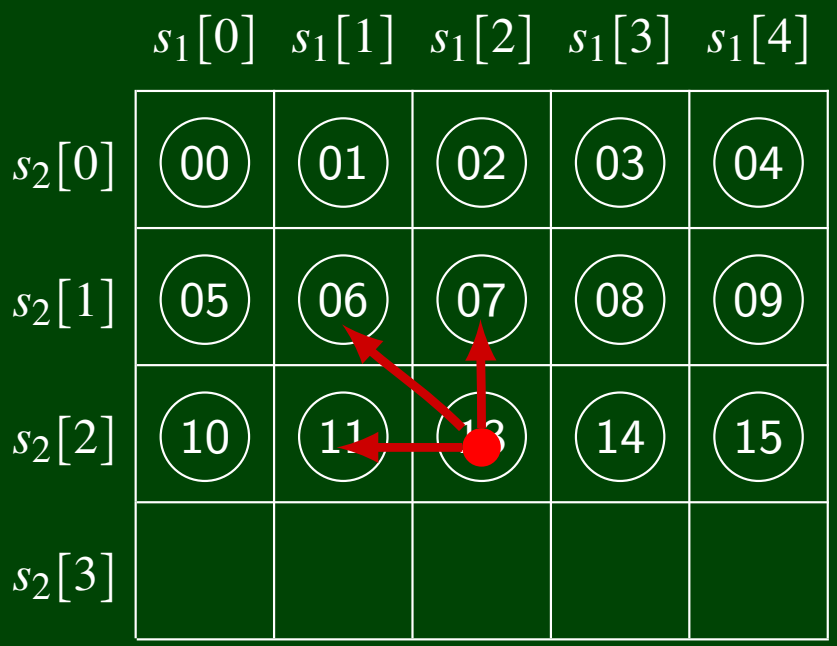
$$\texttt{lcs(n, m)} = \begin{cases} \texttt{lcs(n - 1, m - 1)} + 1 & \text{if } s_1[n] = s_2[m] \\ \max(\texttt{lcs(n - 1, m)}, \texttt{lcs(n, m - 1)}) & \text{otherwise} \end{cases}$$

the length of the lcs of s_1 `[0...n]` and s_2 `[0...m]`

don't use $s_1[n]$

don't use $s_2[m]$

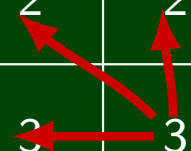
Next up, how do we order the recursive calls?



If we're calculating `lcs(n, m)`, we need to already know:

- `lcs(n - 1, m)`
- `lcs(n, m - 1)`
- `lcs(n - 1, m - 1)`

		a	c	d	e	r
		1	1	1	1	1
r		1	1	1	1	2
c		1	2	2	2	2
e		1	2	2	3	3



If we're calculating `lcs(n, m)`, we need to already know:

- `lcs(n - 1, m)`
- `lcs(n, m - 1)`
- `lcs(n - 1, m - 1)`

Partitions

Given a set of n elements, find S_1 and S_2 such that $|\text{sum}(S_1) - \text{sum}(S_2)|$ is minimized.

Partitions

Given a set of n elements, find the minimum difference: $|\text{sum}(S_1) - \text{sum}(S_2)|$.

Partitions

Given a set of n elements, can we make S_1 and S_2 such that $\text{sum}(S_1) = \text{sum}(S_2)$.

Partitions

Given a set of n elements, can we make S_1 and S_2 such that $\text{sum}(S_1) = \text{sum}(S_2)$.

Interleave

Given three Strings, s_1, s_2, and t, determine if the characters of s_1 and s_2 can be interleaved to make t.

Tiling a $2 \times n$ Board

Tiling

Given 1×2 and 2×1 dominos, how many ways can we tile a $2 \times n$ board?