DYNAMIC PROGRAMMING

RECURSIVE FIBONACCI

If you wanted to write an algorithm that calculated the nth Fibonacci number, you might do it like this:

```
Int Fibonacci(int n)
    If n < 3 Then Return 1
    Return Fibonacci(n-1)+Fibonacci(n-2)
```

What is the recurrence relation for this function?

ANALYSIS

$f(n) = f(n-1) + f(n-2) + \Theta(1)$

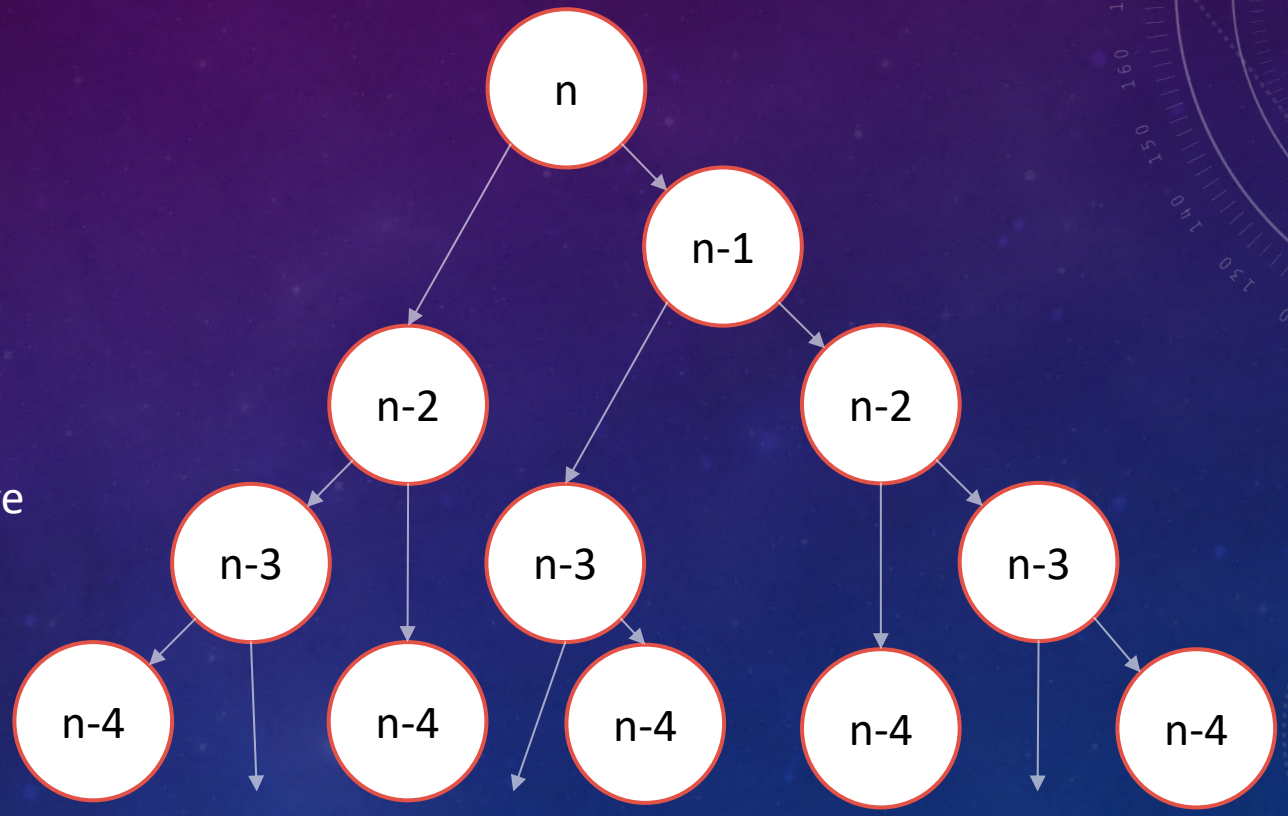
Can Master Theorem solve this?

- No, it's not in the correct form.

What is dumb about this tree that we've drawn?

- We're solving the same problem over and over!

The runtime is difficult to calculate, but it's **really** bad.



MEMOIZATION

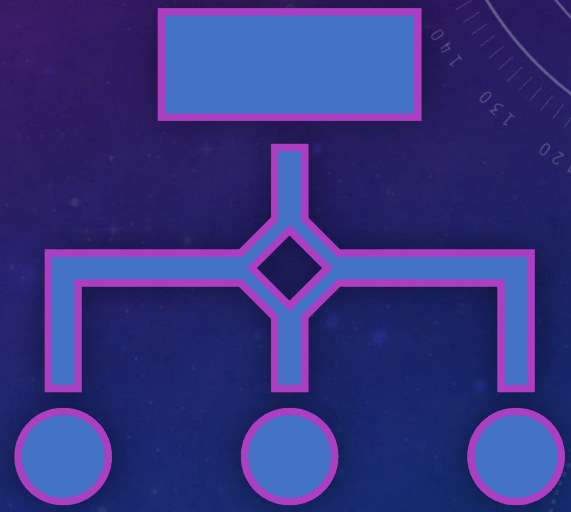
Memoization is the process of writing down intermediate result to refer to later.

- Our Fibonacci algorithm would greatly benefit from this!

What order do I need to solve the subproblems?

- Fibonacci(1), then Fibonacci(2), then Fibonacci(3), etc.

Clearly, we should solve this problem iteratively, rather than recursively. Then we can build up these values as we go.



F[1] = 1

F[2] = 1

For i = 3 to n

    A[i] = A[i-1] + A[i-2]

Return A[n]

Runtime: $\Theta(n)$

We have just done **dynamic programming**.

ITERATIVE FIBONACCI

DYNAMIC PROGRAMMING

Dynamic Programming is the process of transforming a recursive function which replicates work, into an iterative one.

- The iterative solution will solve each subproblem once and write down the answer for future reference.

Dynamic programming has a reputation for being challenging, but the concept is very simple: you are already familiar with it.

Arriving at a dynamic programming solution begins by writing a recursive function.

- This is the hardest part, and ironically, the dynamic programming part is significantly easier.
- It's not that dynamic programming is hard. **Recursion** is hard.

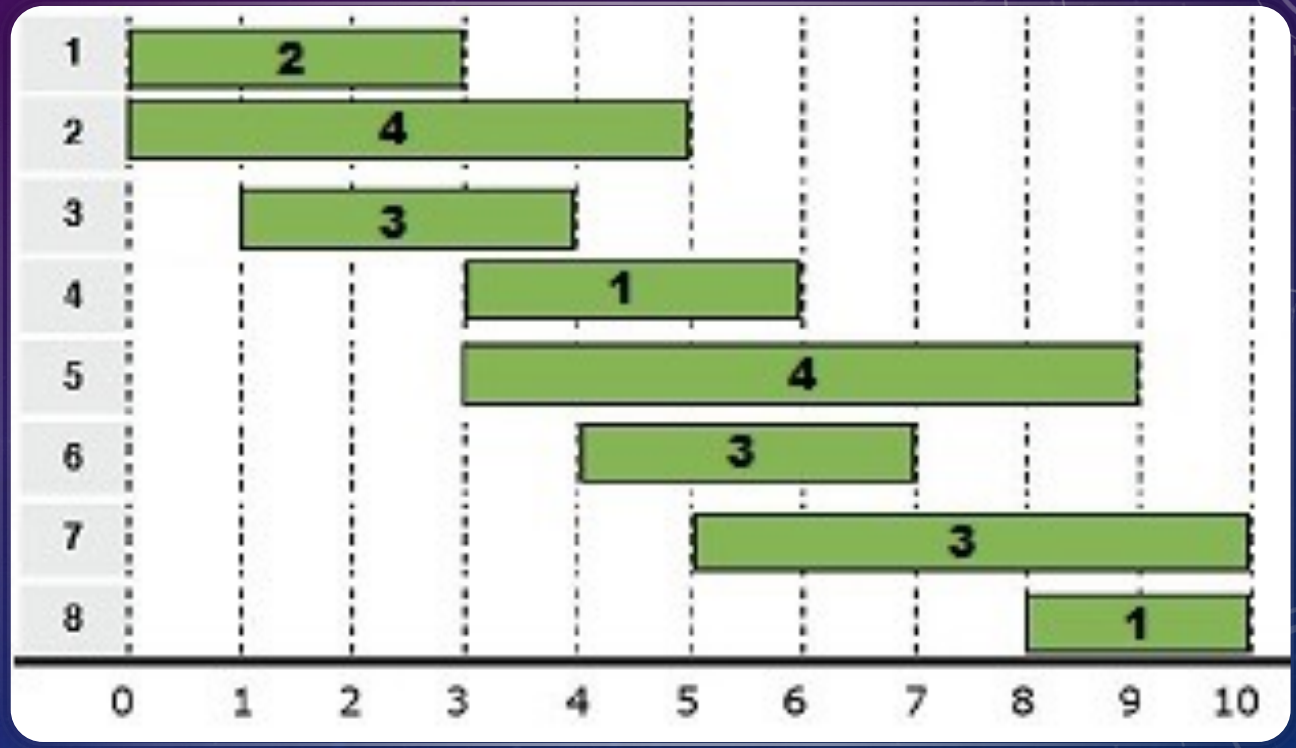
WEIGHTED INTERVAL SCHEDULING

In the **weighted interval scheduling** problem, we have n jobs.

- Each job j has a start time s_j, a finish time f_j, and a value v_j.
- Two jobs are incompatible if their times overlap by any amount.
- We want to find the max-valued subset of mutually compatible jobs.

What is the optimal solution for this instance?

- 3, 6, 8, or 2, 7



FINDING A RECURSIVE SOLUTION

Finding the entire optimal solution is a daunting task.

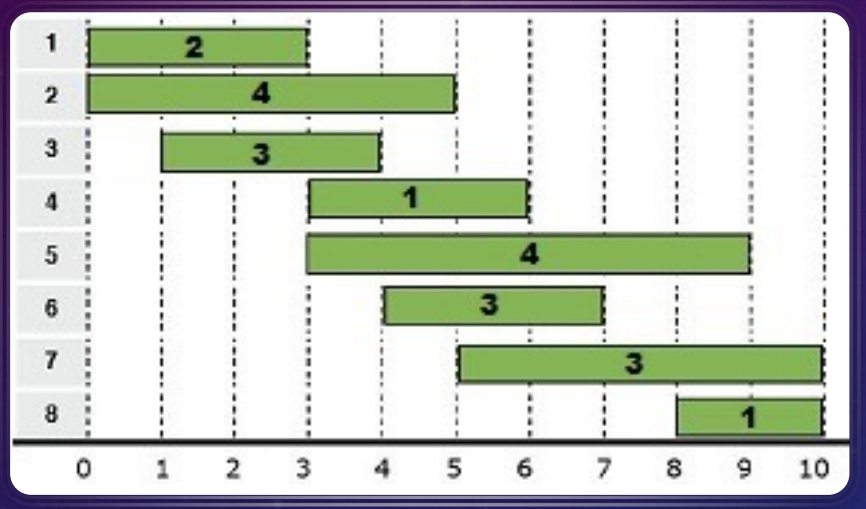
- If you're going to solve it recursively, split the problem up into smaller, bite-size pieces.
- Piece 1: Do I include interval 1?
- Piece 2: Do I include interval 2?, etc.

Your recursive algorithm will tackle exactly one piece.

- It will pass the rest of the problem to itself, to handle recursively.

You *don't yet know* if you should include interval 1. But you could recursively solve the rest of the problem under the assumption that you do or don't take interval 1, and then choose the better.

FINDING THE RECURSIVE FORMULA



If I include interval 1, what remaining intervals am I allowed to take?

- 4-8

If I don't include interval 1, what intervals am I allowed to take?

- 2-8

So, we will try solving the problem on intervals 2-8, and checking what value we get.

We will compare that to solving the problem on intervals 4-8 and adding v_1.

- Therefore, we need a function WIS(i) which calculates the value of the best solution for intervals i through 8 (or, more generally, i through n).

THE RECURSIVE SOLUTION

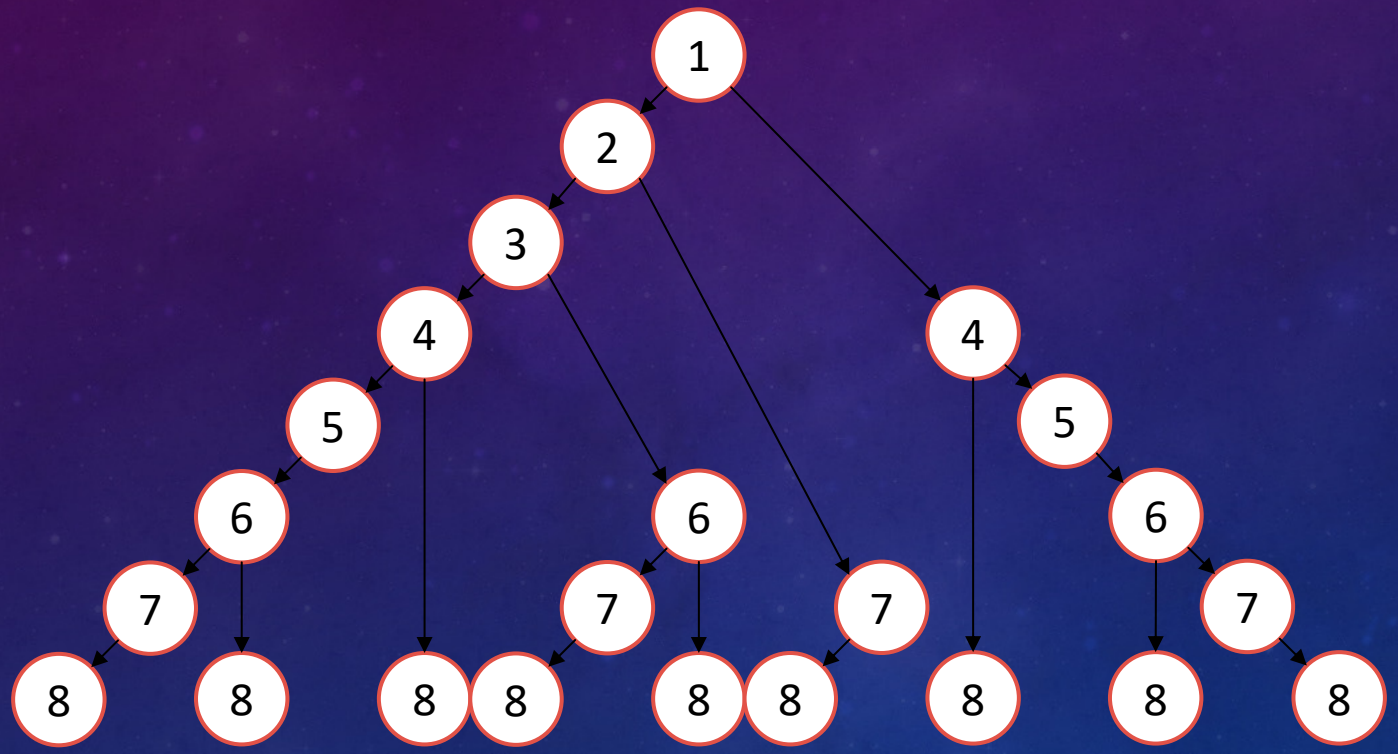
```
int WIS(int i)
    If i > n Then Return 0
    x = WIS(i+1)
    y = v[i] + WIS( the first interval j where s_j >= f_i )
    Return max(x, y)
```

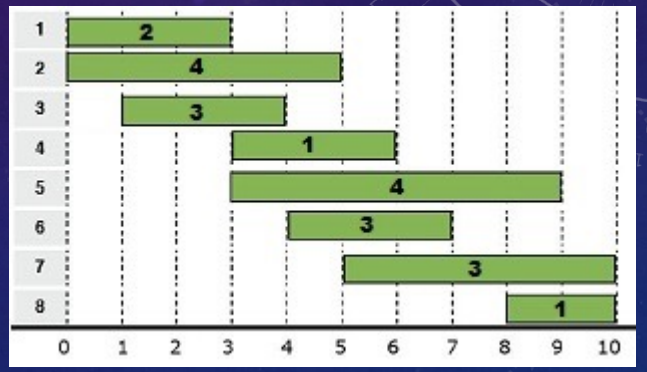
$y = v[i] + WIS(S[i])$

We can calculate the first interval j where $s_j >= f_i$, for all i, before we start the recursive procedure, and store them in an array S[1:n]

This is a bad solution, because it repeats work unnecessarily.

WASTED WORK





TRANSFORMING INTO AN ITERATIVE SOLUTION

We’re past the difficult part.

- We will re-envision WIS(i) as an array, not a function.
- W[i] will store exactly what WIS(i) would have returned.
- We need to determine the order to fill this array, so that we have what we need, when we need it.
- That’s actually quite easy in basic dynamic programming problems: its always reverse order.
- We’ll calculate W[n+1] (our base case), then W[n], then W[n-1], etc.

AN ITERATIVE SOLUTION

W[n+1] = 0

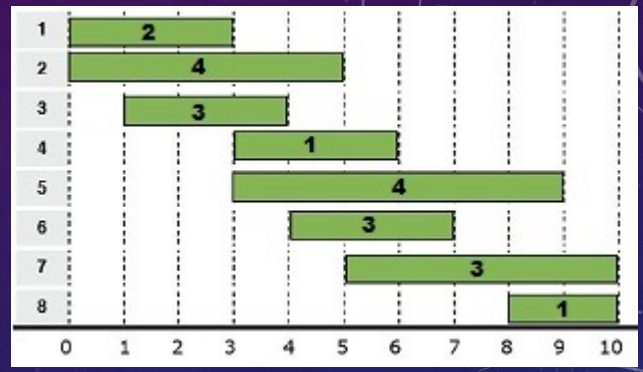
For i = n to 1

    W[i] = max(W[i+1], v[i] + W[S[i]])

Return W[1]

- We manually calculate our “base case”,
- We use the exact same recursive formula to calculate W[i]
- We return the initial parameter we passed into the function

USING THE ALGORITHM



W[n+1] = 0

For i = n to 1

W[i] = max(W[i+1], v[i] + W[S[i]])

Return W[1]

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

HOW TO WRITE YOUR SOLUTIONS

You must include these pieces of information:

1. State what parameters your function/array accepts, what those parameters mean, and what value it is calculating/storing specifically
   - W[i] stores the largest value attainable on job i through n
2. Give the recursive formula to calculate the intended output. This is the hardest step.
   - W[i] = max(W[i+1], v[i] + W[S[i]])
3. Give the base case(s) for your recursive formula
   - W[n+1] = 0
4. State what order you fill the array in.
   - For i = n to 1 calculate W[i]
5. State where the answer is stored in your final array
   - Return W[1]

THE DESIGN PROCESS

1. Reduce the problem to a series of ordered, bite-size decisions.
2. Figure out what subproblem(s) you will have to solve, based on each possible outcome of the bite-sized decision.
3. Represent those subproblems with as few input parameters to your recursive function as possible (more parameters = more complicated, and more runtime)
4. Design the complete recursive procedure
5. Figure out the order you will need to solve each subproblem, so that you have what you need, when you need it.
6. Figure out what indices of the array will store the final answer.
7. Design the complete iterative procedure

A HISTORICAL NOTE

The father of Dynamic Programming, a guy named Bellman, was originally going to call it “Planning over time”.

- He needed to extract funding from the Secretary of Defense.
- The Secretary of Defense had a pathological fear and hatred of the word “research”
- The Secretary of Defense would actually get red in the face, and violent, if the word was used in his presence.
- He **especially** hated it when the word was preceded by “mathematical”
- It is impossible to use the word “dynamic” in a negative manner.
- Bellman figured that not even a congressman could object to something called “dynamic programming”

So, the reason why it is called Dynamic Programming, is politics.

LONGEST INCREASING SUBSEQUENCE

Given a sequence of numbers s_1, …, s_n, delete the fewest numbers possible so that what is left is in increasing order.

3 4 1 2 8 6 7 5 9

3 4 ~~1 2~~ 8 ~~6 7 5~~ 9

3 4 ~~1 2 8~~ 6 7 ~~5~~ 9

~~3 4~~ 1 2 ~~8~~ 6 7 ~~5~~ 9

BITE-SIZE DECISIONS

Possible bite-size question: Do I include s_i or not?

If you include s_1, what number do you check next?

- The first number greater than s_1. In this case, s_2.

If you don't include s_1, what number do you check next?

- The next number. In this case, s_2.

Something seems wrong here…

3 ~~4~~ 1 2 ~~8~~ 6 7 ~~5~~ 9

LET'S TRY AGAIN…

We need to both keep track of which number we're considering, as well as the previous number we've chosen.

If you include s_i, what number do you check next?

- The first number greater than s_i.

If you don't include s_i, what number do you check next?

- The next number that is larger than the previous number we've chosen.

We can accomplish both of these with a single parameter if we re-envision the bite-size decision. **Less parameters = better**

BITE-SIZE, TAKE 2

Bite-size decision: if we include s_i, what number do we include next?

- We'll loop over all numbers that come after s_i, and consider them if they are also greater than s_i.

We only need a single parameter: i

LIS(int i) returns the length of the longest increasing subsequence that uses s_i as its first number.

We **could** have done this differently, where we only figure out if we're going to include the current value, and we keep track both of the current value and the previous number we chose.

- This requires 2-parameter dynamic programming, which we aren't ready to talk about yet. This solution would take the same amount of time, but would use more memory than the solution we're going to tackle.

THE RECURSIVE FORMULA

$LIS(i) = 1+\max_{k : k > i,\ s_k > s_i} LIS(k)$

That is, loop over all k, and consider the k that satisfy both $k > i$ and $s_k > s_i$. For those values return the largest possible value of LIS(k)

- Add 1 to account for including s_i in our sequence.

We also need a base case:

- $LIS(i) = 0$, if $i > n$

This will work perfectly… except it's slow, because it repeats work. We need to turn this into an iterative dynamic programming solution.

THE ITERATIVE VERSION

L[n+1] = 0

For (i = n to 1)

$L[i] = 1 + \max_{k : k > i,\ s_k > s_i} L[k]$

Return $\max_i L[i]$

What is the runtime?

- $\Theta(n^2)$

PSEUDO-POLYNOMIAL RUNTIMES

```
Primality(X)
    For i = 2 to X – 1
        If X % i = 0 Then Return False
    Return True
```

What's the runtime of this function?

- $\Theta(X)$

Is that a polynomial runtime?

- **No**. Wait… what?

PSEUDO-POLYNOMIAL RUNTIMES

When you say a runtime is “polynomial” or “linear” or “logarithmic”, that has to be **in relation** to something.

- It means in relation to the **length of the input**.

When you have a $\Theta(m+n)$ graph algorithm, that’s a **linear** runtime, because the size of the graph (which is the input) is m+n.

- I could introduce a new variable X such that log X = m+n. It would be correct to say the runtime is $\Theta(\log X)$. That does not somehow make the runtime logarithmic.

What was the length of the input for our Primality algorithm?

- We passed in 1 variable: X. What is the **length** of X?
- log X, because that’s the number of bits it takes to represent X.

RUNTIME OF PRIMALITY

So the length of the input is n = log X.

- The runtime is $\Theta(X) = \Theta(2^{\log X}) = \Theta(2^n)$
- Thus the runtime of our algorithm was **exponential**, even though it **looked** linear.

X was the **value** of the input. When you have a runtime that is polynomial in terms of the **value** of the input, then we say it has a **pseudo-polynomial** runtime.

- These runtimes **are** exponential, but they tend to not be as bad as other exponential algorithms.

When you are asked if a runtime is polynomial, always double-check to make sure the variables being used refer to the **length** of the input, rather than something else.

- We typically just refer to the length of the input as the number of inputs. However, when that value is a constant, you will need to look at the number of **bits** of those inputs.

COIN-CHANGING

You have a target T, n denominations of coins $1 = d_1 < \ldots < d_n$, and you have an unlimited amount of each coin. Determine the fewest number of coins needed to make T cents change.

Suppose I'm using American currency ($d_1 = 1$, $d_2 = 5$, $d_3 = 10$, $d_4 = 25$). What would be an optimal algorithm for this currency?

- A greedy algorithm that always uses the largest coin possible. $37 = 25 + 10 + 1 + 1$
- This works because each denomination is a multiple of the previous… except 25 vs. 10, but it is possible to prove that one case works as well.

Does this algorithm work for arbitrary currencies?

- **No**
- Let $d_1 = 1$, $d_2 = 10$, $d_3 = 15$, and $T = 20$.

COIN-CHANGING: BITE-SIZE DECISIONS

What should my first decision be?

- What coin do I choose next?
    - That's the simplest question, but you could also successfully do "Do I use this coin next?" or "How many of this coin should I take?"

What should the parameters of my function be?

- I need to know the remaining target I am aiming to sum to.

CC(x) will return the fewest number of coins needed to make x cents change.

If I choose denomination d_i, what recursive call should I make?

- $CC(x - d_i)$

COIN-CHANGING: RECURSIVE ALGORITHM

What should our recursive formula be?

- We need to consider every possible denomination.
- $CC(x) = 1 + \min_{i\,:\,d_i \le x} CC(x - d_i)$

Don't forget the base case!

- $CC(0) = 0$

However, this will call the same subproblem multiple times, wasting work. Time to make this iterative!

COIN-CHANGING: MAKING IT ITERATIVE

C[0] = 0

For x = 1 to T

$$C[x] = 1 + \min_{i\,:\,d_i \le x} C[x - d_i]$$

Return C[T]

What is the runtime?

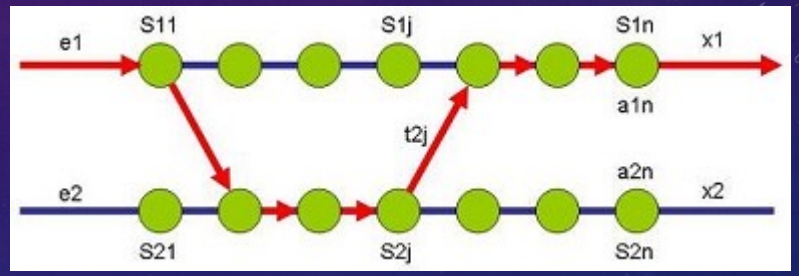
- $\Theta(nT)$

Is this a polynomial-runtime?

- **No**. The ‘n’ is okay, but the ‘T’ is the value of the input, not the length of the input.

ASSEMBLY-LINE SCHEDULING

There are 2 assembly lines, each with n stations.



- The i^{th} station on each line is denoted $S_{1,i}$ and $S_{2,i}$.
- $S_{i,j}$ has a processing time of $a_{i,j}$.
- An automobile can start at your choice of $S_{1,1}$ or $S_{2,1}$.
- After station $S_{i,j}$, the automobile can go to $S_{i,j+1}$ for no cost, or switch to the other line for cost $t_{i,j}$.
- The automobile is done after being handled by station $S_{1,n}$ or $S_{2,n}$.

If we want to find the shortest path across the assembly lines, what algorithm could we use?

- Dijkstra's! We modify the graph slightly, but this is basically a shortest-path problem.
- **However!** We'll be able to improve upon this by using Dynamic Programming!

ASSEMBLY-LINE SCHEDULING: RECURSION

What should our bite-size decision be?

- Do I stay on the same line, or switch to the other line?

What information do I need to pass into the function as input parameters?

- The current station we're at, which is represented by two values: line and station number.

ALS(i, j) will return the length of the shortest path that starts at $S_{i,j}$.

- ALS(1,j) = $a_{1,j}$ + min(ALS(1, j+1), $t_{1,j}$ + ALS(2, j+1))
- ALS(2,j) = $a_{2,j}$ + min(ALS(2, j+1), $t_{2,j}$ + ALS(1, j+1))
- ALS(i,n) = $a_{i,n}$

ASSEMBLY-LINE SCHEDULING: ITERATIVE

$A[1,n] = a_{1,n}$

$A[2,n] = a_{2,n}$

For j = n-1 to 1

    For i = 1 to 2

        $A[i, j] = a_{i,j} + \min(A[i, j+1], t_{i,j} + A[3 – i, j+1])$

Return min(A[1, 1], A[2, 1])

What is the runtime?

- $\Theta(n)$. Dijkstra's would have attained $\Theta(n \log n)$.

SEQUENCE ALIGNMENT

You are given two strings:

- $X = x_1 \ldots x_n$
- $Y = y_1 \ldots y_m$

You want to determine how similar these strings are

X = OCURRANCE
Y = OCCURRENCE

We could simply check how many positions i satisfy $x_i = y_i$

- This suggests that there are 7 differences, so the strings are 30% similar.
- That seems like a dumb system. How many differences are there really?

EDIT DISTANCE

The **edit distance** between two strings X and Y is the minimum possible number of changes to transform X into Y.

There are two changes: add the 2nd C, and change the A to an E.

- You might decide to transform a letter (such as an A to an E)
- You might decide to add a letter (such as the C)
- You might decide to remove a letter (such as if X had an S at the end)

The challenge is: how to efficiently calculate the edit distance?

SEQUENCE ALIGNMENT: BITE-SIZE DECISIONS

We need to figure out which letters line up.

- You could ask a broad question, such as “which letter (if any) does x_i match with?”
  - Note that if x_i matches nothing, that means we are deleting that letter.
  - We’d also have to worry about adding letters to X, which would make this bite-size decision fairly challenging to solve.
- A smaller and simpler question is: “Do I match x_i and y_j?” There are a few options:
  - Match the letters
  - Transform x_i into y_j
  - Delete x_i (matching it with nothing)
  - Add y_j (matching it with nothing)

SEQUENCE ALIGNMENT: RECURSIVE SOLUTION

What do we need to keep track of in our call to Sequence Alignment?

- Our function will be SA(i,j), where i is the current character of X, and j is the current character of Y
- SA(i,j) will solve the problem from this point forwards. Therefore, it will *return the edit distance of* $X = x_i \dots x_n$ *and* $Y = y_j \dots y_m$

What is the recursive formula for SA(i,j)?

- If $x_i = y_j$, then we return SA(i+1,j+1). It is possible to formally prove that this is always optimal (proof omitted).
- If we decide to transform x_i into y_j, we return 1+SA(i+1,j+1)
- If we remove x_i, we return 1+SA(i+1,j)
- If we add y_j, we return 1+SA(i,j+1)

SEQUENCE ALIGNMENT: RECURSION

SA(i,j) = SA(i+1,j+1), if $x_i = y_j$

SA(i,j) = 1+min(SA(i+1,j+1), SA(i,j+1), SA(i+1,j)), otherwise

Don’t forget the base cases!

SA(i,m+1) = n-i+1 (the number of characters left in X)

SA(n+1,j) = m-j+1 (the number of characters left in Y)

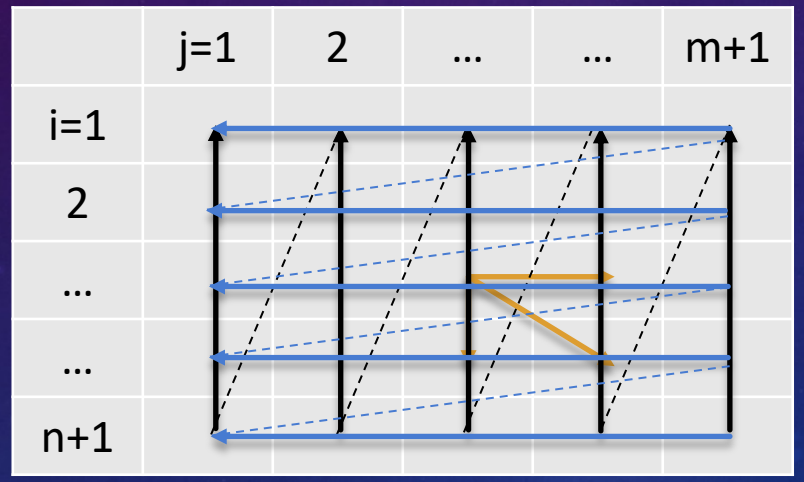
Now we need to make this iterative

SEQUENCE ALIGNMENT: ITERATIVE SOLUTION

What order do we need to fill the array?

We need to complete the destination entries before the source entry

```
For i = n to 1
        For j = m to 1
                Calc SA[i,j]
```




```
For j = m to 1
        For i = n to 1
                Calc SA[i,j]
```

Return SA[1,1]

SEQUENCE ALIGNMENT: THE ALGORITHM

For i = 1 to n+1

    S[i,m+1] = n+1-i

For j = 1 to m+1

    S[n+1,j] = m+1-j

For i = n to 1

    For j = m to 1

        If $x_i = y_j$ Then S[i,j] = S[i+1,j+1]

        Else S[i,j] = 1+min(S[i+1,j+1], S[i,j+1], S[i+1,j])

Return S[1,1]

SEQUENCE ALIGNMENT: EXAMPLE

Runtime = $\Theta(mn)$

Space = $\Theta(mn)$

How could we improve the space requirements?

We only need the current column and the previous column.

Serious drawback: we cannot reconstruct the solution if we do that.

	A	C	A	C	A	C	T	A	
A	2	3	3	4	5	6	7	7	8
G	2	2	3	3	4	5	6	6	7
C	2	1	2	2	3	4	5	5	6
A	3	2	1	2	2	3	4	4	5
C	4	3	2	1	2	2	3	3	4
A	5	4	3	2	1	2	2	2	3
C	6	5	4	3	2	1	1	1	2
A	7	6	5	4	3	2	1	0	1
	8	7	6	5	4	3	2	1	0

A_CACACTA
AGCACAC_A

SUBSET SUM

Given positive integers w_1, …, w_n and a target W, is there a subset of the integers which adds up exactly to W?

- Let our integers be {2, 5, 7, 13, 16, 17, 23, 39}, and W = 50. Is there a solution?
- 2, 5, 7, 13, 23

What should my bite-size decision be?

- Do I include the current integer w_i or not?

What parameters do I need in my function call?

- The current integer, and the remaining target
- SS(i, x) returns 1 if there is a subset of w_i, …, w_n that add exactly to x, and 0 otherwise.

SUBSET SUM: RECURSION

If I include w_i, what is my recursive call?

- $SS(i+1, x - w_i)$

If I don't include it, what is my recursive call?

- $SS(i+1, x)$

What base cases do I need?

- $SS(i, 0) = 1$
- $SS(n+1, x) = 0$, for $x \neq 0$
- $SS(i, x) = \max(SS(i+1, x - w_i), SS(i+1, x))$

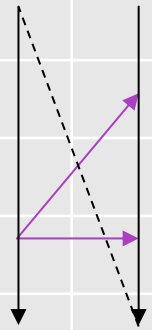
SUBSET SUM: ITERATIVE

For i=n to 1

For x=0 to W

Calc S[i, x]

	i=1	2	…	…	n+1
x=0					
1					
…					
…					
W					



Answer = S[1, W]

$\Theta(nW)$

Is this a polynomial runtime?

No: the n is fine, the W is not.

RNA SECONDARY STRUCTURE

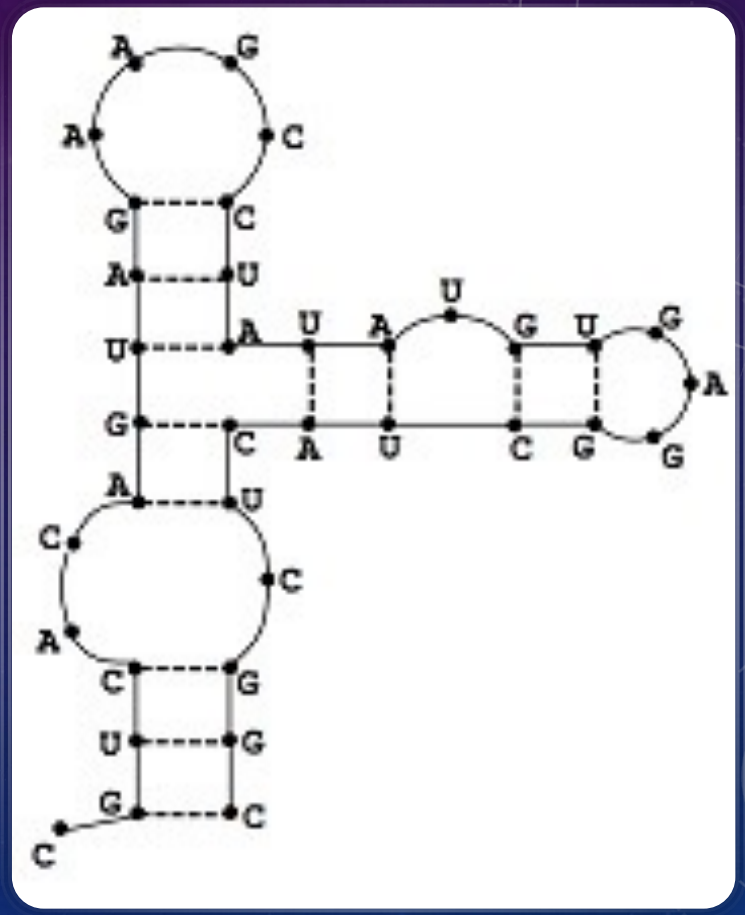
RNA is a string $B = b_1 \ldots b_n$, where $b_i \in \{A,C,G,U\}$

- RNA folds back in on itself and forms pairs with itself.
- The “secondary structure” is the structure that arises from this folding.

CGUCACAGUAGAAGCCUAUAUGUGAGGCUACUCGGC

The above string might take this structure.

- How RNA bonds with itself is very complicated.
- We will simplify it, getting an approximation of how it looks.
- We will be reconstructing one of the first algorithmic attempts to predict RNA secondary structure.



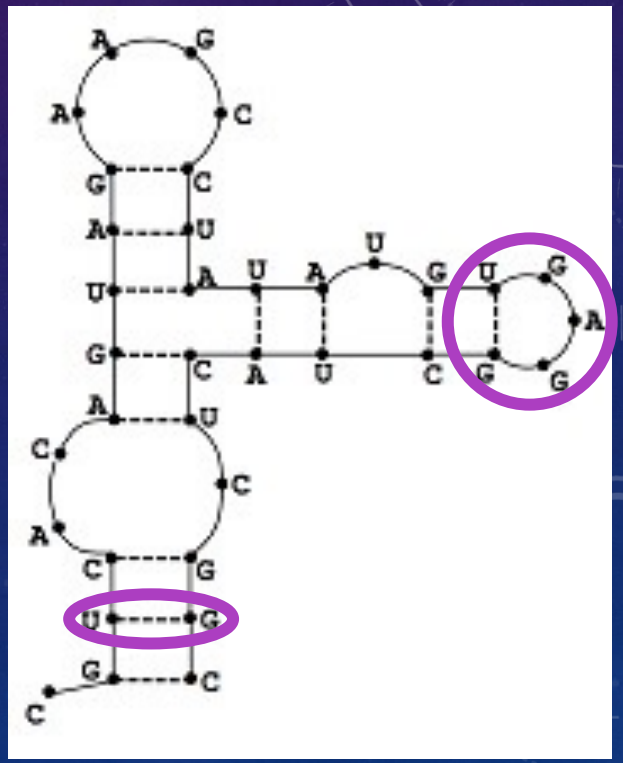
RNA: THE RULES

We want to identify S, the set of pairs for input string B.

We will assume that RNA always follows these rules:

- Each pairing is AU or CG
- If $\langle b_i, b_j \rangle \in S$, then $i < j-4$ (the ends of each pair are separated by 4+ bases)
- If $\langle b_i, b_j \rangle, \langle b_k, b_l \rangle \in S$, it is not the case that $i < k < j < l$ (no crossed pairs)

We will assume RNA forms the maximum possible pairs subject to the above rules.

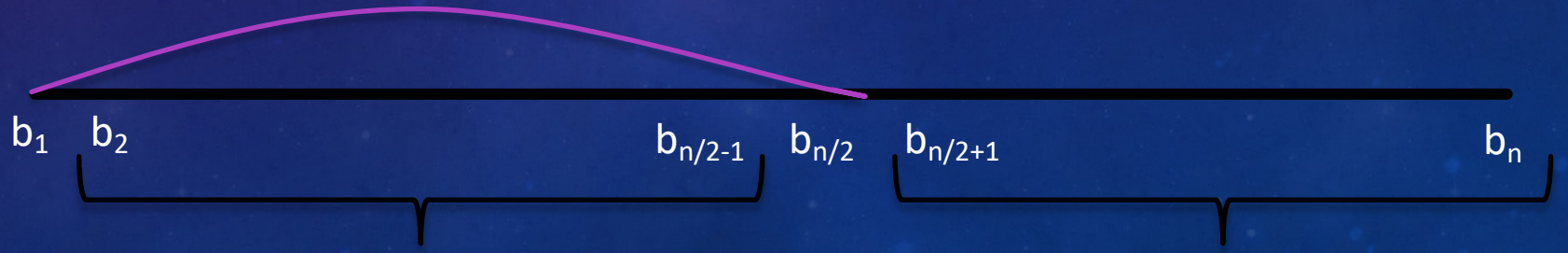


RNA: BITE-SIZE DECISIONS

We need to figure out which letters form pairs.

- You could ask a narrow question such as “do these two letters match?”
  - If the answer is no, it’s rather challenging to figure out the progression of the questions. It will work, but it’s difficult.
- Instead we will ask a broad question: “what, if anything, does this letter match with?”, starting with b_1.

Suppose b_1 pairs with $b_{n/2}$. What subproblems do I need to worry about?



RNA: RECURSION

What parameters do we need to pass into our function?

- RNA(i, j) returns the max possible matched pairs on the substring $b_i \ldots b_j$.

If b_i doesn't pair with anything, what recursive call should we make?

- RNA(i+1, j)

If b_i pairs with b_k, what is my recursive formula?

- 1 + RNA(i+1, k-1) + RNA(k+1, j)

I try all possible k, and take the best option:

- RNA(i, j) = max(RNA(i+1, j),
  1 + $\max_k$(RNA(i+1, k-1) + RNA(k+1, j)))

What choices of k are valid?

RNA: THE ALGORITHM

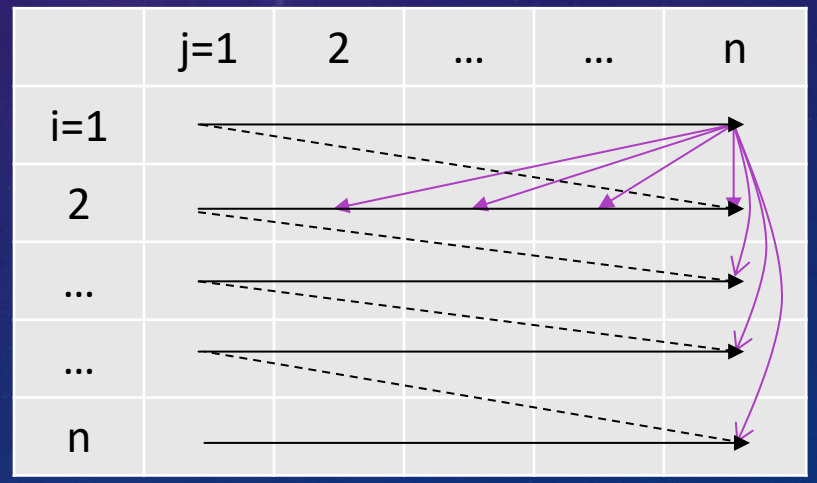
The full recursive formula:

- RNA(i, j) = max(RNA(i+1, j),
  $1 + \max_{k : k > i+4,\ \{b_i, b_k\} = \{A, U\} \text{ or } \{C,G\}} (RNA(i+1, k-1) + RNA(k+1, j)))$
- RNA(i, j) = 0, if $i \geq j-4$

For i = n to 1

For j = 1 to n

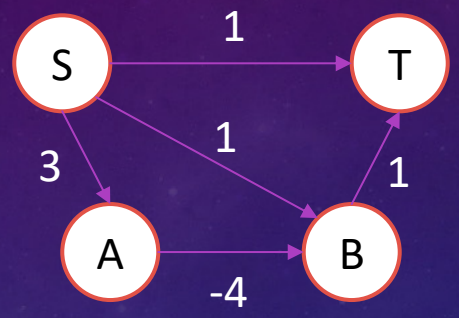
Calc R[i ,j]



Return R[1, n]

$\Theta(n^3)$

SHORTEST PATH (AGAIN!)



What is the length of the shortest path from S to T?

- 0

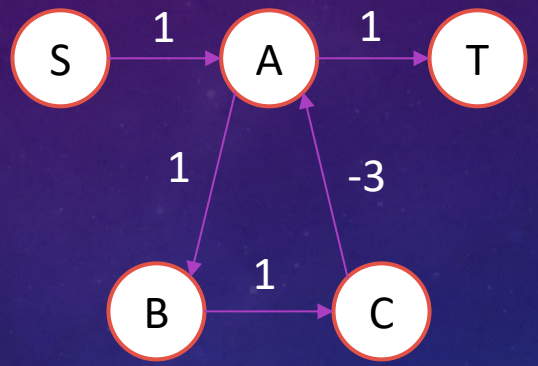
What is the length of the path that Dijkstra's Algorithm will find?

- 1

Why didn't Dijkstra's Algorithm work?

- Negative weight edges.

SHORTEST PATH: CONSTRAINTS



What is the length of the shortest walk from S to T?

- $-\infty$

While we will allow negative-weight edges, we will disallow negative-weight cycles, as this produces nonsensical answers.

SHORTEST PATH: BITE-SIZE DECISIONS

What should my bite-size decision be?

- What node do I go to next?

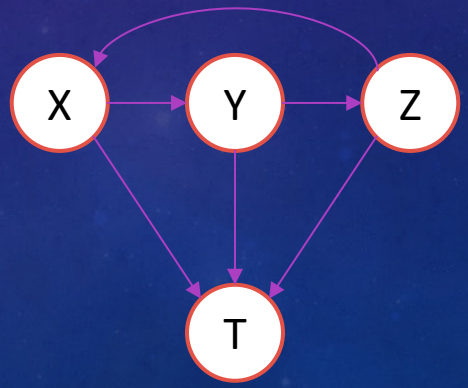
What information do you need to pass in as input parameters?

- The current node. Anything else?

SP(x) will return the length of the shortest path from x to t.

- SP(t) = 0
- $SP(x) = \min_{\langle x,y\rangle \in E}(c_{\langle x,y\rangle}+SP(y))$

What order do I fill the array?



SHORTEST PATH: TAKE TWO

We need to prevent endlessly going around cycles.

- One way to solve this is to keep track of how many edges we've traversed, and stop if we traversed more edges than necessary.

SP(i,x) = The length of the shortest path from x to t, using no more than i edges.

- $SP(i,t) = 0$, for $i \geq 0$
- $SP(0,x) = \infty$, for $x \neq t$
- $SP(i,x) = \min_{\langle x,y \rangle \in E}(c_{\langle x,y \rangle} + SP(i-1,y))$

What values should I initially pass into this function?

- SP(n-1,s)
- If we traverse n edges, we have revisited a node (and thus traversed a cycle)

SHORTEST PATH: ITERATIVE ALGORITHM

For i = 1 to n-1

For all nodes x

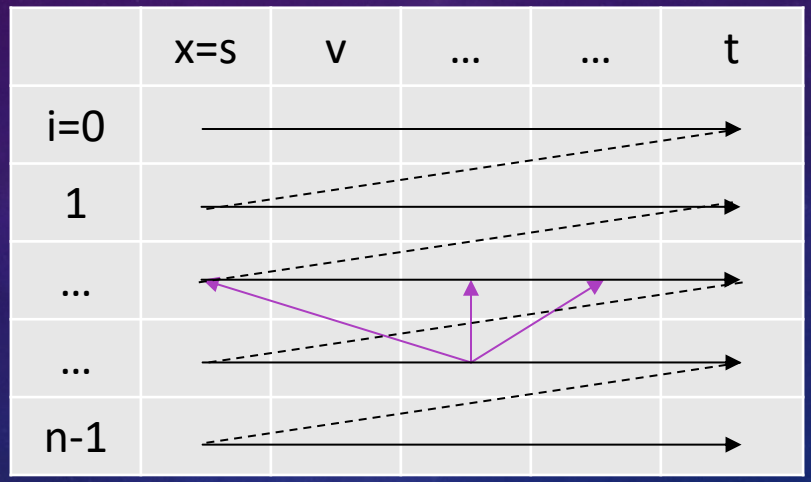
Calc SP[i,x]

Runtime = $\Theta(mn)$

Is this polynomial?

- Yes

This is known as the Bellman-Ford Algorithm



FINDING NEGATIVE-WEIGHT CYCLES

If SP[i,x] = SP[i+1,x], for all x, then there are no more updates to be made for larger values of i.

- If SP[n-1,x] ≠ SP[n,x], then there must exist a negative-weight cycle!
- Run Bellman-Ford one more iteration, and check to see if any values have updated.

ALL-PAIRS SHORTEST PATHS

Suppose we want to return the shortest path between **all** pairs of points

- So, we will return $n \cdot (n-1)$ answers, one for each pair.

How could we do this, using existing algorithms?

- BFS finds the shortest path from s to all nodes. So, on an unweighted graph, we can run BFS n times, once for each start node: $\Theta(n(m+n))$
- Dijkstra's finds the shortest path from s to all nodes. So, on a graph with no negative-weight edges, we can run Dijkstra's n times: $\Theta(mn \log n)$
- Bellman-Ford finds the shortest path from all nodes to t. So, we can run Bellman-Ford n times, once for each end node: $\Theta(mn^2)$

Maybe we can calculate this directly, and find a way to save runtime?

ALL-PAIRS SHORTEST PATHS: ALGORITHM

ASP(i,x,z) = the length of the shortest path from x to z, using no more than i edges.

- ASP(i,z,z) = 0
- ASP(0,x,z) = ∞, for $x \neq z$
- ASP(i,x,z) = $\min_{\langle x,y\rangle \in E}(c_{\langle x,y\rangle}+ASP(i-1,y,z))$

For i = 1 to n-1

    For all nodes z

        For all nodes x

            Calc ASP[i,x,z]

Return ASP[n-1]

Runtime = $\Theta(mn^2)$

- We need a new programming design paradigm before we can improve upon this.

EXTRA PRACTICE

Chapter 6

Exercises 1, 4, 6, 19, 20, 24, 26, 27