ساختمان داده و الگوریتم ها

جلسه مرور میان ترم

سجاد شیرعلی شهرضا

بهار 1402

سه شنبه،29 فروردین 1402

THE ALGORITHMIC TOOLKIT

Algorithm **design** paradigms

Recognizing these patterns will help you design algorithms for problems you encounter in a variety of domains, even outside of this class

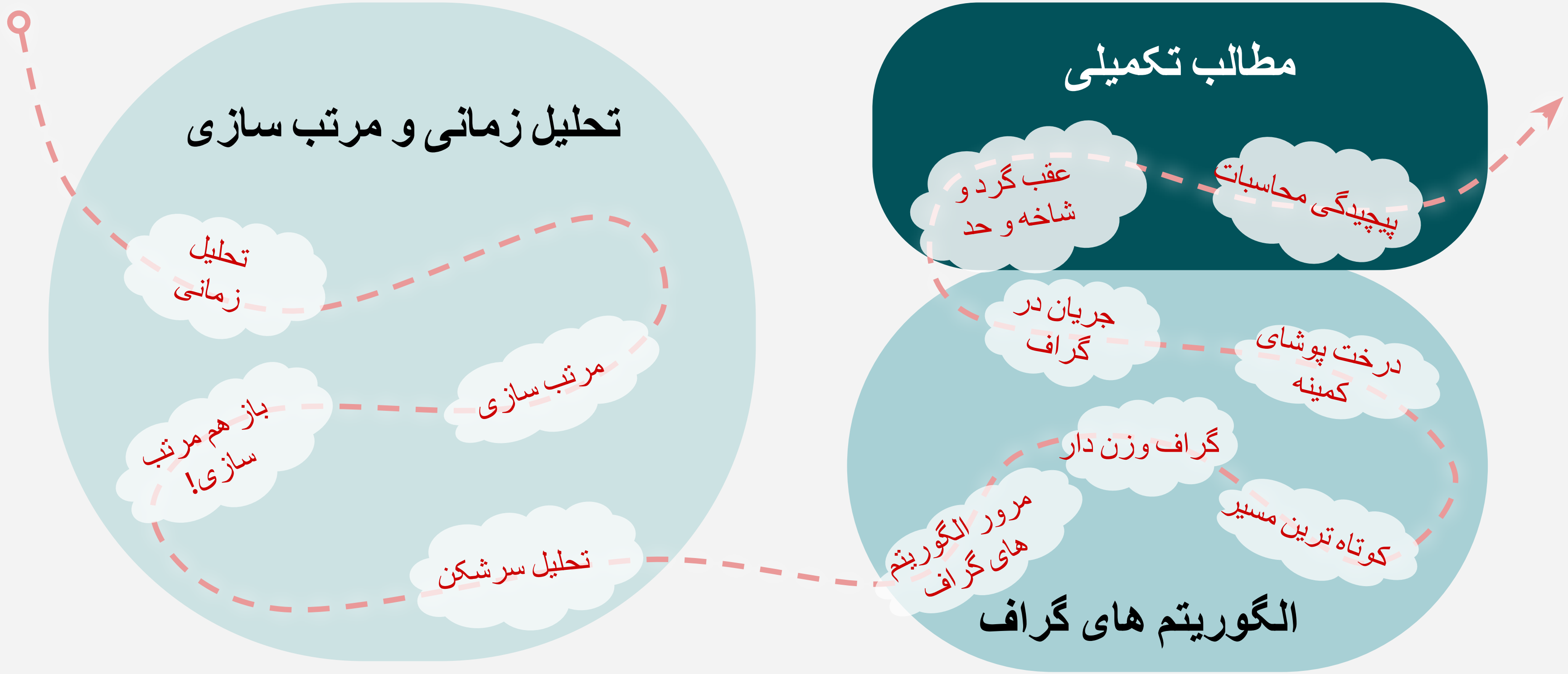
Rigorous algorithm **analysis** skills

What makes an algorithm fast? Correct? How can we prove this mathematically?

Better ways to **communicate** technical ideas

How can we describe an algorithm? How can we make our proofs compelling?

نقشه راهنما



تحلیل زمانی و مرتب سازی
تحلیل زمانی
مرتب سازی
باز هم مرتب سازی!
تحلیل سرشکن
الگوریتم های گراف
مرور الگوریتم های گراف
گراف وزن دار
کوتاه ترین مسیر
درخت پوشای کمینه
جریان در گراف
مطالب تکمیلی
عقب گرد و شاخه و حد
پیچیدگی محاسبات

تحلیل زمانی

FROM DATA STRUCTURE COURSE

THE POINT OF ASYMPTOTIC NOTATION

$$\textbf{suppress } \underbrace{\textbf{constant factors}}_{\textit{too system dependent}} \textbf{ and } \underbrace{\textbf{lower-order terms}}_{\textit{irrelevant for large inputs}}$$

- **Some guiding principles:** we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime's *rate of growth*), and we want some measure of runtime that's independent of hardware, programming language, memory layout, etc.
    - We want to reason about high-level algorithmic approaches rather than lower-level details

BIG-O NOTATION

Let $T(n)$ & $f(n)$ be functions defined on the positive integers.

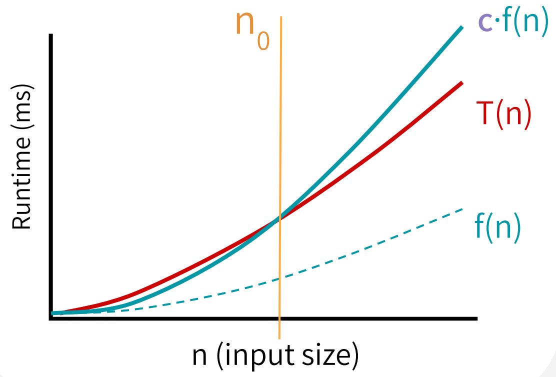
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is O(f(n))"?

In English

$T(n) = O(f(n))$ if and only if $T(n)$ is *eventually* **upper bounded** by a constant multiple of $f(n)$

In Pictures



In *Math*

$$T(n) = O(f(n))$$

"if and only if" $\rightarrow$ $\Leftrightarrow$

"for all"

$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$

"there exists"

"such that"

$$T(n) \leq c \cdot f(n)$$

PROVING BIG-O BOUNDS

If you're ever asked to formally prove that T(n) is O(f(n)), use the *MATH* definition:

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$\mathbf{T(n) \leq c \cdot f(n)}$$

must be constants!
i.e. c & n_0 cannot
depend on n!

- To **prove** T(n) = O(f(n)), you need to announce your c & n_0 up front!
  - Play around with the expressions to find appropriate choices of c & n_0 (positive constants)
  - Then you can write the proof! Here how to structure the start of the proof:

"Let c = __ and n_0 = __. We will show that T(n) ≤ c·f(n) for all n ≥ n_0."

DISPROVING BIG-O BOUNDS

If you're ever asked to formally disprove that T(n) is O(f(n)), use **proof by contradiction!**

For sake of contradiction, assume that T(n) is O(f(n)). In other words, assume there does indeed exist a choice of c & n_0 s.t. $\forall\ n \geq n_0$, $\mathbf{T(n) \leq c \cdot f(n)}$

pretend you have a friend that comes up and says "I have a c & n_0 that will prove T(n) = O(f(n))!!!", and you say "ok fine, let's assume your c & n_0 does prove T(n) = O(f(n))"

$\downarrow$

Treating c & n_0 as variables, derive a contradiction!

although you are skeptical, you'll entertain your friend by saying: "let's see what happens. [some math work... and then...] AHA! regardless of what your constants c & n_0, trusting you has led me to something *impossible!!!*"

$\downarrow$

Conclude that the original assumption must be false, so **T(n) is *not* O(f(n))**.

you have triumphantly proven your silly (or lying) friend wrong.

BIG-O EXAMPLES

lower order terms don't matter!

$$\log_2 n + 15 = O(\log_2 n)$$

remember, big-O is upper bound!

$$3^n = O(4^n)$$

Polynomials

Say $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ is a polynomial of degree $k \geq 1$.

Then:

i. $p(n) = O(n^k)$
ii. $p(n)$ is **not** $O(n^{k-1})$

constant multipliers & lower order terms don't matter

$$6n^3 + n \log_2 n = O(n^3)$$

$$25 = O(1)$$

[any constant] = O(1)

BIG-Ω NOTATION

Let T(n) & f(n) be functions defined on the positive integers.

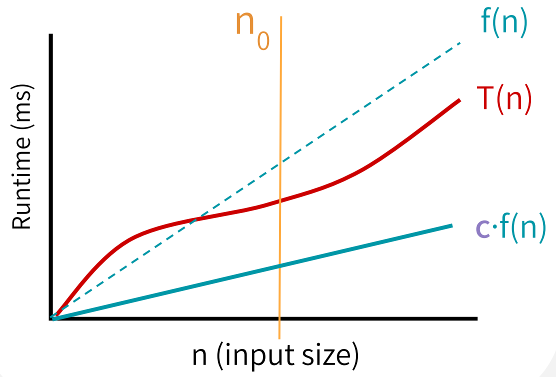
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

What do we mean when we say "T(n) is Ω(f(n))"?

In English

T(n) = Ω(f(n)) if and only if T(n) is eventually **lower bounded** by a constant multiple of f(n)

In Pictures



In *Math*

$$T(n) = \Omega(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$T(n) \geq c \cdot f(n)$$

inequality switched directions!

BIG-Θ NOTATION

We say **"T(n) is Θ(f(n))"** if and only if both

$$\mathbf{T(n) = O(f(n))}$$

and

$$\mathbf{T(n) = \Omega(f(n))}$$

$$T(n) = \Theta(f(n))$$
$$\Leftrightarrow$$
$$\exists\ c_1, c_2, n_0 > 0 \text{ s.t. } \forall\ n \geq n_0,$$
$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

ASYMPTOTIC NOTATION CHEAT SHEET

BOUND	DEFINITION (HOW TO PROVE)	WHAT IT REPRESENTS
$T(n) = O(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \geq n_0,\ T(n) \leq c \cdot f(n)$	upper bound
$T(n) = \Omega(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0$ s.t. $\forall\ n \geq n_0,\ T(n) \geq c \cdot f(n)$	lower bound
$T(n) = \Theta(f(n))$	$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$	tight bound

سوال؟

حل با روش جایگذاری

الگوریتم، اثبات درستی، زمان اجرا

SUBSTITUTION METHOD

1. Guess what the answer is (expand for a few iterations)
2. Prove your guess is correct (using induction)

4 INGREDIENTS OF INDUCTION

INDUCTIVE HYPOTHESIS (IH)

This is a statement that's basically what you're trying to prove, except it's written in terms of some variable (e.g. **i**). We need to set up the inductive hypothesis clearly, and our goal in the next three steps is to prove that the IH holds for a whole *range* of values for **i**.

BASE CASE

First establish that the inductive hypothesis holds for some base case value(s) of **i**.

INDUCTIVE STEP *(weak induction version)*

Next, assume that the inductive hypothesis holds when **i** takes on some value **k**.
Now prove that the IH holds as well when **i** takes on the value **k+1**.

CONCLUSION

By induction, conclude that the IH holds across the range of **i** you're dealing with.

SUBSTITUTION METHOD: EXAMPLE

$T(n) = 2 \cdot T(n/2) + n$

$T(1) = 1$

$\rightarrow$

Our guess from Step 1:

$\mathbf{T(n) = n \log n + n}$

STEP 2: Try to prove your guess!

- **Inductive Hypothesis**: $T(n) = n \log n + n$
- **Base case**: Prove IH holds for $n = 1$. $T(1) = 1 = 1 \log 1 + 1$.
- **Inductive step**:
  - Let $k > 1$. Assume that the IH holds for all n such that $1 \le n < k$.
  - $$\begin{aligned} T(k) &= 2 \cdot T(k/2) + k \\ &= 2 \cdot ((k/2)(\log (k/2)) + (k/2)) + k \\ &= 2 \cdot ((k/2)(\log k - 1 + 1)) + k \\ &= 2 \cdot (k/2)(\log k) + k \\ &= k \log k + k \end{aligned}$$
- **Conclusion:** By induction, $T(n) \le n \log n + n$ for all $n > 0$.

This satisfies the Big-O definition for $O(n \log n)$ (imagine choosing $c = 2$, $n_0 = 1$)

SUBSTITUTION METHOD: EXAMPLE 2

$T(n) = T(n/5) + T(7n/10) + n$

$T(n) = 1$ when $1 \le n \le 10$

$\longrightarrow$

Our guess from Step 1:

T(n) is O(n)

STEP 2: Prove it!

Use a placeholder **C** constant in the big-O proof. We don't know what C should be yet, but let's go through the proof leaving it as C and then figure out what works.

- **Inductive Hypothesis**: $T(n) \le \mathbf{C}n$
- **Base case**: Prove IH holds for $1 \le n \le 10$. $T(n) = 1 \le \mathbf{C}n$
- **Inductive step**:
  - Let $k > 10$. Assume that the IH holds for all n such that $1 \le n < k$.
  - $$\begin{aligned} T(k) &= k + T(k/5) + T(7k/10) \\ &\le k + \mathbf{C} \cdot (k/5) + \mathbf{C} \cdot (7k/10) \\ &= k \cdot (1 + \mathbf{C}/5 + 7\mathbf{C}/10) \\ &\le \mathbf{C}k \quad ??? \end{aligned}$$
  - (If we find the right C, then we've shown IH holds for n = k)

Whatever we choose C to be, we know C needs to be at least 1

We can just solve for C:

$1 + \mathbf{C}/5 + 7\mathbf{C}/10 \le \mathbf{C}$

$1 + 9\mathbf{C}/10 \le \mathbf{C}$

$1 \le \mathbf{C}/10$

So let's choose C = 10!

SUBSTITUTION METHOD: EXAMPLE 2

$T(n) = T(n/5) + T(7n/10) + n$

$T(n) = 1$ when $1 \le n \le 10$

Our guess from Step 1:

T(n) is O(n)

STEP 2: Prove it!

We can choose C = 10!

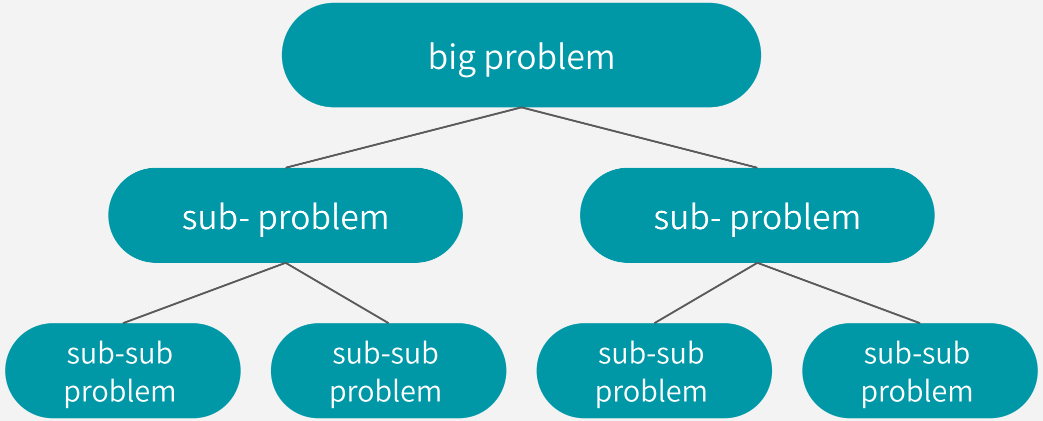
- **Inductive Hypothesis**: $T(n) \le \mathbf{10}n$
- **Base case**: Prove IH holds for $1 \le n \le 10$. $T(n) = 1 \le \mathbf{10}n$
- **Inductive step**:
  - Let $k > 10$. Assume that the IH holds for all n such that $1 \le n < k$.
  - $$\begin{aligned} T(k) &= k + T(k/5) + T(7k/10) \\ &\le k + \mathbf{10} \cdot (k/5) + \mathbf{10} \cdot (7k/10) \\ &= k + 2k + 7k \\ &= \mathbf{10}k \end{aligned}$$
  - Thus, the IH holds for $n = k$
- **Conclusion:** With $C = 10$ and $n_0 = 1$, $T(n) \le Cn$ for all $n \ge n_0$. By the Big-O definition, $T(n) = O(n)$.

یادآوری روش تقسیم و حل و مرتب سازی ادغامی

MERGESORT

FROM DATA STRUCTURE COURSE!

- **DIVIDE-AND-CONQUER: an algorithm design paradigm**
  1. break up a problem into smaller subproblems
  2. solve those subproblems *recursively*
  3. combine the results of those subproblems to get the overall answer



MERGESORT: PSEUDOCODE

Intuition: Divide and Conquer. If you sort your left and right halves, it's easier to "Merge" them into a sorted list.

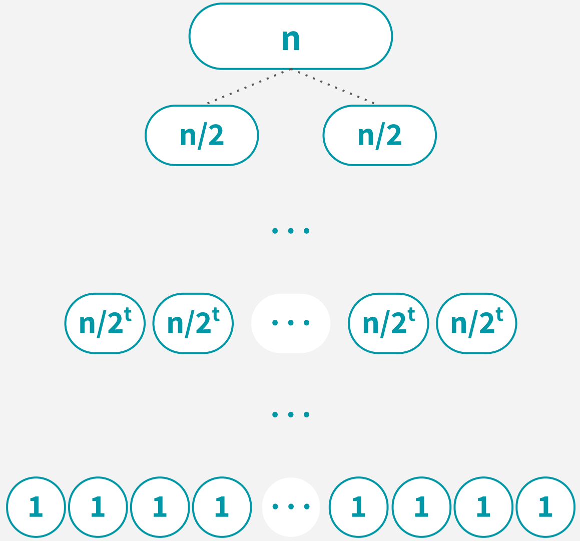
```
MERGESORT(A):
    n = len(A)
    if n <= 1:
        return A
    L = MERGESORT(A[0:n/2])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

```
MERGE*(L,R):
    result = length n array
    i = 0, j = 0
    for k in [0,...,n-1]:
        if L[i] < R[j]:
            result[k] = L[i]
            i += 1
        else:
            result[k] = R[j]
            j += 1
    return result
```

* Not complete! Some corner cases are missing.

MERGESORT RECURSION TREE

If a subproblem is of size **n**, then the work done in that subproblem is **O(n)**.
⇒ **Work ≤ c · n** (c is a constant)



Level	# of Problems	Size of each Problem	Work done per Problem	Total work on this level
0	1	n	$c \cdot n$	**O(n)**
1	2^1	$n/2$	$c \cdot (n/2)$	$2^1 \cdot c \cdot (n/2) =$ **O(n)**
…				
t	2^t	$n/2^t$	$c \cdot (n/2^t)$	$2^t \cdot c \cdot (n/2^t) =$ **O(n)**
…				
$\log_2 n$	$2^{\log_2 n} = n$	1	$c \cdot (1)$	$n \cdot c \cdot (1) =$ **O(n)**

We have ($\log_2 n + 1$) levels, each level has O(n) work total ⇒ **O(n log n)** work overall!

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations ⇒ yay!

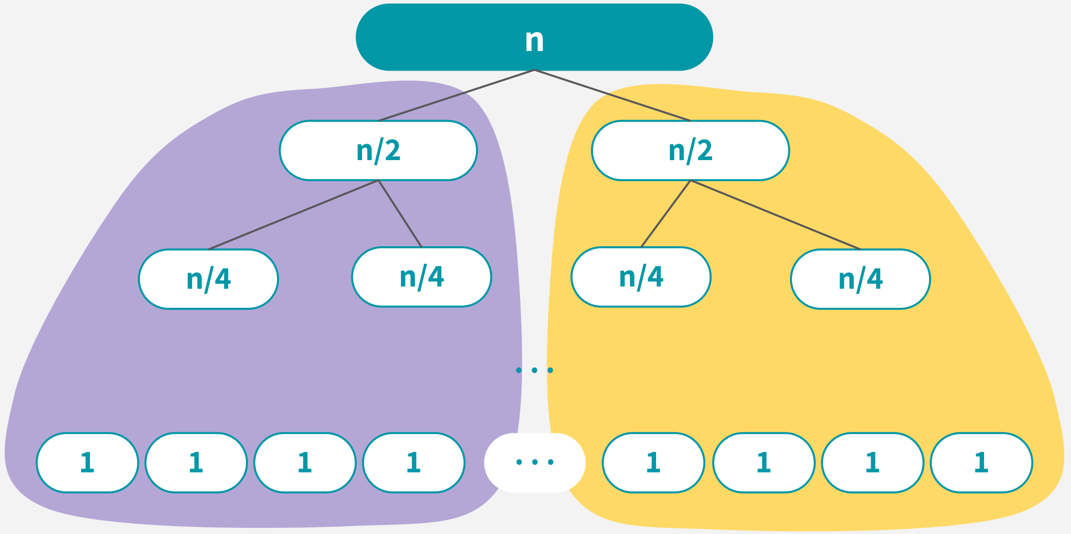
RECURSIVE ALGORITHMS

1. **Inductive hypothesis**: your algorithm is correct for sizes *up to* **i**
2. **Base case**: IH holds for i < small constant
3. **Inductive step**:
    - assume IH holds for k ⇒ prove k+1, *OR*
    - assume IH holds for {1,2,...,k-1} ⇒ prove k.
4. **Conclusion**: IH holds for i = n ⇒ yay!

رابطه بازگشتی و قضیه اصلی

RECURRENCE RELATIONS

To build the recurrence relation for MergeSort, we can think of its runtime as follows:



Work in the whole tree =

total work in LEFT recursive call (left subtree)

+

total work in RIGHT recursive call (right subtree)

+

work done *within* top problem

work to create suproblems & "merge" their solutions

THE MASTER THEOREM

Suppose that **$a \geq 1$**, **$b > 1$**, and **d** are constants (i.e. independent of **n**).

Suppose **$T(n) = a \cdot T(n/b) + O(n^d)$**. The Master Theorem states:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

a: number of subproblems (branching factor)

b: factor by which input size shrinks (shrinking factor)

d: need to do $O(n^d)$ work to create subproblems + “merge” their solutions

MASTER THEOREM "INTUITION"

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

amount of work at each level ~ same

highest level "dominates": work per level decreases (subproblem work shrinks more!)

leaves "dominate": work per level increases (branch more!)

a: number of subproblems (branching factor)

b: factor by which input size shrinks (shrinking factor)

d: need to do $O(n^d)$ work to create subproblems + "merge" their solutions

SUBSTITUTION METHOD

1. Guess what the answer is (expand for a few iterations)
2. Prove your guess is correct (using induction)

This is a good technique to turn to if you find that the Master Theorem doesn't work. It's also especially helpful with recurrences that have differently sized subproblems (i.e. when the recursion tree & table aren't helpful either).

سوال؟

مرتب سازی سریع

QUICKSORT OVERVIEW

EXPECTED RUNNING TIME

$O(n \log n)$

WORST-CASE RUNNING TIME

$O(n^2)$

In practice, it works great! It's competitive with MergeSort (& often better in some contexts!), and it runs *in place* (no need for lots of additional memory)

QUICKSORT: THE IDEA

Let's use DIVIDE-and-CONQUER again!

Select a pivot *at random*

Partition around it

Recursively sort L and R!

QUICKSORT RECURRENCE RELATION

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Recurrence Relation for QUICKSORT

$$T(n) = T(|L|) + T(|R|) + O(n)$$
$$T(0) = T(1) = O(1)$$

EXPECTED RUNTIME = O(n log n)

AN **INCORRECT** PROOF:

- E[|**L**|] = E[|**R**|] = (n - 1)/2
- T(n) = T(|**L**|) + T(|**R**|) + O(n) could be written as
  T(n) = 2T(n/2) + O(n).
- Therefore, the expected running time is O(n log n)!

Why is this wrong?

EXPECTED RUNTIME = O(n log n)

AN

Basically:

E[f(x)] is *not necessarily* the same as f(**E[**x**]**)

e.g. $E[X^2]$ is not the same as $(E[X])^2$

We were reasoning about T(**E[**x**]**) instead of **E[**T(x)**]**

Why is this wrong?

HOW MANY COMPARISONS?

Each pair of elements is compared either **0** or **1** times.

Let $\mathbf{X_{a,b}}$ be a Bernoulli/indicator random variable such that:

$\mathbf{X_{a,b} = 1}$ if **a** and **b** are compared

$\mathbf{X_{a,b} = 0}$ otherwise

In our example, $\mathbf{X_{2,5}}$ took on the value **1** since **2** and **5** were compared.
On the other hand, $\mathbf{X_{3,7}}$ took on the value **0** since **3** and **7** are *not* compared.

Total number of comparisons =

$$\mathbb{E}\left[\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} X_{a,b}\right] \underset{\text{by linearity of expectation!}}{=} \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1} \boxed{\mathbb{E}\left[X_{a,b}\right]}$$

We need to figure out this value!

QUICKSORT EXPECTED RUNTIME

Total number of comparisons =

$$\sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\mathbb{E}[X_{a,b}] = \sum_{a=0}^{n-2}\sum_{b=a+1}^{n-1}\frac{2}{b-a+1}$$

We just computed E[$X_{a,b}$] = P($X_{a,b,}$= 1)

$$= \sum_{a=0}^{n-2}\sum_{c=1}^{n-a-1}\frac{2}{c+1}$$

Introduce c = b – a to make notation nicer

$$\leq \sum_{a=0}^{n-1}\sum_{c=1}^{n-1}\frac{2}{c+1}$$

Increase summation limits to make them nicer (hence the ≤)

$$= 2n\sum_{c=1}^{n-1}\frac{1}{c+1}$$

Nothing in the summation depends on a, so pull 2 out

$$\leq 2n\sum_{c=1}^{n-1}\frac{1}{c}$$

decrease each denominator ⟶ we get the harmonic series!

$$= O(n\log n)$$

If E[# comparisons] = O(n log n), does this mean E[running time] is also O(n log n)?

YES! Intuitively, the runtime is dominated by comparisons.

QUICKSORT

```
QUICKSORT(A):
    if len(A) <= 1:
        return
    pivot = random.choice(A)
    PARTITION A into:
        L (less than pivot) and
        R (greater than pivot)
    Replace A with [L, pivot, R]
    QUICKSORT(L)
    QUICKSORT(R)
```

Worst case runtime:
$O(n^2)$

Expected runtime:
$O(n \log n)$

QUICKSORT vs. MERGESORT

	QuickSort (random pivot)	**MergeSort** (deterministic)
Runtime	**Worst-case: O(n^2)** **Expected: O(n log n)**	**Worst-case: O(n log n)**
Used by	Java (primitive types), C (qsort), Unix, gcc…	Java for objects, perl
In-place? **(i.e. with O(log n) extra memory)**	Yes, pretty easily!	Easy if you sacrifice runtime (O(nlogn) MERGE runtime). Not so easy if you want to keep runtime & stability.
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

سوال؟

انتخاب kامین عضو

LINEAR SELECTION: THE IDEA

Let's use DIVIDE-and-CONQUER!

Select a pivot

Partition around it

Recurse!

kind of like a "binary search" for the kth smallest element (except that the array isn't sorted!)

LINEAR SELECTION: PSEUDOCODE

Base Case:
if len(A) = 1, then just go ahead and return the element itself

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

Case 1:
We got lucky and found exactly the kth smallest!

Case 2:
The kth smallest is in the first part of the array (L)

Case 3:
The kth smallest is in the second part of the array (R)

RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = GET_PIVOT(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

Recurrence Relation for SELECT

For now, assume we'll pick the pivot in time O(n)

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases}$$

THE WORST PIVOT

The WORST pivot: picking the max or the min each time!

Then, in the worst case, the recurrence relation looks like T(n) = T(n-1) + O(n).

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \Rightarrow \mathbf{T(n) \leq T(n-1) + O(n)}$$

This ends up being $\Omega(n^2)$!

A call to SELECT(A, n/2) would already consist of ~n/2 recursive calls (each with a subarray of length at least n/2)!

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

`len(L)` = `len(R)` = (n-1)/2

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \Longrightarrow \mathbf{T(n) \leq T(n/2) + O(n)}$$

a = 1
b = 2
d = 1

$$\mathbf{a < b^d}$$

Suppose $\mathbf{T(n) = a \cdot T(n/b) + O(n^d)}$. The Master Theorem states:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

`len(L)` … (n-1)/2

$$T(n) = \begin{cases} O(n) & \texttt{l}\ldots \\ T(\texttt{len(L)}) + O(n) & \texttt{l}\ldots \\ T(\texttt{len(R)}) + O(n) & \texttt{l}\ldots \end{cases}$$

With the ideal pivot, the runtime would be:

O(n)

$$\mathbf{T(n) \leq T(n/2) + O(n)}$$

a = 1
b = 2
d = 1

$$\mathbf{a < b^d}$$

Suppose **T(n) = a · T(n/b)** … Master Theorem states:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

THE IDEAL PIVOT

The IDEAL pivot: splits the input array exactly in half!

$$T(n) = \begin{cases} O(\\ T(1 \\ T(1 \end{cases}$$

Sadly, the pivot to divide the input in half is the

MEDIAN

aka ***SELECT(A, n/2)***

aka exactly the problem we're trying to solve...

+ O(n)

$\mathbf{b^d}$

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

THE GOOD-ENOUGH PIVOT

The GOOD-ENOUGH pivot: splits the input array kind of in half!

3n/10 < `len(L)` < 7n/10

3n/10 < `len(R)` < 7n/10

If we could fetch this good-enough pivot in time O(n), let's say, the recurrence looks like:

$$T(n) = \begin{cases} O(n) & \texttt{len(L) == k-1} \\ T(\texttt{len(L)}) + O(n) & \texttt{len(L) > k-1} \\ T(\texttt{len(R)}) + O(n) & \texttt{len(L) < k-1} \end{cases} \Longrightarrow \mathbf{T(n) \leq T(7n/10) + O(n)}$$

a = 1
b = 10/7
d = 1

$\mathbf{a < b^d}$

Suppose $\mathbf{T(n) = a \cdot T(n/b) + O(n^d)}$. The Master Theorem states:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

میانه ی میانه ها!

MEDIAN-OF-MEDIANS

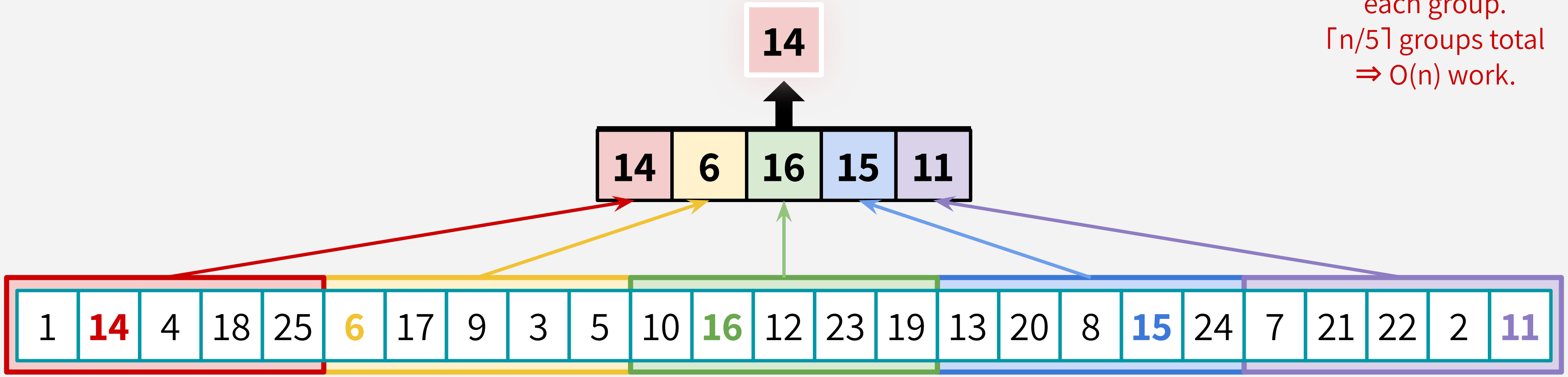
GOAL: get a proxy for the true median by finding the exact median of all the sub-medians!

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the sub-median of each small group (3rd smallest out of the 5)

Find the median of all the sub-medians (call SELECT)

constant work for each group. ⌈n/5⌉ groups total ⇒ O(n) work.



ANALYZING RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else:
        return SELECT(R, k-len(L)-1)
```

O(n) work outside of recursive calls
(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

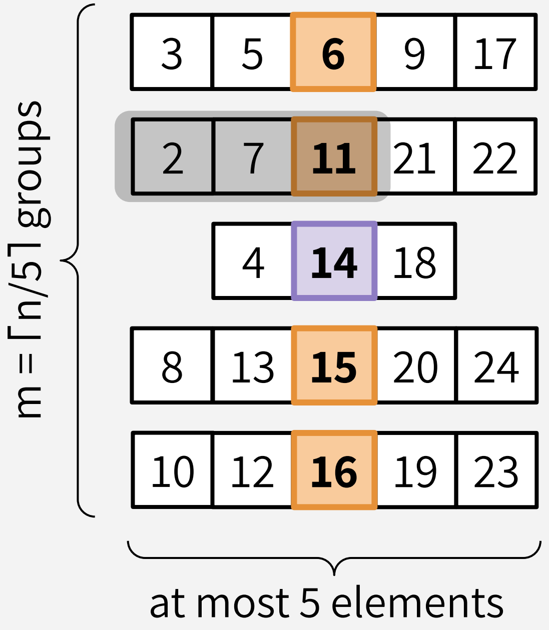
T(n/5) work hidden in this recursive call
(remember, MEDIAN_OF_MEDIANS calls SELECT on ⌈n/5⌉-size array)

T(???) work hidden in this recursive call
What is the maximum size of either L or R?

ANALYZING RUNTIME

Claim: MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians** + ~~2 elements from the group containing the **median of medians**~~

$$3 \cdot (\lceil m/2 \rceil - 1 - 1) \cancel{+2}$$

To exclude the group with the **median of medians**

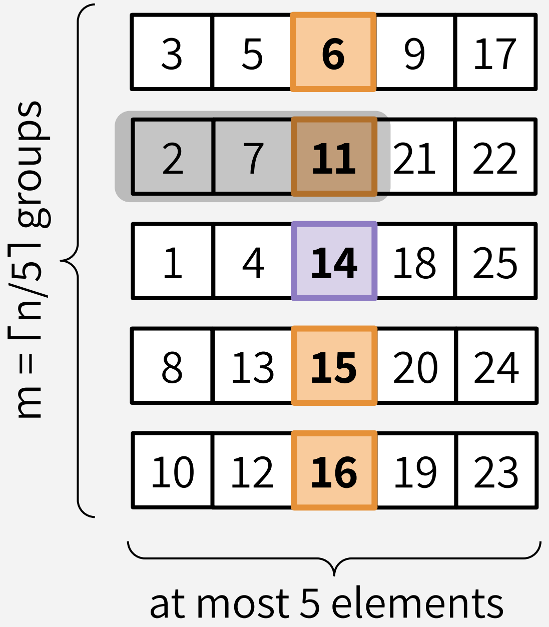
To exclude any of those groups that might be a "leftover" group!

The group with the **median of medians** might be a "leftover" group! Might as well just get rid of the +2 to be safe

ANALYZING RUNTIME

Claim: MEDIAN_OF_MEDIANS will choose a pivot greater than at least 3n/10 - 6 elements

(The same reasoning we're about to do also shows that the pivot will be less than at least 3n/10 - 6 elements)



At least how many elements are guaranteed to be **smaller** than the median of medians?

3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians**

$$\begin{aligned} &3 \cdot (\lceil m/2 \rceil - 2) \\ = \ &3 \cdot (\lceil \lceil n/5 \rceil /2 \rceil - 2) \\ \geq \ &3 \cdot (n/10 - 2) \\ = \ &3n/10 - 6 \end{aligned}$$

ANALYZING RUNTIME

We can similarly show the inverse:

3n/10 - 6 ≤ `len(L)` ≤ 7n/10 + 5

3n/10 - 6 ≤ `len(R)` ≤ 7n/10 + 5

What does the recurrence relation for T(n) look like?

T(n) ≤ T(n/5) + T(7n/10) + O(n)

LINEAR-TIME SELECTION

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

O(n)

Worst-case Runtime!

سوال؟

کران پایین برای مرتب سازی

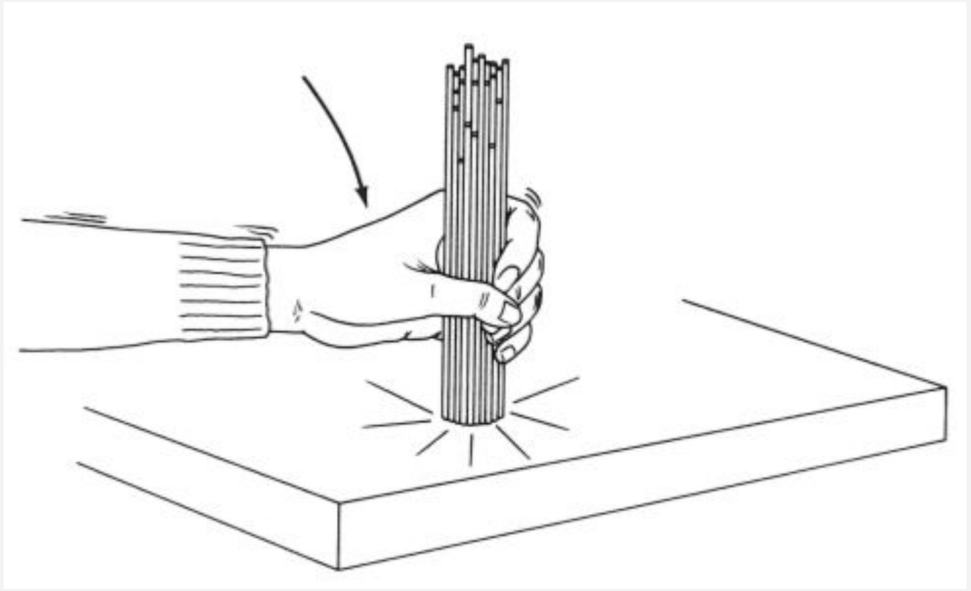
INTRODUCING... SPAGHETTI SORT?

Input: A sequence of real numbers

Algorithm:

- For each number, break off a piece of spaghetti whose length is that number **O(n)**
- Take all the spaghetti in your fist, and push their lower sides against the table **O(1)**
- Lower your other hand on the bundle of spaghetti - the first spaghetto you touch is the longest one. Remove it, transcribe its length, and repeat until all spaghetti have been removed. **O(n)**

Total Runtime:O(n)



WHAT IS OUR MODEL OF COMPUTATION?

Input: array of elements

Output: sorted array

Operations allowed: comparisons

Input: some real numbers

Output: sorted real numbers

Operations allowed: breaking spaghetti, dropping on tables, lowering hand

In a CS class where we're more concerned with what computers can do, the first model seems more reasonable.

COMPARISON-BASED SORTING

- **You want to sort an array of items**
- **You can't access the items' values directly: you can only *compare* two items and find out which is bigger or smaller.**
- Examples: Insertion Sort, MergeSort, QuickSort

"Comparison-based sorting algorithms" are general-purpose.

The algorithm makes no assumption about the input elements other than that they belong to some totally ordered set.

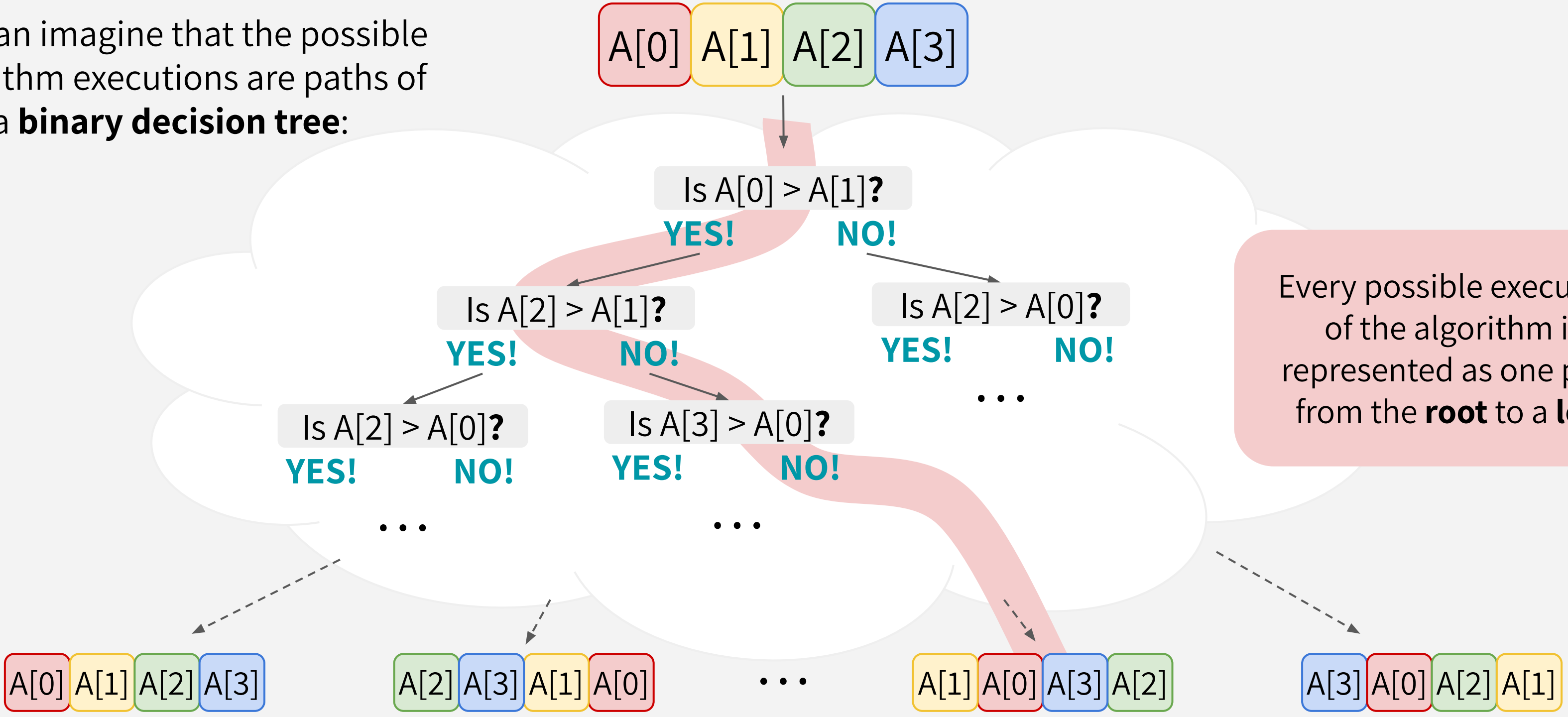
COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decision tree**:

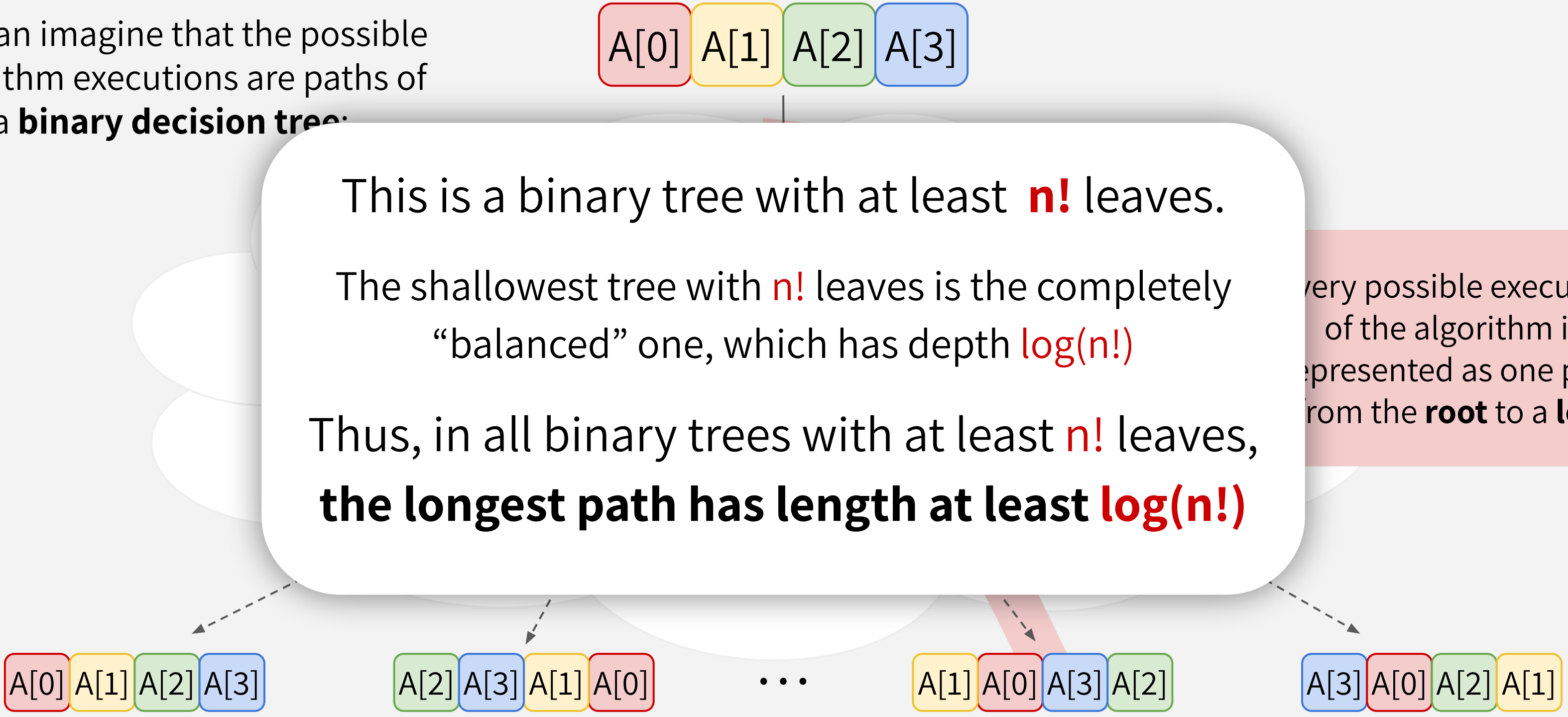


Every possible execution of the algorithm is represented as one path from the **root** to a **leaf**

Your algorithm needs to be able to output any one of __**n!**__ possible orderings

COMPARISON-BASED SORTING

You can imagine that the possible algorithm executions are paths of a **binary decision tree**:



This is a binary tree with at least **n!** leaves.

The shallowest tree with n! leaves is the completely "balanced" one, which has depth log(n!)

Thus, in all binary trees with at least n! leaves, **the longest path has length at least log(n!)**

very possible execution of the algorithm is epresented as one path rom the **root** to a **leaf**

Your algorithm needs to be able to output any one of **n!** possible orderings

COMPARISON-BASED SORTING

The longest path has length at least log(n!)

Consequently, any execution of a comparison-based sorting algorithm has to perform at least log(n!) steps.

The worst-case runtime is at least log(n!) = Ω(n log n).

$$\begin{aligned}
\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\
&\geq \log\left(\frac{n}{2}+1\right) + \log\left(\frac{n}{2}+2\right) + \cdots + \log(n-1) + \log n \\
&\geq \left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) \\
&= \frac{1}{2}n\big(\log n - \log 2\big) \\
&= \Omega(n \log n)
\end{aligned}$$

PROOF IDEA

Theorem:

Any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time.

- Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves
- The worst-case runtime is at least the length of the longest path in the decision tree
- All decision trees with n! leaves have a longest path with length at least $\log(n!) = \Omega(n \log n)$
- So, any comparison-based sorting algorithm must have worst-case runtime at least $\Omega(n \log n)$

سوال؟

مرتب سازی خطی

الگوریتم های مرتب سازی که بر مبنای مقایسه نیستند!

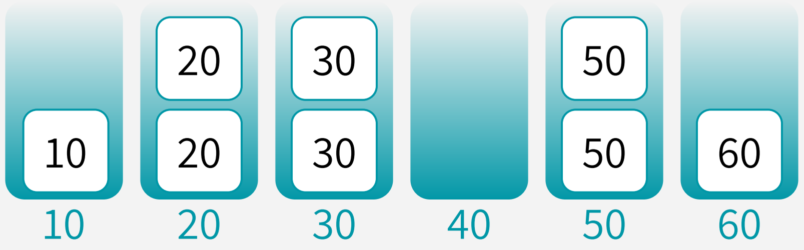
COUNTING SORT

We assume that there are only k different possible values in the array (and we know these k values in advance)

For example: elements are integers in {10, 20, 30, 40, 50, 60}

Input: 30 50 20 30 10 60 50 20

Buckets:



Output: 10 20 20 30 30 50 50 60

Sorted in time:
O(n)

COUNTING SORT

Assumptions:

We are able to know what bucket to put something in.

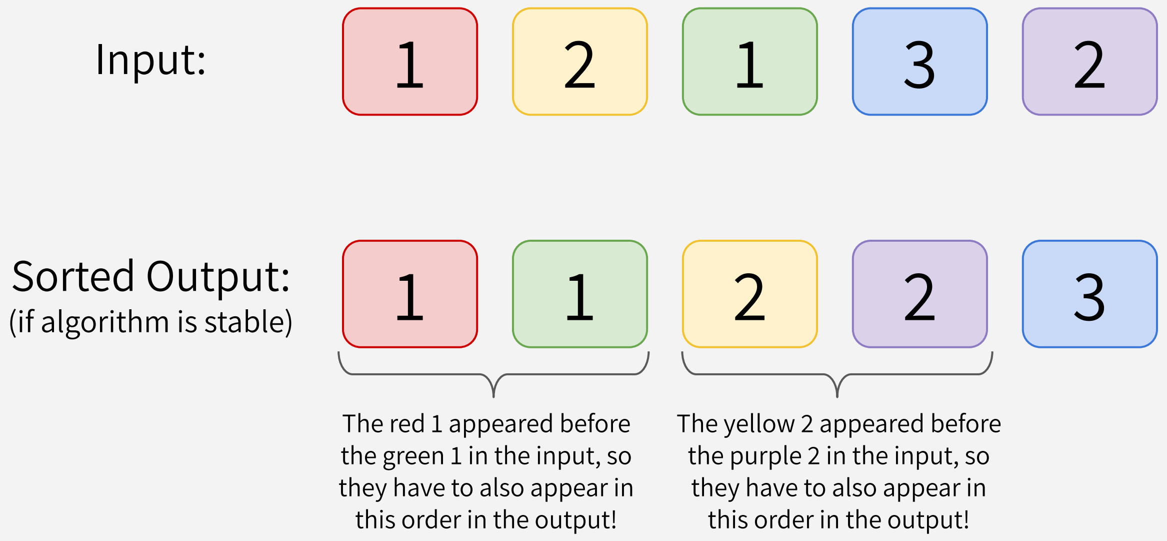
We know what values might show up ahead of time.

There aren’t too many such values.

If there are too many possible values that could show up, then we need a bucket per value…
This can easily amount to a lot of space.

STABLE SORTING

We say a sorting algorithm is STABLE if two objects with equal values appear in the same order in the sorted output as they appear in the input.



Input:
1
2
1
3
2
Sorted Output:
(if algorithm is stable)
1
1
2
2
3
The red 1 appeared before the green 1 in the input, so they have to also appear in this order in the output!
The yellow 2 appeared before the purple 2 in the input, so they have to also appear in this order in the output!

RADIX SORT

For sorting integers where the maximum value of any integer is M.
(This can be generalized to lexicographically sorting strings as well)

IDEA:

Perform CountingSort on the least-significant digit first,
then perform CountingSort on the next least-significant, and so on...

Instead of a bucket per possible value, **we just need to maintain a bucket per possible value that a single digit (or character) can take on!**

e.g. 10 buckets labeled 0, 1, …, 9

USING A DIFFERENT BASE

RadixSort with base 10 doesn't seem so good… How does the base affect the runtime?

Let's say base **r**

How many iterations are there?

d = $\lfloor \log_r M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **r** buckets + put n numbers in **r** buckets ⇒ **O(n + r)**

What is the total running time?

$O(d \cdot (n+r)) = O((\lfloor \log_r M \rfloor + 1) \cdot (n + r))$

Bigger base r ⇒ fewer iterations, but more buckets to initialize!

USING A DIFFERENT BASE

A reasonable sweet spot: **let r = n**

How many iterations are there?

$d = \lfloor \log_n M \rfloor + 1$ iterations

How long does each iteration take?

Initialize **n** buckets + put n numbers in **n** buckets ⇒ $O(n+n) = O(n)$

What is the total running time?

$O(d \cdot n) = O((\lfloor \log_n M \rfloor + 1) \cdot n)$

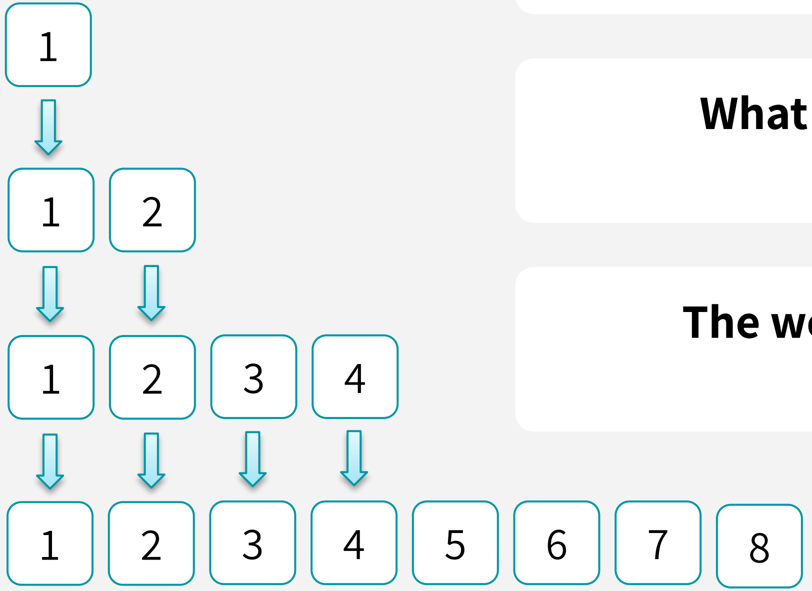
This term is a constant!

If $M \leq n^c$ for some constant c, then $O((\lfloor \log_n M \rfloor + 1) \cdot n) = O(n)$

سوال؟

تحلیل سرشکن

DYNAMIC ARRAY



We fill it with n elements. When it is FULL, we replaced it with a new array that has 2*n capacity.

What is the cost of EACH INSERTION?
What is the WORST CASE?

The worst insertion doubles the array!
So, In worst case O(n)?

AMORTIZED vs. PROBABILISTIC

- **Probabilistic analysis:**
  - Average case running time: average over all possible inputs for one algorithm (operation)
  - If using probability, called Expected Running Time.

- **Amortized analysis:**
  - No involvement of probability
  - Average performance on a sequence of operations
  - **Guarantee average performance of each operation among the sequence in worst case**

AMORTIZED ANALYSIS METHODS

- **Aggregate analysis**:
  - Total cost of n operations/n,
- **Accounting Method**:
  - Pay extra credit in each operation
  - Save extra credit on elements
  - Use extra credit for expensive operations
- **Potential method**:
  - Same as accounting method
  - But store the credit in one place as **potential energy**

AGGREGATE ANALYSIS

The **simplest** way to perform amortized analysis

How to calculate? $\frac{\text{Total cost}}{\text{\# of operations}}$

$$O\left(\sum Cost\ of\ n\ operations\right) = O\left(\begin{array}{c}\sum Cost\ of\ \boldsymbol{Cheap}\ operations \\ + \\ \sum Cost\ of\ \boldsymbol{Expensive}\ operations\end{array}\right)$$

ACCOUNTING METHOD

- Save your money for a rainy day!
- Assign every operation a **cost**
  - Use part of it for the operation
  - Save surplus **beside** new item
- **Cheap** operations will have **extra** cost
  - Will help to afford **Expensive** operations
- **Challenge**: Bank balance must always be **0 or positive**

POTENTIAL METHOD

Same as Accounting method

Pay extra for cheap operations and store extra credit.
Use stored credit for expensive operations.

Different from Accounting method

The prepaid work not as credit,
but as "potential energy", or "potential"

Potential: associated with the **whole data structure**
Credit: associated with **specific objects** in the data structure

DIFFERENCE FROM ACCOUNTING

In Accounting method, Bank balance of particular state is dependent on previous state

Potential Method uses Potential Function $\Phi(h)$

Potential function:
independently derive the potential at any state

Can compute the **potential difference**:
The change in cost between two operations

سوال؟

گراف

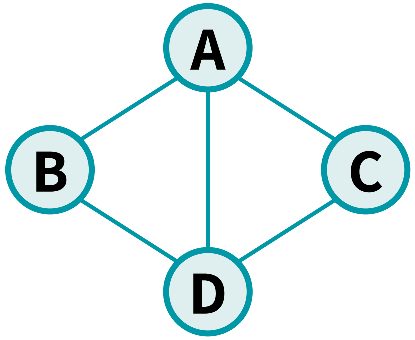
2 KINDS OF GRAPHS

We'll deal with both kinds of graphs in this class.

UNDIRECTED GRAPHS

An undirected graph has
a set of vertices (V) & a set of edges (E)

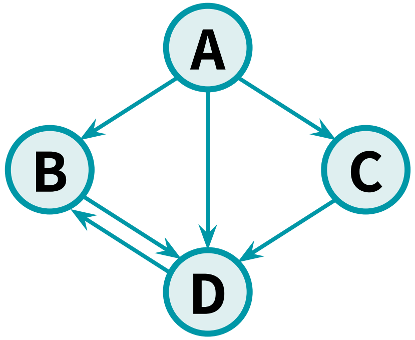
Formally,
G = (V, E)



V = {A, B, C, D}
E = { {A, B}, {A, C}, {A, D}, {B, D}, {C, D}}

DIRECTED GRAPHS

A directed graph has
a set of vertices (V) & a set of **DIRECTED** edges (E)

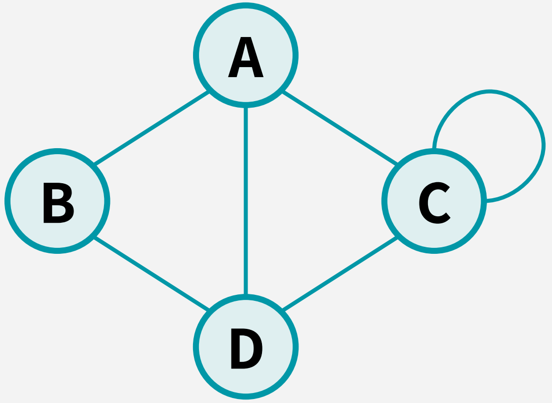


Formally,
G = (V, E)

V = {A, B, C, D}
E = { [A, B], [A, C], [A, D], [B, D], [C, D], [D, B]}

GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(An undirected graph)

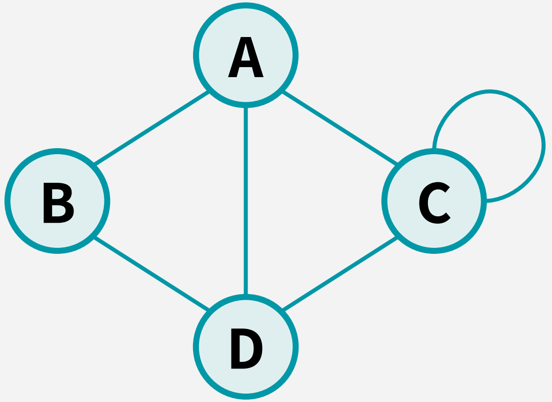
(destination)

(source)

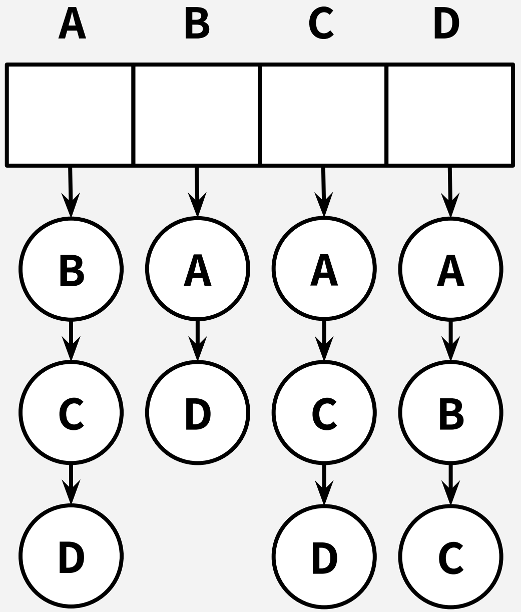
$$\begin{array}{c|cccc} & \textbf{A} & \textbf{B} & \textbf{C} & \textbf{D} \\ \hline \textbf{A} & 0 & 1 & 1 & 1 \\ \textbf{B} & 1 & 0 & 0 & 1 \\ \textbf{C} & 1 & 0 & 1 & 1 \\ \textbf{D} & 1 & 1 & 1 & 0 \end{array}$$

GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**

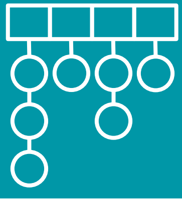


(An undirected graph)



Each list stores a node's neighbors

GRAPH REPRESENTATIONS

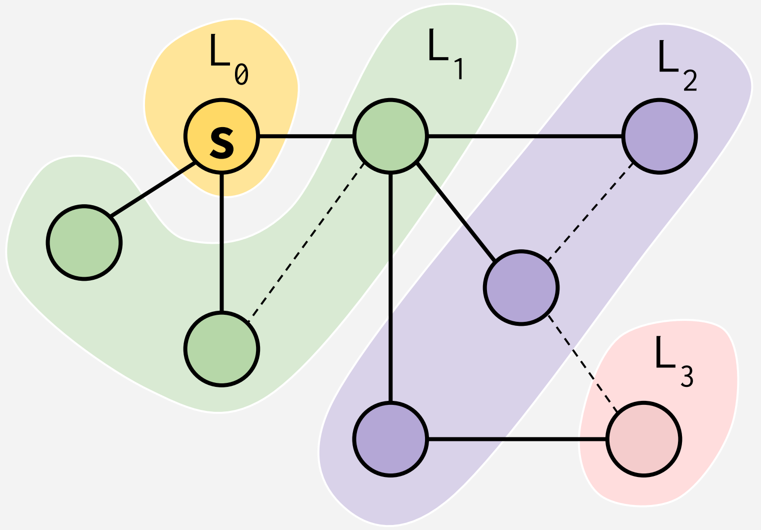
For a graph G = (V, E) where |V| = **n**, and |E| = **m**	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
EDGE MEMBERSHIP Is e = {v, w} in E?	**O(1)**	**O(deg(v))** or **O(deg(w))**
NEIGHBOR QUERY Give me v's neighbors	**O(n)**	**O(deg(v))**
SPACE REQUIREMENTS	**$O(n^2)$**	**O(n + m)**

Generally, better for sparse graphs (where m << n^2).

We'll assume this representation, unless otherwise stated.

جستجوی سطح اول (BFS)

BREADTH-FIRST SEARCH



L_i = The set of nodes we can reach in i steps from s

```
BFS(s):
    Set $L_i$ = [] for i = 0, ..., n-1
    $L_0$ = s
    for i = 0, ..., n-1:
        for u in $L_i$:
            for v in u.neighbors:
                if v not yet visited:
                    mark v as visited
                    add v to $L_{i+1}$
```

Go through all nodes in L_i and add their unvisited neighbors to L_{i+1}

BREADTH-FIRST SEARCH: RUNTIME

To explore a graph's i^{th} **connected component** (n_i nodes, m_i edges):

We visit each vertex in the CC exactly once ("visit" = grab from its L_i).
At each vertex v, we:

- Do some bookkeeping: **O(1)**
- Loop over v's neighbors & check if they are visited (& then potentially mark the neighbor & place in L_{i+1}): O(1) per neighbor → **O(deg(v))** total.

Total: $\sum_v O(\deg(v)) + \sum_v O(1) = \mathbf{O(m_i + n_i)}$

BREADTH-FIRST SEARCH: RUNTIME

To explore **the entire graph** (n nodes, m edges):

A graph might have multiple connected components! To **explore the whole graph**, we would call our BFS routine once for each connected component (note that each vertex and each edge participates in exactly one connected component). The combined running time would be:

$$O(\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

SINGLE-SOURCE SHORTEST PATH

How long is the shortest path between vertices v & *all other vertices* w?

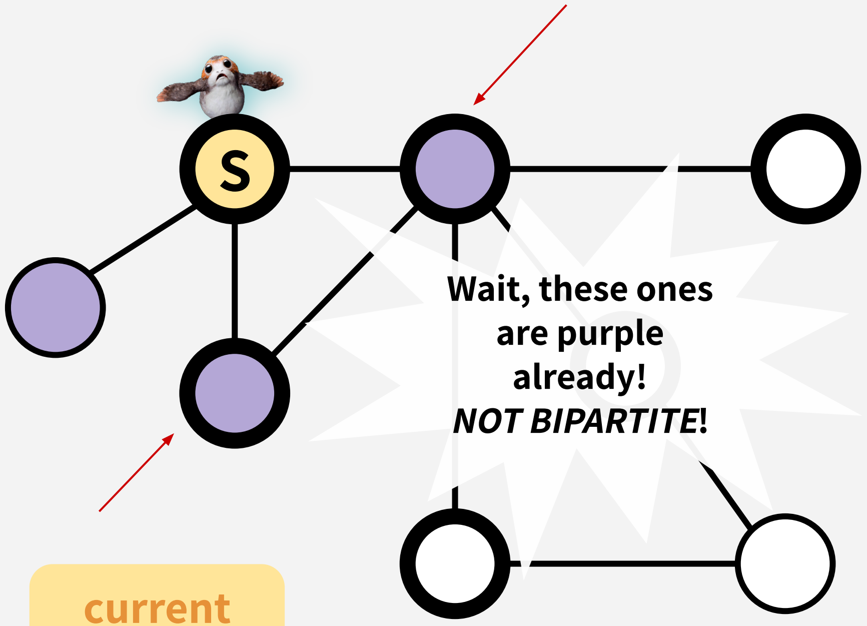
```
findAllDistances(v):
    perform BFS(v) → gives us all $L_i$
    for all w in V:
        d[w] = ∞
    for each $L_i$:
        for all w in $L_i$:
            d[w] = i
```

Runtime: O(m+n)

BIPARTITE GRAPHS

Application of BFS:

- Color the levels of the BFS tree in alternating colors (i.e. run BFS from any vertex, and alternate colors for each layer)
- If you attempt to color the same vertex different colors (i.e. revisit a node that's a different color than what you would have colored it), then the graph isn't bipartite!
- If you successfully color the whole graph without conflicts, then it is bipartite!

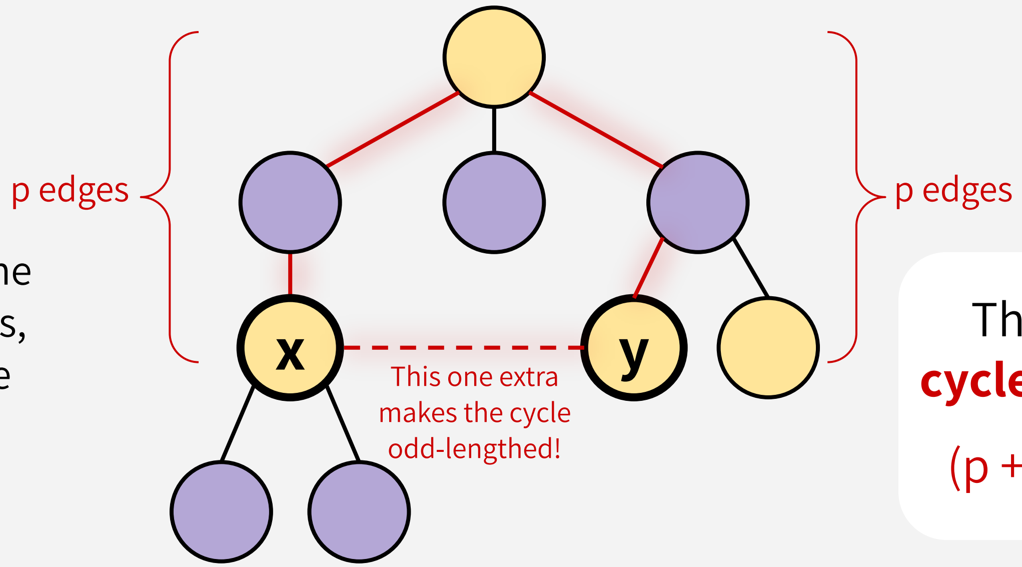


current color

BIPARTITE GRAPHS

If BFS tries to color two neighbors the same color, then it's found a **cycle of odd length** in the graph

If **x** and **y** are the same color & are neighbors, then they are on the same level.



Thus, there is a **cycle of odd length**

$(p + p + 1) = \text{odd } \#$

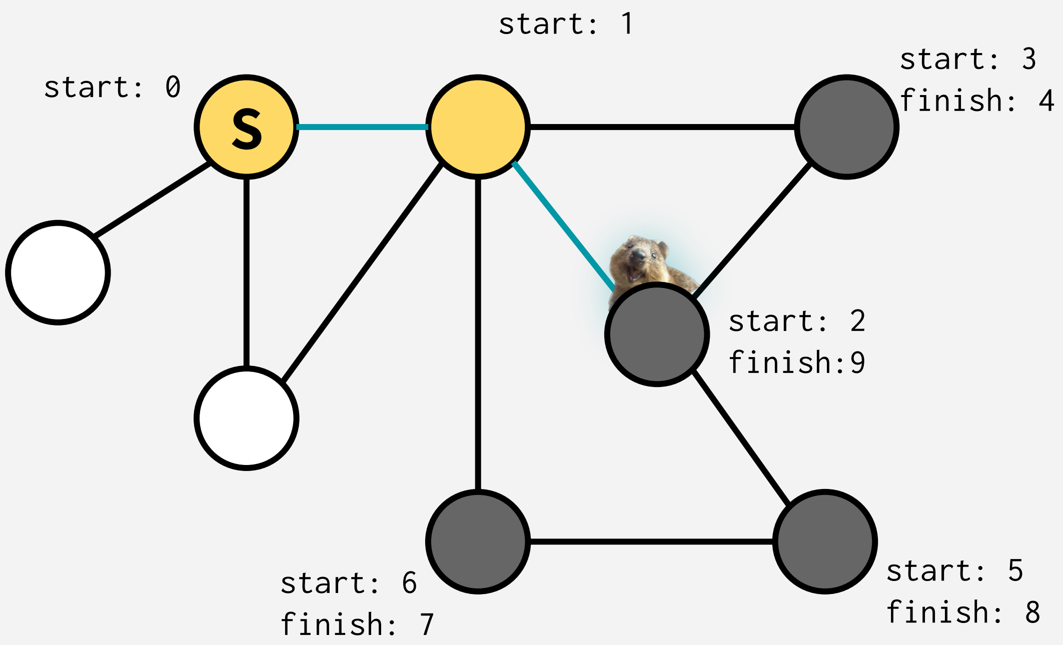
سوال؟

جستجوی عمق اول (DFS)

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)

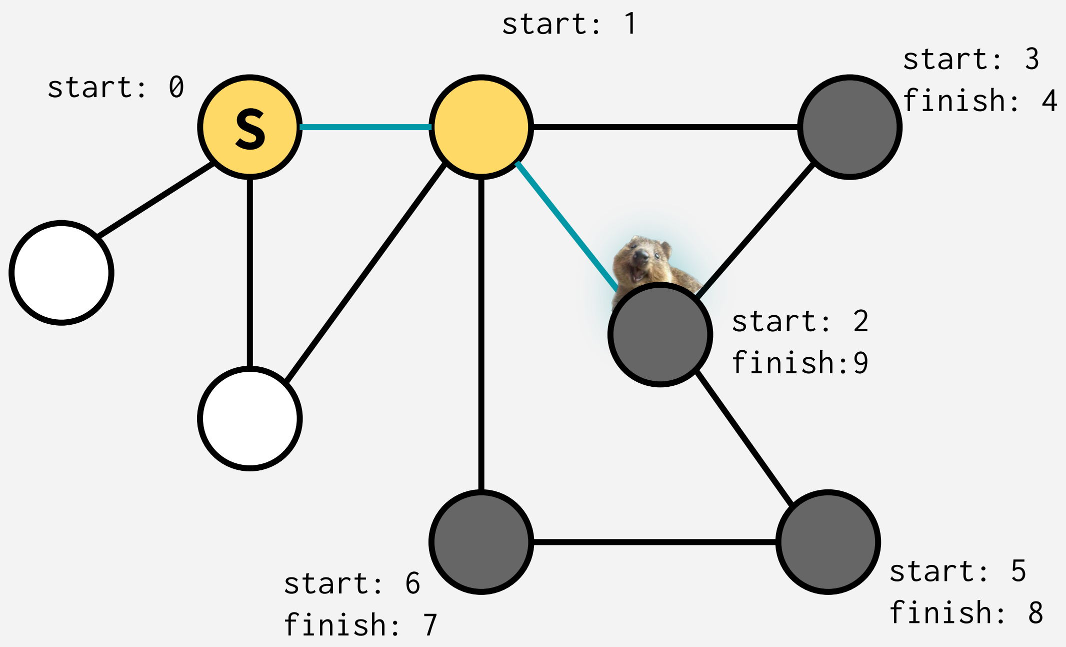


```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH

An analogy:

A smart quokka is exploring a labyrinth with chalk (to mark visited destinations) & thread (to retrace steps)



```
DFS(w, currTime):
    w.start = currTime
    currTime++
    mark w as visited
    for v in w.neighbors:
        if v is unvisited:
            currTime =
                DFS(v, currTime)
            currTime++
    w.finish = currTime
    mark w as finished
    return currTime
```

DEPTH-FIRST SEARCH: RUNTIME

To explore a graph's **ith connected component** (n_i nodes, m_i edges):

We visit each vertex in the CC exactly once ("visit" = "call DFS on").
At each vertex v, we:

- Do some bookkeeping: **O(1)**
- Loop over v's neighbors & check if they are visited (& then potentially make a recursive call): O(1) per neighbor → **O(deg(v))** total.

Total: $\sum_v O(\deg(v)) + \sum_v O(1) = \mathbf{O(m_i + n_i)}$

DEPTH-FIRST SEARCH: RUNTIME

To explore **the entire graph** (n nodes, m edges):

A graph might have multiple connected components! To **explore the whole graph**, we would call our DFS routine once for each connected component (note that each vertex and each edge participates in exactly one connected component). The combined running time would be:

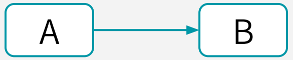
$$O(\textstyle\sum_i m_i + \sum_i n_i) = \mathbf{O(m + n)}$$

TOPOLOGICAL SORTING: THE TASK

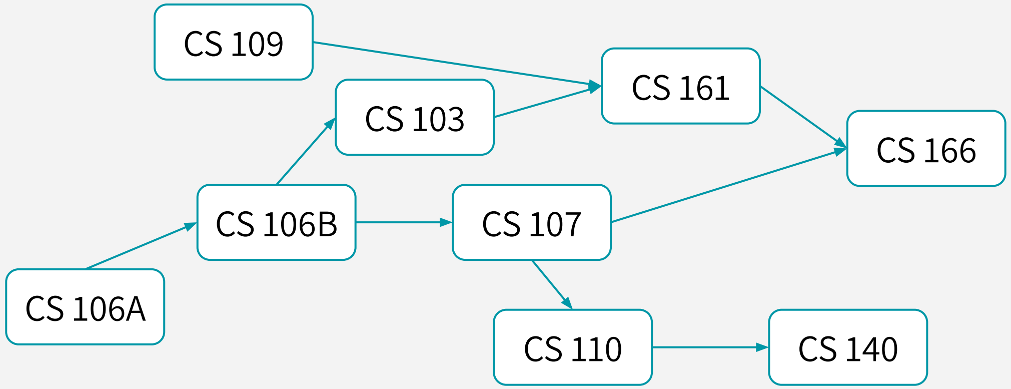
Given a DAG, find an ordering of vertices so that all of the dependency requirements are met

Example applications:

Given a package dependency graph, in what order should packages be installed?
Given a course prerequisites graph, in what order should we take classes?



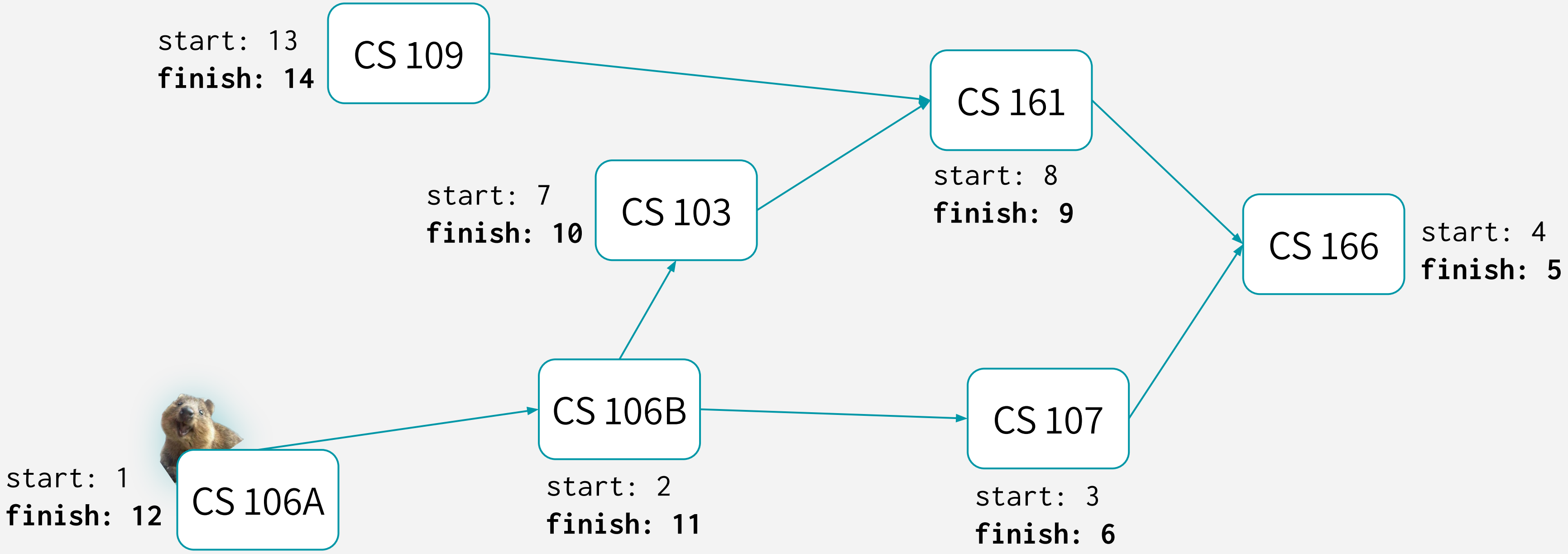
means B "depends" on A
(i.e. take class B after A)



This prerequisite graph is a DAG!
(directed & acyclic)

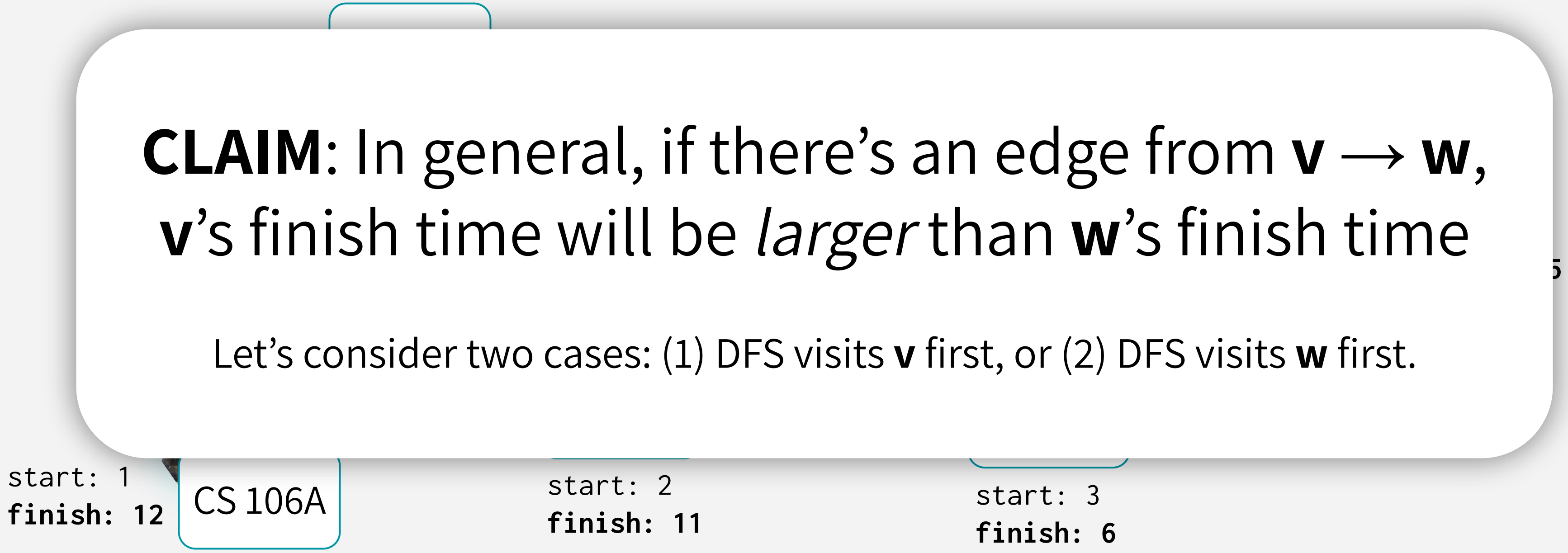
DFS WILL GET US A TOPOSORT

Let’s run DFS. What do you notice about the finish times? What does it have to do with toposort?



DFS WILL GET US A TOPOSORT

Let’s run DFS. What do you notice about the finish times? What does it have to do with toposort?



CLAIM: In general, if there’s an edge from **v** → **w**, **v**’s finish time will be *larger* than **w**’s finish time

Let’s consider two cases: (1) DFS visits **v** first, or (2) DFS visits **w** first.

سوال؟

برنامه نویسی پویا

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.
e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!
This means we can save time by solving a sub-problem once & cache the answer.
(this is sometimes called “memoization”)
e.g. **Lots of different entries in the row $\mathbf{d^{(k)}}$ may ask for $\mathbf{d^{(k-1)}[v]}$**

DYNAMIC PROGRAMMING

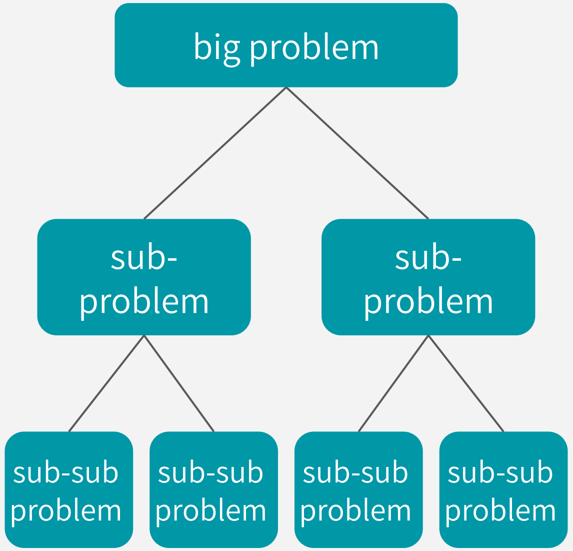
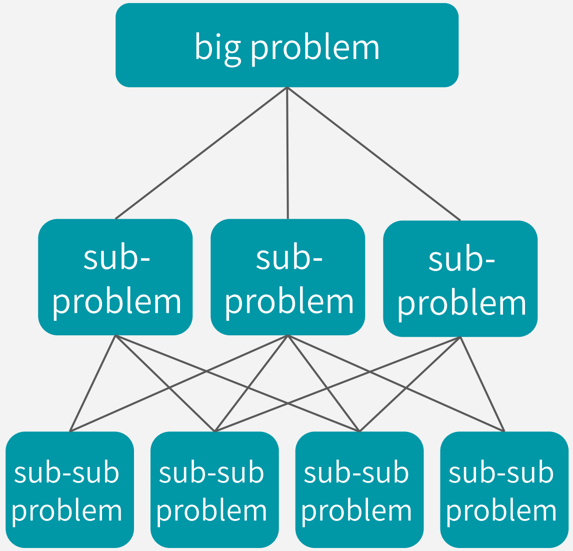
Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see next week) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)
We will see a way later to implement **Bellman-Ford** using a top-down approach.

DIVIDE & CONQUER vs DP

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING
	

RECIPE FOR APPLYING DP

1. **Identify optimal substructure.** What are your overlapping subproblems?
2. **Define a recursive formulation.** Recursively define your optimal solution in terms of sub-solutions. *Always write down this formulation.*
3. **Use dynamic programming.** Turn the recursive formulation into a DP algorithm.
4. **If needed, track additional information.** You may need to solve a related problem, e.g. step 3 finds you an optimal *value/cost*, but you need to recover the actual optimal *solution/path/subset/substring/etc*. Go back and modify your algorithm in step 3 to make this happen.

LONGEST COMMON SUBSEQUENCE

A sequence **Z** is a **SUBSEQUENCE** of **X** if **Z** can be obtained from **X** by deleting symbols

BDFH is a subsequence of **ABCDEFGH**

C is a subsequence of **ABCDEFGH**

ABCDEFGH is a subsequence of **ABCDEFGH**

A sequence **Z** is a **LONGEST COMMON SUBSEQUENCE (LCS)** of **X** and **Y**
if **Z** is a subsequence of both **X** and **Y**
and any sequence longer than **Z** is not a subsequence of at least one of **X** or **Y**.

ABDFGH is the LCS of

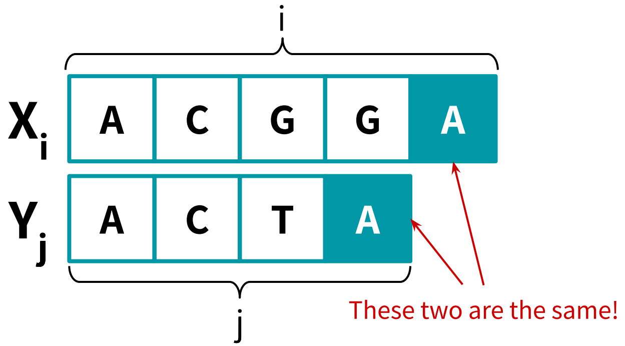
ABCDEFGH and **ABDFGHI**

STEP 1: OPTIMAL SUBSTRUCTURE

Let **C[i, j] = length_of_LCS(X_i, Y_j)**

Consider the ends of our prefixes, X[i] and Y[j]. We have two cases:

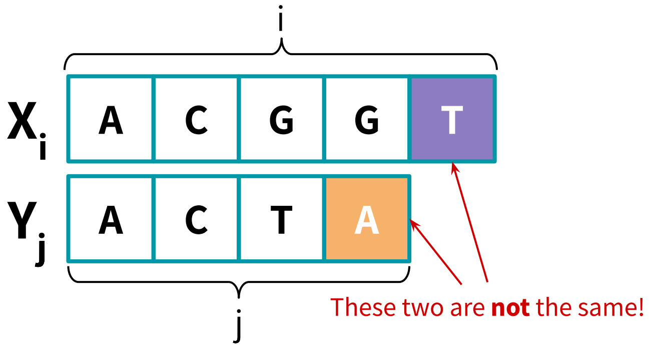
Case 1: X[i] = Y[j]



Then, **C[i, j] = 1 + C[i–1, j–1]**

because LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) followed by A .

Case 2: X[i] ≠ Y[j]



Then, **C[i, j] = max{ C[i–1, j], C[i, j–1] }**

Give A a chance to "match": LCS(X_i, Y_j) = LCS(X_{i-1}, Y)

Give T a chance to "match": LCS(X_i, Y_j) = LCS(X_i, Y_{j-1})

STEP 2: RECURSIVE FORMULATION

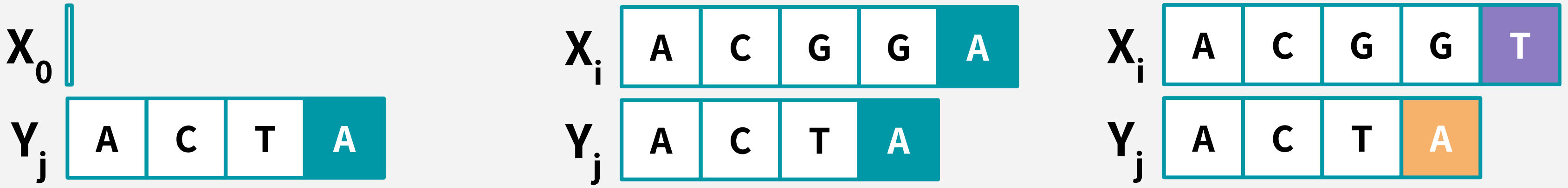
Our recursive formulation:

$$C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\ C[i-1, j],\ C[i, j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

CASE 0 (base case)

CASE 1

CASE 2



STEP 3: WRITE A DP ALGORITHM

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\ C[i-1, j],\ C[i, j-1]\ \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

len(X) = m & len(Y) = n

```
LCS(X,Y):
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

Make sure that our base cases are set up

Case 1

Case 2

Final answer

Runtime: O(mn)

Constant amount of work to fill out each of the mn entries in C

STEP 4: FIND ACTUAL LCS

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

```
RECOVER_LCS(X, Y, C):
    // C is already filled out
    L = []
    i = m
    j = n
    while i > 0 and j > 0:
        if X[i] = X[j]:
            append X[i] to the beginning of L
            i = i-1
            j = j-1
        else if C[i,j] = C[i,j-1]:
            j = j-1
        else:
            i = i-1
    return L
```

This extra subroutine takes O(m+n) time!

سوال؟

یافتن کوتاه ترین مسیر در گراف

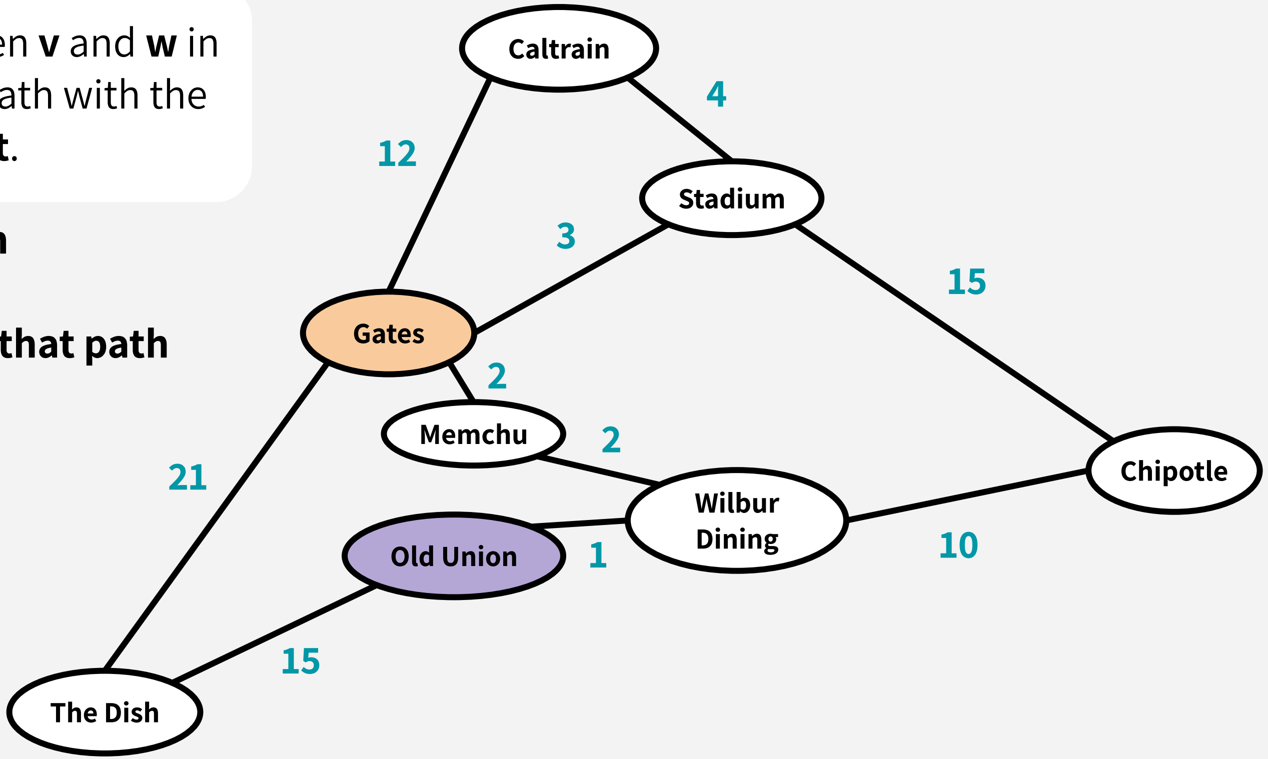
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

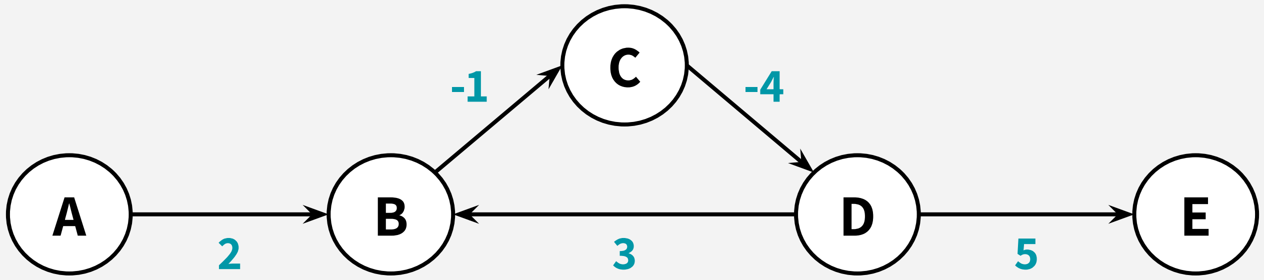
=

sum of weights along that path



NEGATIVE CYCLES

If negative cycles exist in the graph, we'll say *no solution exists.* Why?



What's the shortest path from A to E?

Is it: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$? Cost = 2 - 1 - 4 + 5 = **2**.

Or is it: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C \rightarrow D \rightarrow E$?

Basically, shortest paths aren't defined if there are negative cycles!

الگوریتم بلمن-فورد

BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

How do we use $\mathbf{d}^{(0)}$ to update $\mathbf{d}^{(1)}[\mathbf{b}]$?

Case 1: the shortest path from s to b with at most k edges could be one with at most k–1 edges! In other words, allowing k edges is not going to change anything. Then:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{d}^{(k-1)}[\mathbf{b}]$$

Case 2: the shortest path from s to b with at most k edges could be one with exactly k edges!
I.e. this length-k shortest path is [length k–1 shortest path to some incoming neighbor a] + w(a,b).
Which of b's incoming neighbors will offer this shortest path? Let's check them all:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{min}_{\text{a in b's incoming neighbors}} \{ \mathbf{d}^{(k-1)}[\mathbf{a}] + \mathbf{w(a,b)} \}$$

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    d^(k) = [] for k = 0, ..., n-1
    d^(0)[v] = ∞ for all v in V (except s)
    d^(0)[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            d^(k)[b] ← min{ d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
    return d^(n-1)
```

CASE 1: $d^{(k-1)}[b]$

CASE 2: $\min_a\{d^{(k-1)}[a] + w(a,b)\}$

Keeping all n–1 rows is a simplification to make the pseudocode straightforward. In practice, we'd only keep 2 of them at a time!

Fill the first row

Take the minimum over all incoming neighbors a (i.e. all a s.t. (a, b) ∈ E)
This takes O(deg(b))!!!

The answer

Runtime: O(m·n)

TOP-DOWN BELLMAN-FORD

```
RECURSIVE_BELLMAN_FORD(G,s):
    $d^{(k)}$ = [None] * n for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for b in V:
        $d^{(n-1)}$[b] ← RECURSIVE_BF_HELPER(G, b, n-1)

RECURSIVE_BF_HELPER(G, b, k):
    A = {a such that (a,b) in E} ∪ {b}    // b's in-neighbors
    for a in A:
        if $d^{(k-1)}$[a] is None:                // not yet solved
            $d^{(k-1)}$[a] ← RECURSIVE_BF_HELPER(G, a, k-1)
    return min{ $d^{(k-1)}$[b], $\min_a$\{$d^{(k-1)}$[a] + w(a,b)\} }
```

Think of this as a table/cache that holds the computed answers of our subproblems.

if the answer to this subproblem hasn't been computed yet, then we'll first solve it! It immediately gets saved in our cache, so we won't ever solve it twice.

Runtime: O(m·n)

سوال؟

الگوریتم فلوید-وارشال

ALL-PAIRS SHORTEST PATHS (APSP)

Find the shortest paths from **v** to **w** for ALL pairs **v, w** of vertices in the graph

Naive algorithm (if we want to handle negative edge weights):

```
For all s in G:
      Run Bellman-Ford on G starting at s
```

Runtime: O(n ·mn) = **O(mn^2)**... this may be as bad as n^4 if m = n^2

Can we do better?

What's a naive algorithm?

FLOYD-WARSHALL: A DP APPROACH

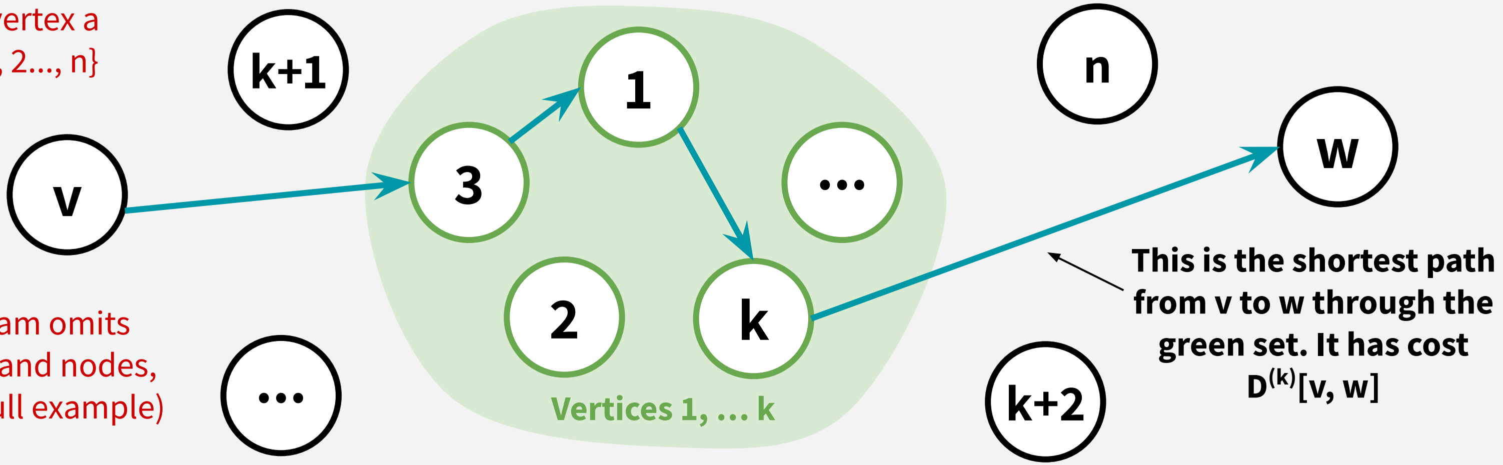
We need to define the optimal substructure: Figure out what your subproblems are, and how you'll express an optimal solution in terms of optimal solutions to subproblems.

Subproblem(k): for all pairs **v**, **w**, find the cost of the shortest path from **v** to **w** so that all the internal vertices on that path are in {1, …, k}

Let $\mathbf{D^{(k)}[v, w]}$ be the solution to Subproblem(k)

Assign each vertex a number in {1, 2…, n}

(This diagram omits many edges and nodes, so it's not a full example)



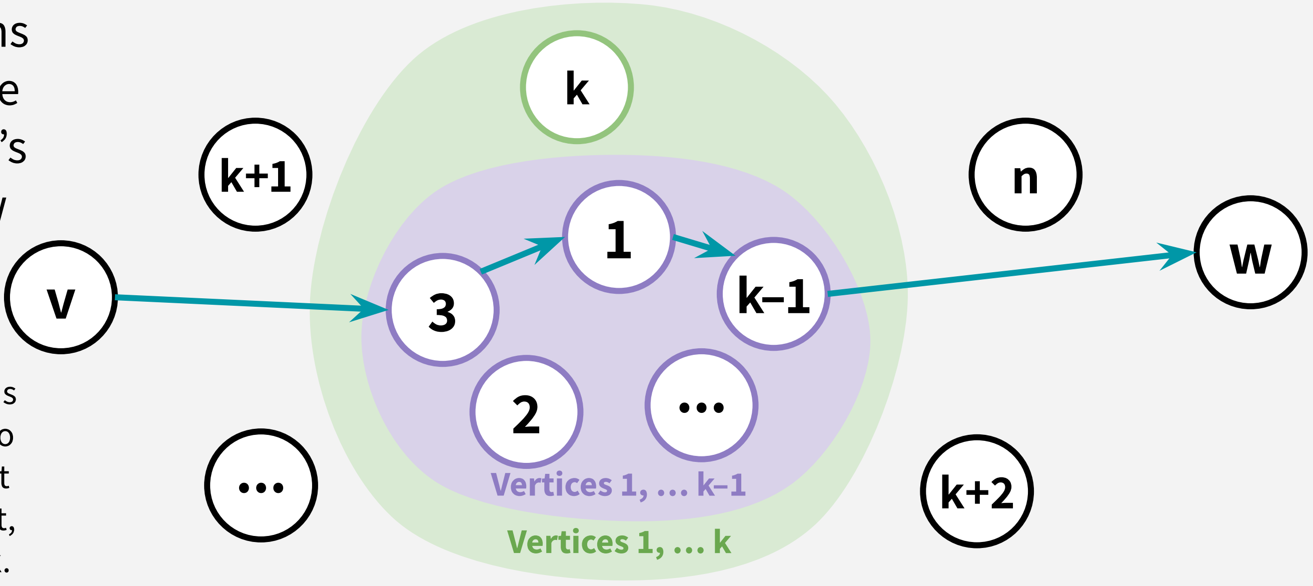
This is the shortest path from v to w through the green set. It has cost $D^{(k)}[v, w]$

FLOYD-WARSHALL: A DP APPROACH

$\mathbf{D^{(k)}[v, w]}$ = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 1: We don't need vertex k! So, $\mathbf{D^{(k)}[v, w] = D^{(k-1)}[v, w]}$

In this case, this means that **this path** was the shortest before *and* it's still the shortest now



In other words, allowing paths to go through k (in addition to nodes 1, …, k–1) now doesn't change the shortest path cost, since it doesn't need to use k.

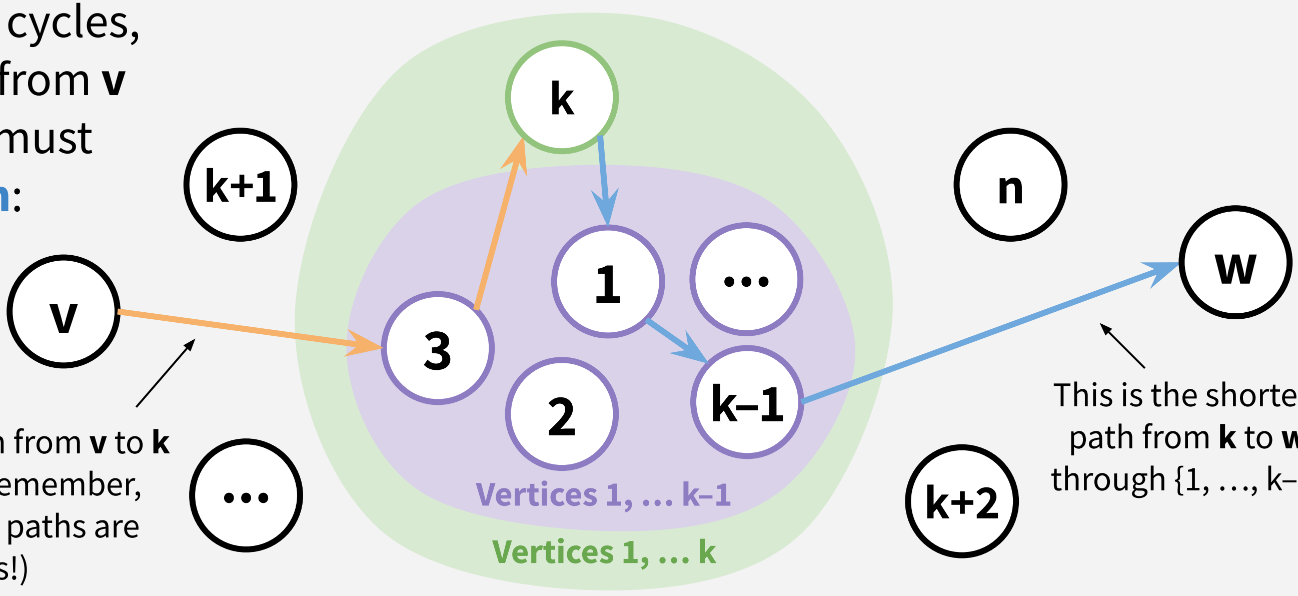
FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}$[v, w] = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 2: We need vertex k! So, **$D^{(k)}$[v, w] = $D^{(k-1)}$[v, k] + $D^{(k-1)}$[k, w]**

If there are no negative cycles, then the shortest path from **v** to **w** is *simple*, and it must look like **this path**:

(we also know that neither of these subpaths contains nodes greater than k–1.)



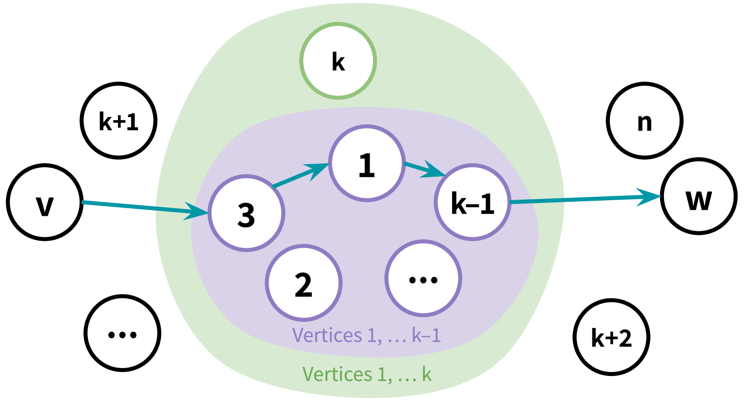
This is the shortest path from **v** to **k** through {1, …, k–1} (remember, sub-paths of shortest paths are shortest paths!)

This is the shortest path from **k** to **w** through {1, …, k–1}

FLOYD-WARSHALL: A DP APPROACH

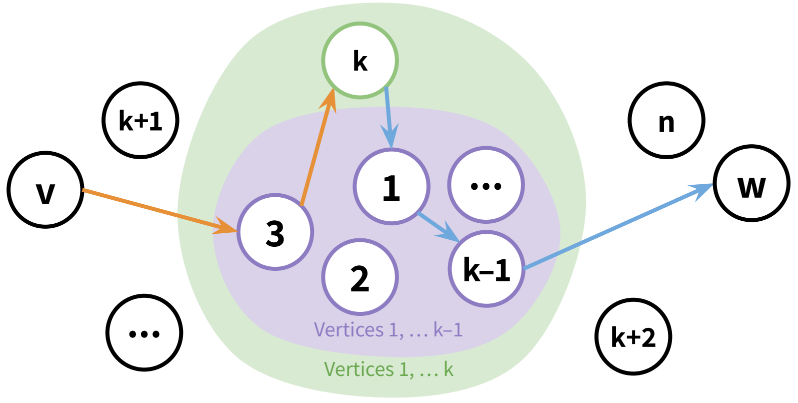
How do we find $D^{(k)}[v, w]$ using $D^{(k-1)}$? Choose the minimum of these 2 cases:

CASE 1: We don't need vertex **k**



$$D^{(k)}[v, w] = D^{(k-1)}[v, w]$$

CASE 2: We need vertex **k**



$$D^{(k)}[v, w] = D^{(k-1)}[v, k] + D^{(k-1)}[k, w]$$

FLOYD-WARSHALL: A DP APPROACH

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}[v,v] = 0$ for all v, for all k
        $D^{(k)}[v,w] = \infty$ for all $v \neq w$, for all k
        $D^{(0)}[v,w]$ = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}[v,w] = \min\{ D^{(k-1)}[v,w], D^{(k-1)}[v,k] + D^{(k-1)}[k,w] \}$
    return $D^{(n)}$
```

Keeping all these n x n arrays would be a waste of space. In practice, only need to store 2!

Take the minimum over our two cases!

Runtime: $O(n^3)$

(Better than running Bellman-Ford n times!)

WHAT ABOUT NEGATIVE CYCLES?

Negative cycle means there's some **v**
s.t. there is a path from **v** to **v** that has cost < 0

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}$[v,v] = 0 for all v, for all k
        $D^{(k)}$[v,w] = ∞ for all v ≠ w, for all k
        $D^{(k)}$[v,w] = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}$[v,w] = min{ $D^{(k-1)}$[v,w], $D^{(k-1)}$[v,k] + $D^{(k-1)}$[k,w]
}
    for v in V:
        if $D^{(n)}$[v,v] < 0:
            throw "NEGATIVE CYCLE!"
    return $D^{(n)}$
```

SHORTEST-PATH ALGORITHMS

$n = |V|$
$m = |E|$

BFS	DFS	DIJKSTRA	BELLMAN-FORD	FLOYD-WARSHALL
$O(m+n)$	$O(m+n)$	$O(m+n\log n)$*	$O(mn)$	$O(n^3)$
Unweighted (or weights don't matter)	Unweighted (or weights don't matter)	Weighted (weights must be *non-negative*)	Weighted (can handle *negative* weights)	Weighted (can handle *negative* weights)
Single source shortest path Test bipartiteness Find connected components	Path finding (s,t) Toposort (DAG!!) Find SCC's Find connected components	***Single source shortest paths:*** Compute shortest path from a source s to all other nodes	***Single source shortest paths:*** Compute shortest path from source s to all other nodes Detect negative cycles	***All pairs shortest paths:*** Compute shortest path between every pair of nodes (v,w)

سوال؟