ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و چهارم:
مرور مطالب

سجاد شیرعلی شهرضا

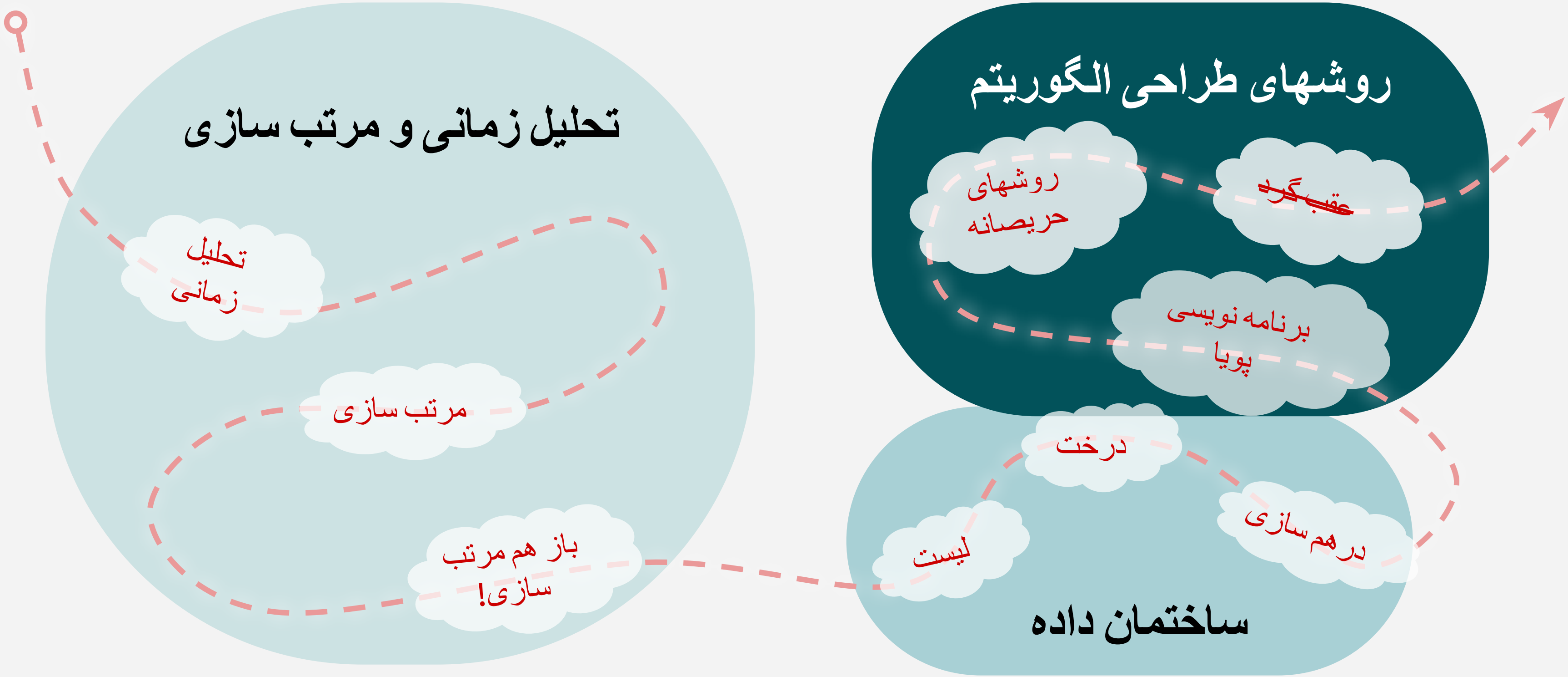
پاییز 1400

دوشنبه،13 دی 1400

اطلاع رسانی

- مهلت ارسال تمرین 4: ساعت 8 شب روز چهارشنبه (15 دی 1400)
- آخرین هفته (و جلسه!) کلاس
- هفته آینده: امتحان پایان ترم
- جلسه مراجعه مجازی برای رفع اشکال: روز جمعه، 17 دی 1400، ساعت 8 تا 9 شب
  - از طریق سامانه دروس (lmshome.aut.ac.ir) در قالب یک کلاس جبرانی

آنچه گذشت!


تحلیل زمانی و مرتب سازی
تحلیل زمانی
مرتب سازی
باز هم مرتب سازی!
روشهای طراحی الگوریتم
روشهای حریصانه
~~عقبگرد~~
برنامه نویسی پویا
درخت
لیست
درهم سازی
ساختمان داده

جلسه اول:
مقدمه

شنبه، 27 شهریور 1400

جلسه دوم:
ضرب

شنبه، 3 مهر 1400

ASYMPTOTIC ANALYSIS (High Level Idea)

We'll express the asymptotic runtime of an algorithm using

BIG-O NOTATION

"big-oh of n squared"
or
"Oh of n squared"

We would say Grade-school Multiplication **"runs in time $O(n^2)$"**

Informally, this means that the runtime "scales like" n^2

We'll discuss the formal definition of Big-O (math-y stuff) next week

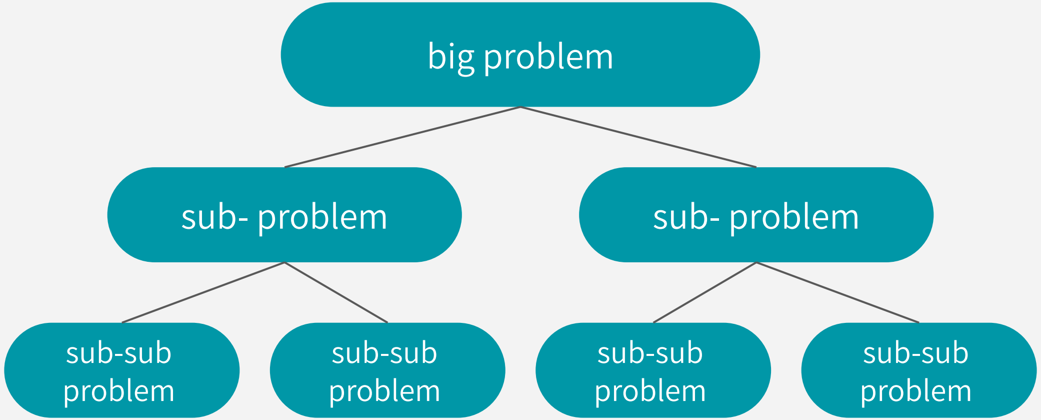
THE POINT OF ASYMPTOTIC NOTATION

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

DIVIDE AND CONQUER

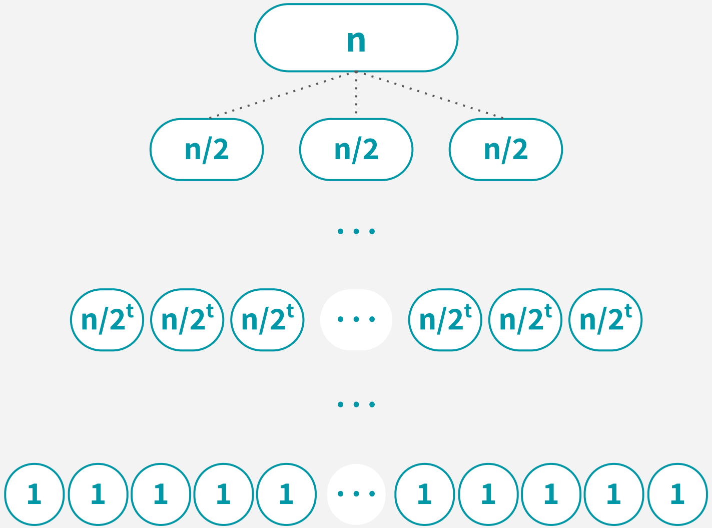
An algorithm design paradigm:

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



WHAT'S THE RUNTIME?

Karatsuba Multiplication Recursion Tree



Level 0: 1 problem of size n

Level 1: 3^1 problems of size n/2

Level t: 3^t problems of size $n/2^t$

Level $\log_2 n$: $\underline{\mathbf{n^{1.6}}}$ problems of size 1

$\log_2 n$ levels
(you need to cut n in half $\log_2 n$ times to get to size 1)

# of problems on last level (size 1)
$= 3^{\log_2 n} = n^{\log_2 3}$
$\approx \mathbf{n^{1.6}}$

Thus, the runtime is $O(n^{1.6})$!

جلسه سوم:
تحلیل زمانی الگوریتمها

شنبه،10 مهر 1400

BIG-O NOTATION

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

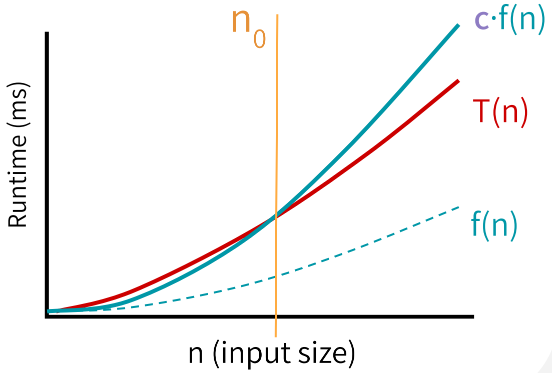
(In this class, we'll typically write T(n) to denote the worst case runtime of an algorithm)

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" $\Leftrightarrow$

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

T(n) ≤ c · f(n)

"there exists" "such that" "for all"

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 $n \geq n_0$."

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))"

↓

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!!!*"

↓

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.

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

جلسه چهارم:
مرتب سازی درجی و ادغامی

شنبه، 10 مهر 1400

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 *(strong/complete induction version)*

Next, assume that the IH holds when **i** takes on any value *between [base case value(s)] and some number* **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.

INSERTION SORT: IS IT FAST?

Instead of counting every little operation, we can think about:
How many iterations take place
How much work happens within each iteration

```
InsertionSort(A):
    for i in range(1, len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = cur_value
```

At most n outer for-loop iterations

At most n inner while-loop iterations

OVERALL RUNTIME OF INSERTION SORT: $O(n^2)$

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.

PROVE CORRECTNESS w/ INDUCTION

ITERATIVE ALGORITHMS

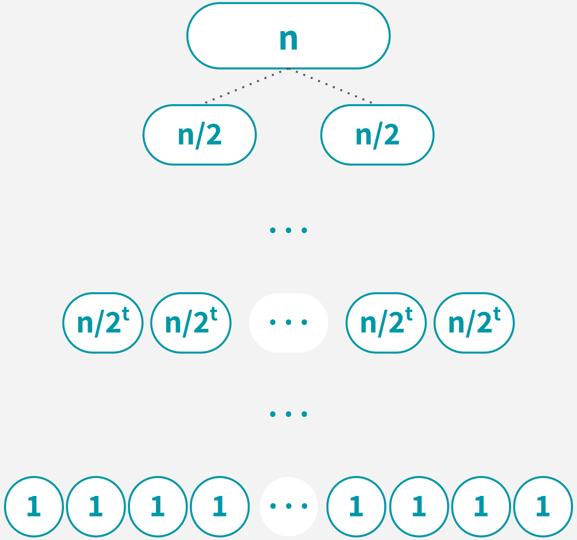
1. **Inductive hypothesis**: some state/condition will always hold throughout your algorithm by any iteration **i**
2. **Base case**: show IH holds for iteration 0 (i.e. start of algorithm)
3. **Inductive step**: Assume IH holds for k ⇒ prove k+1
4. **Conclusion**: IH holds for i = # total iterations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

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)**

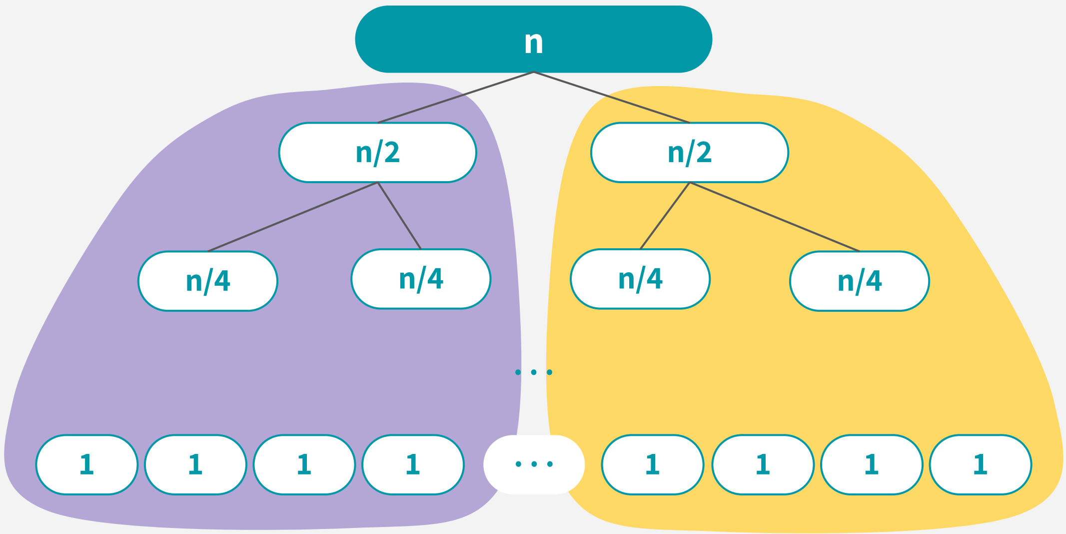
We have ($\log_2 n + 1$) levels, each level has O(n) work total ⇒ **O(n log n)** work overall!

جلسه پنجم:
رابطه بازگشتی و قضیه اصلی

دوشنبه، 12 مهر 1400

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 $\mathbf{a \geq 1}$, $\mathbf{b > 1}$, and $\mathbf{d}$ are constants (i.e. independent of $\mathbf{n}$).

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

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

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

جلسه ششم و هفتم: حل با روش جایگذاری و انتخاب kامین عضو

شنبه، 17 مهر 1400 و شنبه، 24 مهر 1400

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).

Let's try it on some example recurrences...

LINEAR SELECTION: RUNTIME

O(n)
Non-recursive "shallow" work!

Select a pivot: **Median of (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** (via recursive call to SELECT!!)

T(n/5)
Recursive work: we call SELECT on an array of size n/5

Partition around pivot

Recurse!

T(7n/10)
Recursive work: we call SELECT on either L or R (size ≤ 7n/10)

جلسه هشتم:
الگوریتم های تصادفی

دوشنبه، 26 مهر 1400

LAS VEGAS vs. MONTE CARLO

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

MONTE CARLO ALGORITHMS

Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

We'll focus on these algorithms for now
(BogoSort, QuickSort, QuickSelect)

We'll see some examples of these later!

RUNTIME FOR RANDOMIZED ALGS

"Expected value over *dice outcomes*"

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

"The worst possible *dice outcomes*"

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

ساختمان داده و الگوریتم ها (CE203)

جلسه نهم: مرتب سازی سریع

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 1 آبان 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

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 ($O(n\log n)$ 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

You do not need to understand any of this stuff

ساختمان داده و الگوریتم ها (CE203)

جلسه دهم:
کران پایین برای مرتب سازی

سجاد شیرعلی شهرضا

پاییز 1400

دوشنبه، 3 آبان 1400

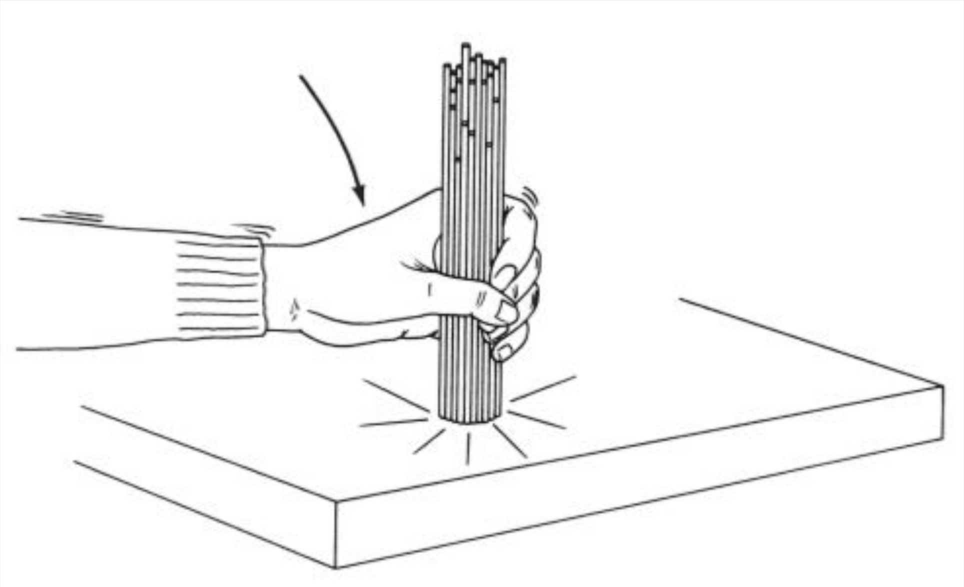
Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

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)**



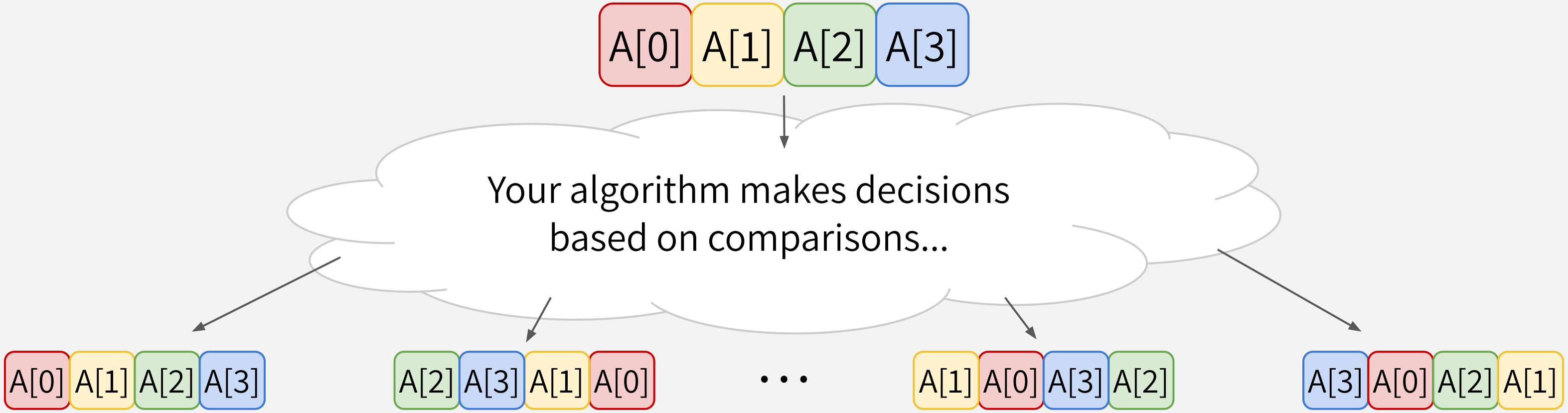
While you shouldn't take this algorithm too seriously… it does raise some important questions!

COMPARISON-BASED SORTING

Theorem:

Any deterministic comparison-based sorting algorithm must take Ω(n log n) time.

Think about it like this: this is the input format that your algorithm is ready to accept.



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

ساختمان داده و الگوریتم ها (CE203)

جلسه یازدهم: مرتب سازی خطی

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 8 آبان 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

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:

10	20	30	40	50	60
	20	30		50	
10	20	30		50	60

Output:	10	20	20	30	30	50	50	60

Sorted in time:
O(n)

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

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?

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

This term is a constant!

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

RADIX SORT RECAP

Radix Sort can sort **n integers of size at most n^{100}** (or n^C for any constant c) in time **O(n)**.

If your sorting task involves integers that have size much bigger than n (or n^C), like 2^n, maybe you shouldn't use Radix Sort because you wouldn't get linear time.

It matters how you pick the base! In general, if you have **n** elements, **M** = max size of any element, and **r** is the base:

Runtime of Radix Sort = $\mathbf{O\left(\left(\lfloor \log_r M \rfloor + 1\right) \cdot n \right)}$

ساختمان داده ها و الگوریتم ها (CE203)

جلسه دوازدهم:
لیست، پشته و صف

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 15 آبان 1400

Based on Washington University CSE 373 Slides from Autumn 2020 by Hunter Schafer

Comparing ADT Implementations: List

	ArrayList	LinkedList
`add (front)`	linear	constant
`remove (front)`	linear	constant
`add (back)`	(usually) constant	linear
`remove (back)`	constant	linear
`get`	constant	linear
`put`	linear	linear

- Important to be able to come up with this, and understand why
- But only half the story: to be able to make a design decision, need the context to understand which of these we should prioritize

Implementing a Stack with Linked Nodes

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

LinkedStack<E>

State

`Node top`
`size`

Behavior

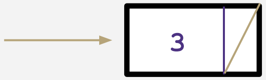
`push` add new node at top
`pop` return & remove node at top
`peek` return node at top
`size` return size
`isEmpty` return size == 0

Big-Oh Analysis

pop()	O(1) Constant
peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
push()	O(1) otherwise

```
push(3)
push(4)
pop()
```

top



size = 1

Implementing a Stack with an Array

STACK ADT

State

Collection of ordered items
Count of items

Behavior

`push(index)` add item to top
`pop()` return & remove item at top
`peek()` return item at top
`size()` count of items
`isEmpty()` is count 0?

ArrayStack<E>

State

`data[]`
`size`

Behavior

`push` data[size] = value, if out of room grow data
`pop` return data[size - 1], size -= 1
`peek` return data[size - 1]
`size` return size
`isEmpty` return size == 0

0	1	2	3
3	5		

size = 2

```
push(3)
push(4)
pop()
push(5)
```

Big-Oh Analysis

pop()	O(1) Constant
peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
push()	O(n) linear if you have to resize, O(1) otherwise

Implementing a Queue with Linked Nodes

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

<u>add(item)</u> add item to back
<u>remove()</u> remove and return item at front
<u>peek()</u> return item at front
<u>size()</u> count of items
<u>isEmpty()</u> count is 0?

LinkedQueue<E>

State

```
Node front
Node back
size
```

Behavior

<u>add</u> – add node to back
<u>remove</u> – return and remove node at front
<u>peek</u> – return node at front
<u>size</u> – return size
<u>isEmpty</u> – return size == 0

Big-Oh Analysis

remove()	O(1) Constant
peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
add()	O(1) Constant

size = 1

```
add(5)
add(8)
remove()
```

Implementing a Queue with an Array (v1)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV1<E>

State

`data[]`
`size`

Behavior

`add` - data[size] = value, if out of room grow
`remove` - return/remove at 0, shift everything
`peek` - return node at 0
`size` - return size
`isEmpty` - return size == 0

Big-Oh Analysis

peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
add()	O(n) Linear: if we need to resize O(1) Constant: otherwise
remove()	O(n) Linear

0	1	2	3	4
8	9			

size = 2

```
add(5)
add(8)
add(9)
remove()
```

Implementing a Queue with an Array (v2)

QUEUE ADT

State

Collection of ordered items
Count of items

Behavior

`add(item)` add item to back
`remove()` remove and return item at front
`peek()` return item at front
`size()` count of items
`isEmpty()` count is 0?

ArrayQueueV2<E>

State

`data[]`, `front`,
`size`, `back`

Behavior

`add` – data[back] = value, back++, size++, if out of room grow
`remove` – return data[front], size--, front++
`peek` – return data[front]
`size` – return size
`isEmpty` – return size == 0

Big-Oh Analysis

`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(n) Linear: if we need to resize O(1) Constant: otherwise
`remove()`	O(1) Constant

ساختمان داده ها و الگوریتم ها (CE203)

جلسه سیزدهم:
هرم و مرتب سازی هرمی

سجاد شیرعلی شهرضا

پاییز 1400

دوشنبه، 17 آبان 1400

Based on Washington University CSE 373 Slides from Autumn 2020 by Hunter Schafer

Priority Queue Implementations

LinkedList

`add()`	**put new element at front – O(1)**
`poll()`	**must search the list – O(n)**
`peek()`	**must search the list – O(n)**

LinkedList that is always sorted

`add()`	**must search the list – O(n)**
`poll()`	**highest priority element at front – O(1)**
`peek()`	**same – O(1)**

A Heap..

Is a binary tree satisfying 2 properties

1) **Completeness. Every level of the tree (except last) is completely filled. All holes in last level are all the way to the right.**
2) **Heap-order.**

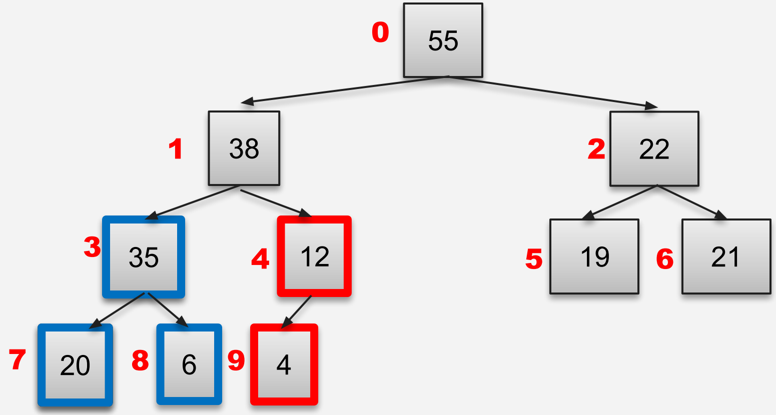
Max-Heap: every element in tree is <= its parent

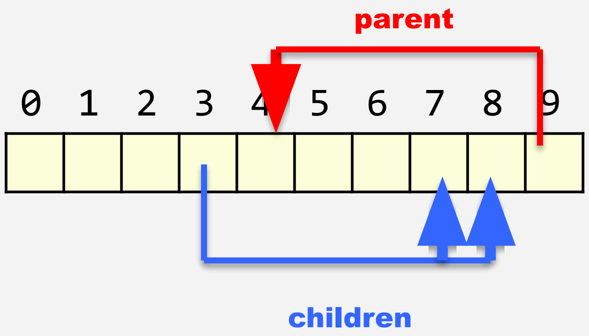
Primary operations:

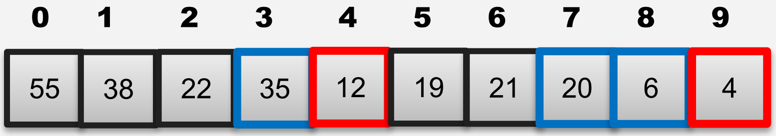
1) **add(e): add a new element to the heap**
2) **poll(): delete the max element and return it**
3) **peek(): return the max element**

Represent tree with array

- Store node number i in:
  b[i]
- Children of **b[k]** are:
  b[2k +1] and **b[2k +2]**
- Parent of **b[k]** is **b[**$\lfloor (k-1)/2 \rfloor$**]**

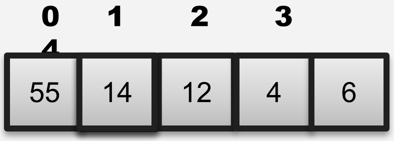


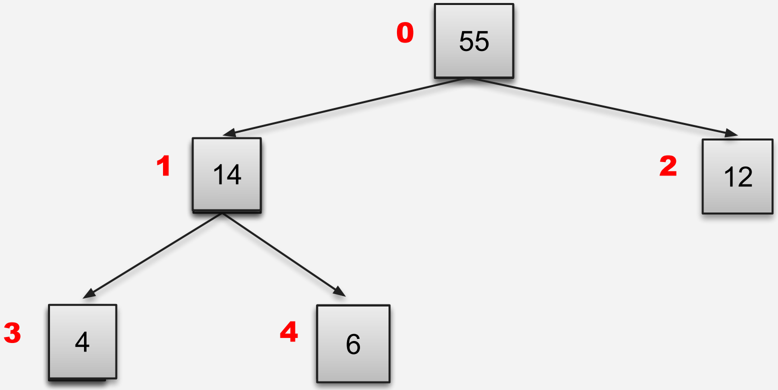




Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is sorted
    for (k= n-1; k > 0; k= k-1) {
            b[k]= poll  – i.e., take max element out of heap.
    }
```





ساختمان داده ها و الگوریتم ها (CE203)

جلسات چهاردهم و پانزدهم: درخت

سجاد شیرعلی شهرضا

پاییز 1400

شنبه 22 و دوشنبه 24 آبان 1400

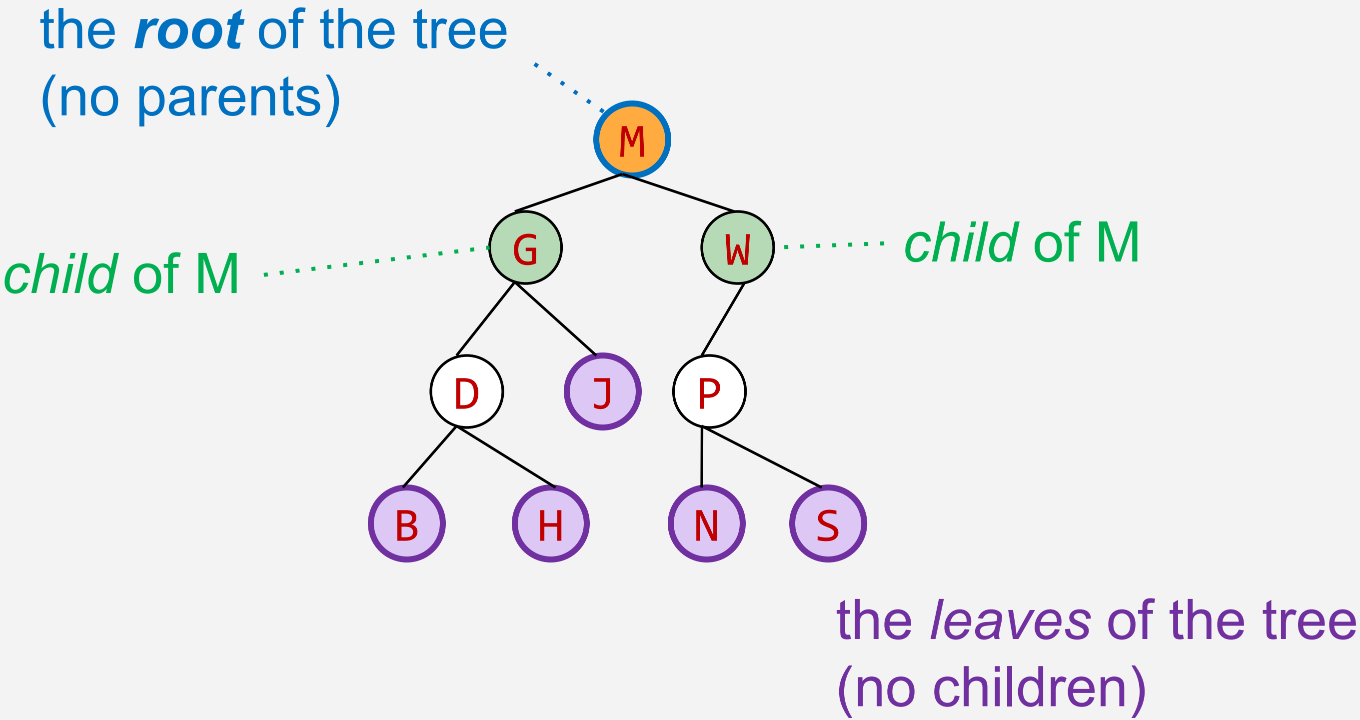
Based on Washington University CSE 373 Slides from Autumn 2020 by Hunter Schafer

Example Data Structures

Data Structure	add(val v)	get(int i)	contains(val v)
Array [2 | 1 | 3 | 0]	$O(n)$	$O(1)$	$O(n)$
Linked List 2 → 1 → 3 → 0	$O(1)$	$O(n)$	$O(n)$

```
add(v): append v
get(i): return element at position i
contains(v): return true if contains v
```

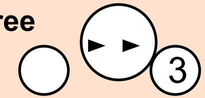
Tree Terminology: Parent, Child, Leaves, Root



Iterate through data structure

Iterate: process elements of data structure

- Sum all elements
- Print each element
- …

Data Structure	**Order to iterate**
Array 2 | 1 | 3 | 0	Forwards: 2, 1, 3, 0 Backwards: 0, 3, 1, 2
Linked List 2 → 1 → 3 → 0	Forwards: 2, 1, 3, 0
Binary Tree 	**???**

Tree traversals

- Iterating through tree is aka tree traversal
- Well-known recursive tree traversal algorithms:
  - Preorder
  - Inorder
  - Postorder
- Another, non-recursive: level order

Recover tree from traversals

Suppose inorder is B C A E D
preorder is A B C D E
Can we determine the tree uniquely? Yes!

- What is root? Preorder tells us: A
- What comes before/after root A?
  - Inorder tells us:
    - Before: B C
    - After: E D
- Now recurse! Figure out left/right subtrees using same technique.

ساختمان داده و الگوریتم ها (CE203)

جلسه شانزدهم:
درخت دودویی جستجو

سجاد شیرعلی شهرضا
پاییز 1400
شنبه، 6 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

BINARY SEARCH TREE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	BST (WORST CASE)	BST (BALANCED)
SEARCH	O(log(n))	O(n)	O(n)	O(log(n))
DELETE	O(n)	O(n)	O(n)	O(log(n))
INSERT	O(n)	O(1)	O(n)	O(log(n))

(Balanced) Binary Search Trees can give us the best of both worlds!

BINARY TREE TERMINOLOGY

Each node has two children

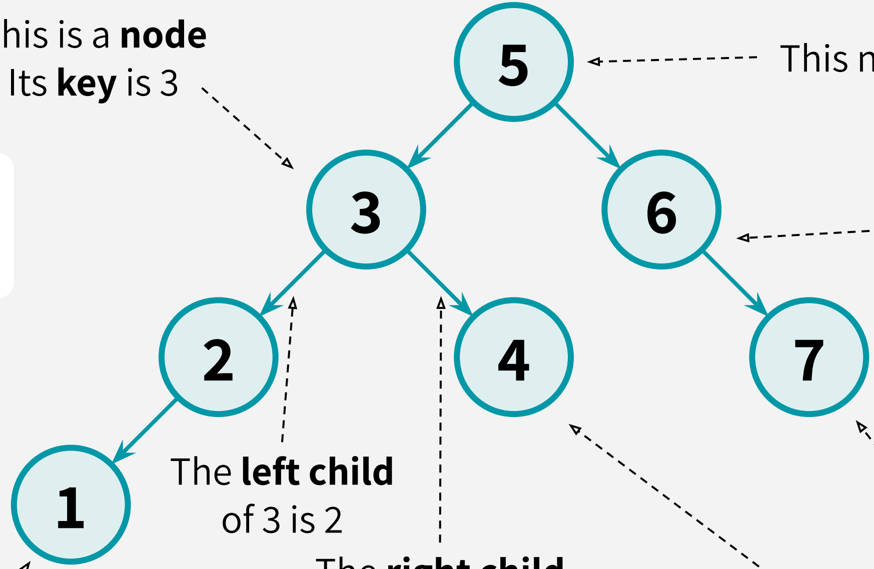
Each node has a pointer to its left child, right child, and parent

The **left descendants** of 5 are 1, 2, 3, and 4

The **ancestors** of 1 are 2, 3, and 5

This is a **node**
Its **key** is 3

This node is the **root**



6 is the **parent** of 7

The **height** of this tree is 3
(max number of edges from root to a leaf)

The **left child** of 3 is 2

The **right child** of 3 is 4

Both children of a leaf node are **NIL**
(I usually won't draw them)

Nodes without children are **leaves** (aka **leaf nodes**)

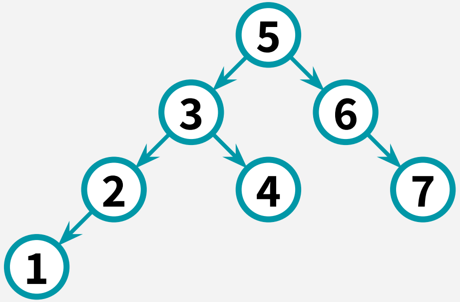
THE BST PROPERTY

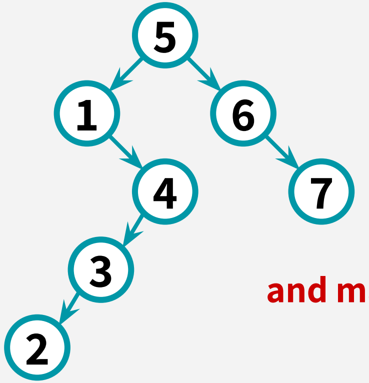
A Binary Search Tree (BST) is a binary tree such that:

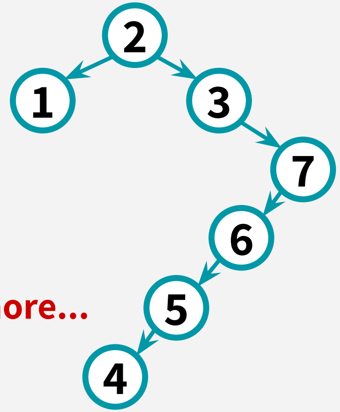
Every LEFT descendant of a node has key less than that node
Every RIGHT descendant of a node has key larger than that node

There exist many valid BSTs that contain these numbers:

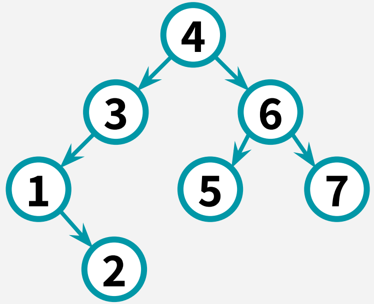
5 3 6 2 1 7 4







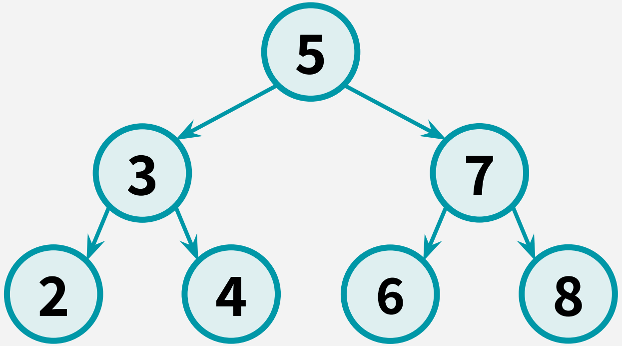
and many more…



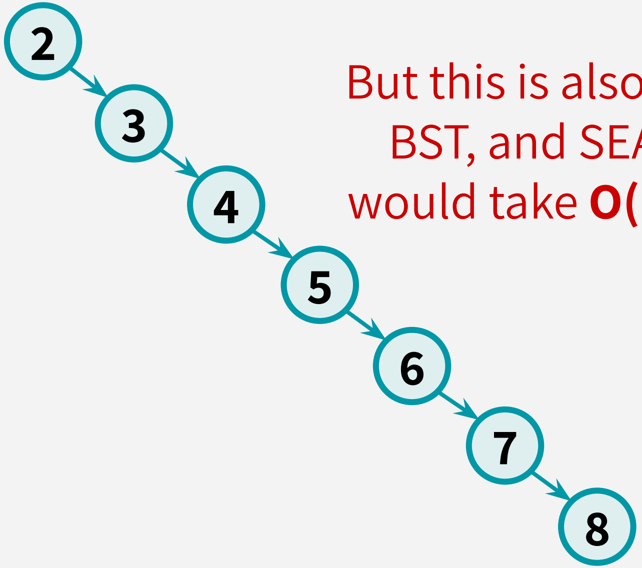
RUNTIME OF SEARCH/INSERT/DELETE

INSERT and **DELETE** both call **SEARCH** (and then do some O(1)-time operation)

Runtime of **SEARCH** = **O(height)**



Sometimes SEARCH takes **O(log n)**



But this is also a valid BST, and SEARCH would take **O(n)** here

ساختمان داده و الگوریتم ها (CE203)

جلسه هفدهم:
درخت قرمز-سیاه

سجاد شیرعلی شهرضا

پاییز 1400

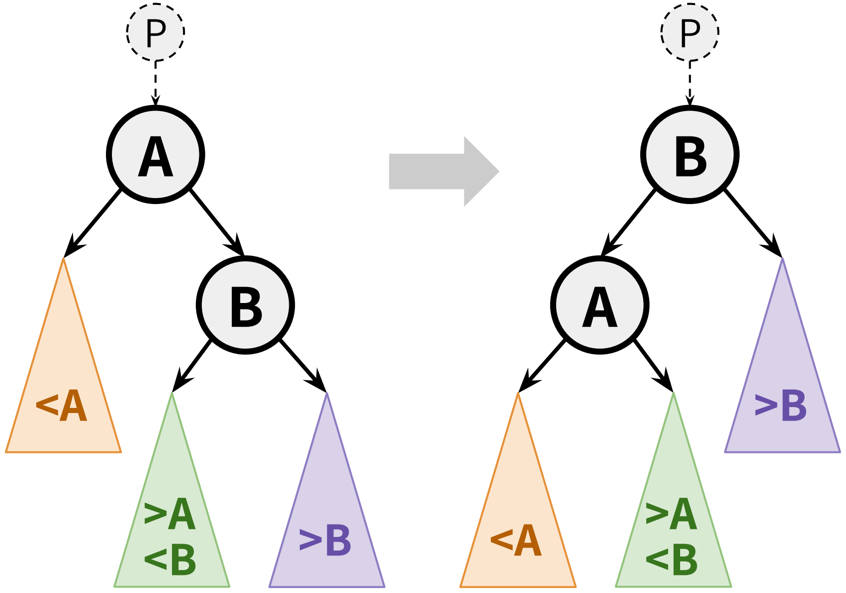
دوشنبه، 8 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

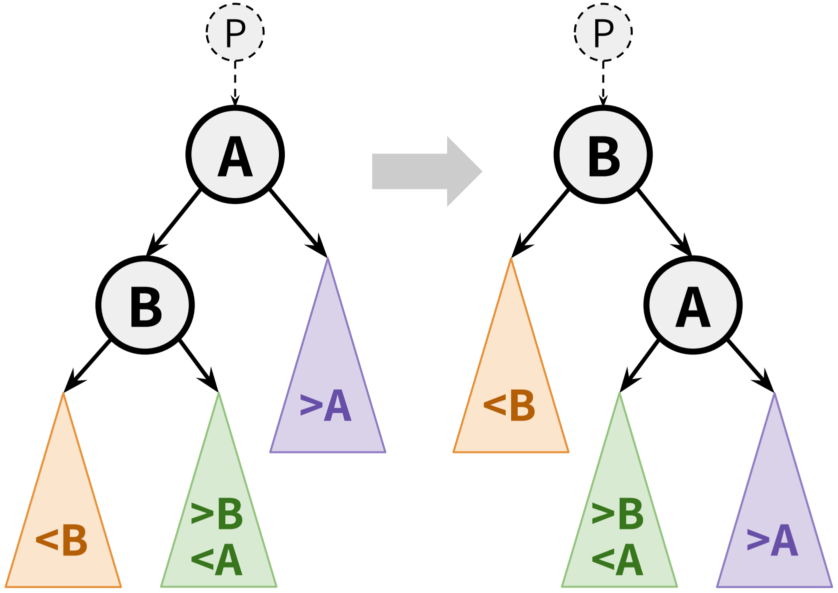
ROTATIONS

IDEA: locally rebalance a node's subtree in O(1) time while maintaining BST property

LEFT ROTATION



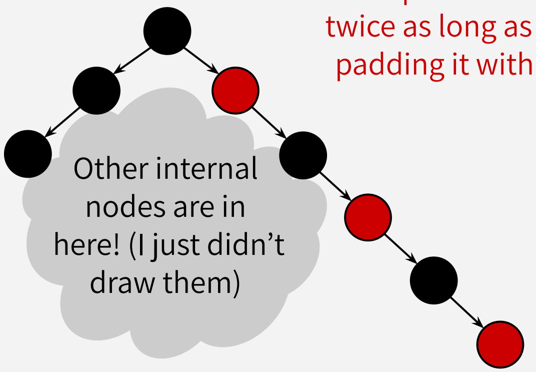
RIGHT ROTATION



WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

Rules 3 & 4 guarantee that one path can be at most twice as long as another by padding it with red nodes



Intuitively, these rules are a *proxy* for balance:
The **black** nodes are ~balanced across the tree.
And the **red** nodes might elongate paths but not by much!

More formally…

WHAT'S THE POINT OF THESE RULES?

1. Every node is either **red** or **black**
2. The root is a **black** node
3. No **red** node has a **red** child
4. Every root-NIL path has the same number of **black** nodes on them

THEOREM: Any Red-Black Tree with **n** nodes has height **O(log n)**

PROOF IDEA: We can show that any RB tree with **n** nodes has height $\leq 2 \cdot \log_2(n+1)$

WHAT HAVE WE LEARNED?

The height of an RB Tree is O(log n).

Runtime of **SEARCH** in an RB Tree = **O(height)**

= O(log n)

What about INSERT/DELETE?

These are the two operations that actually modify the RB Tree, so we need to make sure that we insert & delete without violating our precious RB Tree properties...

INSERT IN RB TREES

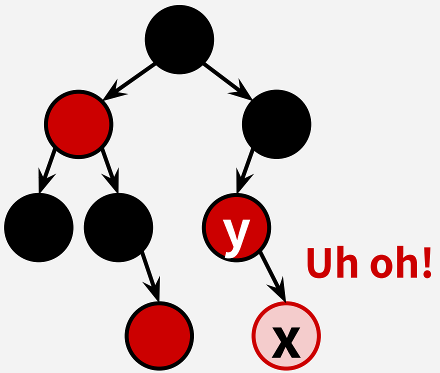
High-level plan

Insert as normal (same insert as BST), and then fix.

Fix = recolor and/or apply rotations until RB Tree properties are met.

INSERT(x):

- Insert **x** normally (**x** becomes a leaf)
- Color **x** **red**
- If **x**'s parent **y** is **black**, then we're done!
- Otherwise, **y** is **red**, so we have two red nodes in a row and need to do some fixing!



ساختمان داده و الگوریتم ها (CE203)

جلسه هجدهم:
درهم سازی

سجاد شیرعلی شهرضا
پاییز 1400
شنبه، 13 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

HASH TABLE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	HASH TABLES (HOPEFULLY)
SEARCH	O(log(n))	O(n)	O(1)
DELETE	O(n)	O(n)	O(1)
INSERT	O(n)	O(1)	O(1)

SOME TERMINOLOGY

There exists a universe **U** of keys, with size M.

Generally, M is *really big*. Examples:

- U = the set of all ASCII strings of length 20. M = 26^{20}
- U = the set of all IPv4 addresses. M = 2^{32}
- U = the set of all possible YouTube view stats. M = 8.6 billion

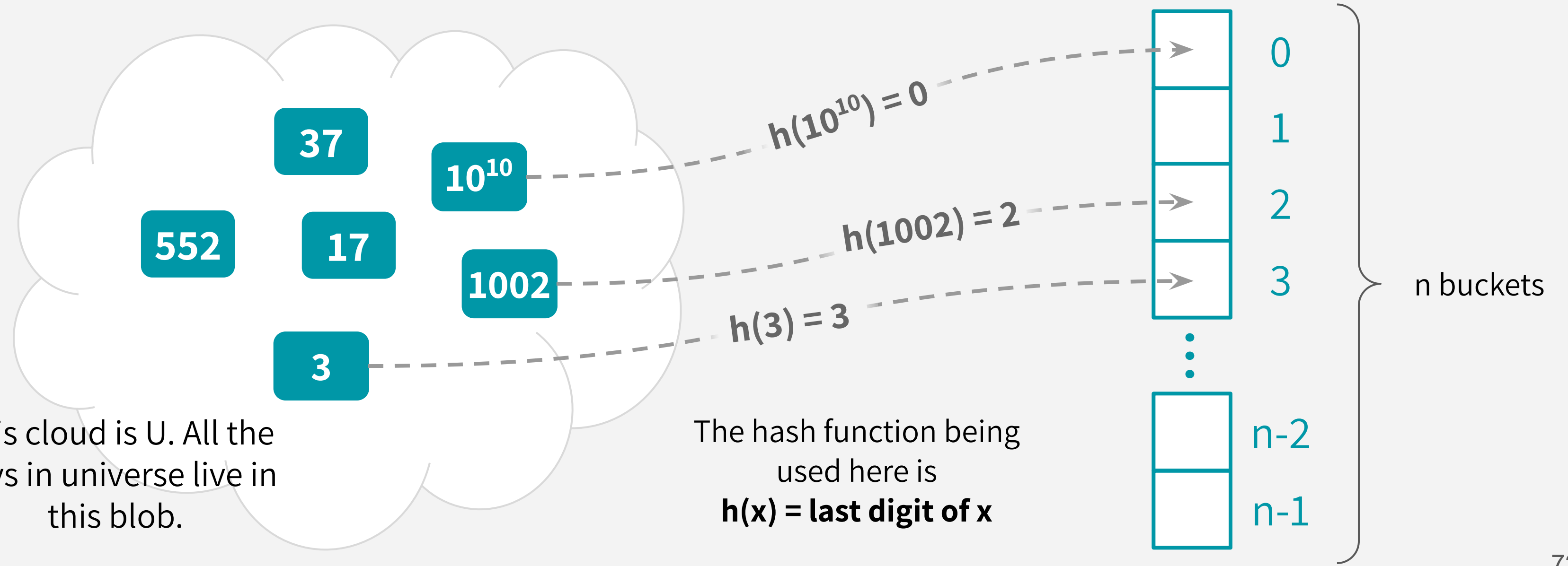
Our job is to store **n** keys, and we assume M >> n

Only a few (at most n) elements of U are ever going to show up. We don't know which ones will show up in advance.

A hash function **h: U → {1, ..., n}**
maps elements of U to buckets 1, ..., n

SOME TERMINOLOGY

A hash function **h: U → {1, …, n}**
maps elements of U to buckets 1, …, n

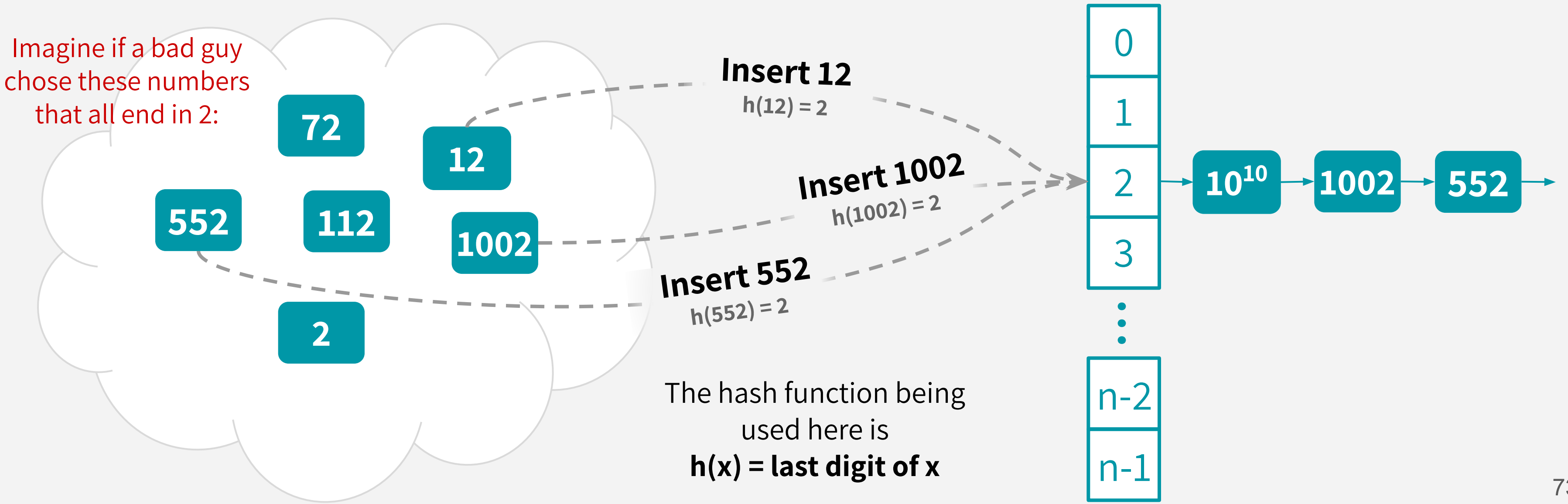


This cloud is U. All the keys in universe live in this blob.

The hash function being used here is
h(x) = last digit of x

COLLISION RESOLUTION: CHAINING

But if the items are all clumped together in a single bucket, SEARCH/DELETE may be very slow because of the linked list traversal…



ساختمان داده و الگوریتم ها (CE203)

جلسه نوزدهم:
درهم سازی تصادفی

سجاد شیرعلی شهرضا

پاییز 1400

دوشنبه، 15 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

INTUITION

Intuitively, the adversary can't foil a hash function that they don't yet know.

So, our strategy is to define a set of hash functions, and then we randomly choose a hash function **h** from this set to use!

You can think of it like a game:

1. You announce your set of hash functions, **H**.
2. The adversary chooses **n** items for your hash function to hash.
3. You then randomly pick a hash function **h** from **H** to hash the **n** items.

UNIVERSAL HASH FAMILY

A **hash family** is a fancy name for a set of hash functions.

A hash family **H** is a **universal hash family** if,
when **h** is chosen uniformly at random from **H**,

$$\text{for all } u_i, u_j \in U \text{ with } u_i \neq u_j,$$

$$P_{h \in H}\left[h(u_i) = h(u_j)\right] \leq \frac{1}{n}$$

Then if we randomly choose **h** from a universal hash family **H**, we'll be guaranteed that:

E[# of items in u_i's bucket] ≤ 2 = O(1)

AN EXAMPLE

Here is one of the more well-studied universal hash families:

Pick a prime $\mathbf{p \geq M}$

Define $\mathbf{h_{a,b}(x) = ((ax + b) \bmod p) \bmod n}$

$$\mathbf{H = \{\, h_{a,b} : a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\} \,\}}$$

To draw a hash function **h** from **H**:

Pick a random **a** in {1, …, p - 1}.

&

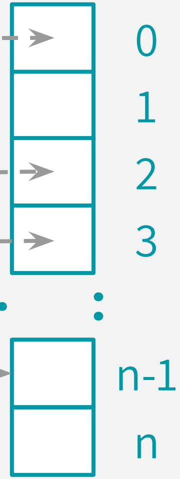
Pick a random **b** in {0, …, p - 1}.

THE WHOLE SCHEME

You choose your set of hash functions **H**, a universal hash family like H = mod p mod n.

When the client initializes a hash table, randomly pick a hash function **h** from **H** to use in the hash table to hash the items.

Now you're ready to start hashing values from the universe U!



We can now expect that these buckets will be pretty balanced

From now on, any operation on the hash table uses that same **h** that you randomly selected from **H**

HASH TABLE MOTIVATION

OPERATION	SORTED ARRAY	UNSORTED LINKED LIST	HASH TABLES (WORST-CASE)	HASH TABLES (EXPECTED)*
SEARCH	O(log(n))	O(n)	O(n)	O(1)
DELETE	O(n)	O(n)	O(n)	O(1)
INSERT	O(n)	O(1)	O(1)	O(1)

*** Assuming we implement it cleverly with a "good" hash function**

ساختمان داده و الگوریتم ها (CE203)

جلسه بیستم:
برنامه نویسی پویا

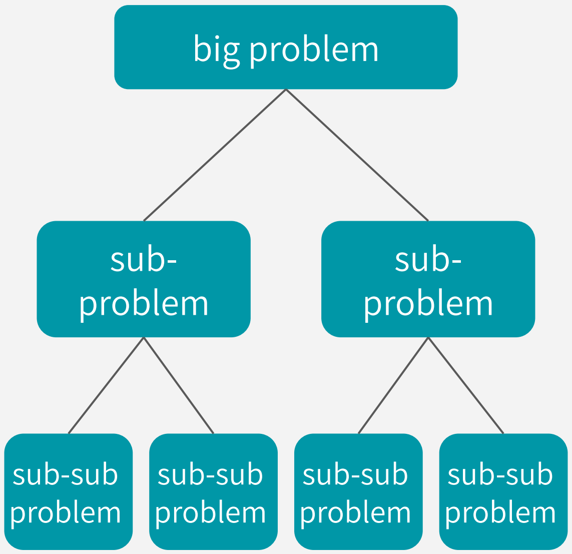
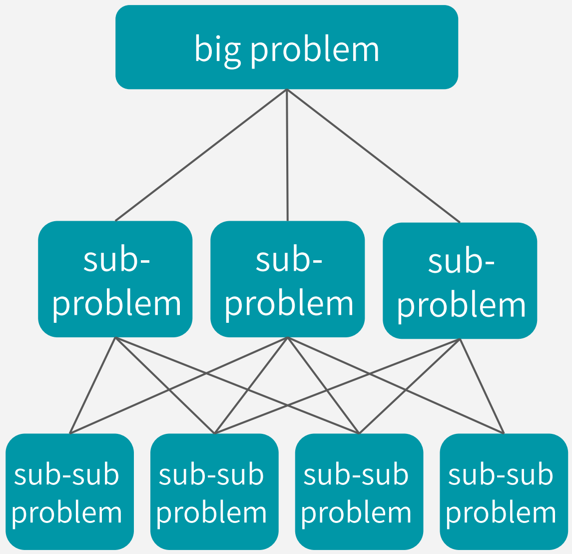
سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 20 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

DIVIDE & CONQUER vs DP

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING
	

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 $d^{(k)}$ may ask for $d^{(k-1)}[v]$**

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 shortly!) 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.

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!

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

Runtime: O(m·n)

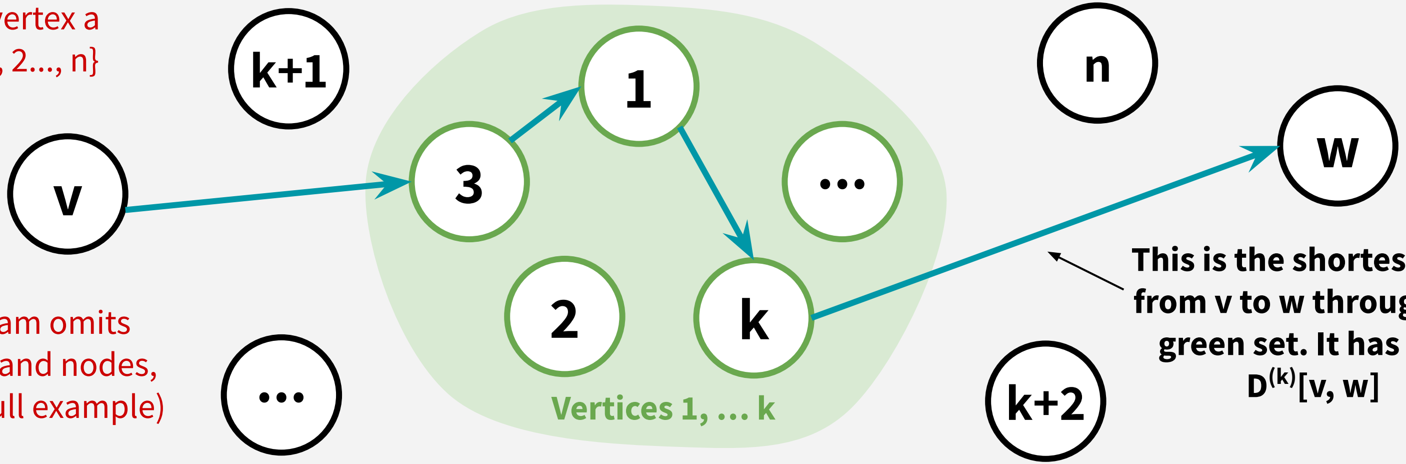
FLOYD-WARSHALL: A DP APPROACH

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}



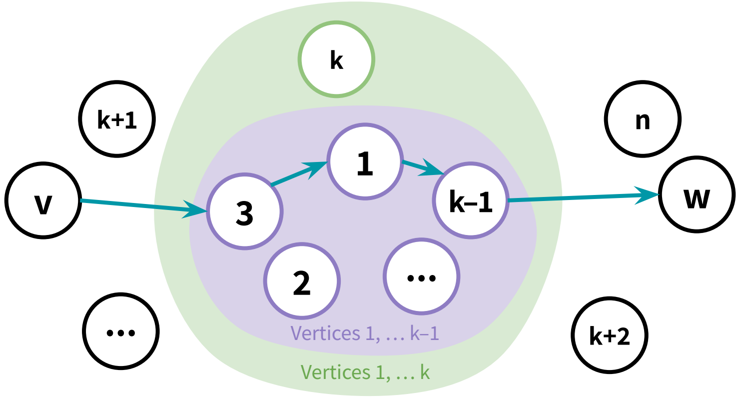
This is the shortest path from v to w through the green set. It has cost $D^{(k)}[v, w]$

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

FLOYD-WARSHALL: A DP APPROACH

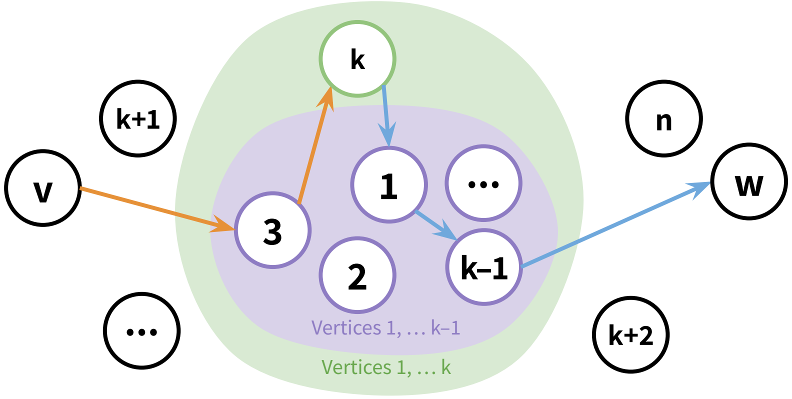
How do we find $D^{(k)}[v, w]$ using $D^{(k-1)}$? Choose the minimum of these 2 cases:

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] = ∞ for all v ≠ 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:
            return "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)

ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و یکم:
مسئله کوله پشتی

سجاد شیرعلی شهرضا

پاییز 1400

دوشنبه، 22 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

KNAPSACK PROBLEM: TWO VERSIONS

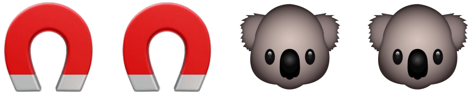
Capacity: **10**

Item:					
Weight:	**6**	**2**	**4**	**3**	**11**
Value:	**20**	**8**	**14**	**13**	**35**

UNBOUNDED KNAPSACK

We have infinite copies of all the items.

What's the most valuable way to fill the knapsack?



Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

0/1 KNAPSACK

We have only one copy of each item.

What's the most valuable way to fill the knapsack?

Total weight: **2 + 4 + 3 = 9**

Total value: **8 + 14 + 13 = 35**

STEP 1: OPTIMAL SUBSTRUCTURE

SUBPROBLEMS:

Unbounded Knapsack with a smaller knapsack

K[x] = optimal value you can fit in a knapsack of capacity **x**

Why does this make sense, and how can subproblems help me find an optimal solution for K[x]? Basically, I would like to take the maximum outcome over all the available possibilities:

My knapsack has capacity x. Which item should I put in my knapsack for now?

Well, if I put in item **i** with weight $\mathbf{w_i}$, the best value I could achieve is the value of item **i**, $\mathbf{v_i}$, *plus the optimal value for a smaller knapsack* that has capacity $\mathbf{x - w_i}$ (i.e. the remaining space once I put item **i** in).

Item:					
Weight:	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	…	$\mathbf{w_n}$
Value:	$\mathbf{v_1}$	$\mathbf{v_2}$	$\mathbf{v_3}$		$\mathbf{v_n}$

STEP 2: RECURSIVE FORMULATION

K[x] = optimal value you can fit in a knapsack of capacity **x**

Our recursive formulation:

$$\mathbf{K[x]} = \begin{cases} \mathbf{0} & \text{if there are no i where } w_i \le x \\ \mathbf{max_i \{ \ K[x - w_i] + v_i \ \}} & \text{otherwise} \end{cases}$$

The maximum is over all items i s.t. $w_i \le x$ (i.e. over all the items that could actually fit)

Optimal way to fill the smaller knapsack

The value of item i

STEP 3: WRITE A DP ALGORITHM

$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \le x \\ \max_i \{ K[x-w_i] + v_i \} & \text{otherwise} \end{cases}$$

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize a size W+1 array, K
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if wᵢ ≤ x:
                K[x] = max{ K[x], K[x-wᵢ] + vᵢ }
    return K[W]
```

Make sure that our base case is set up (0 capacity means 0 value)

Iterate over each knapsack size from smallest to largest

Iterate over each possible item & only process those that could actually fit in a size x knapsack

Runtime: O(nW)

You do O(n) work to fill out each of the W entries in the array

STEP 4: FIND ACTUAL ITEMS

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] ∪ {item i}
    return ITEMS[W]
```

STEP 1: OPTIMAL SUBSTRUCTURE

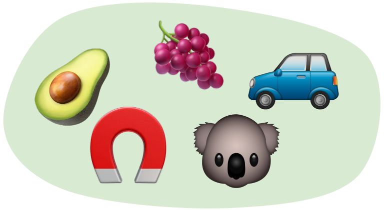
SUBPROBLEM (ATTEMPT #2):

0/1 Knapsack with a smaller knapsack & *fewer items*

Our subproblems will be indexed by x and j:

K[x, j] = optimal solution for a knapsack of size x using only the first j items

Capacity x



First j items

First sol
for

Ther

Tl
mor

STEP 2: RECURSIVE FORMULATION

K[x, j] = optimal value you can fit in a knapsack of capacity **x** with items 1 through **j**

Our recursive formulation:

$$K[x,j] = \begin{cases} 0 & \text{if x or j are 0} \\ \max\{\ K[x, j-1],\ \ K[x-w_j, j-1] + v_j\ \} & \text{otherwise} \end{cases}$$

Optimal way to fill the same size knapsack without using item j

Optimal way to fill the smaller knapsack when we no longer have access to item j

value gained by using item j

STEP 3: WRITE A DP ALGORITHM

$$K[x,j] = \begin{cases} 0 & \text{if x or j are 0} \\ \max\{\, K[x,j-1],\ K[x-w_j, j-1] + v_j \,\} & \text{otherwise} \end{cases}$$

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
   return K[W,n]
```

Make sure that our base case is set up (0 value for entries where we have 0 capacity or 0 items)

Iterate over knapsack sizes from smallest to largest

Iterate over items we can consider

Default case: we don't use item j

But if item j can fit, then we'll consider using it!

Runtime: O(nW)

You do O(1) work to fill out each of the nW entries in the table

ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و دوم:
روش های حریصانه

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 4 دی 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

THE GREEDY PARADIGM

Commit to choices one-at-a-time,

never look back,

and hope for the best.

Greedy doesn't always work.

We'll see some non-examples where a tempting greedy approach won't work.
Then, we'll see some examples where a greedy solution exists!

THE GREEDY PARADIGM

DISCLAIMER: It's often surprisingly easy to come up with ideas for greedy algorithms, they're usually pretty easy to write down, and their runtimes are straightforward to analyze! But you'll end up wondering, "how am I supposed to know *when* I can use greedy algorithms?" The answer may not be satisfying: *a lot of the times, greedy algorithms are not correct, and whenever they are correct, it can be difficult to prove its correctness.* This aspect of greedy algorithms is why we've waited until the end of class to discuss this design paradigm!

Then, we'll see some examples where a greedy solution exists.

NON-EXAMPLE: GREEDY KNAPSACK?

Can we design a greedy algorithm for Unbounded Knapsack?

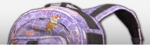
UNBOUN

We have inf

What's the mos

Total w

Total va



0

3

13

11

35

This doesn't work! We ended up "regretting" our greedy choices. By the time we put in the third koala, we realized that a magnet would have been better (even though it doesn't immediately seem as valuable at the time) because it would have left enough space for a fourth object that could bump up our overall value!

Greedy approach? Here's an idea: koalas have the best value/weight ratio, so keep using koalas!

Total weight: **3 + 3 + 3 = 9**

Total value: **13 + 13 + 13 = 39**

ACTIVITY SELECTION: PSEUDOCODE

```
ACTIVITY_SELECTION(activities A with start and finish times):
   A = MERGESORT_BY_FINISHTIMES(A)
   result = {}
   busy_until = 0
   for a in A:
      if a.start >= busy_until:
         result.add(a)
         busy_until = a.finish
   return result
```

Runtime: O(n log n)

WHY IS IT GREEDY?

What makes our algorithm a **greedy** algorithm?

At each step in the algorithm, we make a choice (pick the available activity with the smallest finish time) and never look back.

How do we know that this greedy algorithm is correct?
(Proving correctness is the hard part!)

THE BIG IDEA:
Whenever we make a choice, we don't rule out an optimal solution.

ACTIVITY SELECTION: CORRECTNESS

We want to prove that the algorithm finds an optimal set of activities (i.e. there isn't a better set available)

Note: there could be other optimal solutions, too! We're just proving that ours is at least as good as any optimal solution.

High-level proof idea:

At every step of the algorithm, the greedy choice we make doesn't rule out an optimal solution. By the end of the algorithm, we've got some solution, so it must be optimal!

In other words, at every step of the algorithm, there is always an optimal solution that *extends* the set of choices we made so far.

We'll perform induction on the # of greedy choices we make!

A STRATEGY FOR GREEDY PROOFS

The inductive step (If you haven't ruled out success after choice t, then show that you won't rule out success after choice t+1) **will often look like:**

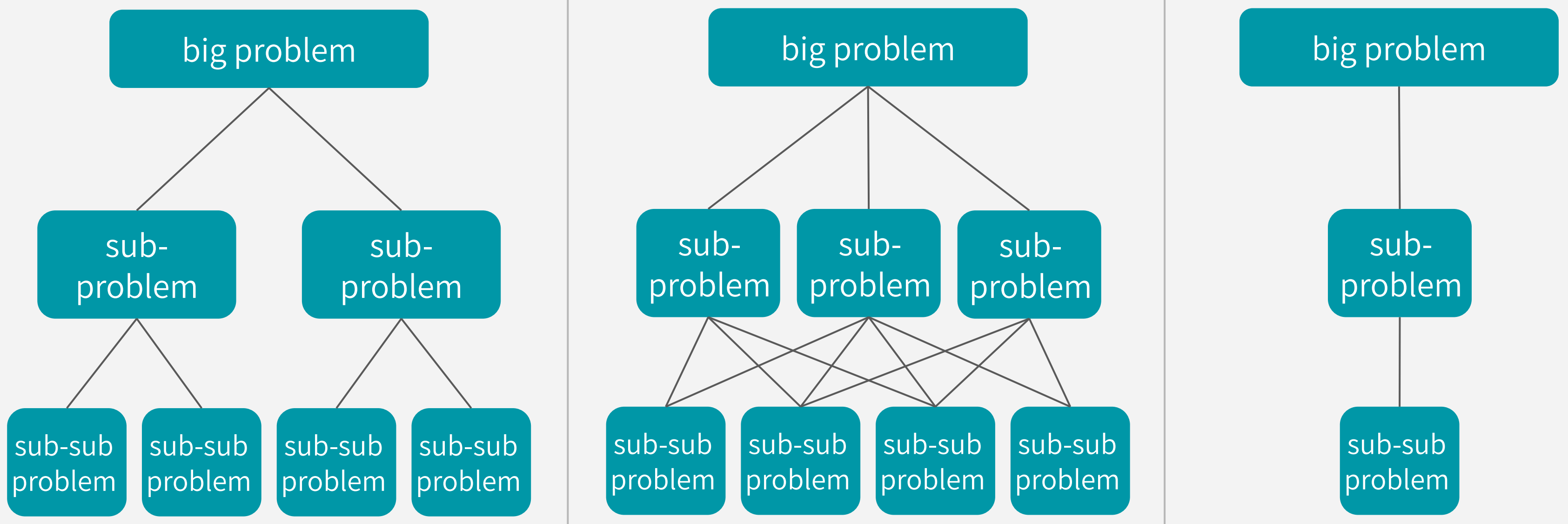
Suppose we're on track to make some optimal solution T*
(e.g. after we've picked k–1 activities, we're still on track)

Suppose that T* disagrees with our next greedy choice
(e.g. T* doesn't involve activity k)

Manipulate T* in order to make another solution T that's not worse (i.e. also optimal) but now *agrees* with our greedy choice!
(e.g. replace whatever activity T* had picked next with our greedy choice of activity k)

D&C vs. DP vs. GREEDY

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING	GREEDY



ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و سوم:
نمونه های دیگر الگوریتمهای حریصانه

سجاد شیرعلی شهرضا

پاییز 1400

شنبه، 11 دی 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

SCHEDULING: "PSEUDOCODE"

Our greedy choice: always choose the job with the next biggest ratio:

$$\frac{\textbf{cost (per hour until finished)}}{\textbf{time it takes}}$$

SCHEDULING(n jobs with times & costs):

    Compute cost/time ratios for all jobs
    Sort jobs in descending order of cost/time ratios
    Return sorted jobs!

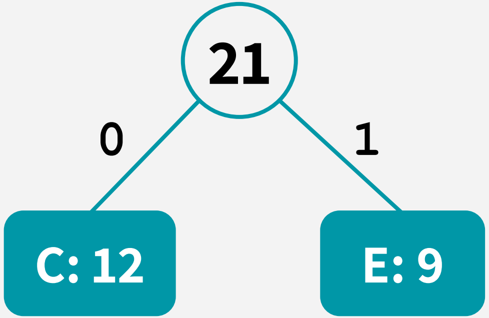
Runtime: O(n log n)

HUFFMAN CODING: THE IDEA

IDEA: Greedily build sub-trees from the bottom up, where the “greedy goal” is to have less frequent letters further down in the tree.

To ensure that less frequent letters are further down in the tree, we’ll greedily build subtrees, by “merging” the 2 node with the smallest frequency count, and then repeating until we’ve merged everything!

A **“merge”** between 2 nodes creates a common parent node whose key is the sum of those 2 nodes frequencies:



HUFFMAN CODING: PSEUDOCODE

```
HUFFMAN_CODING(Characters C, Frequencies F):
    Create a node for each character (key is its frequency)
    CURRENT = {set of all these nodes}
    while len(CURRENT) > 1:
        X and Y ← the 2 nodes in CURRENT with the smallest keys
        Create a new node Z with Z.key = X.key + Y.key
        Z.left = X, Z.right = Y
        Add Z to CURRENT, and remove X and Y from CURRENT
    return CURRENT[0]
```

Pre-sorting frequencies using RADIXSORT (if frequencies are appropriate!!!) and using 2 queues (can you figure this out?)

Runtime: O(n)

سوال؟