

Table of Contents

- Lecture 01 - intro
- Lecture 02 - dynamic - arrays
- Lecture 03 - heaps
- Lecture 04 - bst
- Lecture 05 - treaps
- Lecture 06 - kd - trees
- Lecture 07 - lsh
- Lecture 08 - union - find
- Lecture 09 - greedy
- Lecture 10 - backtracking
- Lecture 11 - dp
- Lecture 12 - dp - lcs
- Lecture 13 - pytest
- Lecture 14 - rabin - karp
- Lecture 15 - tries
- Lecture 16 - suffix - trees
- Lecture 17 - bloom - filters
- Lecture 18 - count - min - sketch
- Lecture 19 - complexity

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 1 | Part 1

Welcome!

Advanced Data Structures and Algorithms

(for data science)

- ▶ Second time being taught.
- ▶ Modeled (partly) after CSE 100/101.
- ▶ But with more data science flavor.

Roadmap

- ▶ Advanced Data Structures
 - ▶ Dynamic Arrays
 - ▶ AVL Trees
 - ▶ Heaps
 - ▶ Disjoint Set Forests
- ▶ Nearest Neighbor Queries
 - ▶ KD-Trees
 - ▶ Locality Sensitive Hashing

Roadmap

- ▶ Strings
 - ▶ Tries and Suffix Trees
 - ▶ Knuth-Morris-Pratt and Rabin-Karp string search
- ▶ Algorithm Design
 - ▶ Divide and Conquer
 - ▶ Greedy Algorithms
 - ▶ Dynamic Programming (Viterbi Algorithm)
 - ▶ Backtracking, Branch and Bound
 - ▶ Linear Time Sorting; Sort with Noisy Comparator

Roadmap

- ▶ Sketching and Streaming
 - ▶ Count-min-sketch
 - ▶ Bloom filters
 - ▶ Reservoir Sampling?
- ▶ Theory of Computation
 - ▶ NP-Completeness and NP-Hardness
 - ▶ Computationally-hard problems in ML/DS

Roadmap?

- ▶ Other
 - ▶ Regular Expressions
 - ▶ Linear Programming
 - ▶ ?

Prerequisite Knowledge

- ▶ Python
- ▶ Basic Data Structures and Algorithms
 - ▶ DSC 30, DSC 40B

(syllabus)

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 1 | Part 2

Review of Time Complexity Analysis

Time Complexity Analysis

- ▶ Determine efficiency of code **without** running it.
- ▶ Idea: find a formula for time taken as a function of input size.

Advantages of Time Complexity

1. Doesn't depend on the computer.
2. Reveals which inputs are slow, which are fast.
3. Tells us how algorithm scales.

Counting Operations

- ▶ Abstraction: certain basic operations take **constant time**, no matter how large the input data set is.
- ▶ Example: addition of two integers, assigning a variable, etc.
- ▶ Idea: count basic operations

Example

```
def mean(numbers):
    total = 0
    n = len(numbers)
    for x in numbers:
        total += x
    return total / n
```

Theta Notation, Informally

- ▶ $\Theta(\cdot)$ forgets constant factors, lower-order terms.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

Theta Notation, Informally

- ▶ $f(n) = \Theta(g(n))$ if $f(n)$ “grows like” $g(n)$.

$$5n^3 + 3n^2 + 42 = \Theta(n^3)$$

Theta Notation Examples

► $4n^2 + 3n - 20 = \Theta(n^2)$

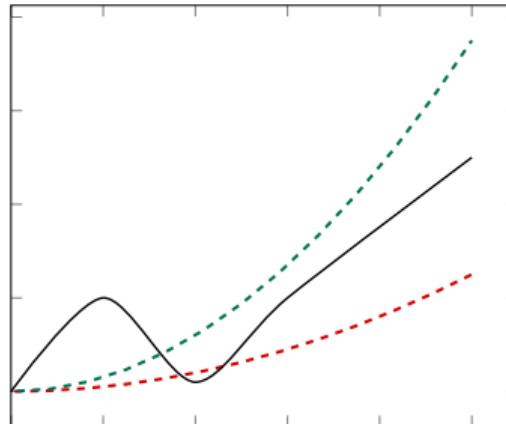
► $3n + \sin(4\pi n) = \Theta(n)$

► $2^n + 100n = \Theta(2^n)$

Definition

We write $f(n) = \Theta(g(n))$ if there are positive constants N , c_1 and c_2 such that for all $n \geq N$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Main Idea

If $f(n) = \Theta(g(n))$, then f can be “sandwiched” between copies of g when n is large.

Other Bounds

- ▶ $f = \Theta(g)$ means that f is both **upper** and **lower** bounded by factors of g .
- ▶ Sometimes we only have (or care about) upper bound or lower bound.
- ▶ We have notation for that, too.

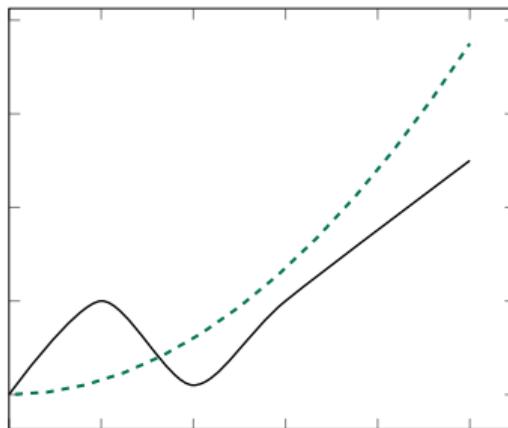
Big-O Notation, Informally

- ▶ Sometimes we only care about upper bound.
- ▶ $f(n) = O(g(n))$ if $f(n)$ “grows at most as fast” as $g(n)$.
- ▶ Examples:
 - ▶ $4n^2 = O(n^{100})$
 - ▶ $4n^2 = O(n^3)$
 - ▶ $4n^2 = O(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = O(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

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



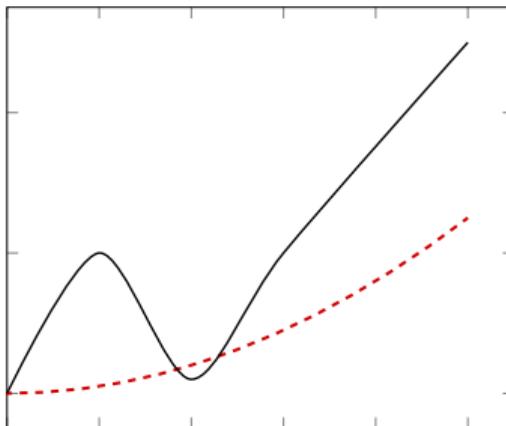
Big-Omega Notation

- ▶ Sometimes we only care about lower bound.
- ▶ Intuitively: $f(n) = \Omega(g(n))$ if $f(n)$ “grows at least as fast” as $g(n)$.
- ▶ Examples:
 - ▶ $4n^{100} = \Omega(n^5)$
 - ▶ $4n^2 = \Omega(n)$
 - ▶ $4n^2 = \Omega(n^2)$ and $4n^2 = \Theta(n^2)$

Definition

We write $f(n) = \Omega(g(n))$ if there are positive constants N and c such that for all $n \geq N$:

$$c_1 \cdot g(n) \leq f(n)$$



Sums of Theta

- ▶ If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$\begin{aligned}f_1(n) + f_2(n) &= \Theta(g_1(n) + g_2(n)) \\&= \Theta(\max(g_1(n), g_2(n)))\end{aligned}$$

- ▶ Useful for sequential code.

Products of Theta

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then

$$f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$$

Example

```
def foo(n):
    for i in range(3*n + 4, 5n**2 - 2*n + 5):
        for j in range(500*n, n**3):
            print(i, j)
```

Linear Search

- ▶ **Given:** an array arr of numbers and a target t.
- ▶ **Find:** the index of t in arr, or **None** if it is missing.

```
def linear_search(arr, t):
    for i, x in enumerate(arr):
        if x == t:
            return i
    return None
```

Exercise

What is the time complexity of `linear_search`?

The Best Case

- ▶ When t is the very first element.
- ▶ The loop exits after one iteration.
- ▶ $\Theta(1)$ time?

The Worst Case

- ▶ When t is not in the array at all.
- ▶ The loop exits after n iterations.
- ▶ $\Theta(n)$ time?

Time Complexity

- ▶ `linear_search` can take vastly different amounts of time on two inputs of the **same size**.
 - ▶ Depends on **actual elements** as well as size.
- ▶ There is no single, overall time complexity here.
- ▶ Instead we'll report **best** and **worst** case time complexities.

Best Case Time Complexity

- ▶ How does the time taken in the **best case** grow as the input gets larger?

Definition

Define $T_{\text{best}}(n)$ to be the **least** time taken by the algorithm on any input of size n .

The asymptotic growth of $T_{\text{best}}(n)$ is the algorithm's **best case time complexity**.

Best Case

- ▶ In `linear_search`'s **best case**, $T_{\text{best}}(n) = c$, no matter how large the array is.
- ▶ The **best case time complexity** is $\Theta(1)$.

Worst Case Time Complexity

- ▶ How does the time taken in the **worst case** grow as the input gets larger?

Definition

Define $T_{\text{worst}}(n)$ to be the **most** time taken by the algorithm on any input of size n .

The asymptotic growth of $T_{\text{worst}}(n)$ is the algorithm's **worst case time complexity**.

Worst Case

- ▶ In the worst case, `linear_search` iterates through the entire array.
- ▶ The **worst case time complexity** is $\Theta(n)$.

Faux Pas

- ▶ Asymptotic time complexity is not a **complete** measure of efficiency.
- ▶ $\Theta(n)$ is not always better than $\Theta(n^2)$.
- ▶ Why?

Faux Pas

- ▶ **Why?** Asymptotic notation “hides the constants”.
- ▶ $T_1(n) = 1,000,000n = \Theta(n)$
- ▶ $T_2(n) = 0.00001n^2 = \Theta(n^2)$
- ▶ But $T_1(n)$ is **worse** for all but really large n .

Main Idea

Asymptotic time complexity is not the **only** way to measure efficiency, and it can be misleading.

Sometimes even a $\Theta(2^n)$ algorithm is better than a $\Theta(n)$ algorithm, if the data size is small.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 1 | Part 3

Arrays and Linked Lists

Memory

- ▶ To access a value, we must know its **address**.



Sequences

- ▶ How do we store an **ordered sequence**?
 - ▶ e.g.: 55, 22, 12, 66, 60
- ▶ Array? Linked list?

Arrays

- ▶ Store elements **contiguously**.
 - ▶ e.g.: 55, 22, 12, 66, 60



- ▶ NumPy arrays are... arrays.

Allocation

- ▶ Memory is shared resource.
- ▶ A chunk of memory of fixed size has to be reserved (**allocated**) for the array.
- ▶ The size has to be known beforehand.



Arrays

- ▶ To access an element, we need its address.
- ▶ **Key:** Addresses are easily calculated.
 - ▶ For k th element: address of first + $(k \times 64)$ bits
- ▶ Therefore, arrays support $\Theta(1)$ -time access.

Downsides of Arrays

- ▶ Homogeneous; every element must be same size.
- ▶ To **resize** the array, a totally new chunk of memory has to be found; old values copied over¹.



¹In worst case: see realloc

Array Time Complexities

- ▶ Retrieve k th element: $\Theta(1)$ (**good**).
- ▶ Append element at end: $\Theta(n)$ (**bad**)².
- ▶ Insert/remove in middle: $\Theta(n)$ (**bad**).
- ▶ Allocation: $\Theta(n)$ if initialized,³ else $\Theta(1)$

²At least on average. See: `realloc`

³On Linux this is done lazily, as can be seen by timing `np.zeros`

Aside: np.append

```
»> arr = np.array([1, 2, 3])
»> np.append(arr, 4) # takes Theta(n) time!
array([1, 2, 3, 4])
```

Aside: np.append

```
results = np.array([])
for i in np.arange(100):
    result = run_simulation()
    results = np.append(results, result)
```

Aside: np.append

- ▶ This was **bad** code!
- ▶ We allocate/copy a **quadratic** number of elements:

$$\underbrace{1}_{\text{1st iter}} + \underbrace{2}_{\text{2nd iter}} + \underbrace{3}_{\text{3rd iter}} + \dots + \underbrace{100}_{\text{last iter}} = \frac{100 \times 101}{2} = 5050$$

Aside: np.append

- ▶ Better: pre-allocate.

```
results = np.empty(100)
for i in np.arange(100):
    results[i] = run_simulation()
```

(Doubly) Linked Lists

- ▶ Scatter elements throughout memory.
- ▶ For each, store address of next/previous.



Linked Lists

- ▶ Each element has an **address**.
- ▶ Keep track of the address of first/last elements.
- ▶ Have to **find** address of middle elements by looping.

Linked List Time Complexities

- ▶ Retrieve k th element:
 - ▶ $\Theta(k)$ if you don't know address (**bad**)⁴
 - ▶ $\Theta(1)$ if you do
- ▶ Append/pop element at start/end: $\Theta(1)$ (**good**).
- ▶ Insert/remove k th element:
 - ▶ $\Theta(k)$ if you don't know address (**bad**)
 - ▶ $\Theta(1)$ if you do
- ▶ Allocation not needed! (**good**)

⁴assumes search starts from beginning

Tradeoffs

- ▶ Arrays are better for numerical algorithms.
 - ▶ Arrays have good cache performance.
- ▶ Linked lists are better for stacks and queues.

Main Idea

Different data structures optimize for different operations.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 1

More about Memory

Is appending an array really so slow?

- ▶ `malloc()` vs. `realloc()`



Is appending an array really so slow?

- ▶ If realloc doesn't copy: $\Theta(1)$.
- ▶ If realloc copies: $\Theta(n)$.
- ▶ Assume p is probability that realloc copies.
- ▶ **Expected time** is still¹ $\Theta(n)$.

¹If p doesn't depend on n .

How is empty memory found?

- ▶ Basically: a linked list.



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 2

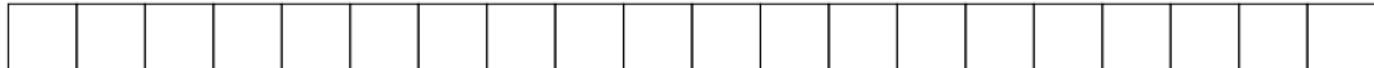
Dynamic Arrays

Motivation

- ▶ Can we have the best of both worlds?
- ▶ $\Theta(1)$ time access like an array.
- ▶ $\Theta(1)$ time append like a linked list.
- ▶ **Yes!** (sort of)

The Idea

- ▶ Allocate memory for an **underlying array**.
 - ▶ say, 512 elements
 - ▶ This is the **physical size**.
- ▶ To append element, insert into first unused slot.
 - ▶ Number of elements used is the **logical size**.
 - ▶ $\Theta(1)$ time.



The Idea

- ▶ We'll eventually run out of unused slots.
- ▶ Fix: allocate a new underlying array whose physical size is γ times as large.
 - ▶ γ is the **growth factor**.
 - ▶ Commonly, $\gamma = 2$; i.e., double its size.
 - ▶ Takes $\Theta(k)$ time, where k is current size.

Example



```
»> arr = DynamicArray(initial_physical_size=4)
»> arr.append(1)
»> arr.append(2)
»> arr.append(3)
»> arr.append(4)
»> arr.append(5)
```

(notebook)

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 3

Amortized Analysis

Analysis

- ▶ Appending takes $\Theta(1)$ time *usually...*
- ▶ ...but takes $\Theta(k)$ time when we run out of slots.
 - ▶ Where k is current size of sequence.

The Key

- ▶ Resizing is expensive, but rare.
 - ▶ If $\gamma = 2$, each new resize is twice as expensive, but happens half as often.
- ▶ Thus, the **cost per append** is small.
- ▶ **Amortize** the cost over all previous appends.

Amortized Time Complexity

- ▶ The **amortized** time for an append is:

$$T_{\text{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

- ▶ We'll see that $T_{\text{amort}}(n) = \Theta(1)$.

Amortized Analysis

total time for n appends

=

total time for **non-growing** appends

+

total time for **growing** appends

Counting Growing Appends

- ▶ Want to calculate time taken by growing appends.
- ▶ First: how many appends caused a resize?
 - ▶ β : initial physical size
 - ▶ γ : growth factor

Counting Growing Apps

- ▶ Suppose initial physical size is $\beta = 512$, and $\gamma = 2$
- ▶ Resizes occur on append #:
 $512, 1024, 2048, 4096, \dots$
- ▶ In general, resizes occur on append #:
 $\beta\gamma^0, \beta\gamma^1, \beta\gamma^2, \beta\gamma^3, \dots$

Counting Growing Appends

- ▶ In a sequence of n appends, how many caused the physical size to grow?
- ▶ Simplification: Assume n is such that n th append caused a resize. Then, for some $x \in \{0, 1, 2, \dots\}$:

$$n = \beta\gamma^x$$

- ▶ If $x = 0$ there was 1 resize; if $x = 1$ there were 2; etc.

Counting Growing Appends

- ▶ Solving for x :

$$x = \log_{\gamma} \frac{n}{\beta}$$

- ▶ Check: without assumption, $x = \lfloor \log_{\gamma} \frac{n}{\beta} \rfloor$
- ▶ Number of resizes is $\lfloor \log_{\gamma} \frac{n}{\beta} \rfloor + 1$

Counting Growing Apps

- ▶ Number of resizes is $\lfloor \log_{\gamma} \frac{n}{\beta} \rfloor + 1$
- ▶ Check with $\gamma = 2, \beta = 512, n = 400$
 - ▶ Correct # of resizes: 0
- ▶ Check with $\gamma = 2, \beta = 512, n = 1100$
 - ▶ Correct # of resizes: 2

Time of Growing Appends

- ▶ How much time was taken across all appends that caused resizes?
- ▶ Assumption: resizing an array with physical size k takes time $ck = \Theta(k)$.
 - ▶ c is a constant that depends on γ .

Time of Growing Appends

- ▶ Time for first resize: $c\beta$.
- ▶ Time for second resize: $c\gamma\beta$.
- ▶ Time for third resize: $c\gamma^2\beta$.
- ▶ Time for j th resize: $c\gamma^{j-1}\beta$.
- ▶ This is a **geometric progression**.

Time of Growing Appends

- ▶ Time for j th resize: $c\gamma^{j-1}\beta$.
- ▶ Suppose there are r resizes.
- ▶ Total time:

$$c\beta \sum_{j=1}^r \gamma^{j-1} = c\beta \sum_{j=0}^r \gamma^j$$

Recall: Geometric Sum

- ▶ From some class you've taken:

$$\sum_{p=0}^N x^p = \frac{1 - x^{N+1}}{1 - x}$$

- ▶ Example:

$$1 + 2 + 4 + 8 + 16 = \sum_{p=0}^4 2^p = \frac{1 - 2^5}{1 - 2} = 31$$

Time of Growing Appends

- ▶ Total time:

$$c\beta \sum_{j=0}^r \gamma^j = c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma}$$

Time of Growing Appends

- ▶ Remember: in n appends there are $r = \lfloor \log_{\gamma} \frac{n}{\beta} \rfloor + 1$ resizes.
- ▶ Total time:

$$\begin{aligned} c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma} &= c\beta \frac{1 - \gamma^{\lfloor \log_{\gamma} \frac{n}{\beta} \rfloor + 2}}{1 - \gamma} \\ &= \Theta(n) \end{aligned}$$

Amortized Analysis

total time for n appends

=

total time for **non-growing** appends

+

$\Theta(n)$ ← total time for **growing** appends

Time of Non-Growing Appends

- ▶ In a sequence of n appends, how many are **non-growing**?

$$n - \left(\lfloor \log_{\gamma} \frac{n}{\beta} \rfloor + 1 \right) = \Theta(n)$$

- ▶ Time for one such append: $\Theta(1)$.
- ▶ Total time: $\Theta(n) \times \Theta(1) = \Theta(n)$.

Amortized Analysis

total time for n appends

=

$\Theta(n)$ \leftarrow total time for **non-growing** appends

+

$\Theta(n)$ \leftarrow total time for **growing** appends

Amortized Time Complexity

- ▶ The **amortized** time for an append is:

$$\begin{aligned} T_{\text{amort}}(n) &= \frac{\text{total time for } n \text{ appends}}{n} \\ &= \frac{\Theta(n)}{n} \\ &= \Theta(1) \end{aligned}$$

Dynamic Array Time Complexities

- ▶ Retrieve k th element: $\Theta(1)$
- ▶ Append/pop element at end:
 - ▶ $\Theta(1)$ best case
 - ▶ $\Theta(n)$ worst case (where n = current size)
 - ▶ $\Theta(1)$ amortized
- ▶ Insert/remove in middle: $O(n)$
 - ▶ May or may not need resize, still $O(n)$!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 4

Practicalities

Advantages

- ▶ Great cache performance (it's an array).
- ▶ Fast access.
- ▶ Don't need to know size in advance of allocation.

Downsides

- ▶ Wasted memory.
- ▶ Expensive deletion in middle.

Implementations

- ▶ Python: `list`
- ▶ C++: `std::vector`
- ▶ Java: `ArrayList`

Exercise

Why do we need `np.array`? Python's `list` is a dynamic array, isn't that better?

In defense of np.array

- ▶ Memory savings are one reason.
- ▶ Bigger reason: using Python's `list` to store numbers does not have good `cache` performance.



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 1

Abstract Data Types

Python's `list`

- ▶ You can go a long time without ever knowing how `list` is **implemented**.
- ▶ But you knew its **interface**.
 - ▶ supports `.append`, random access, is ordered, etc.

Abstract vs. Concrete

- ▶ An **abstract data type** (ADT) is a formal description of a type's **interface**.
- ▶ A **data structure** is a concrete strategy for implementing an abstract data type.
 - ▶ Describes how data is stored in memory.
 - ▶ How to access the data.

Example: Stacks

- ▶ A **stack** is an ADT which supports two operations:
 - ▶ push: put a new object on to the “top”
 - ▶ pop: remove and return item at the “top”
- ▶ Most often implemented using **linked lists**.
- ▶ But can also be implemented with **(dynamic) arrays**.

Main Idea

A given abstract data type can be implemented in several ways, but some data structures are more natural choices than others.

Main Idea

The data structure (not the abstract data type) determines the time complexity of operations.

Building Blocks

- ▶ Data structures are used to implement ADTs.
- ▶ But they are also used to implement more advanced data structures.
 - ▶ Example: arrays used to implement dynamic arrays.
- ▶ Arrays, linked lists are basic building blocks.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 2

Priority Queues

Priority Queues

- ▶ A **priority queue** is an abstract data type representing a collection.
- ▶ Each element has a **priority**.
- ▶ Supports operations¹:
 - ▶ `.pop_highest_priority()`
 - ▶ `.insert(value, priority)`
 - ▶ `.is_empty()`

¹and possibly more, like `.increase_priority`

Example

```
»> er = PriorityQueue()  
»> er.insert('flu', priority=1)  
»> er.insert('heart attack', priority=20)  
»> er.insert('broken hand', priority=10)  
»> er.pop_highest_priority()  
'heart attack'  
»> er.pop_highest_priority()  
'broken hand'
```

Applications

- ▶ Scheduling.
- ▶ Simulations of future events.
- ▶ Useful in algorithms.
 - ▶ Example: Prim's MST algorithm

Array Implementation

- ▶ We can implement a priority queue with a **(dynamic) array**.
- ▶ `.insert(k, p)`
 - ▶ append (value, priority) pair: $\Theta(1)$ time
- ▶ `.pop_highest_priority()`
 - ▶ find entry with highest priority: $\Theta(n)$ time
 - ▶ remove it: $O(n)$ time

Array Implementation (Variant)

- ▶ Alternatively, maintain dynamic array in sorted order of priority.
- ▶ `.insert(k, p)`
 - ▶ find place in sorted order: $\Theta(\log n)$ time worst case
 - ▶ actually insert: $\Theta(n)$ time worst case
- ▶ `.pop_highest_priority()`
 - ▶ remove/return last entry: $\Theta(1)$ time

Main Idea

If we made no insertions/deletions, a sorted array would be great. But we want a data structure with quick remove/return even after being modified.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 3

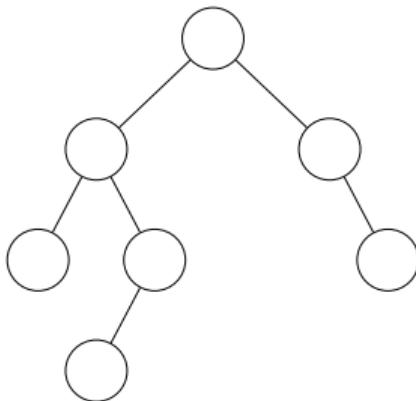
Binary Heaps

Binary Heaps

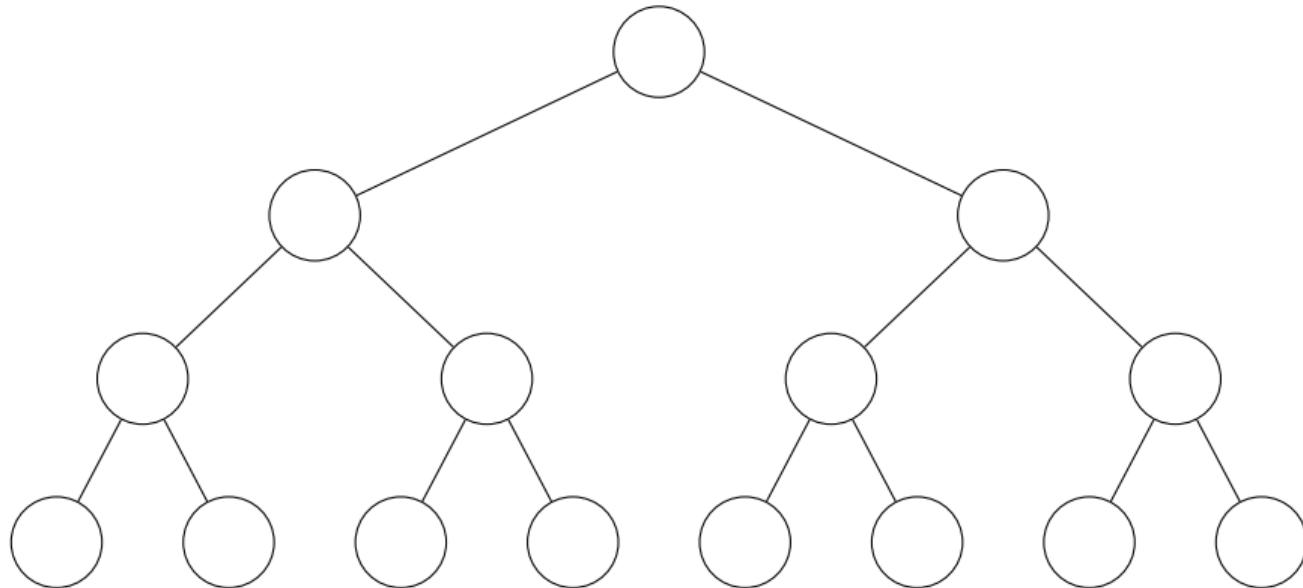
- ▶ A **binary heap** is a **binary tree** data structure often used to implement **priority queues**.

Binary Trees

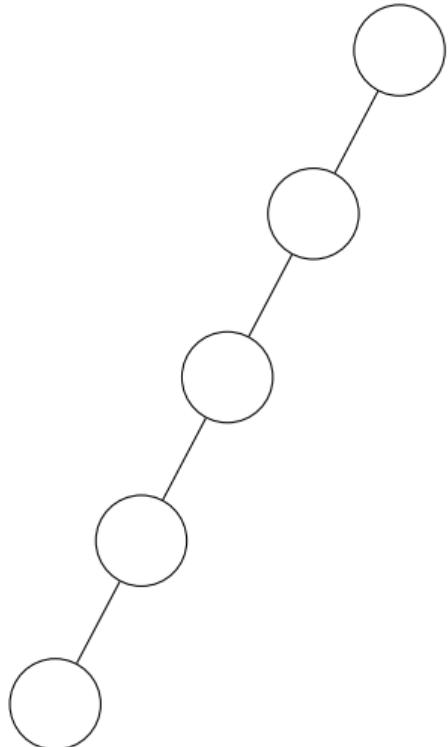
- ▶ Each node has **at most** two children (left, right).



Example

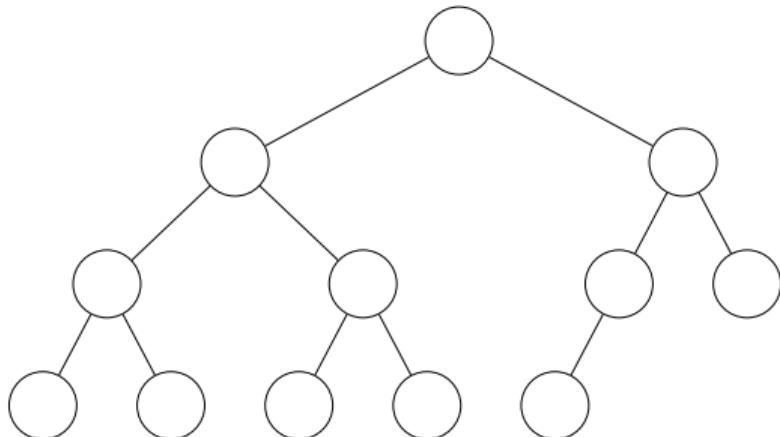


Example



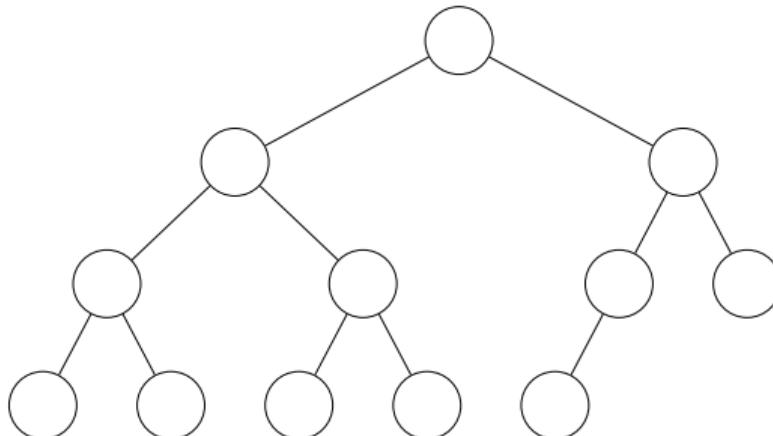
Complete Binary Trees

- ▶ A binary tree is **complete** if every level is filled, except for possibly the last (which fills from left to right).



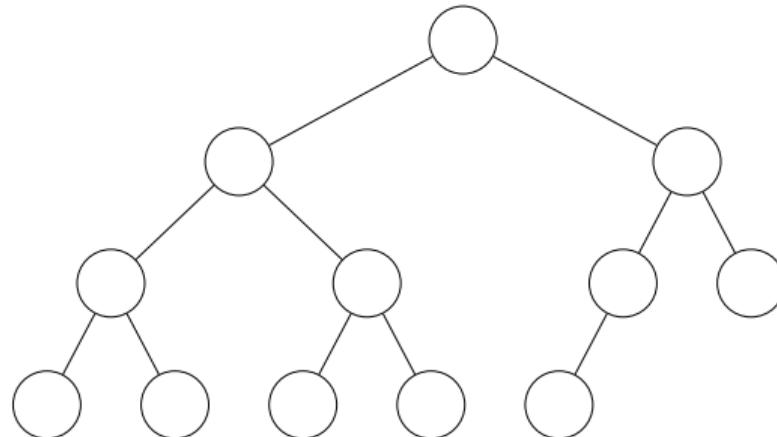
Node Height

- ▶ The **height** of node in a tree is the largest number of edges along any path to a leaf.
- ▶ The **height** of a tree is the height of the root.



Complete Tree Height

- ▶ The height of a complete binary tree with n nodes is $\Theta(\log n)$.



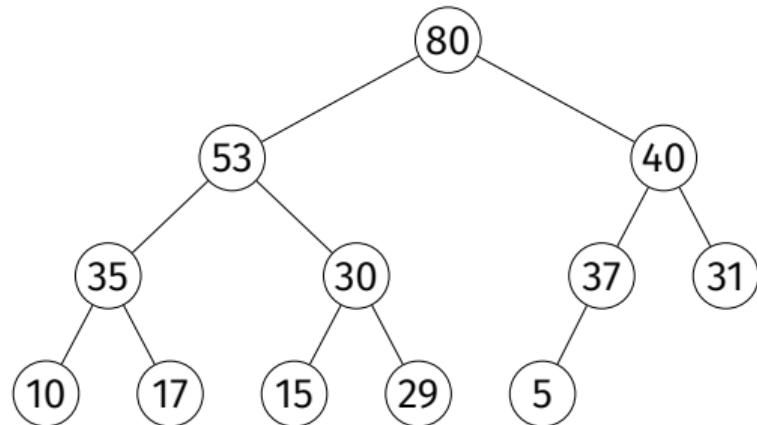
Binary Heap Properties

- ▶ A **binary max heap**² is a binary tree with three additional properties:
 1. Each node has a **key**.
 2. **Shape**: the tree is complete.
 3. **Max-Heap**: the key of a node is \geq the key of each of its children.

²There's also a min heap, of course.

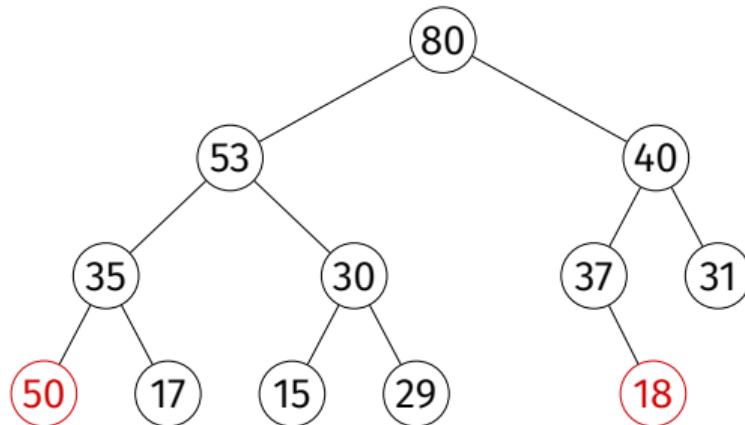
Example

- ▶ This is a binary max-heap.



Example

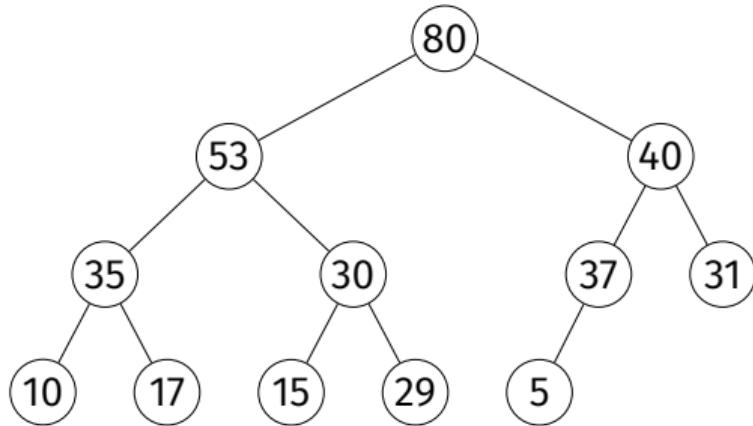
- ▶ This is **not** a binary max-heap.



Representation

- ▶ One representation: nodes are objects with pointers to children.
- ▶ But due to completeness property, we can store a binary heap in a (dynamic) array.

Array Representation



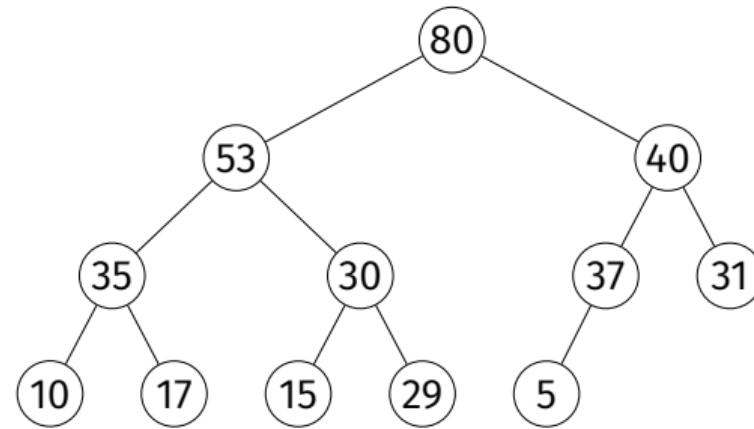
- ▶ `.left_child(i)`
- ▶ `.right_child(i)`
- ▶ `.parent(i)`



Operations

- ▶ `.max()`
 - ▶ Return (but do not remove) the max key
- ▶ `.increase_key(i, new_key)`
 - ▶ Increase key of node i , maintaining heap
- ▶ `.insert(key)`
 - ▶ Insert new node, maintaining heap
- ▶ `.pop_max()`
 - ▶ Remove max-key node, return key

.max



| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 80 | 53 | 40 | 35 | 30 | 37 | 31 | 10 | 17 | 15 | 29 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

.max

```
class MaxHeap:

    def __init__(self, keys=None):
        if keys is None:
            keys = []
        self.keys = keys

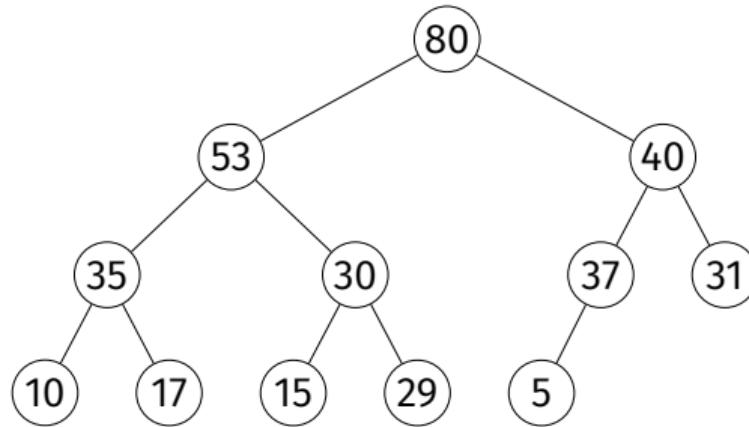
    def max(self):
        return self.keys[0]
```

.max

- ▶ Takes $\Theta(1)$ time.

.increase_key

.increase_key(9, key=60)



| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 80 | 53 | 40 | 35 | 30 | 37 | 31 | 10 | 17 | 15 | 29 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

.increase_key

```
def increase_key(self, ix, key):
    if key < self.keys[ix]:
        raise ValueError('New key is smaller.')

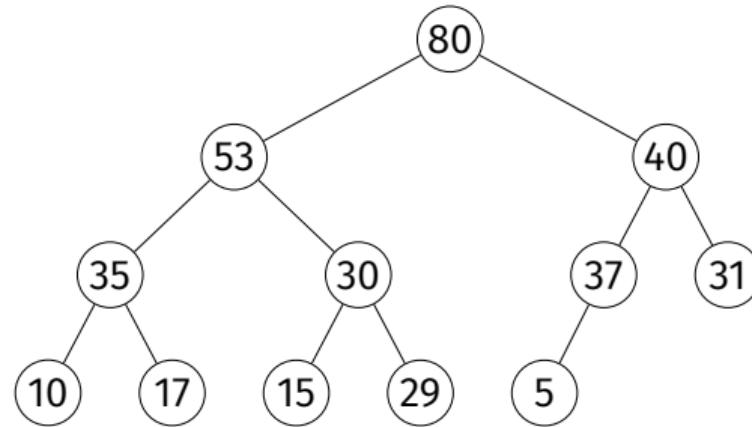
    self.keys[ix] = key
    while (
            parent(ix) >= 0
            and
            self.keys[parent(ix)] < key
    ):
        self._swap(ix, parent(ix))
        ix = parent(ix)
```

.increase_key

- ▶ Takes $O(\log n)$ time.

.insert

.insert(key=60)



| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 80 | 53 | 40 | 35 | 30 | 37 | 31 | 10 | 17 | 15 | 29 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

.insert

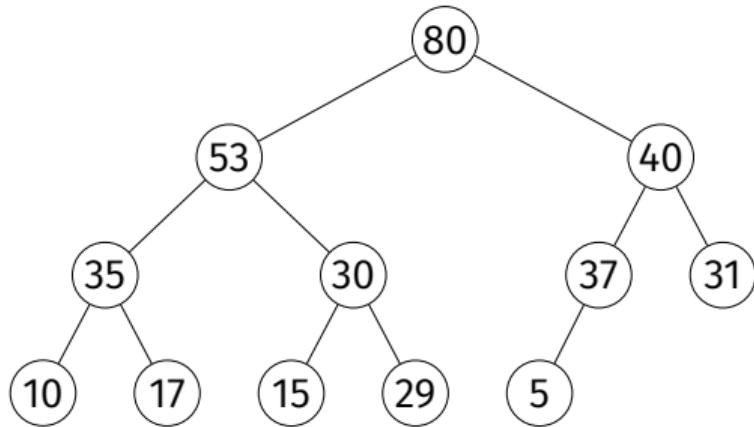
```
def insert(self, key):
    self.keys.append(key)
    self.increase_key(
        len(self.keys)-1, key
    )
```

.insert

- ▶ Takes $O(\log n)$ time (amortized)³.

³If we use a static array the worst case is $\Theta(\log n)$

.pop_max_key



| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 80 | 53 | 40 | 35 | 30 | 37 | 31 | 10 | 17 | 15 | 29 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

.pop_max_key

```
def pop_max_key(self):
    if len(self.keys) == 0:
        raise IndexError('Heap is empty.')
    highest = self.max()
    self.keys[0] = self.keys[-1]
    self.keys.pop()
    self._push_down(0)
    return highest
```

• `_push_down(i)`

- ▶ Assume that left and right subtrees of node i are max heaps, but key of i is possibly too small.
- ▶ Push it down until heap property satisfied.
 - ▶ Recursively swap with largest of left and right child.

• _push_down()

```
def _push_down(self, i):
    left = left_child(i)
    right = right_child(i)
    if (
        left < len(self.keys)
        and
        self.keys[left] > self.keys[i]
    ):
        largest = left
    else:
        largest = i

    if (
        right < len(self.keys)
        and
        self.keys[right] > self.keys[largest]
    ):
        largest = right

    if largest != i:
        self._swap(i, largest)
        self._push_down(largest)
```

- `.pop_max_key`

- ▶ `._push_down(i)` takes $O(h)$ where h is i 's height
- ▶ Since $h = O(\log n)$, `.pop_max_key` takes $O(\log n)$ time.

Summary

For a binary heap⁴:

| | |
|---------------|--------------------|
| .max | $\Theta(1)$ |
| .increase_key | $O(\log n)$ |
| .insert | $O(\log n)$ |
| .pop_max_key | $O(h) = O(\log n)$ |

⁴There are other heap data structures. Fibonacci heaps have $\Theta(1)$ insert and increase key, but slower for small n .

Implementing Priority Queues

- ▶ Can use max heaps to implement priority queues.
- ▶ But a priority queue has values *and* keys.

```
pq.insert('heart attack', priority=20)
```

Trick

- ▶ Heap keys need not be integers.
- ▶ Need only be comparable.
- ▶ Can store key and value with a **tuple**.

Tuple Comparison

- ▶ In Python, tuple comparison is lexicographical.
 - ▶ Compare first entry; if tie, compare second, etc.

```
»> (10, 'test') > (5, 'zzz')
```

True

```
»> (10, 'test') > (10, 'zzz')
```

False

Trick

- ▶ Use 2-tuples: priority in 1st spot, value in 2nd.

```
class PriorityQueue:

    def __init__(self):
        self._heap = MaxHeap()

    def insert(self, value, priority):
        self._heap.insert((priority, value))

    def pop_highest_priority(self):
        return self._heap.pop_max()

    def max(self):
        return self._heap.max()

    def is_empty(self):
        return not bool(self._heap.keys)
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 4

Example: Online Median

Online Median

- ▶ **Given:** a stream of numbers, one at a time.
- ▶ **Compute:** the median of all numbers seen so far.
- ▶ **Design:** a data structure with the following operations:
 - ▶ `.insert(number)`: in $\Theta(\log n)$ time
 - ▶ `.median()`: in $\Theta(1)$ time

Review

- ▶ Given an array, we can compute the median in:
 - ▶ $\Theta(n \log n)$ time by sorting
 - ▶ $\Theta(n)$ (expected) time with quickselect
- ▶ But modifying the array and repeating is costly.

Idea

- ▶ Median is the:
 - ▶ **maximum** of the smallest $\approx n/2$ numbers.
 - ▶ **minimum** of the largest $\approx n/2$ numbers.
- ▶ Keep a max heap for the smallest half.
- ▶ Keep a min heap for the largest half.
- ▶ May become unbalanced.
 - ▶ Move elements between them to balance.

Example

- ▶ Given 5, 1, 9, 8, 10, 7, 3, 6, 2, 4

Analysis

- ▶ Given a stream of n numbers, compute median, insert another, compute median

quickselect (dyn. arr.)

- ▶ $\Theta(n)$ time for n appends
- ▶ $\Theta(n)$ time for quickselect
- ▶ $\Theta(1)$ time for 1 append
- ▶ $\Theta(n)$ time for quickselect

now (double heap)

- ▶ $\Theta(n \log n)$ time for n inserts
- ▶ $\Theta(1)$ time for median
- ▶ $\Theta(\log n)$ time for 1 insert
- ▶ $\Theta(1)$ time for quickselect

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 4 | Part 1

Dynamic Sets and Hashing

Dynamic Set

- ▶ One of the most useful abstract data types.
- ▶ A collection of unique keys which supports:
 - ▶ insertion and deletion
 - ▶ membership queries: $x \text{ in set}$
- ▶ Very similar to **dictionary**.

Implementation #1

- ▶ Store n elements in a dynamic array.
- ▶ Initial cost: $\Theta(n)$.
- ▶ Query: linear search, $O(n)$.
- ▶ Insertion: $\Theta(1)$ amortized.

Implementation #2

- ▶ Store n elements in a **sorted** dynamic array.
- ▶ Initial cost: $O(n \log n)$.
- ▶ Query: binary search, $\Theta(\log n)$.
- ▶ Insertion: $O(n)$
 - ▶ Must maintain sorted order, involves copies.

Better Implementation

- ▶ Store n elements in a **hash table**.
- ▶ Initial cost: $\Theta(n)$ ¹.
- ▶ Query: $\Theta(1)$.
- ▶ Insertion: $\Theta(1)$.

¹All time complexities are average case.

Today's Lecture

- ▶ We'll review hashing.
- ▶ See where hashing is **not** the right thing to do.
- ▶ Review binary search trees as an alternative.
- ▶ Next lecture: introduce **treaps**.

Hashing

- ▶ One of the most important ideas in CS.
- ▶ Tons of uses:
 - ▶ Verifying message integrity.
 - ▶ Fast queries on a large data set.
 - ▶ Identify if file has changed in version control.

Hash Function

- ▶ A **hash function** takes a (large) object and returns a (smaller) “fingerprint” of that object.

How?

- ▶ Looking at certain bits, combining them in ways that look random.

Hash Function Properties

- ▶ Hashing same thing twice returns the same hash.
- ▶ Unlikely that different things have same fingerprint.
 - ▶ But not impossible!

Example

- ▶ MD5 is a **cryptographic** hash function.
 - ▶ Hard to “reverse engineer” input from hash.
- ▶ Returns a *really large* number in hex.

a741d8524a853cf83ca21eabf8cea190

- ▶ Used to “fingerprint” whole files.

Example

```
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin" | md5  
a741d8524a853cf83ca21eabf8cea190  
> echo "My name is Justin!" | md5  
f11eed2391bbdoa5a2355397c089fafd
```

Another Use

- ▶ Want to place images into 100 bins.
- ▶ How do we decide which bin an image goes into?
- ▶ Hash function!
 - ▶ Takes in an image.
 - ▶ Outputs a number in $\{1, 2, \dots, 100\}$.

Hashing for Data Scientists

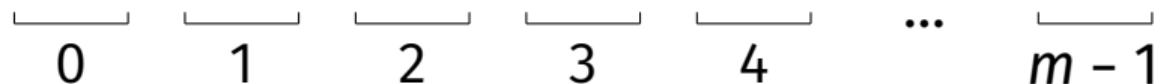
- ▶ Don't need to know much about *how* hash function works.
- ▶ But should know how they are used.

Hash Tables

- ▶ Create an array with pointers to m linked lists.
 - ▶ Usually $m \approx$ number of things you'll be storing.
- ▶ Create hash function to turn input into a number in $\{0, 1, \dots, m - 1\}$.

Example

```
hash('hello') == 3  
hash('data') == 0  
hash('science') == 4
```



Collisions

- ▶ The **universe** is the set of all possible inputs.
- ▶ This is usually much larger than m (even infinite).
- ▶ Not possible to assign each input to a unique bin.
- ▶ If `hash(a) == hash(b)`, there is a **collision**.

Chaining

- ▶ Collisions stored in same bin, in linked list.
- ▶ **Query:** Hash to find bin, then linear search.



The Idea

- ▶ A good hash function will utilize all bins evenly.
 - ▶ Looks like uniform random distribution.
- ▶ If $m \approx n$, then only a few elements in each bin.
- ▶ As we add more elements, we need to add bins.

Average Case

- ▶ n elements in bin.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins².
- ▶ Expected bin size: n/m .

²Of course, they are placed deterministically.

Analysis

- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n/m)$ for linear search.
 - ▶ Total: $\Theta(1 + n/m)$.
 - ▶ We usually guarantee $m = O(n)$, $\implies \Theta(1)$.
- ▶ Insertion: $\Theta(1)$.

Worst Case

- ▶ Everything hashed to same bin.
 - ▶ Really unlikely!
 - ▶ Adversarial attack?
- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n)$ for linear search.
 - ▶ Total: $\Theta(n)$.

Worst Case Insertion

- ▶ We need to ensure that $m \leq c \cdot n$.
 - ▶ Otherwise, too many collisions.
- ▶ If we add a bunch of elements, we'll need to increase m .
- ▶ Increasing m means allocating a new array, $\Theta(m) = \Theta(n)$ time.

Main Idea

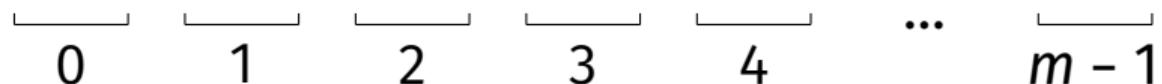
Hash tables support constant (expected) time insertion and membership queries.

Hashing Downsides

- ▶ Hashing is like magic. Constant time access?!
- ▶ Comes at a cost: data now scattered “randomly”.
- ▶ Examples:
 - ▶ find max/min in hash table.
 - ▶ range query: all strings between 'a' and 'c'
- ▶ Must do a full loop over table!

Example

```
hash('apple') == 3  
hash('bill nye') == 0  
hash('cassowary') == 4
```



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 4 | Part 2

Binary Search Trees

Binary Search Trees

- ▶ An alternative way to implement dynamic sets.
- ▶ Slightly slower insertion, query.
- ▶ But preserves data in sorted order.

Binary Search Tree

- ▶ A **binary search tree** (BST) is a binary tree that satisfies the following for any node x :
- ▶ if y is in x 's **left** subtree:

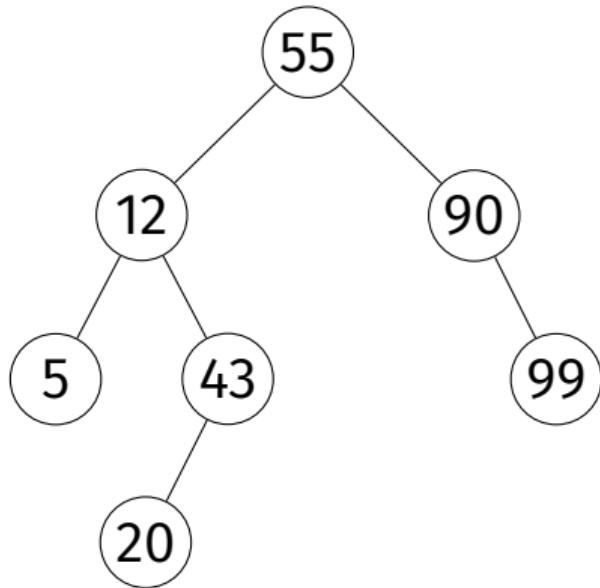
$$y.\text{key} \leq x.\text{key}$$

- ▶ if y is in x 's **right** subtree:

$$y.\text{key} \geq x.\text{key}$$

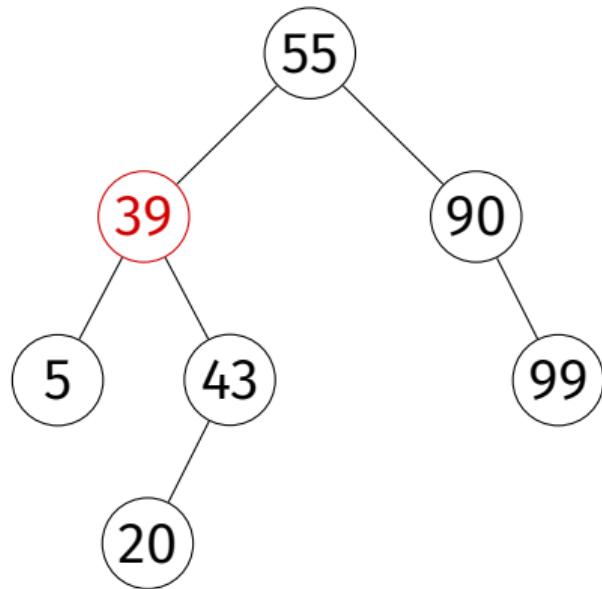
Example

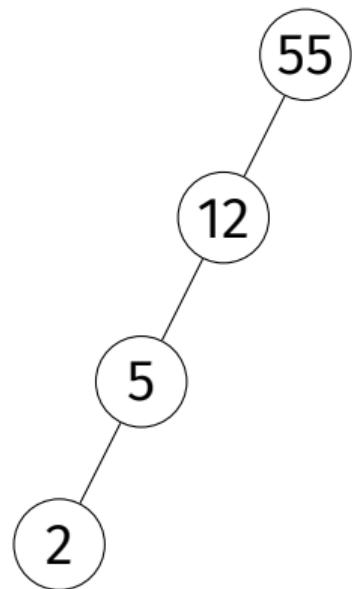
- ▶ This **is** a BST.



Example

- ▶ This is **not** a BST.



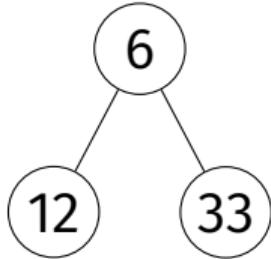


Exercise

Is this is a BST?

Memory Representation

- ▶ Each element stored as a **node** at an arbitrary address in memory.
- ▶ Each node has a **key**³ and pointers to **left child**, **right child**, and **parent** nodes (if they exist).

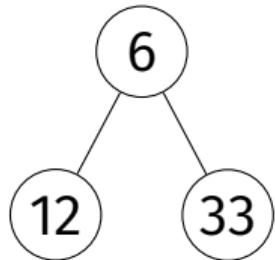


³We'll assume keys are unique, though this can be relaxed.

In Python

```
class Node:  
    def __init__(self, key, parent=None):  
        self.key = key  
        self.parent = parent  
        self.left = None  
        self.right = None  
  
class BinarySearchTree:  
    def __init__(self, root: Node):  
        self.root = root
```

In Python

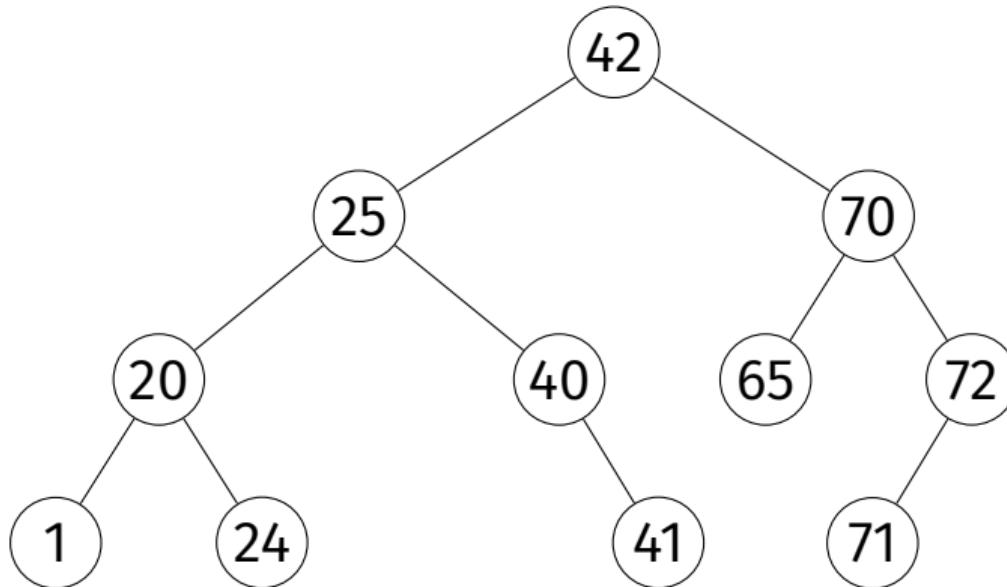


```
root = Node(6)
n1 = Node(12, parent=root)
root.left = n1
n2 = Node(33, parent=root)
root.right = n2
tree = BinarySearchTree(root)
```

Operations on BSTs

- ▶ We will want to:
 - ▶ **traverse** the nodes in sorted order by key
 - ▶ **query** a key (is it in the tree?)
 - ▶ **insert** a new key
 - ▶ **delete** an existing key

Inorder Traversal



```
def inorder(node):
    if node is not None:
        inorder(node.left)
        print(node.key)
        inorder(node.right)
```

Inorder Traversal

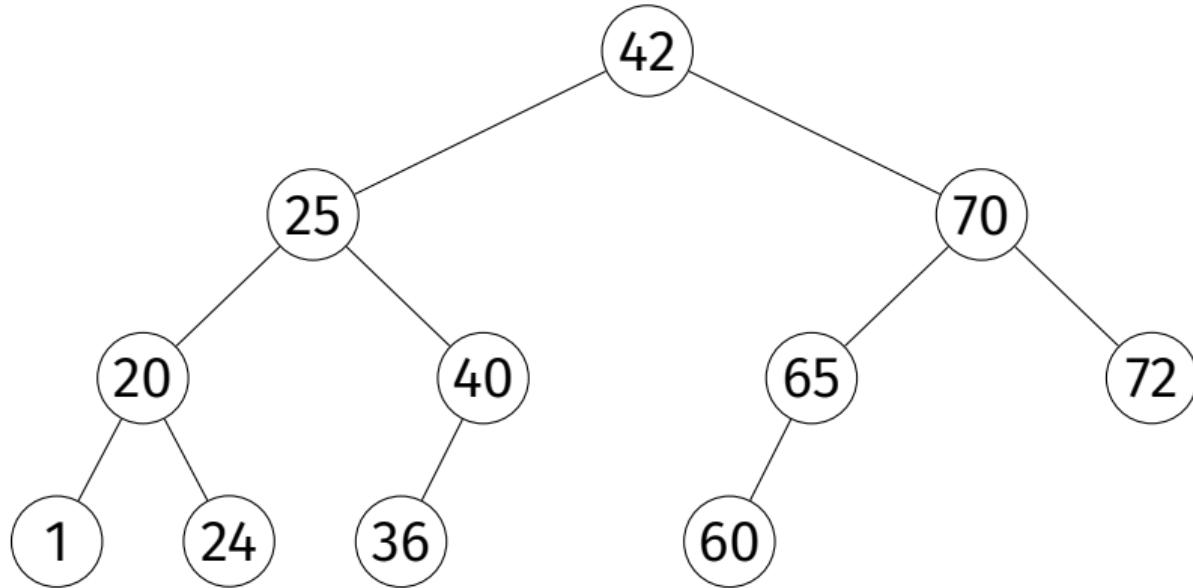
- ▶ Prints nodes in sorted order.
- ▶ Visits each node once, $\Theta(1)$ time in the call.
- ▶ Takes $\Theta(n)$ time.

Queries

- ▶ **Given:** a BST and a target, t .
- ▶ **Return:** **True** or **False**, is the target in the collection?

Queries

- ▶ Is 36 in the tree? 65? 23?



Queries

- ▶ Start walking from root.
- ▶ If current node is:
 - ▶ equal to target, return **True**;
 - ▶ too large ($>$ target), follow left edge;
 - ▶ too small ($<$ target), follow right edge;
 - ▶ **None**, return **False**

Queries, in Python

```
def query(self, target):
    current_node = self.root
    while current_node is not None:
        if current_node.key == target:
            return current_node
        elif current_node.key < target:
            current_node = current_node.right
        else:
            current_node = current_node.left
    return None
```

Queries, Analyzed

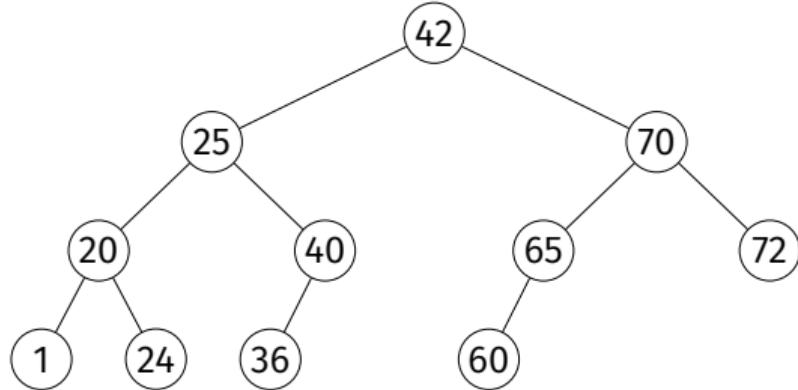
- ▶ Best case: $\Theta(1)$.
- ▶ Worst case: $\Theta(h)$, where h is **height** of tree.

Insertion

- ▶ **Given:** a BST and a new key, k .
- ▶ **Modify:** the BST, inserting k .
- ▶ Must **Maintain** the BST properties.

Insertion

- ▶ Insert 23 into the BST.



```
def insert(self, new_key):
    # assume new_key is unique
    current_node = self.root
    parent = None

    while current_node is not None:
        parent = current_node
        if current_node.key == new_key:
            raise ValueError(f'Duplicate key "{new_key}" not allowed.')
        if current_node.key < new_key:
            current_node = current_node.right
        elif current_node.key > new_key:
            current_node = current_node.left

    new_node = Node(key=new_key, parent=parent)
    if parent is None:
        self.root = new_node
    elif parent.key < new_key:
        parent.right = new_node
    else:
        parent.left = new_node
```

Insertion, Analyzed

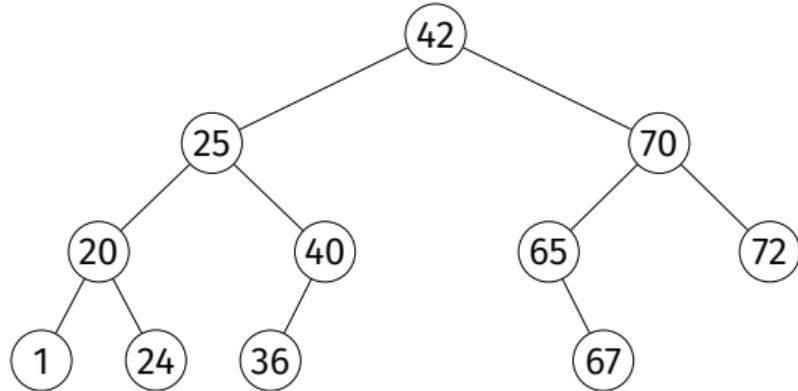
- ▶ Worst case: $\Theta(h)$, where h is **height** of tree.

Deletion

- ▶ **Given:** a key in the BST.
- ▶ **Modify:** the BST, deleting the key.
- ▶ Must **maintain** the BST properties.
- ▶ This is a little trickier.

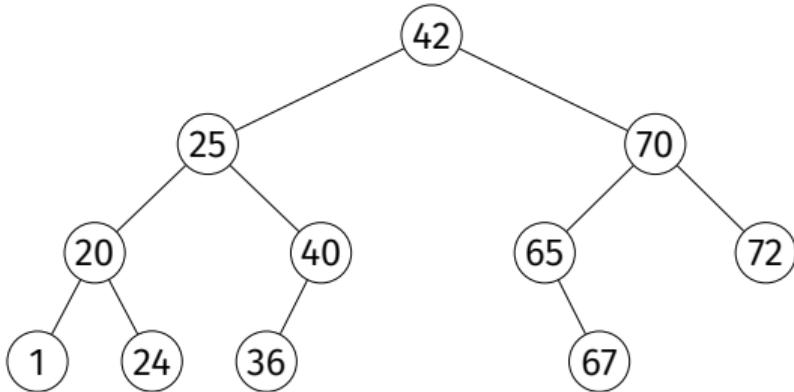
Deletion: Case 1 (Easy)

- ▶ Delete 36 from the BST.



Deletion: Case 2 (Tricky)

- ▶ Delete 42 from the BST.

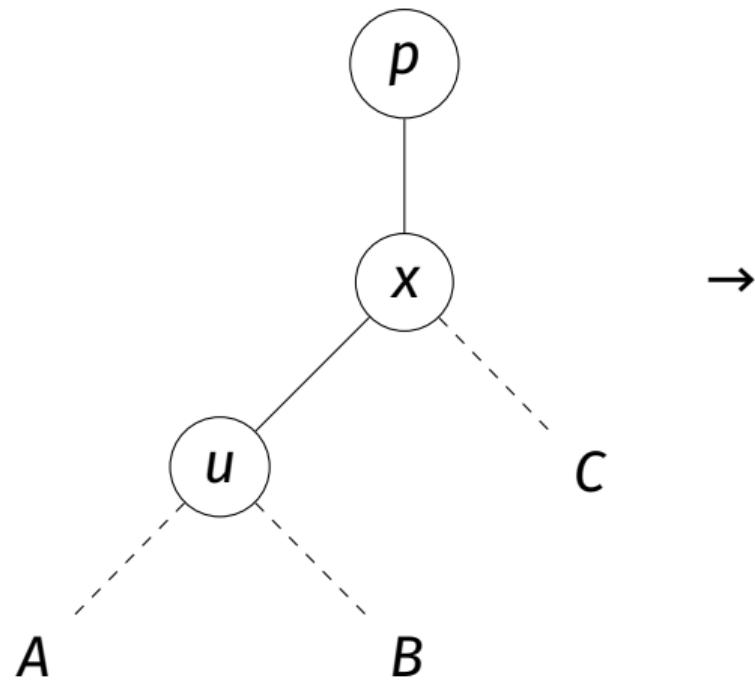


Deletion

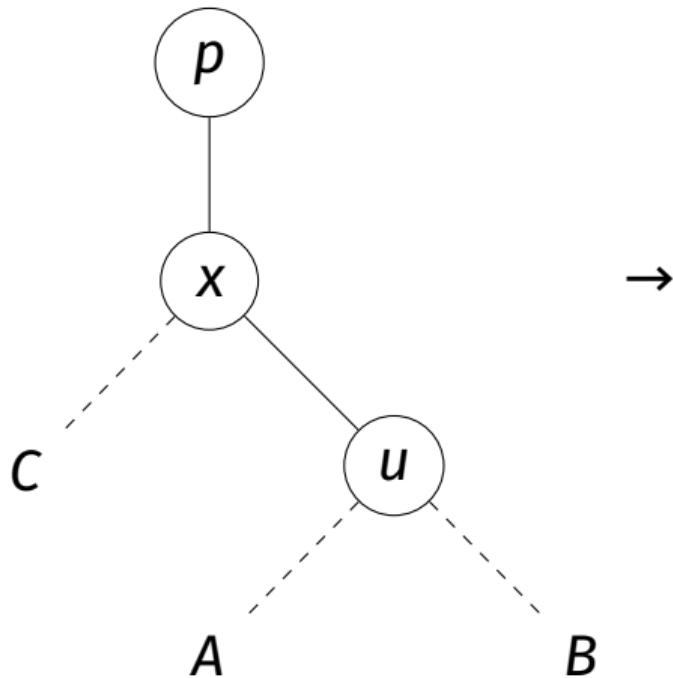
- ▶ If node has no children (leaf), **easy**.
- ▶ Otherwise, a little trickier.
- ▶ Idea: **rotate**⁴ node to bottom, preserving BST.
When it is a leaf, delete.

⁴Most books take a different approach with the same time complexity.

(Right) Rotation



(Left) Rotation



Claim

Left rotate and right rotate preserve the BST property.

```
def _right_rotate(self, x):
    u = x.left
    B = u.right
    C = x.right
    p = x.parent

    x.left = B
    if B is not None: B.parent = x

    u.right = x
    x.parent = u

    u.parent = p

    if p is None:
        self.root = u
    elif p.left is x:
        p.left = u
    else:
        p.right = u
```

Deletion Analyzed

- ▶ Each rotate takes $\Theta(1)$ time.
- ▶ $O(h)$ rotations until node becomes leaf.
- ▶ So $\Theta(h)$ time in the worst case.

Main Idea

Insertion, deletion, and querying all take $\Theta(h)$ time in the worst case, where h is the height of the tree.

DSC 190

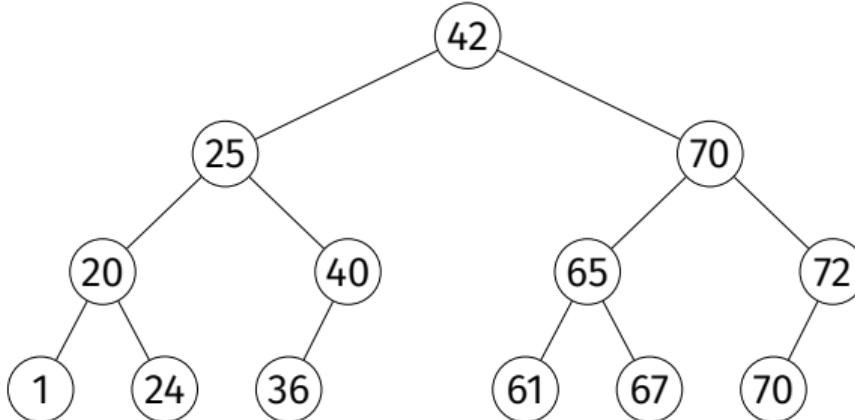
DATA STRUCTURES & ALGORITHMS

Lecture 4 | Part 3

Balanced and Unbalanced BSTs

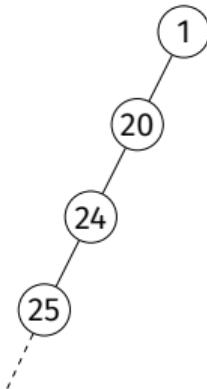
Binary Tree Height

- ▶ In case of very balanced tree, h grows **logarithmically** with n .
 - ▶ $h = \Theta(\log n)$
 - ▶ Query, insertion, deletion take worst case $\Theta(\log n)$ time.



Binary Tree Height

- ▶ In the case of very unbalanced tree, h grows **linearly** with n .
 - ▶ $h = \Theta(n)$
 - ▶ Query, insertion, deletion take worst case $\Theta(n)$ time.



Unbalanced Trees

- ▶ Occurs if we insert items in (close to) sorted or reverse sorted order.
- ▶ This is a **common** situation.

Example

- ▶ Insert 1, 2, 3, 4, 5, 6, 7, 8 (in that order).

Time Complexities

| | |
|-----------|-------------|
| query | $\Theta(h)$ |
| insertion | $\Theta(h)$ |

Where h is height, and $h = \Omega(\log n)$ and $h = O(n)$.

Time Complexities (Balanced)

| | |
|-----------|-------------|
| query | $O(\log n)$ |
| insertion | $O(\log n)$ |

Where h is height, and $h = \Omega(\log n)$ and $h = O(n)$.

Worst Case Time Complexities (Unbalanced)

| | |
|-----------|-------------|
| query | $\Theta(n)$ |
| insertion | $\Theta(n)$ |

- ▶ The worst case is **bad**.
 - ▶ Worse than using a sorted array!
- ▶ The worst case is **not rare**.

Main Idea

The operations take linear time in the worst case **unless** we can somehow ensure that the tree is **balanced**.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 4 | Part 4

Range Queries, Max, and Min

Why use a BST?

- ▶ Even assuming a balanced tree, BSTs seem worse than hash tables.

| | BST | Hash Table ⁵ |
|-----------|-------------|-------------------------|
| query | $O(\log n)$ | $\Theta(1)$ |
| insertion | $O(\log n)$ | $\Theta(1)$ |

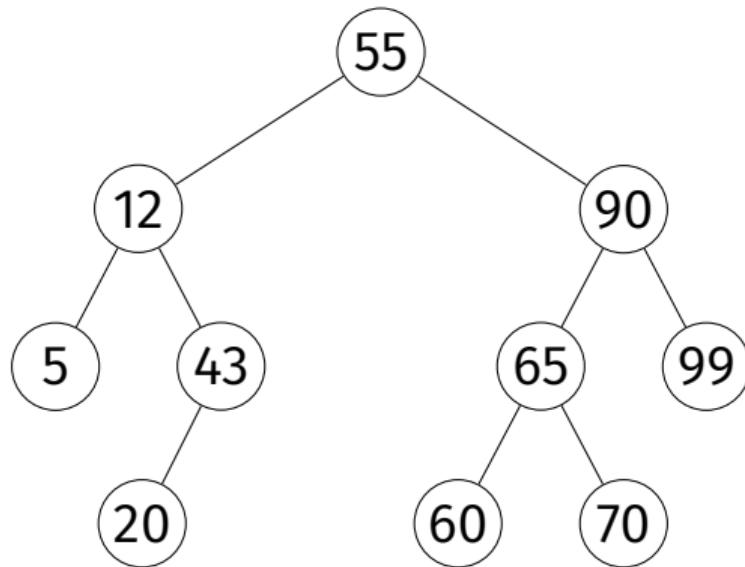
- ▶ So when are BSTs better?

⁵Average case times reported.

Max/Min

- ▶ Consider finding the maximum element.
- ▶ Hash tables: $\Theta(n)$; must loop through all bins.
- ▶ BST: $\Theta(h)$, which is $O(\log n)$ if balanced

Example



Main Idea

Keeping track of the maximum can be done efficiently in any stream of numbers, provided that there are only **insertions**. But if **deletions** are allowed, BSTs can find the *next* maximum efficiently.

Exercise

How well do heaps work for this problem? Are they better? In what sense?

Range Queries

- ▶ **Given:** a collection and an interval $[a, b]$
- ▶ **Retrieve:** all elements in the interval.
- ▶ **Example:**
 - ▶ collection: 55, 12, 5, 43, 20, 90, 65, 99, 60, 70
 - ▶ interval: [1, 30]
 - ▶ result: 5, 12, 20

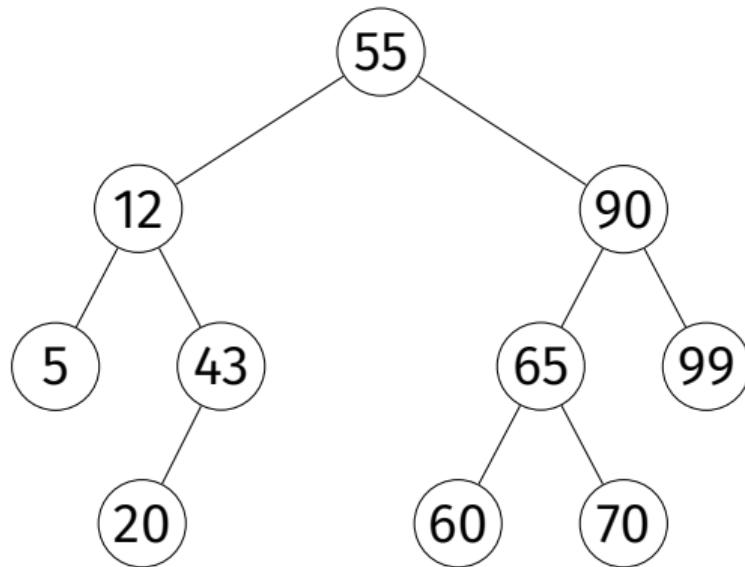
Exercise

How quickly can this be performed with a hash table?

Range Queries in BST

- ▶ Definitions:
 - ▶ The **ceiling** of x in a BST is the smallest key $\geq x$.
 - ▶ The **successor** of node u is the smallest node $> x$.
- ▶ Strategy:
 - ▶ Find the **floor** of a
 - ▶ Repeatedly find the **successor** until $> b$

Example



Range Queries

- ▶ **ceiling** and **successor** both take $O(h) = O(\log n)$ in balanced trees
- ▶ If there are k elements in the range, calling successor k times gives complexity $O(k \log n)$.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 1

Today's Lecture

Last Time

- ▶ Time needed for BST operations is proportional to height.
- ▶ If tree is balanced, $h = \Theta(\log n)$
- ▶ If tree is unbalanced, $h = O(n)$

Today

- ▶ How do we ensure that tree is balanced?
- ▶ Approach 1: Complicated rules, red-black trees.
- ▶ Approach 2: Randomization
- ▶ We'll introduce **treaps**.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 2

Red-Black Trees

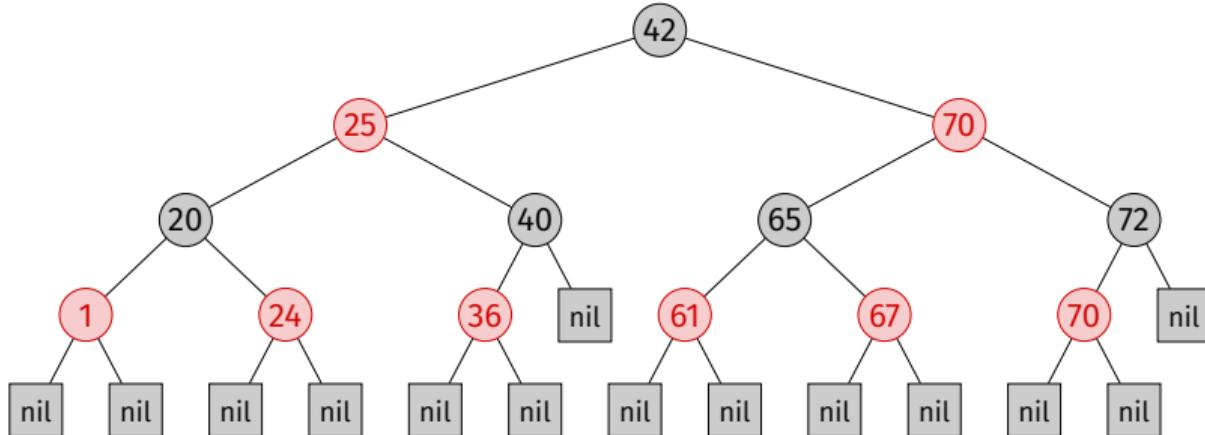
Self-Balancing BSTs

- ▶ We wish to ensure that the tree does not become unbalanced.
- ▶ Idea: If tree becoming unbalanced, it will balance itself.
- ▶ Several strategies, including **red-black** trees and **AVL** trees

Red-Black Trees

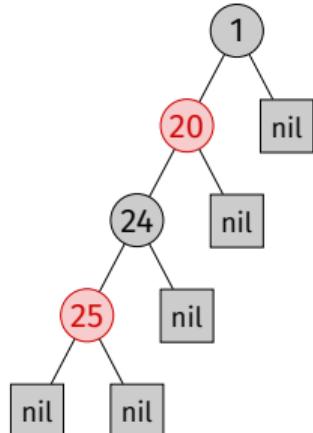
- ▶ A **red-black** tree is a BST whose nodes are colored **red** and **black**.
- ▶ Leaf nodes are “nil”.
- ▶ Must satisfy four additional properties:
 1. The root node is **black**.
 2. Every leaf node is **black**.
 3. If a node is **red**, both child nodes are **black**.
 4. For any node, all paths from the node to a leaf contain the same number of **black** nodes.

Example



Example

- ▶ This **not** a red-black tree.
 - ▶ Violates last property



Claim

If a red-black tree has n internal (non-nil) nodes, then the height is at most $2 \log(n + 1)$.

Proof Intuition¹

- ▶ All paths from root to a leaf are about the same length ($\approx h$).
- ▶ Therefore, the tree is close to balanced.
- ▶ So height is proportional to $\log n$

¹Formal proof proceeds by induction.

Non-Modifying Operations

- ▶ As a result, the non-modifying operations take $\Theta(\log n)$ time in red-black trees.
 - ▶ query
 - ▶ minimum/maximum
 - ▶ next smallest/largest
- ▶ Proof: these take $\Theta(h)$ time in any BST, and in a red-black tree $h = O(\log n)$.

Insertion and Deletion

- ▶ Standard BST `.insert` and `.delete` methods preserve BST, but **not** red-black properties.
- ▶ Insertion/deletion in a red-black tree is considerably more **complicated**.
- ▶ But both take $\Theta(\log n)$ time.

Implementing balanced trees is an exacting task and as a result balanced tree algorithms are rarely implemented except as part of a programming assignment in a data structures class².

Pugh, 1990

²For computer science majors.

Summary

- ▶ For red-black trees, worst cases:

| | |
|-----------------------|------------------|
| query | $\Theta(\log n)$ |
| minimum/maximum | $\Theta(\log n)$ |
| next largest/smallest | $\Theta(\log n)$ |
| insertion | $\Theta(\log n)$ |

- ▶ But they are **tricky** to implement.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 3

Randomization to the Rescue

Order Matters

- ▶ The structure of a BST depends on insertion order.

Example

- ▶ Insert 1,2,3,4,5,6 into BST, in that order.

Example

- ▶ Insert 3, 5, 1, 2, 4, 6 into BST, in that order.

Claim

The expected height of a BST built by inserting the keys in random order is $\Theta(\log n)$.

Idea

- ▶ To build a BST, take all n keys, shuffle them randomly, then insert.
- ▶ No need for Red-Black Trees, right?

Problem

- ▶ Usually don't have all the keys right now.
- ▶ This is a **dynamic set**, after all.
- ▶ The keys come to us in a stream, can't specify order.

Goal

- ▶ Design a data structure that **simulates** random insertion order without actually changing the order.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 4

Treaps

Randomization

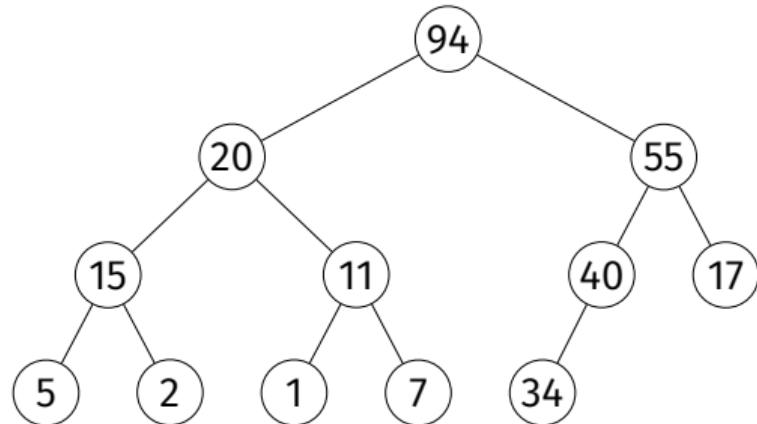
- ▶ If insertions are in a random order, expected depth of a BST is $\Theta(\log n)$.
- ▶ But in **online** operation, we cannot randomize insertion order.
- ▶ Now: an elegant data structure simulating random insertion order in online operation.

First: Recall Heaps

- ▶ A **max heap** is a **binary tree** where:
 - ▶ each node has a priority.
 - ▶ if y is a child of node x , then
$$y.\text{priority} \leq x.\text{priority}$$

Example

- ▶ This is a max heap:

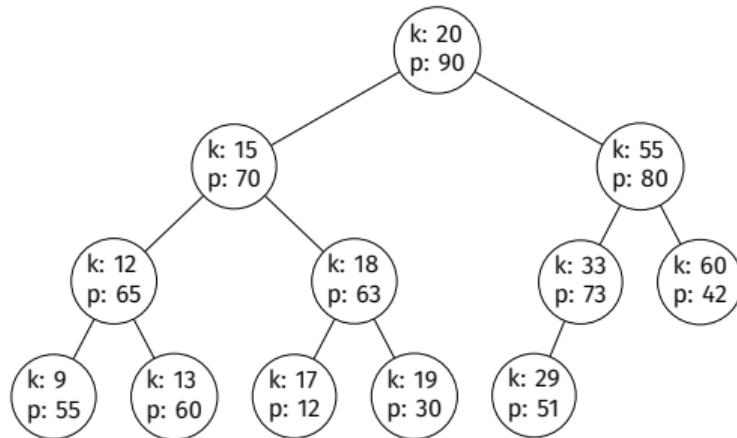


Treaps

- ▶ A **treap** is a binary tree in which each node has both a **key** and a **priority**.
- ▶ It is a **max heap** w.r.t. its priorities.
- ▶ It is a **binary search tree** w.r.t. its keys.

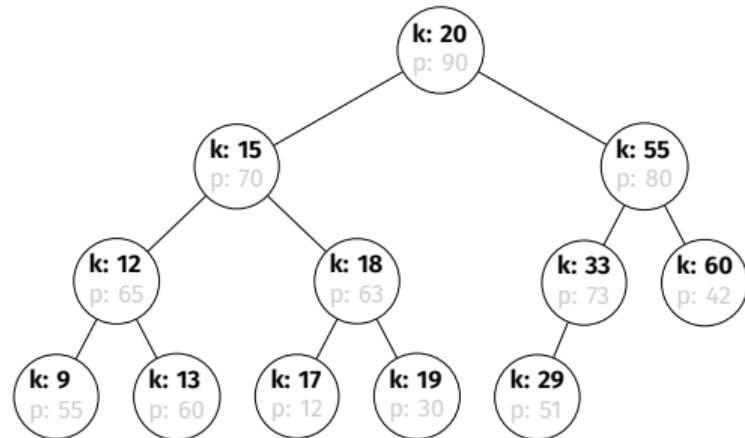
Example

- ▶ This is a treap:



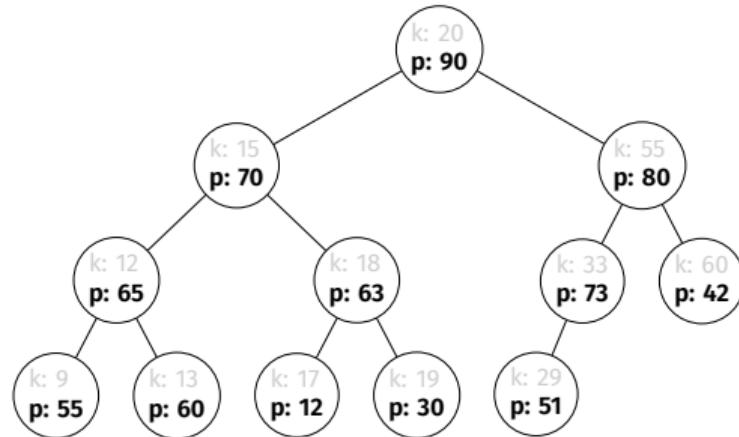
Example

- ▶ This is a treap:



Example

- ▶ This is a treap:



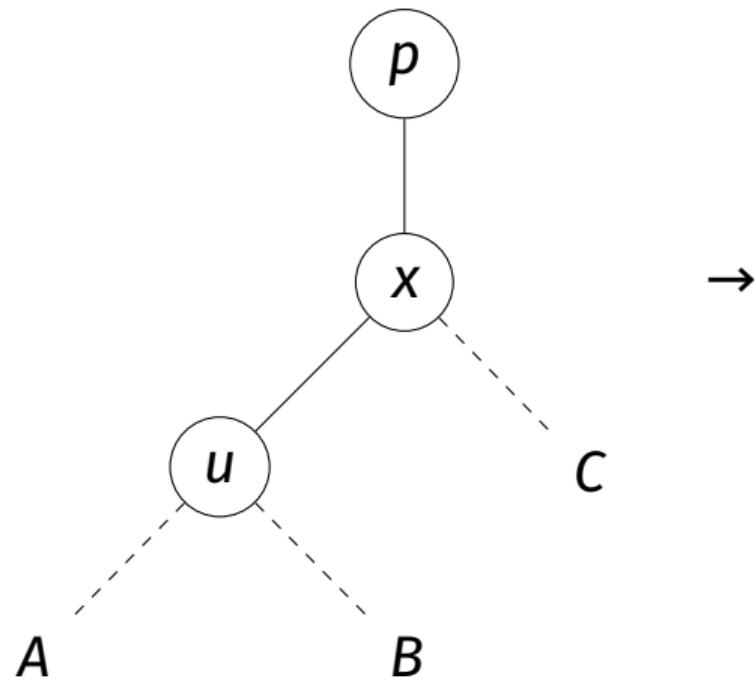
BST Operations

- ▶ Because a treap is a BST, querying, finding max/min by key is done the same.
- ▶ Insertion and deletion require care to preserve **heap** property.

Insertion

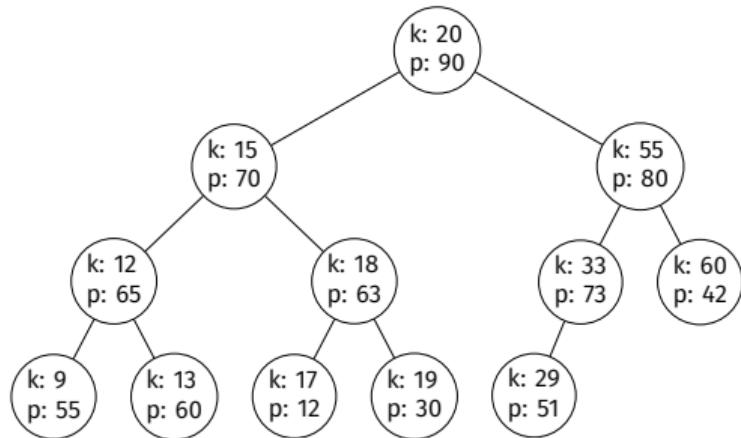
- ▶ Find place to insert node as usual.
- ▶ While priority of new node is $>$ than parent's:
 - ▶ Left rotate new node if it is the right child.
 - ▶ Right rotate new node if it is the left child.
- ▶ Rotate preserves BST, repeat until heap property satisfied.

(Right) Rotation



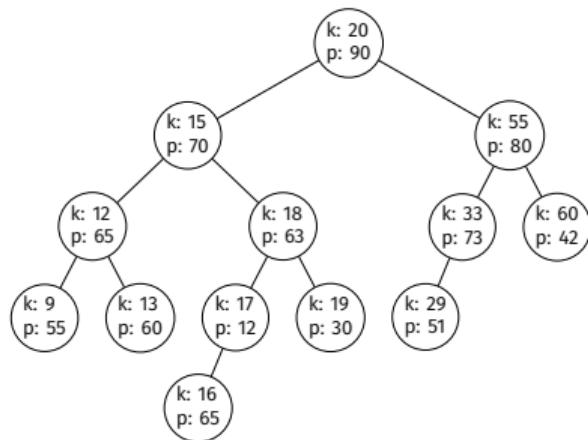
Example: Insertion

- ▶ Insert key: 16, priority: 65.



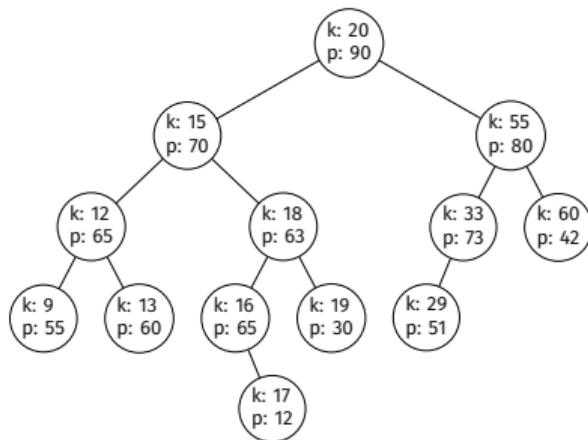
Example: Insertion

- ▶ Insert key: 16, priority: 65.



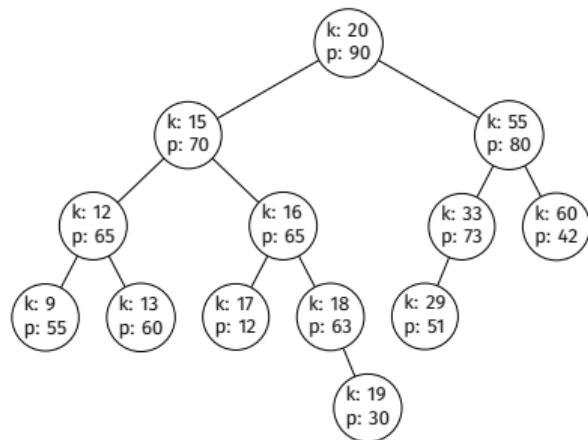
Example: Insertion

- ▶ Insert key: 16, priority: 65.



Example: Insertion

- ▶ Insert key: 16, priority: 65.



Deletion

- ▶ While node is not a leaf:
 - ▶ Rotate it with child of highest priority.
- ▶ Once it is a leaf, delete it.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 5

Treap Properties

Good Question

- ▶ Is it always possible to build a treap?

Claim

Given any set of (key, priority) pairs, inserting them one-by-one into a treap always results in a valid treap (no matter the insertion order).

Proof Idea

- ▶ Start with a treap (possibly empty).
- ▶ Inserting new (key, priority) preserves treap:
 - ▶ **BST**: rotation preserves BST property
 - ▶ **heap**: initially violated, but rotation repeated until it is satisfied

Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then the treap is **unique**.

Claim

Corollary: Given any set of (key, priority) pairs, if both keys and priorities are unique, inserting them one-by-one into a treap results in the same treap, no matter the insertion order.

Example

- ▶ Insert $(3, 40)$, $(1, 20)$, $(10, 50)$, $(6, 30)$, $(5, 100)$, in that order

Example

- ▶ Insert $(5, 100)$, $(10, 50)$, $(3, 40)$, $(6, 30)$, $(1, 20)$, in that order

Proof Idea

- ▶ Root node must be node w/ highest priority.
- ▶ Root's left (right) child must have highest priority among nodes with key $<$ ($>$) root key.
- ▶ Apply recursively.

Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then inserting them one-by-one into a treap (in any order) results in the **same** BST one would obtain by inserting into a BST in decreasing order of priority.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 6

Randomized Binary Search Trees

Claim

Given any set of keys, if they are inserted into a BST in random order, the result is (almost surely) balanced. The expected height is $\Theta(\log n)$.

Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then inserting them one-by-one into a treap (in any order) results in the **same** BST one would obtain by inserting into a BST in decreasing order of priority.

The Idea

- ▶ When inserting a node into a treap, generate priority **randomly**.
- ▶ The resulting treap will be the same tree as a BST built with nodes randomly ordered according to these priorities.
- ▶ It will almost surely be balanced.
- ▶ This is called a **randomized binary search tree**³.

³Sometimes people call these treaps

Warning

- ▶ Randomness does not mean that the result of, for example, a query has some probability of being incorrect.
- ▶ BST operations on treaps are always, 100% correct.
- ▶ The structure is random.

Example

- ▶ Insert 1, 2, 3, 4, 5, 6 into a treap, generating priorities randomly.

Time Complexities

- ▶ For randomized BSTs, expected times:

| | |
|-----------------------|------------------|
| query | $\Theta(\log n)$ |
| minimum/maximum | $\Theta(\log n)$ |
| next largest/smallest | $\Theta(\log n)$ |
| insertion | $\Theta(\log n)$ |
- ▶ Worst case times are $\Theta(n)$, but very rare

Comparison to Red-Black Trees

- ▶ When compared to red-black trees, randomized BSTs are:
 - ▶ same in terms of expected time;
 - ▶ perhaps slightly slower in practice;
 - ▶ **much** easier to implement/modify.
- ▶ Good trade-off for a data scientist!

Priority Hacks

- ▶ Several interesting strategies for generating a new node's priority, beyond simply generating a random number.

Idea #1: Hashing

- ▶ Instead of randomly generating a number, hash the key to get priority.
- ▶ Works, provided hash function looks random.
- ▶ **Careful!** In python, `hash(300) == 300`

Idea #2: “Learning”

- ▶ Idea: Frequently-queried items should be near top of tree.
- ▶ When an item is queried, update its priority:
 $\text{new priority} = \max(\text{old priority}, \text{random number})$

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 7

Order Statistic Trees

Modifying BSTs

- ▶ More than most other data structures, BSTs must be modified to solve unique problems.
- ▶ Red-black trees are a pain to modify.
- ▶ Treaps/randomized BSTs are easy!

Order Statistics

- ▶ Given n numbers, the **k th order statistic** is the k th smallest number in the collection.

Example

[99, 42, -77, -12, 101]

- ▶ 1st order statistic:
- ▶ 2nd order statistic:
- ▶ 4th order statistic:

Exercise

Some special cases of order statistics go by different names. Can you think of some?

Special Cases

- ▶ **Minimum:** 1st order statistic.
- ▶ **Maximum:** n th order statistic.
- ▶ **Median:** $[n/2]$ th order statistic⁴.
- ▶ **p th Percentile:** $\lceil \frac{p}{100} \cdot n \rceil$ th order statistic.

⁴What if n is even?

Computing Order Statistics

- ▶ Quickselect finds any order statistic in linear expected time.
- ▶ This is efficient for a static set.
- ▶ Inefficient if set is dynamic.

Goal

- ▶ Create a dynamic set data structure that supports fast computation of **any** order statistic.

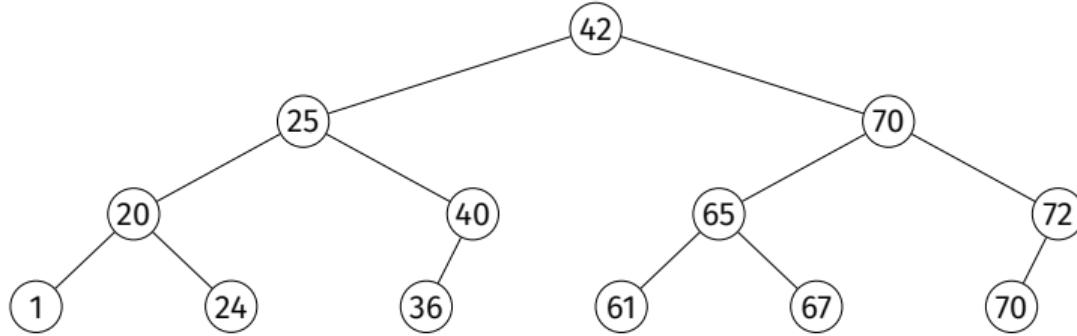
Exercise

Does the “two heaps” trick from before work?

BST Solution

- ▶ For each node, keep attribute `.size`, containing # of nodes in subtree rooted at current node

Example: Insert/Delete



Challenge

- ▶ `.number_lt` changes when nodes are inserted/deleted
- ▶ We must **modify** the code for insertion/deletion
- ▶ A pain with R-B tree; easy with treap!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 8

BSTs vs. Heaps

BSTs vs. Heaps

- ▶ Seemingly similar.
- ▶ Both are binary trees.
- ▶ Similar time complexities.

Summary

| | Balanced BST | Binary Heap |
|-------------------------|-------------------------------|-------------------|
| get minimum/maximum | $\Theta(\log n)$ ⁵ | $\Theta(1)$ |
| extract minimum/maximum | $\Theta(\log n)$ | $\Theta(\log n)$ |
| insertion | $\Theta(\log n)$ | $\Theta(\log(n))$ |

⁵Can actually be optimized to $\Theta(1)$

Comparison

BSTs

- ▶ No cache locality
- ▶ Memory for pointers
- ▶ Maintains sorted order
- ▶ **Used for order statistics, queries**

Heaps

- ▶ Cache locality
- ▶ Use less memory
- ▶ Costly to query
- ▶ **Used for max/min**

DSC 190

DATA STRUCTURES & ALGORITHMS

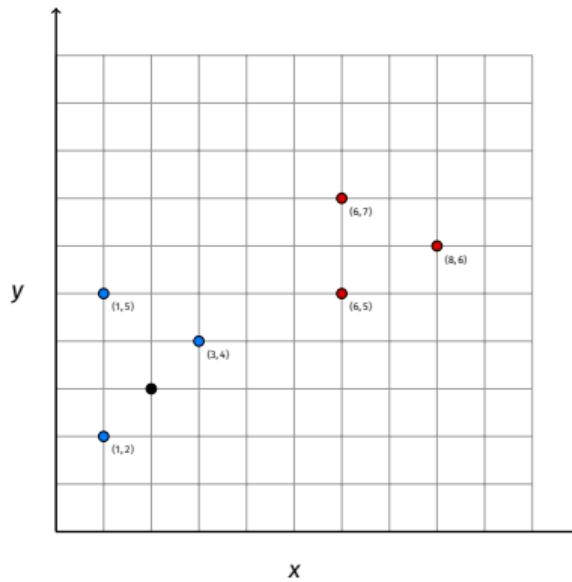
Lecture 6 | Part 1

Today's Lecture

Nearest Neighbors

- ▶ Finding the closest data point to a query point is a common operation.
- ▶ In machine learning, at the core of the **nearest neighbor classifier**.

NN Classifier



NN Query

- ▶ **Given:** a data set X of n points in \mathbb{R}^d and a **query** point, $p \in \mathbb{R}^d$.
- ▶ **Return:** the point in X that is nearest¹ to p

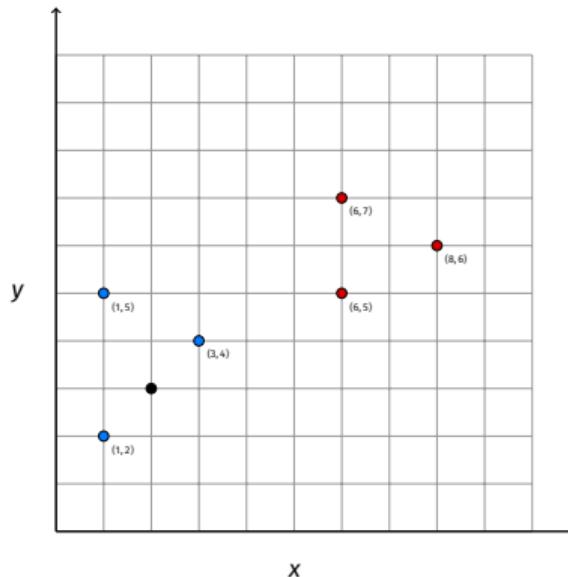
¹In terms of Euclidean distance, though other distances can be considered.

Approach #1: Brute Force

- ▶ Compute distance between p and every point $x \in X$, keep closest.
- ▶ Time: $\Theta(nd)$

Intuitively...

- ▶ ...we can do better. We only need to look at region close to p .



```
def brute_force_nn_search(data, p):
    """Find nearest neighbor.

Parameters
_____
data : np.ndarray
    An  $n \times d$  array of points.
p : np.ndarray
    A  $d$ -array representing the query point.

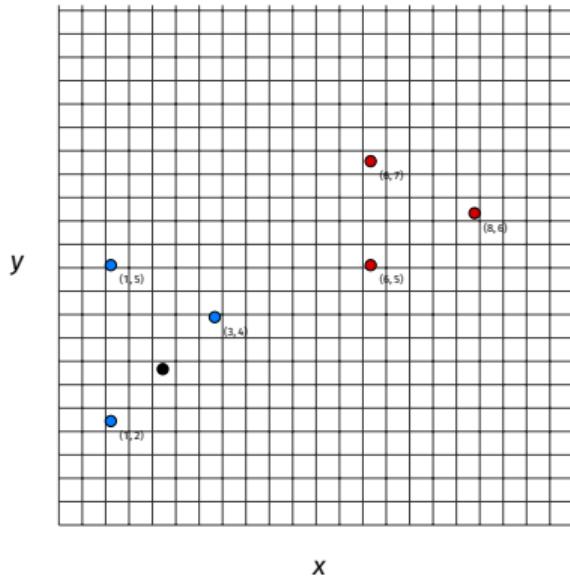
Returns
_____
nn : np.ndarray
    The closest point.
nn_distance : float
    Distance to closest point.

"""
distances = np.sqrt(np.sum((data - p)**2, axis=1))
ix_of_nn = np.argmin(distances)
nn = data[ix_of_nn]
nn_distance = distances[ix_of_nn]
return (nn, nn_distance)
```

Approach #2

- ▶ Build a grid.
- ▶ To query NN, find cell containing p .
- ▶ Start search in p 's cell, move outwards.

Intuitively...



Problems

- ▶ How do we choose grid cell size?
 - ▶ Too big: cells contain a lot of points = brute force.
 - ▶ Too small: Most of the cells are empty.
 - ▶ “Just right” for one region might be too big/small for another region.
- ▶ Number of cells grows **exponentially** with dimension.

Today

- ▶ We'll refine the idea of a grid.
- ▶ Adapt cell placement/size to the data.
- ▶ Result: **k-d trees**.

k-d Trees

- ▶ Will speed up NN queries in low dimensions (<10) from $\Theta(n)$ to $\Theta(\log n)$.
- ▶ But will be just as bad as brute force in high dimensions.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 2

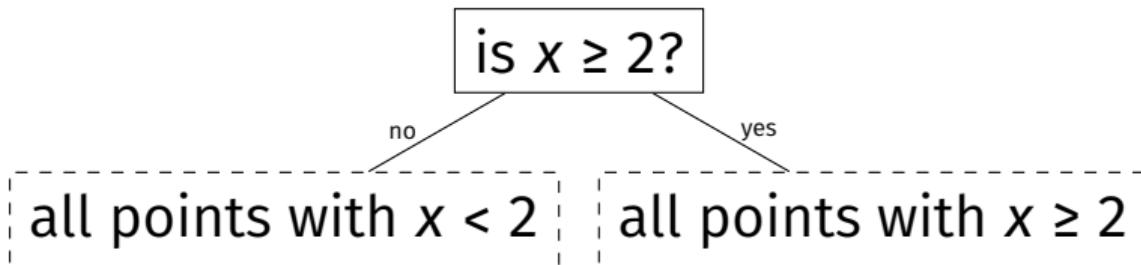
k-d Trees

k-d Trees

- ▶ **Binary search tree** for multidimensional data.
- ▶ Now: structure & properties.
- ▶ Next section: how to query them.
- ▶ Next next section: how to construct them.

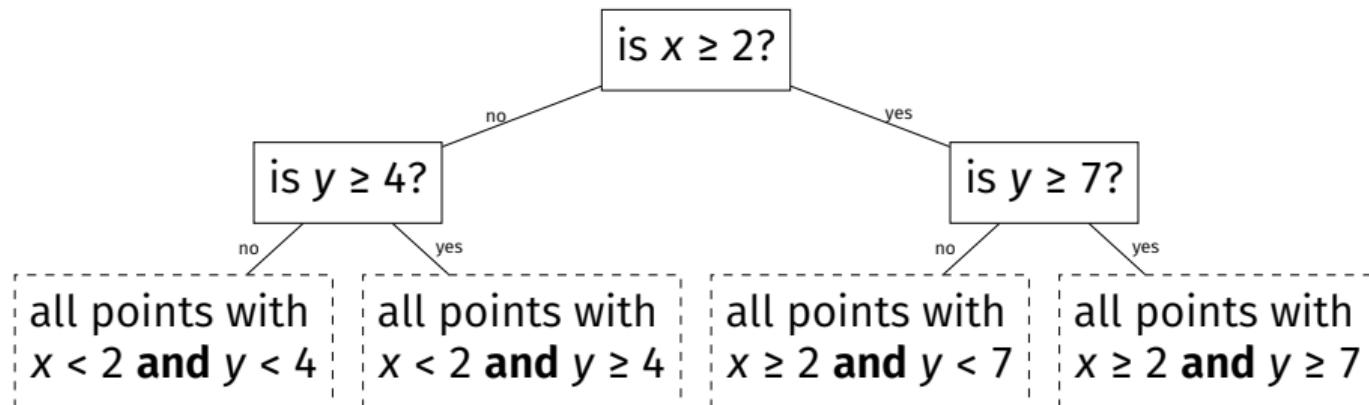
Internal Nodes

- ▶ Internal nodes are **threshold questions**.
 - ▶ can be of form $x \geq 1?$ or $y \geq \tau?$ in 2-d.
 - ▶ can be of form $x \geq \tau?$ or $y \geq \tau?$ or $z \geq \tau?$ in 3-d.
 - ▶ etc.



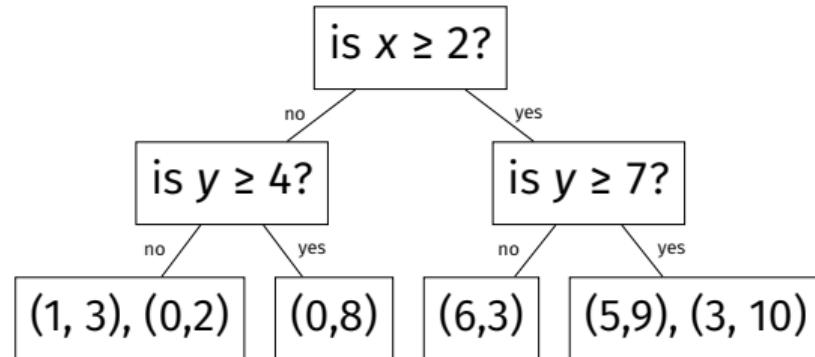
Internal Nodes

- ▶ A path forms a **conjunction**.



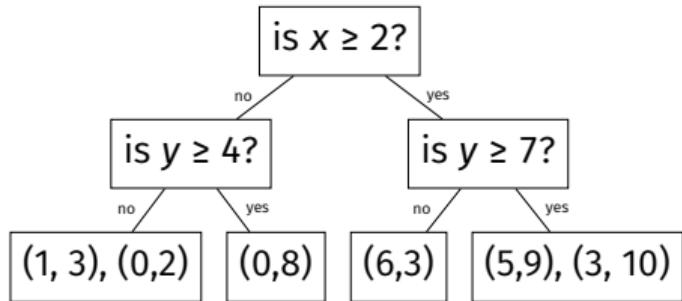
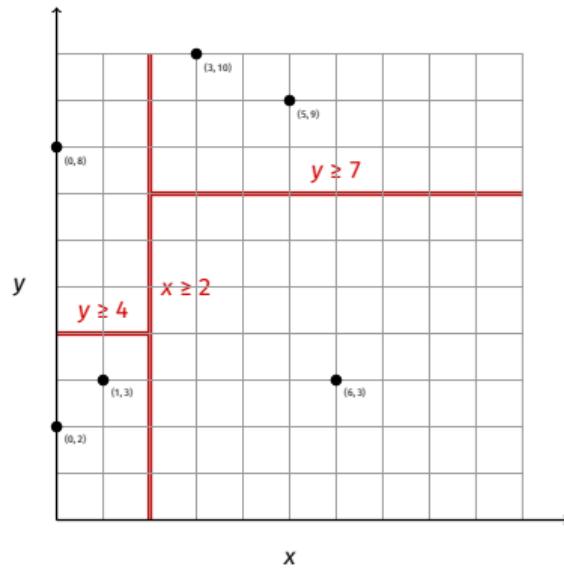
Leaf Nodes

- ▶ Leaf nodes are (collections of) points.



Partitioning

- Each internal node **splits** space.



k-d Trees in Python

```
from dataclasses import dataclass
from typing import Union, Optional
import numpy as np

@dataclass
class KDInternalNode:
    # the left and right children can be internal nodes
    # or numpy arrays of points (leaf nodes)
    left: Union['KDInternalNode', np.ndarray]
    right: Union['KDInternalNode', np.ndarray]

    # the threshold tau in the question
    threshold: float

    # the dimension used in the question
    dimension: int
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 3

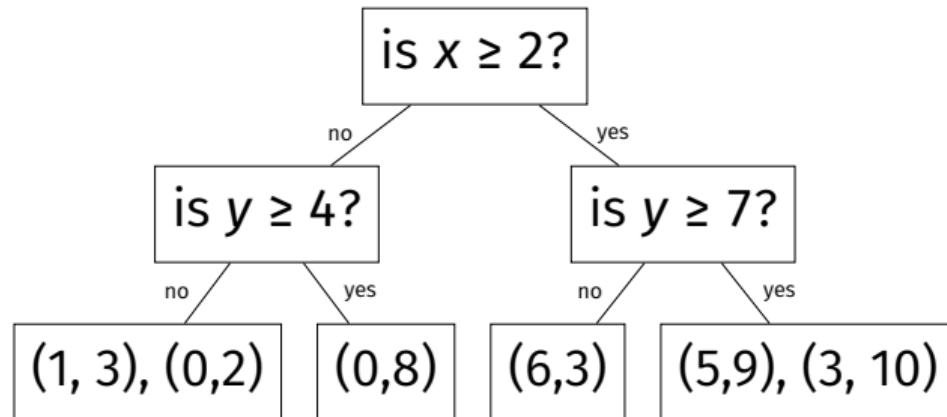
Queries on k-d Trees

Types of Queries

- ▶ Standard query:
 - ▶ Is (1, 5) in the tree?
- ▶ Nearest neighbor query:
 - ▶ Return the nearest neighbor(s) of (1, 5).

Standard Queries

- ▶ Is $(6,3)$ in the tree? Is $(1, 5)$ in the tree?



Standard Queries

- ▶ Similar to BST query.
 - ▶ Recursively choose left/right by answering question.
 - ▶ Brute-force linear search on leaf (if needed).
- ▶ Takes $O(h)$ time, where h is height of the tree².

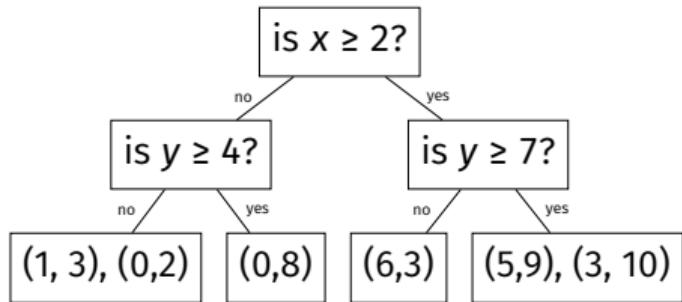
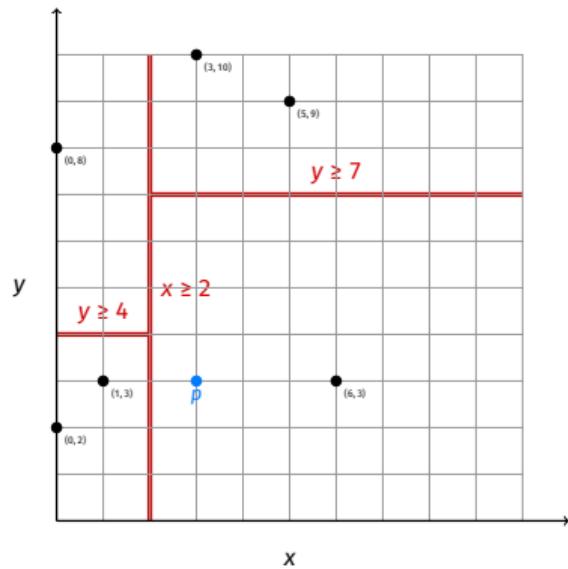
²Assuming each leaf has a bounded number of points.

Nearest Neighbor Queries

- ▶ Given query point $p = (x, y)$, find nearest neighbor in tree.
- ▶ Can we just do a standard query?
 - ▶ Find cell that *would* contain (x, y) .
 - ▶ Return closest neighbor within that cell.

No

- Example: $p = (3, 3)$.



Main Idea

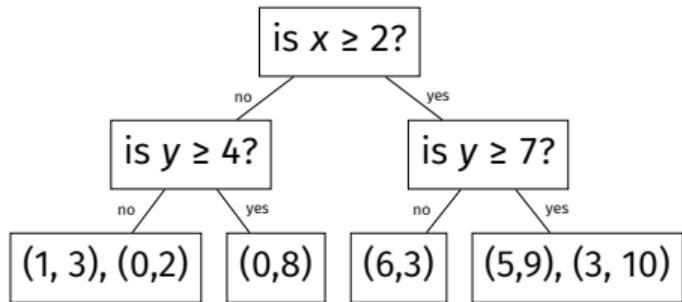
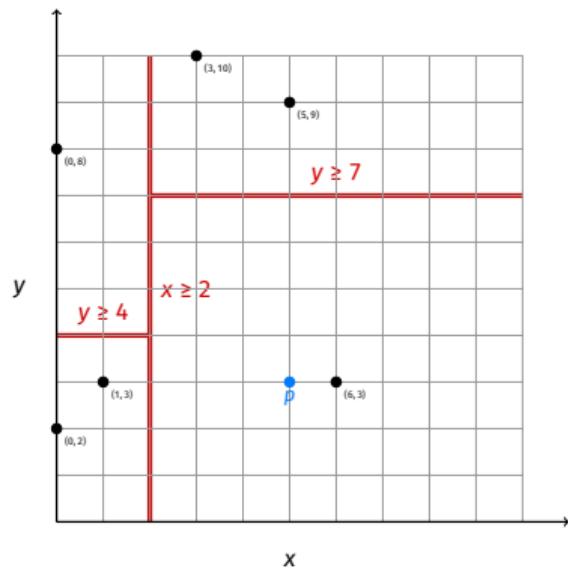
It is not sufficient to only check the cell that p would be placed in. You must also check neighboring cells (which can be very far away in the tree).

Brute Force?

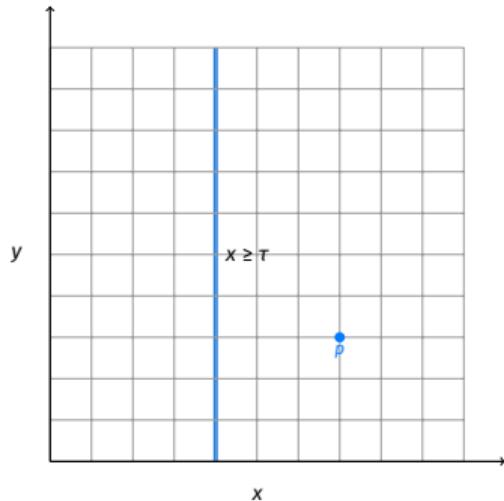
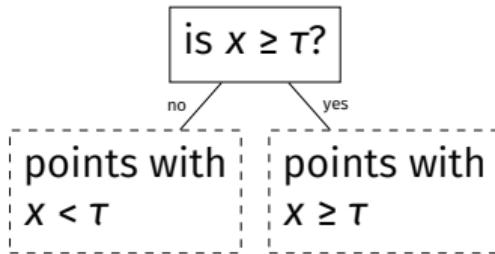
- ▶ This suggests we need to traverse the whole tree.
- ▶ But we can actually do much better.
- ▶ Intuitively, some branches can be ruled out (**pruned**).

Example

- Example: $p = (5, 3)$.



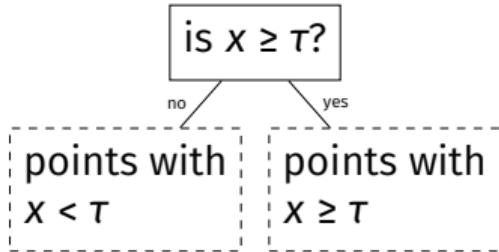
Bounding Branches



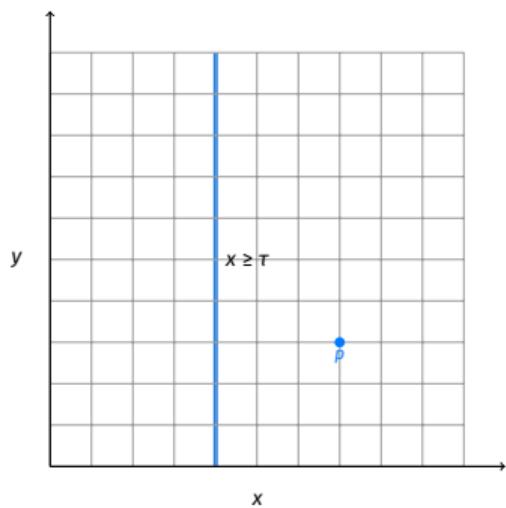
- ▶ **Observation:** let d_{bound} be distance from p to the boundary.
- ▶ Then the closest a point in the other branch can be to p is d_{bound}
- ▶ If we search and find a point whose distance to p is less than d_{bound} , **we do not need to search other branch.**

Bounding Branches

To query NN of (x, y) :



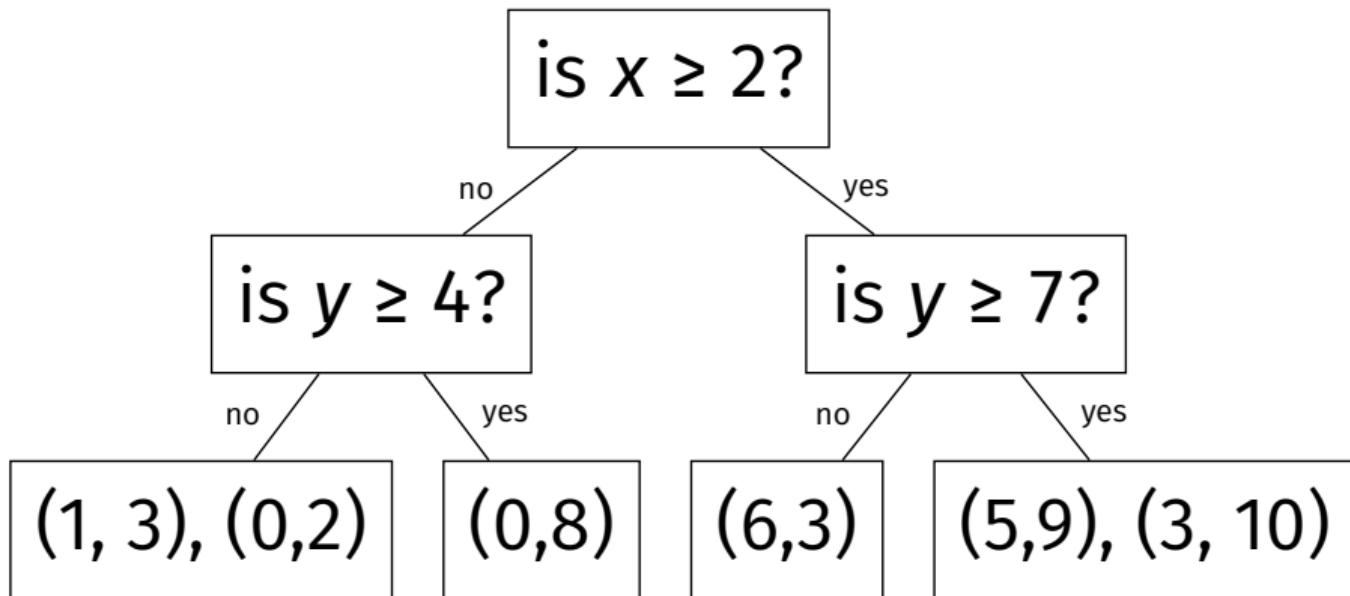
- ▶ Search right branch first if $x \geq t$, otherwise search left branch first.
- ▶ Let d_{nn} be the distance from p to the closest point found.
- ▶ Let d_{bound} be the distance from p to boundary.
- ▶ Search other branch only if $d_{bound} < d_{nn}$.



Apply this idea recursively.

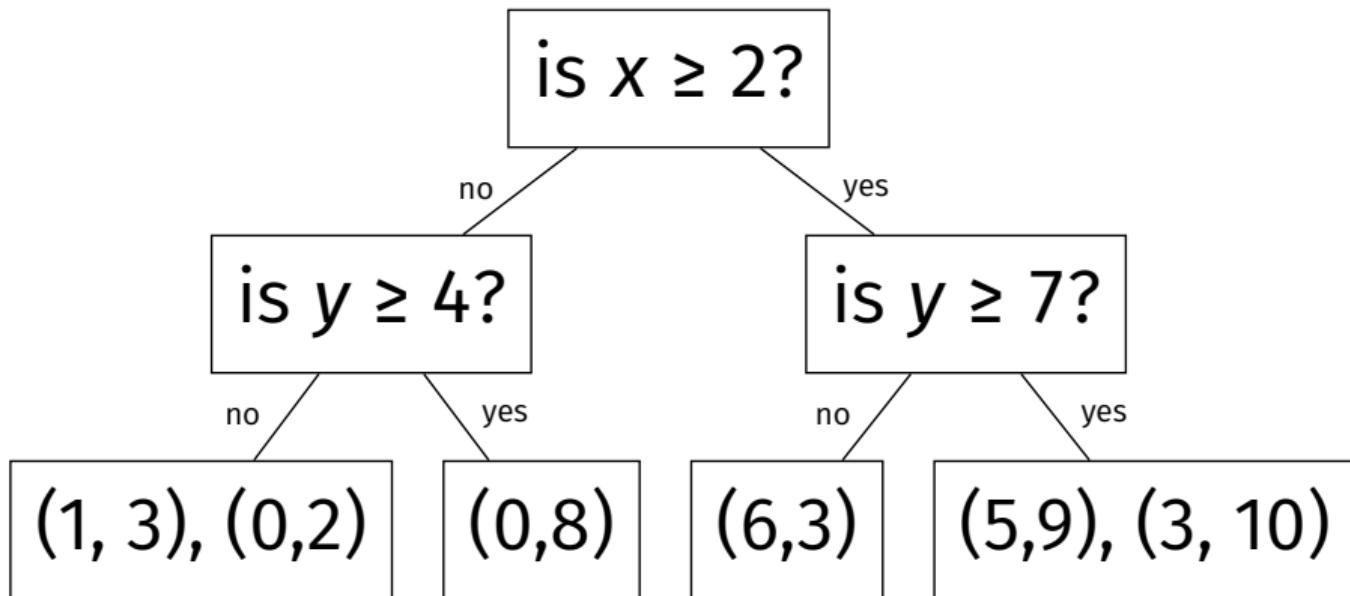
Example

- ▶ NN Query: (5, 3)



Example

- ▶ NN Query: (3, 3)



```
def nn_query(node, p):
    if isinstance(node, np.ndarray):
        return brute_force_nn_search(node, p)
    else:
        # find the most likely branch
        if p[node.dimension] >= node.threshold:
            most_likely_branch, other_branch = node.right, node.left
        else:
            most_likely_branch, other_branch = node.left, node.right

        # compute distance to boundary
        distance_to_boundary = abs(p[node.dimension] - node.threshold)

        # find nn within most likely branch
        nn, nn_distance = nn_query(most_likely_branch, p)

        # check the other branch, but only if necessary
        if distance_to_boundary < nn_distance:
            nn_other, nn_other_distance = nn_query(other_branch, p)

            # check if the nn within this branch is closer
            if nn_other_distance < nn_distance:
                nn = nn_other
                nn_distance = nn_other_distance

    return nn, nn_distance
```

k-NN Search

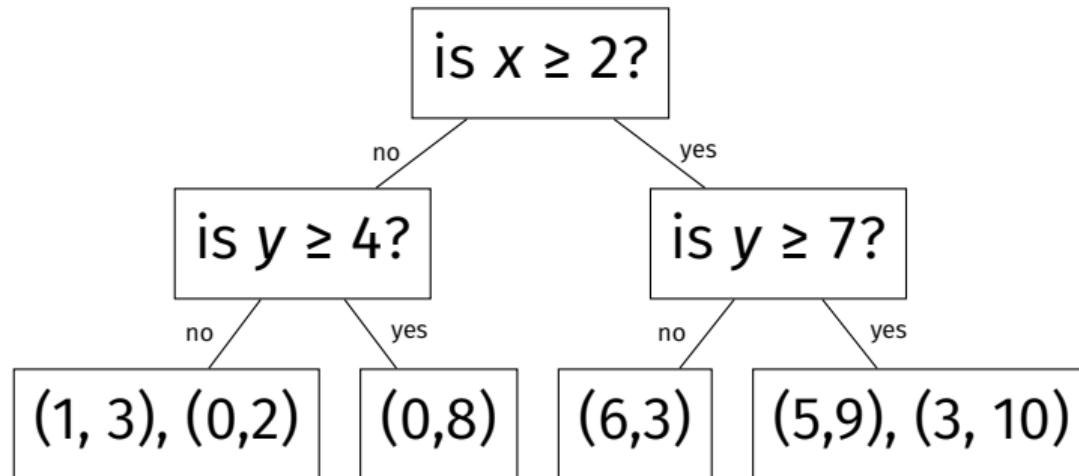
- ▶ Sometimes we want to find k nearest neighbors.
- ▶ Keep a max heap of best k so far.
- ▶ Check branch if distance to boundary $< k$ th closest.

Analysis

- ▶ Assume each leaf has bounded number of points.
- ▶ Best case: $\Theta(h) \rightarrow \Theta(\log n)$ if balanced
- ▶ Worst case: $\Theta(n)$.
 - ▶ We may be unable to rule out many of the branches.
 - ▶ Can occur even if tree is balanced.
 - ▶ Especially if query point far from data.
- ▶ Note: balancing is difficult, but possible.

Example of Worst Case

- ▶ NN Query: (20, 20)
- ▶ Closest point is (5, 9) at distance ≈ 19



Performance Degradation

- ▶ In small dimensions, NN lookup usually takes $\Theta(\log n)$.
- ▶ We'll see performance **degrades** to $\Theta(n)$ (brute force) as dimensionality $\rightarrow \infty$.
- ▶ **Curse of Dimensionality**

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 4

Constructing k-d Trees

Construction

- ▶ **Given:** a set of n data points in \mathbb{R}^d
- ▶ **Construct:** a k-d tree containing these points.

Caveats

- ▶ There are many variations on k-d tree construction.
- ▶ We'll describe one popular approach.
- ▶ **Assumption:** **offline** construction.
 - ▶ Have all of the data at once (no insert/delete).

Idea

- ▶ Starting with n points, either:
 - ▶ make internal node by splitting ($x \geq \tau?$)
 - ▶ make leaf node containing the points
- ▶ Apply this strategy recursively.
- ▶ Questions:
 - ▶ Do we split, or do we make a leaf?
 - ▶ If we split:
 - ▶ What dimension to split on?
 - ▶ What threshold to use?

Q1: Do we split?

- ▶ Take parameter M (max leaf size).
- ▶ If $n < M$, don't split.
- ▶ **Reason:** For small n , brute force is actually faster (less overhead).

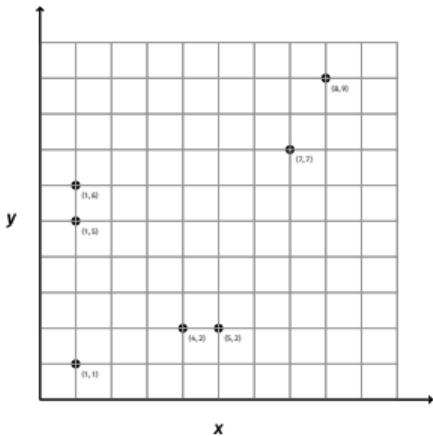
Q2: Which dimension to split on?

- ▶ Choose dimension with largest **spread**.
 - ▶ Difference between largest and smallest values.
 - ▶ Calculated using only points in **this** subtree.
- ▶ **Alternatively:** round-robin. Split x, y, z, x, y, ...

Q3: What threshold to use?

- ▶ Need threshold, τ .
- ▶ Use median value in splitting dimension.
 - ▶ Calculated using only points in **this** subtree.
 - ▶ Guaranteed to produce balanced tree.
- ▶ **Alternatively:** randomly-selected pivot, or median of random selection

Set $M = 2$, use median and spread for splitting. We start with data:

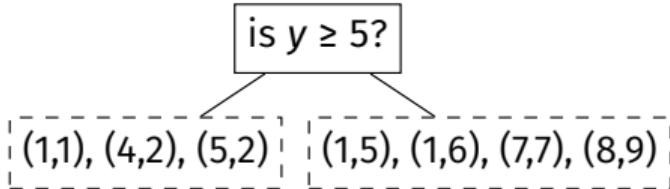
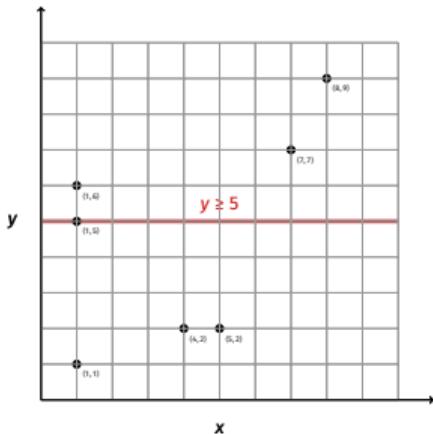


$(1,1), (4,2), (5,2), (1,5), (1,6), (7,7), (8,9)$

| x | y |
|---|---|
| 4 | 2 |
| 1 | 1 |
| 5 | 2 |
| 1 | 6 |
| 7 | 7 |
| 8 | 9 |
| 2 | 5 |

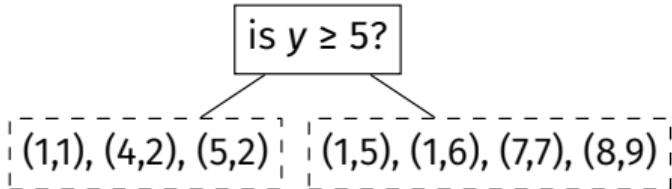
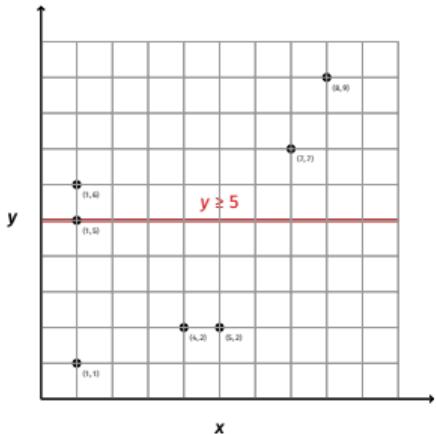
- ▶ Spread of x: 7
- ▶ Spread of y: 8
- ▶ Use y as splitting dimension.
- ▶ Median of y: 5.

Set $M = 2$, use median and spread for splitting. We start with data:



| x | y |
|---|---|
| 4 | 2 |
| 1 | 1 |
| 5 | 2 |
| 1 | 6 |
| 7 | 7 |
| 8 | 9 |
| 2 | 5 |

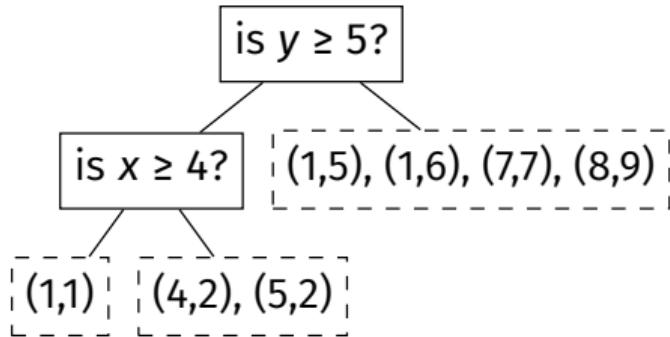
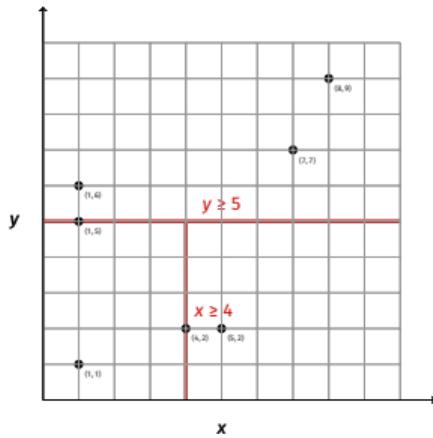
- ▶ Spread of x: 7
- ▶ Spread of y: 8
- ▶ Use y as splitting dimension.
- ▶ Median of y: 5.



Recurse on left child. Data becomes:

| x | y |
|---|---|
| 4 | 2 |
| 1 | 1 |
| 5 | 2 |

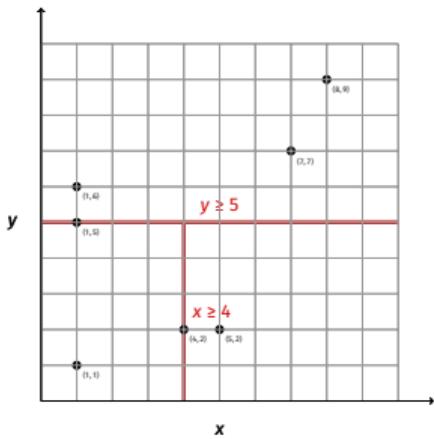
- ▶ Spread of x: 4
- ▶ Spread of y: 1
- ▶ Use x as splitting dimension.
- ▶ Median of x: 4.



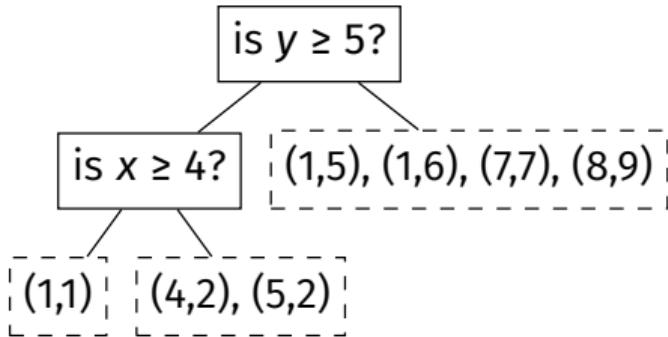
Recurse on left child. Data becomes:

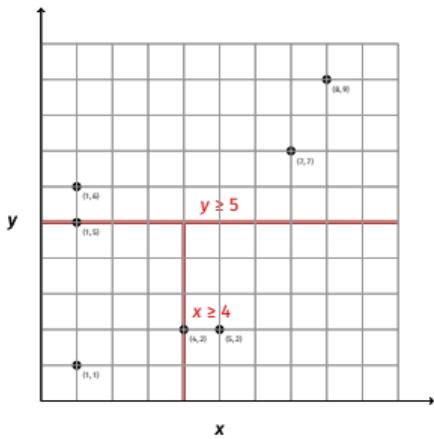
| x | y |
|---|---|
| 4 | 2 |
| 1 | 1 |
| 5 | 2 |

- ▶ Spread of x: 4
- ▶ Spread of y: 1
- ▶ Use x as splitting dimension.
- ▶ Median of x: 4.

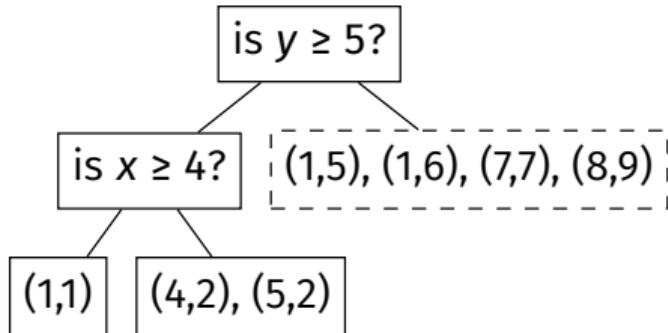


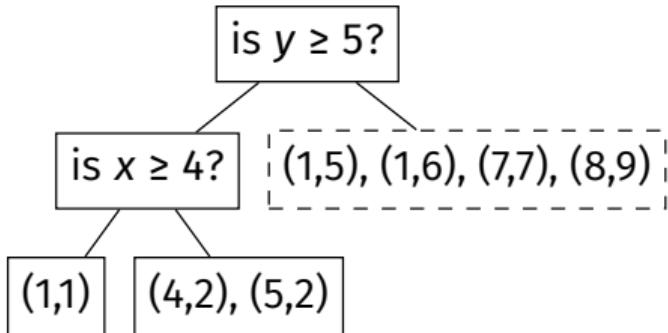
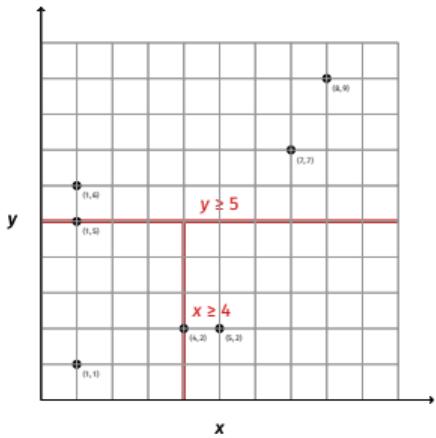
Recurse on children. Since size $\leq M$, these become leaf nodes.





Recurse on children. Since size $\leq M$, these become leaf nodes.

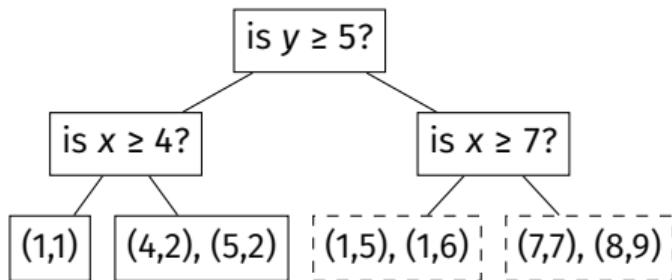
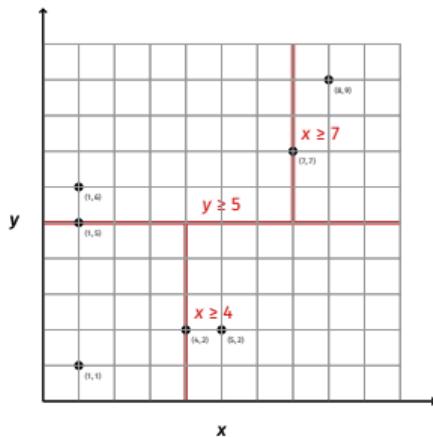




Unroll recursion, now recurse down right side of tree. Data becomes:

| x | y |
|---|---|
| 1 | 6 |
| 7 | 7 |
| 8 | 9 |
| 2 | 5 |

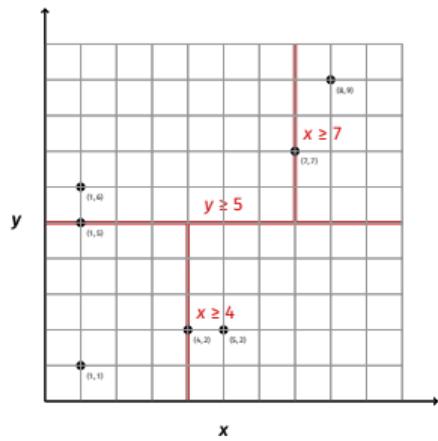
- ▶ Spread of x: 7
- ▶ Spread of y: 4
- ▶ Use x as splitting dimension.
- ▶ Median of x: 7 (or 2).



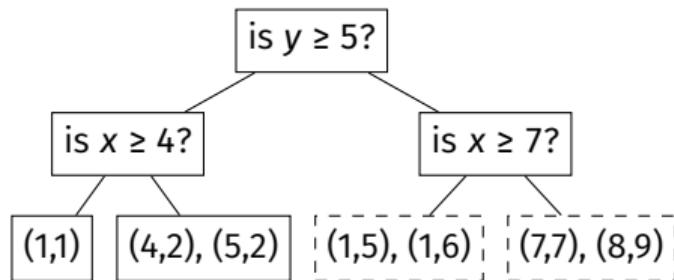
Unroll recursion, now recurse down right side of tree. Data becomes:

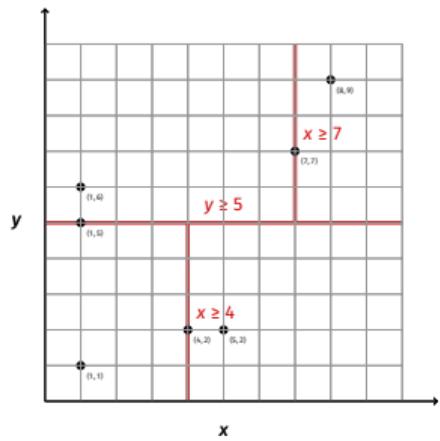
| <hr/> | <hr/> |
|-------|-------|
| x | y |
| 1 | 6 |
| 7 | 7 |
| 8 | 9 |
| 2 | 5 |

- ▶ Spread of x: 7
- ▶ Spread of y: 4
- ▶ Use x as splitting dimension.
- ▶ Median of x: 7 (or 2).

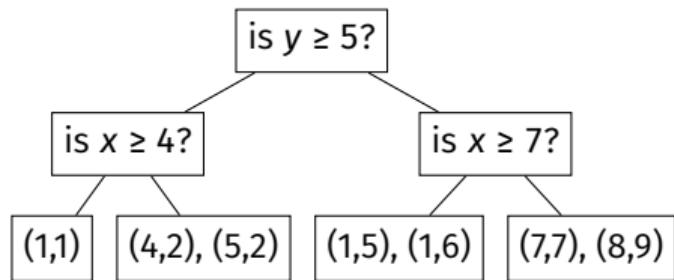


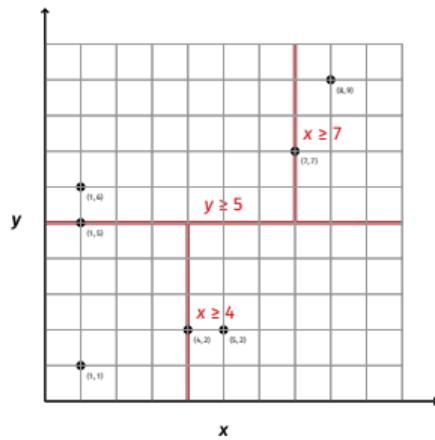
Make leaf nodes, since size $\leq M$.



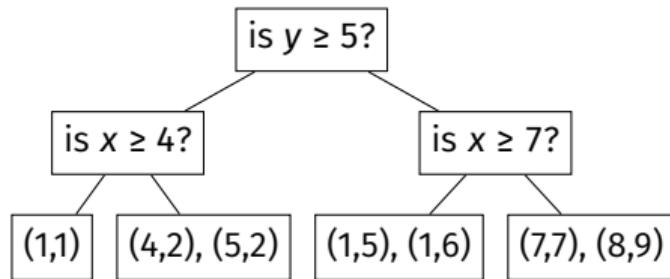


Make leaf nodes, since size $\leq M$.





Tree complete!



```
def build_kd_tree(data, m=2):
    if len(data) <= m:
        return data

    # find the dimension with greatest spread
    spread = data.max(axis=0) - data.min(axis=0)
    splitting_dimension = np.argmax(spread)

    # find the median along this dimension
    median = np.median(data[:, splitting_dimension])

    # separate the data into new left and right sets
    # note that this isn't the most efficient since it will
    # produce a copy... better to do an in-place partition
    left_data = data[data[:, splitting_dimension] < median]
    right_data = data[data[:, splitting_dimension] >= median]

    left = build_kd_tree(left_data)
    right = build_kd_tree(right_data)

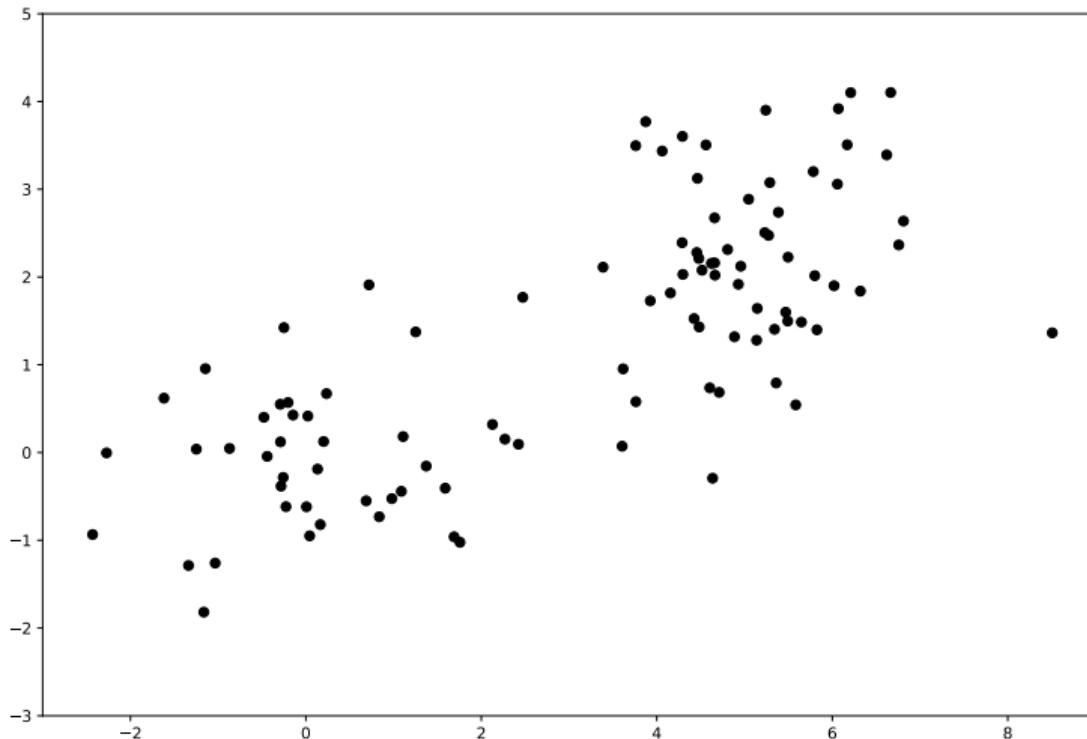
    return KDInternalNode(
        left=left, right=right, threshold=median,
        dimension=splitting_dimension
    )
```

Analysis

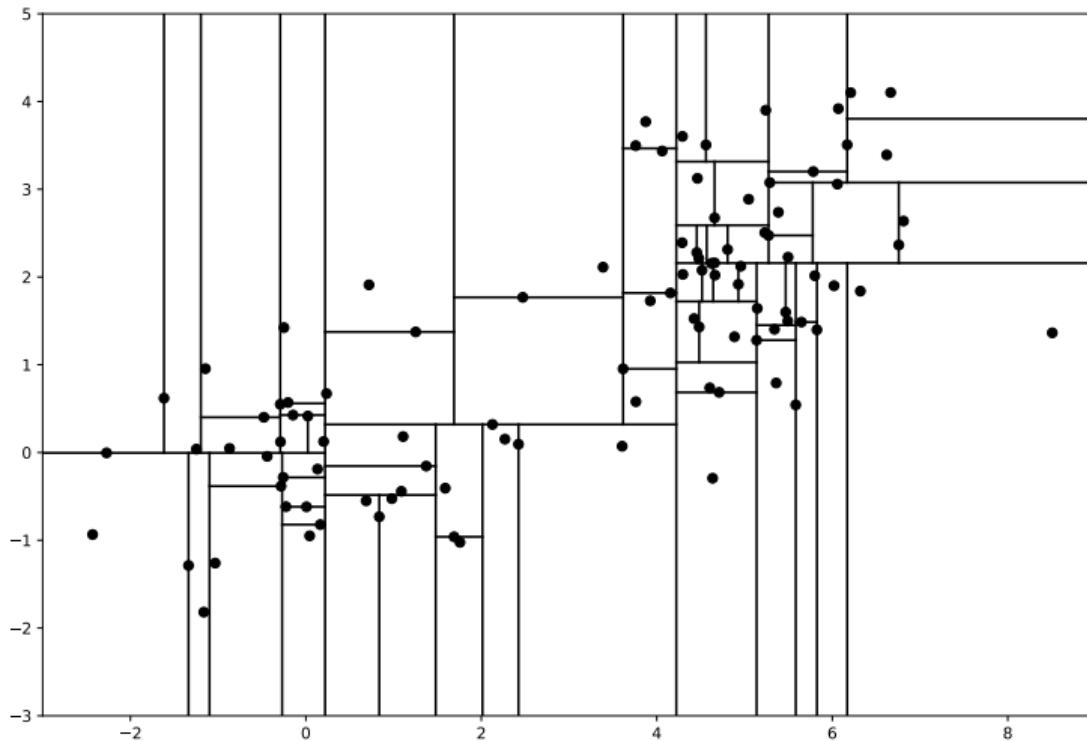
- ▶ $\Theta(k)$ to find median, perform copies, where k is number of points in subtree.
- ▶ Tree has $\Theta(\log n)$ levels (since it is balanced).
- ▶ Total time:

$$\underbrace{n}_{\text{level 1}} + \underbrace{(n/2 + n/2)}_{\text{level 2}} + \underbrace{(n/4 + n/4 + n/4 + n/4)}_{\text{level 3}} + \dots = \Theta(n \log n)$$

Example



Example



DSC 190

DATA STRUCTURES & ALGORITHMS

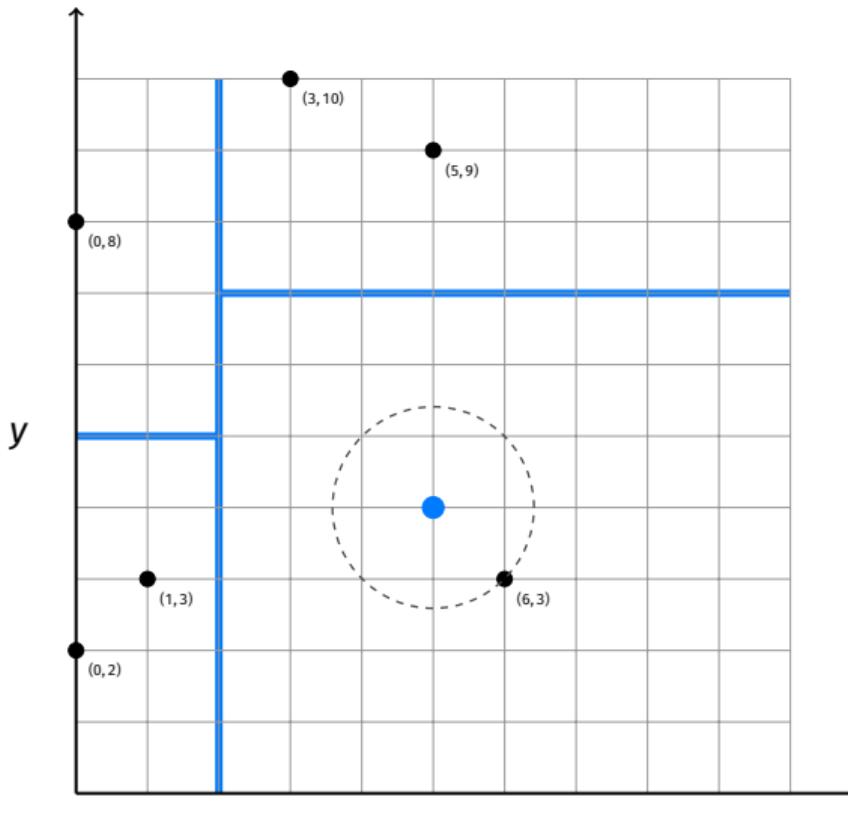
Lecture 6 | Part 5

Curse of Dimensionality

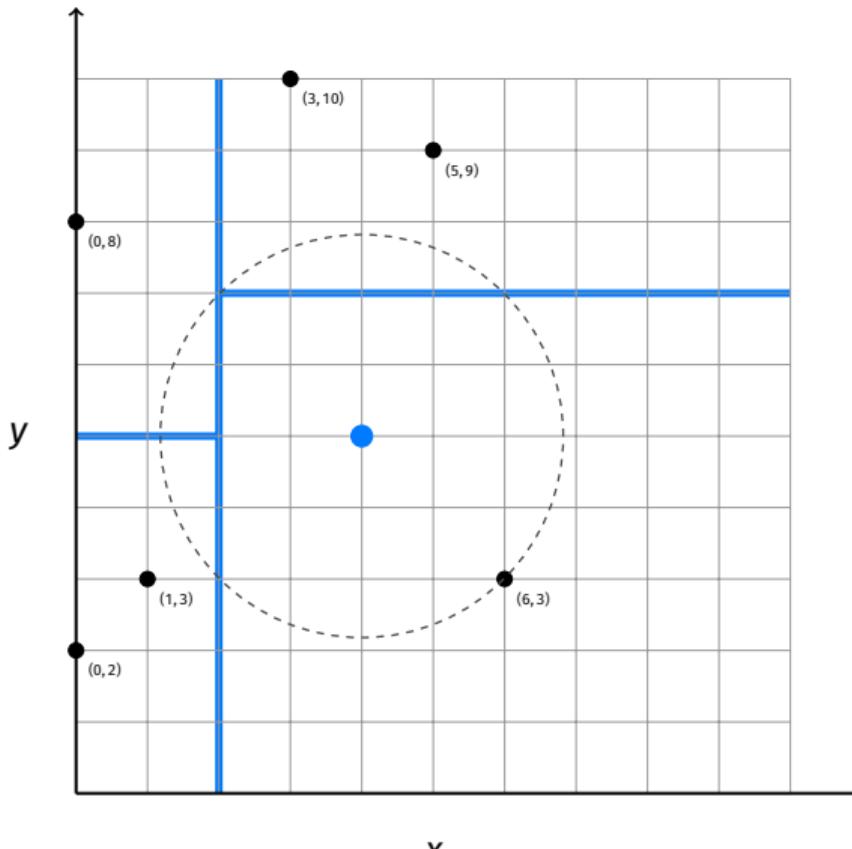
Performance Degradation

- ▶ Brute force NN search takes $\Theta(n)$ time.
- ▶ If dimensionality is small, k-d trees take $\Theta(\log n)$.
 - ▶ Great speedup!
- ▶ As dimensionality grows, performance **degrades**.
 - ▶ At worst, it is $\Theta(n)$.
 - ▶ Becomes just as bad as brute force!
- ▶ Why?

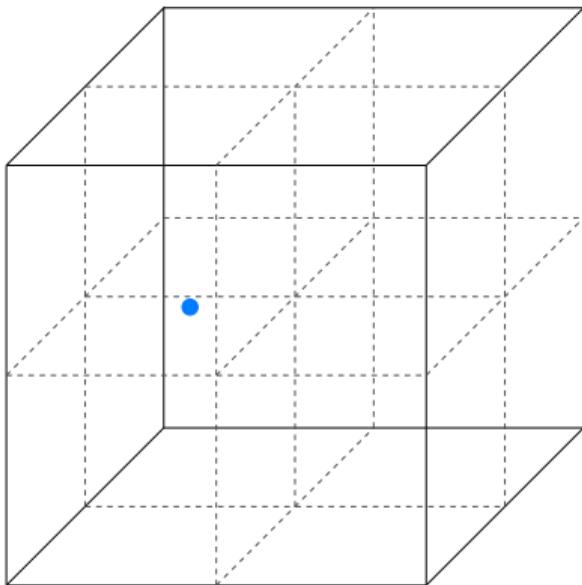
Explanation #1



Explanation #1



Explanation # 1

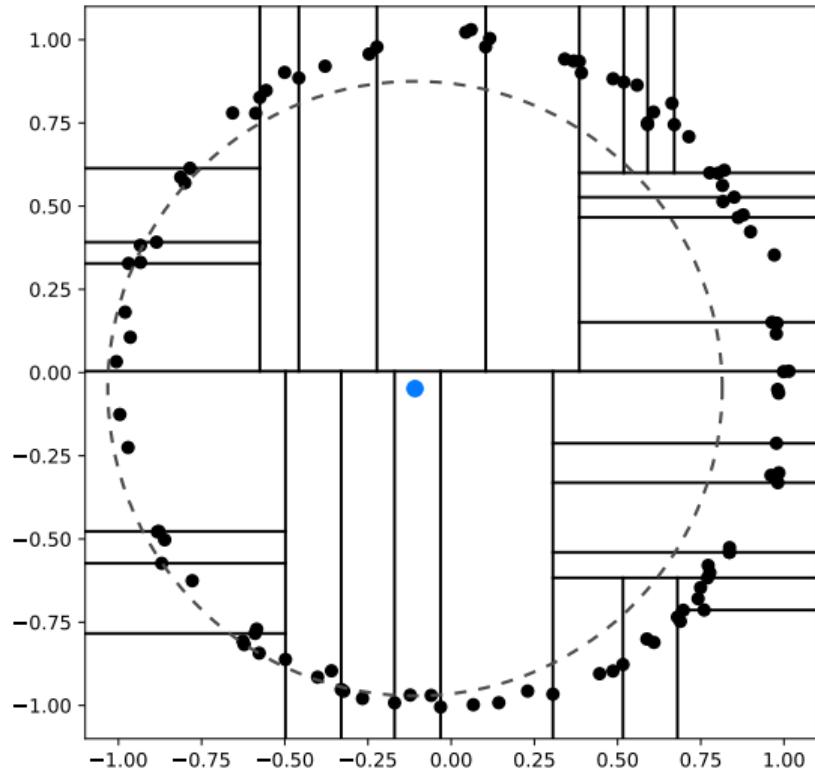


Main Idea

As d grows, the number of neighboring cells that we may need to check grows like 2^d .

Explanation #2

- ▶ We saw that if query point is far away, we cannot rule out branches.
- ▶ The reason? Distance from query to NN is not significantly different from distance between query and other points.



Surprising Fact

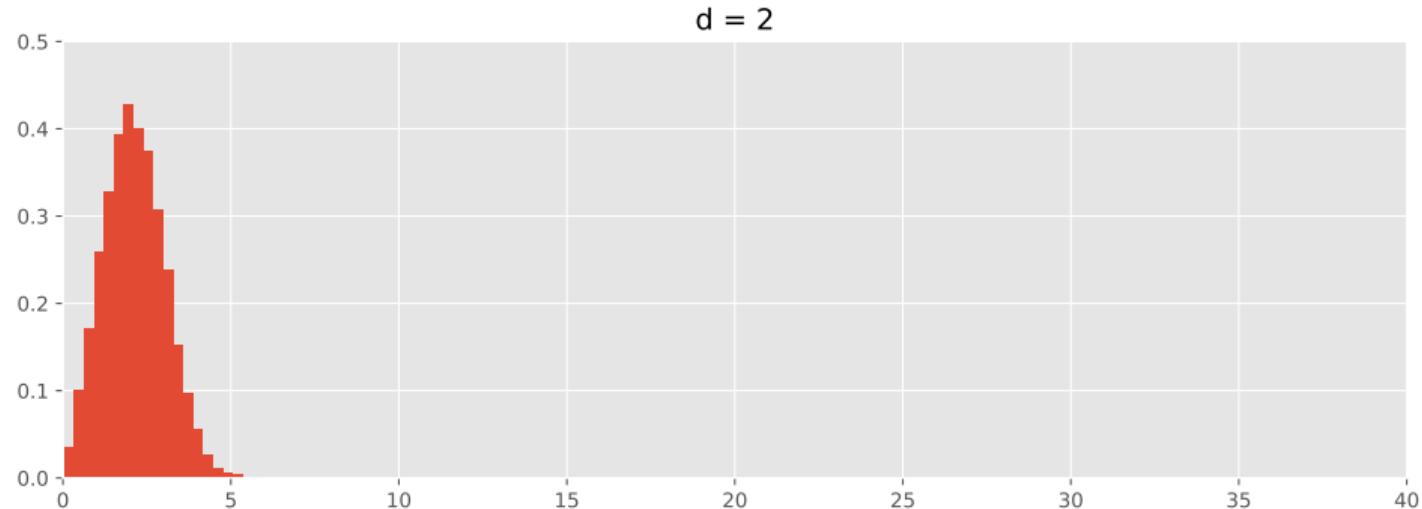
- ▶ In high dimensions³, the ratio of the distance to nearest neighbor and distance to furthest neighbor → 1.

³Under some assumptions on distribution of data.

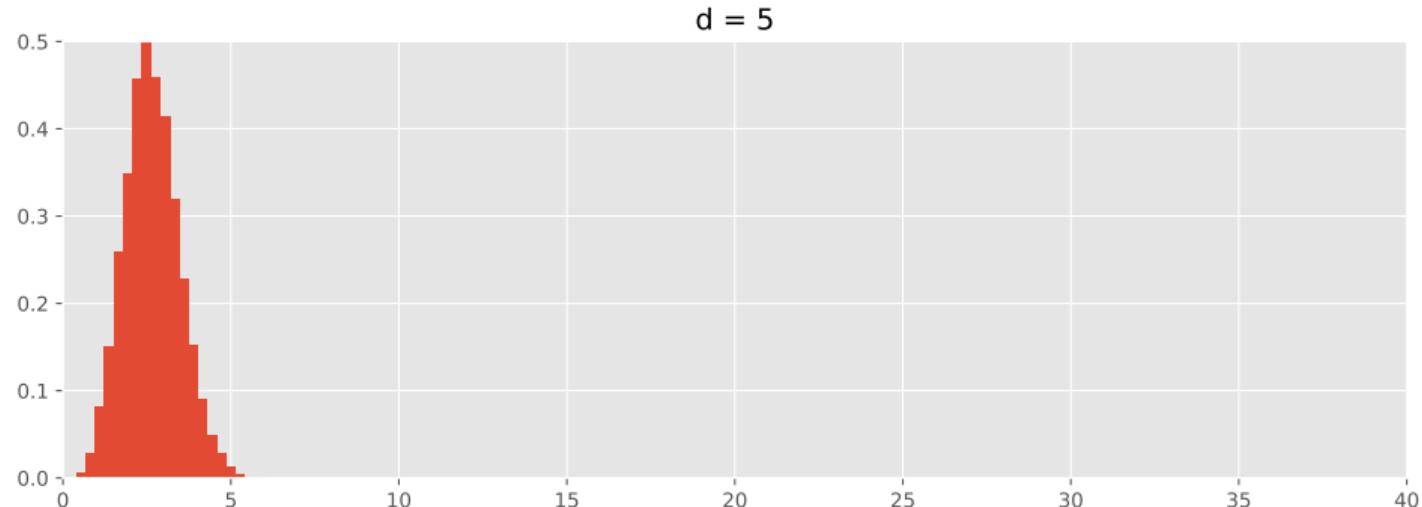
Experiment

- ▶ Generate random d -dimensional query vector from multivariate Gaussian.
- ▶ Generate 1000 d -dimensional data points from same Gaussian.
- ▶ Plot distribution of distances.

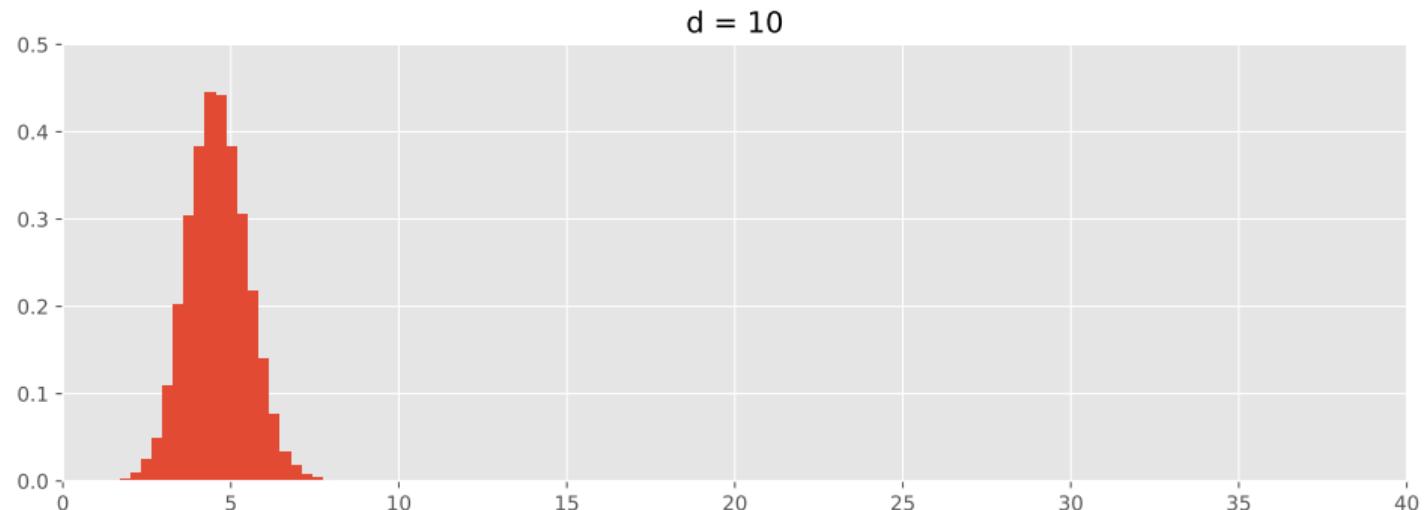
Experiment



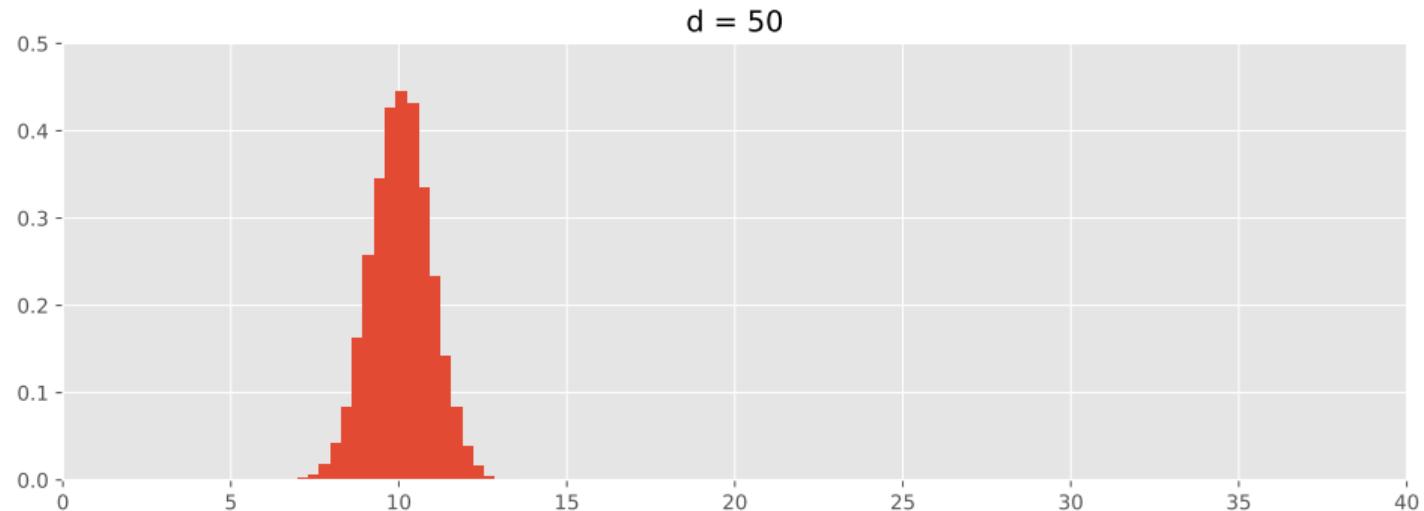
Experiment



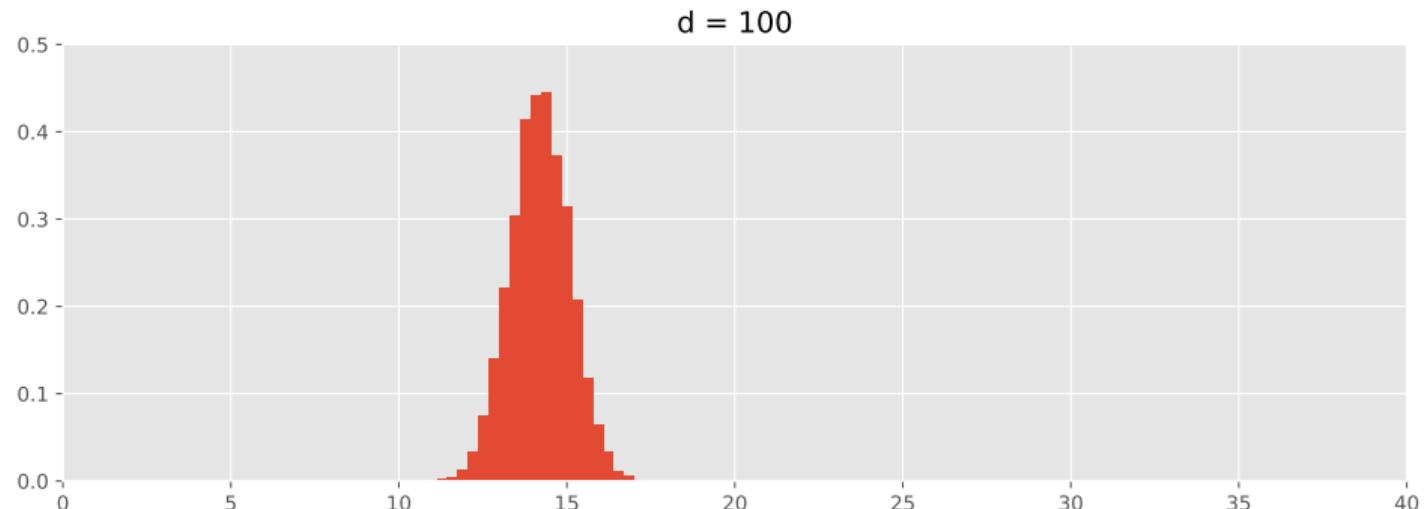
Experiment



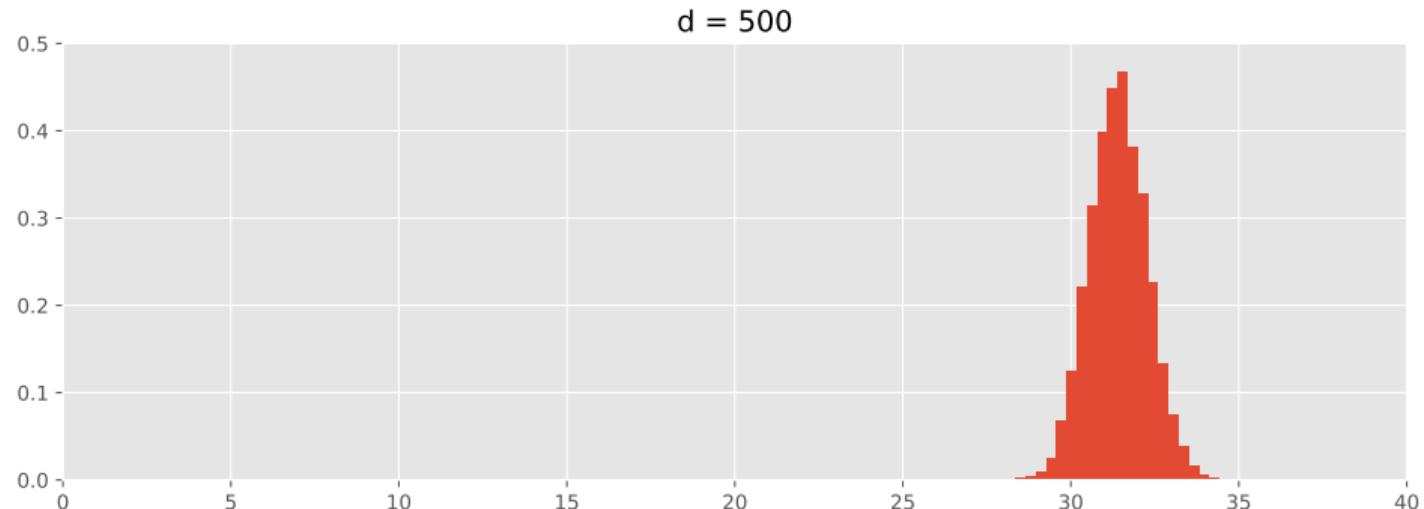
Experiment



Experiment



Experiment



Experiment

- ▶ Notice: width doesn't change, but center increases.
- ▶ So $\min = \max - \delta$, with δ constant.

$$\frac{\min}{\max} = 1 - \frac{\delta}{\max}$$

Explanation #2

- ▶ Every point in data set is approximately equidistant to query point.
- ▶ Can't rule out branches.
- ▶ Have to perform a brute force search.

Main Idea

In high dimensions, every data point is approximately equidistant to the query point, meaning we can't rule out most branches.

Main Idea

Not only are k-d trees **inefficient** in high dimensions, Euclidean distance is **less meaningful** in high dimensions, and therefore so is the concept of NN search itself.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 6

Approximate Nearest Neighbors

Why, exactly?

- ▶ Why do we need the **exact** NN?
- ▶ Often something close would do.
- ▶ Especially if not confident in distance measure.
 - ▶ As is the case in high dimensions.
- ▶ Maybe this can be done faster?

ANN

- ▶ **Given:** A set of points and a query point, p .
- ▶ **Return:** An **approximate nearest neighbor**.

k-D ANNs

- ▶ So far, our k-d trees find **exact** nearest neighbor.
- ▶ But there's a **very** simple way to do ANN query.
- ▶ Idea: prune more aggressively.

Before

- ▶ Let d_{nn} be distance from query point to best so far.
- ▶ Let d_{bound} be distance from query point to boundary.
- ▶ Search branch only if $d_{bound} < d_{nn}$.

Now

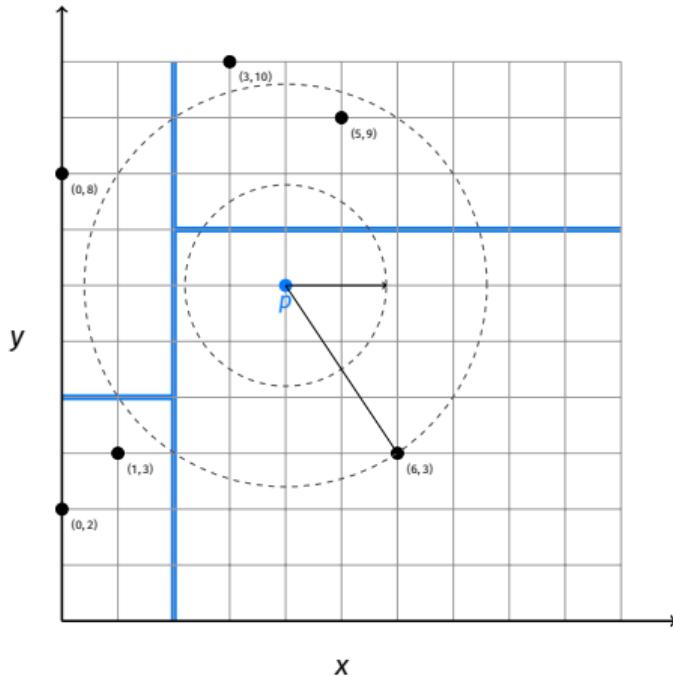
- ▶ Take $\alpha \geq 1$ as a parameter.
- ▶ Search branch only if $d_{\text{bound}} < d_{\text{nn}}/\alpha$.
- ▶ **Idea:** make it easier to toss out branch.
- ▶ If $\alpha = 1$; exact search.
- ▶ If $\alpha > 1$; approximate, faster as α grows.

Theory

- ▶ Let q be exact NN, let q_{ann} be that found by this strategy.

- ▶ Then:

$$d(p, q_{\text{ann}}) \leq \alpha \cdot d(p, q)$$



Next Time

- ▶ ANNs via **Locality Sensitive Hashing**.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 7 | Part 1

Today's Lecture

Approximate Nearest Neighbor

- ▶ **Given:** a set of n points in \mathbb{R}^d and query point p .
- ▶ **Compute:** (approximate) nearest neighbor of p .
- ▶ Last time: k-d trees do not scale well with d .

Today

- ▶ Slightly different problem: given a distance r and query p .
- ▶ Return (approximately) all of the points within distance r of p .
- ▶ Can use to compute ANN.

Today

- ▶ We'll introduce **locality sensitive hashing**.
- ▶ An important idea.
- ▶ We'll see similar themes in remainder of course.

DSC 190

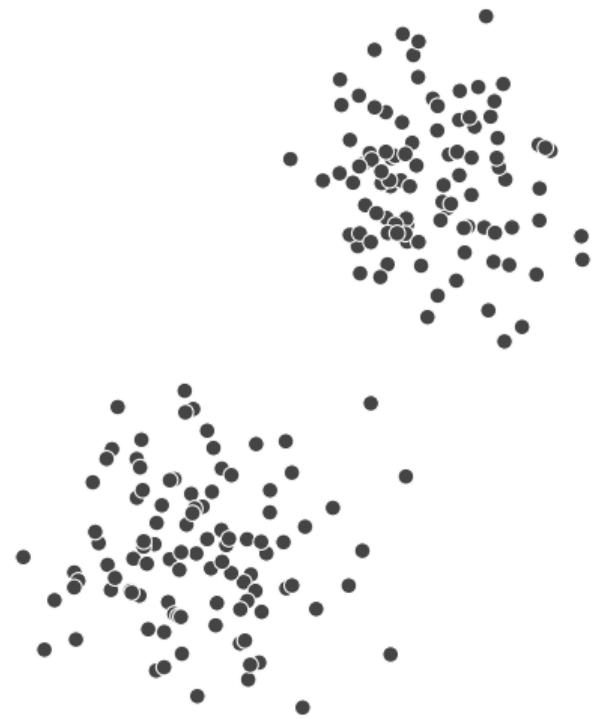
DATA STRUCTURES & ALGORITHMS

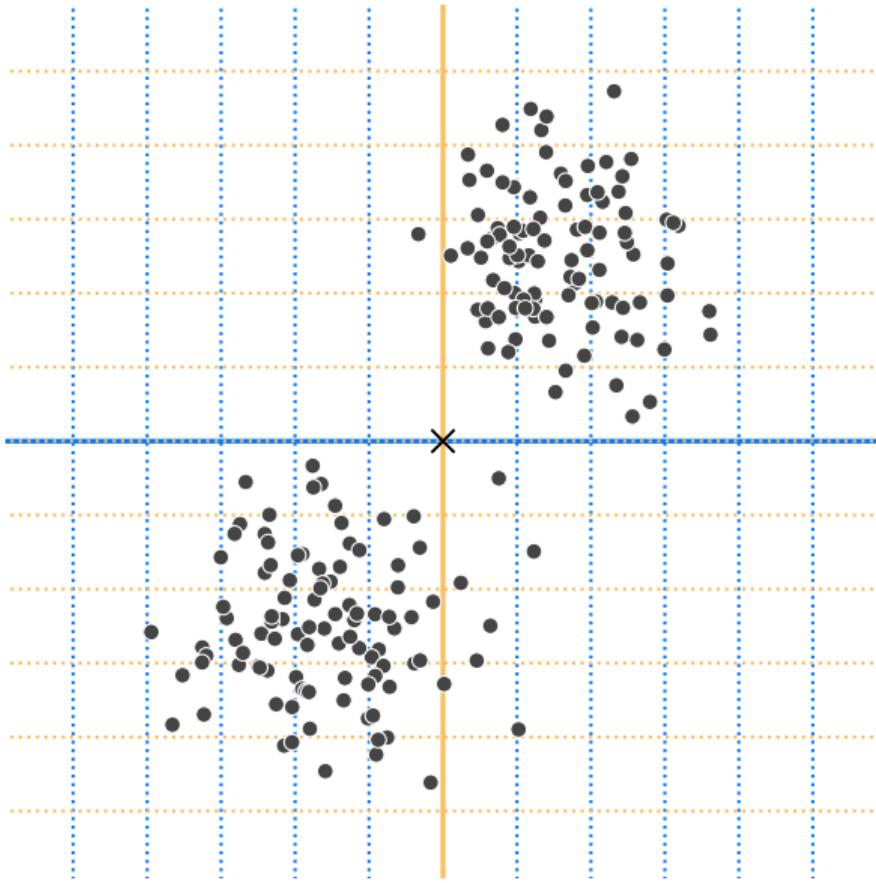
Lecture 7 | Part 2

Implementing a NN Grid

Grids

- ▶ Given input point p , want quick way to find nearby points.
- ▶ Idea: divide space into cells using grid.
- ▶ Find cell containing p , search it.
- ▶ How would we implement this?





Grid Cells

- ▶ Each point (x, y) given cell id: $(\lfloor x \rfloor, \lfloor y \rfloor)$
 - ▶ Example: $(1.2, 6.7)$ given cell id $(1, 6)$.
- ▶ Store (x, y) in dictionary with cell id as key.
 - ▶ Discretization allows multiple points in same cell.
 - ▶ Store collisions in list.
- ▶ Generalizes naturally to d -dimensions.

```
class NNGrid:

    def __init__(self, width):
        self.width = width
        self.cells = {}

    def cell_id(self, p):
        p = np.asarray(p)
        cell_id = np.floor(p / self.width).astype(int)
        return tuple(cell_id)

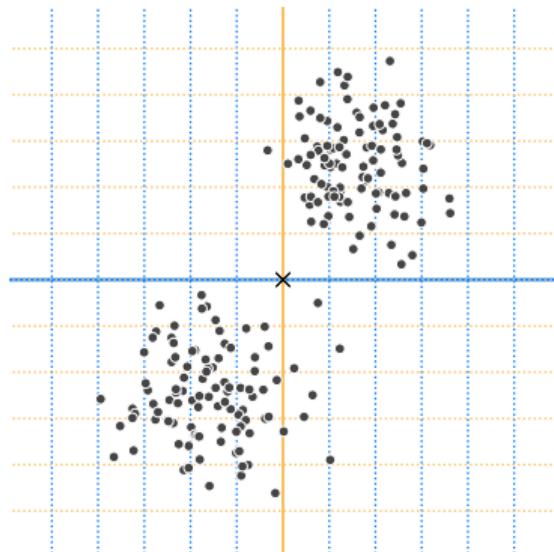
    def insert(self, p):
        """Insert p into the grid."""
        cell_id = self.cell_id(p)
        if cell_id not in self.cells:
            self.cells[cell_id] = []
        self.cells[cell_id].append(p)

    ...
```

```
...  
  
def points_in_cell(self, p):  
    cell_id = self.cell_id(p)  
    if cell_id not in self.cells:  
        return []  
    points_in_cell = self.cells[cell_id]  
    # turn into an array  
    return np.vstack(points_in_cell)  
  
def query(self, p):  
    return brute_force_nn(self.points_in_cell(p), p)
```

Note

- ▶ This may **fail** – NN could be in different cell.



Problems

- ▶ In d dimensions, cell id has d entries.

$$\text{cell-id}(p) = (\lfloor x_1/w \rfloor, \lfloor x_2/w \rfloor, \dots, \lfloor x_d/w \rfloor)$$

- ▶ All entries must be **exactly** the same for two points to have same cell id.
- ▶ This is **very unlikely**. Most cells are empty or contain one point.

High-Dimensional Cuboids

- ▶ One “fix”: increase cell width parameter.
- ▶ Suppose we want to ensure any points within distance r are in same cell.
- ▶ Then cell width must be $2r$.¹

¹Note: Jan 2022, this isn't actually true... but idea of next slide still holds.

High-Dimensional Cuboids

- ▶ But a d -dimensional cuboid of width $2r$ can contain points at distance $2\sqrt{dr}$ from one another!
- ▶ For even modest r , the whole data set is in one cell.

Main Idea

Dividing into a grid of cuboids fails in high dimensions. Either the cells are empty, or contain everything, depending on the width!

DSC 190

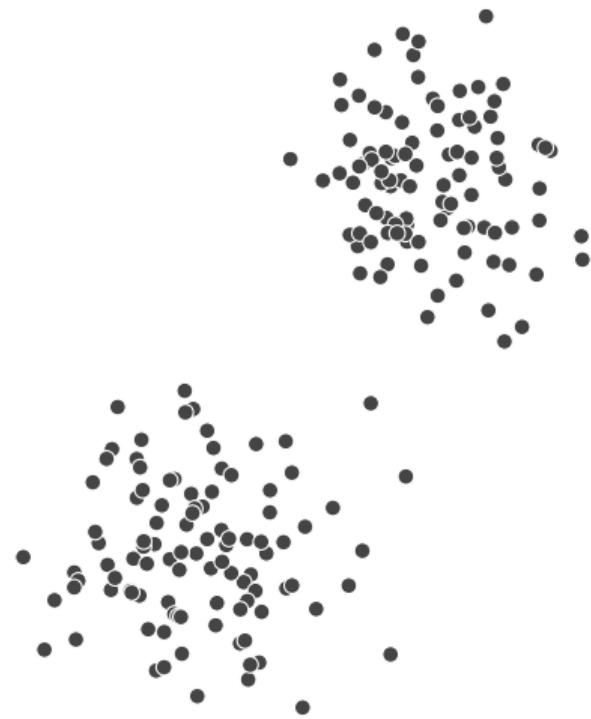
DATA STRUCTURES & ALGORITHMS

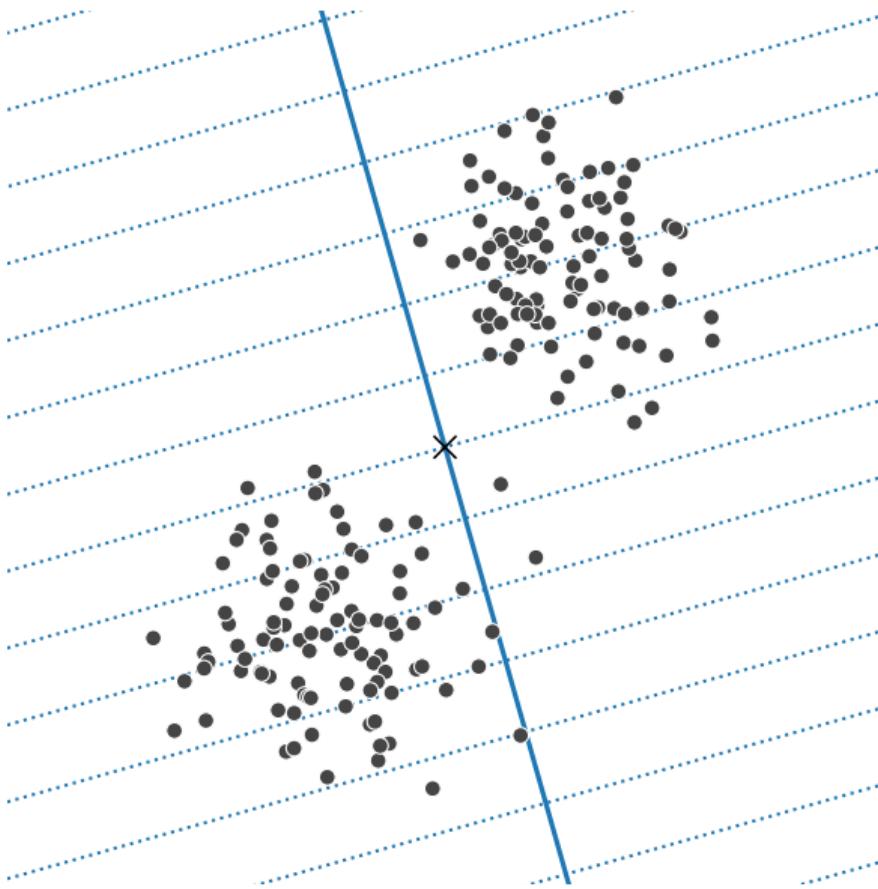
Lecture 7 | Part 3

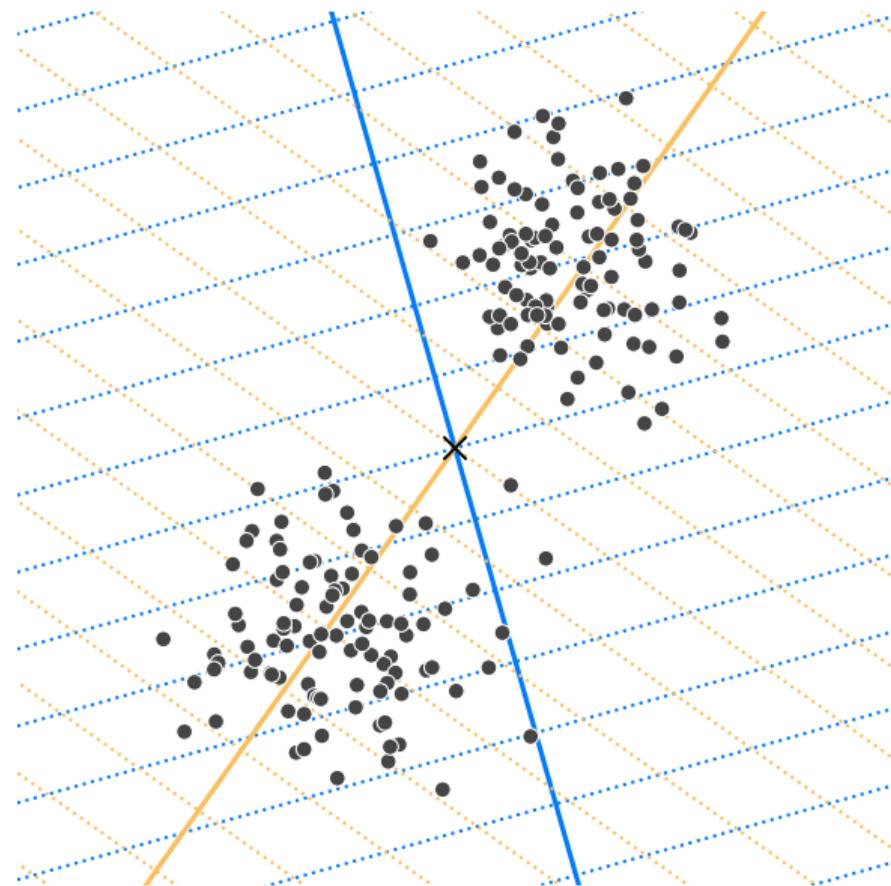
A Randomized “Grid”

A Randomized “Grid”

- ▶ Idea: Instead of axis-aligned grid, divide into cells using $k \ll d$ random directions.

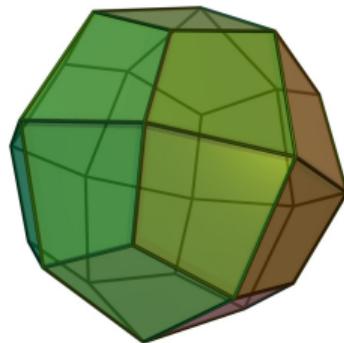






Cell Shape

- ▶ Cells are no longer d -dimensional cuboids.
- ▶ They are random k -dimensional **polytopes**.

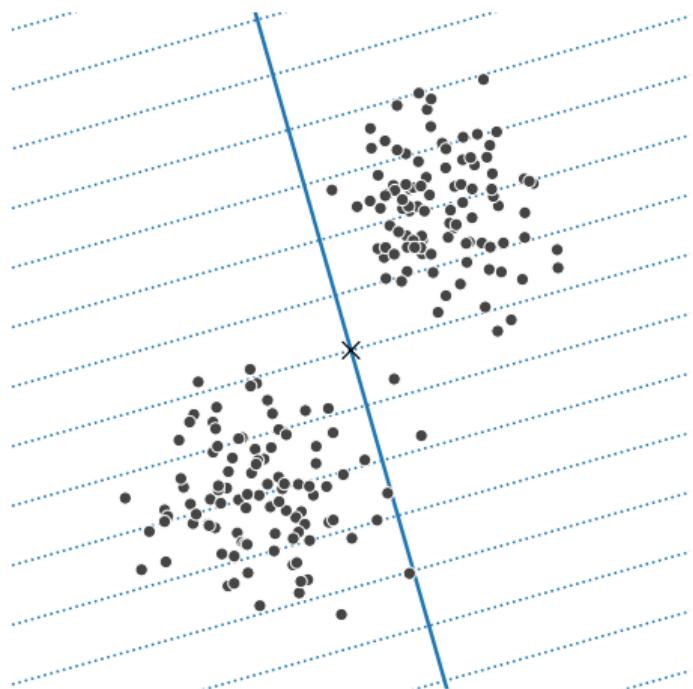


Question

- ▶ Why is this better? We'll see in the next sections.

Projection

- ▶ How do we determine which cell a point lies in?



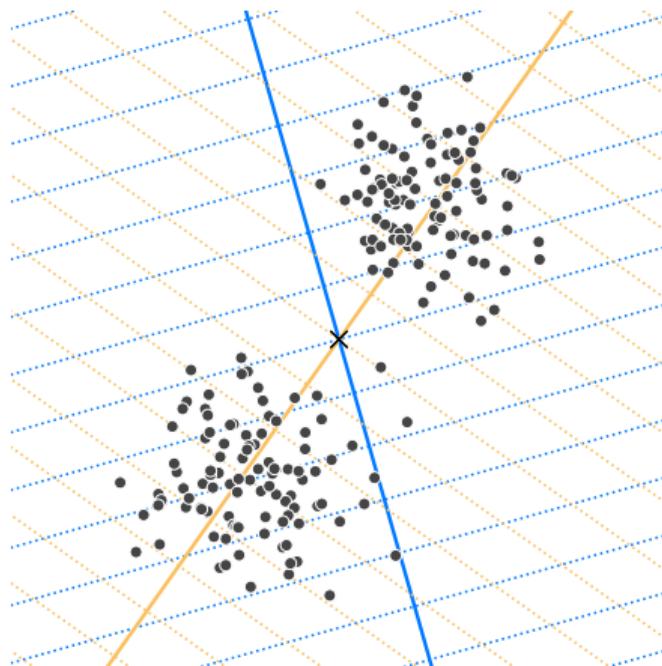
Cell IDs

- ▶ Pick k random unit vectors, $\vec{u}^{(1)}, \dots, \vec{u}^{(k)} \in \mathbb{R}^d$.
- ▶ Pick a width parameter, w .
- ▶ Given any point \vec{p} , its cell id is²:

$$\text{cell-id}(\vec{p}) = \left(\left\lfloor \frac{\vec{u}^{(1)} \cdot \vec{p}}{w} \right\rfloor, \left\lfloor \frac{\vec{u}^{(2)} \cdot \vec{p}}{w} \right\rfloor, \dots, \left\lfloor \frac{\vec{u}^{(k)} \cdot \vec{p}}{w} \right\rfloor, \right)$$

²use same width and unit vectors for all points

Example



Quick Cell-ID Calculation

- ▶ Place $\vec{u}^{(1)}, \dots, \vec{u}^{(k)}$ into a matrix:

$$U = \begin{pmatrix} \leftarrow & (\vec{u}^{(1)})^T & \rightarrow \\ \leftarrow & (\vec{u}^{(2)})^T & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & (\vec{u}^{(k)})^T & \rightarrow \end{pmatrix}$$

- ▶ Then $\text{cell-id}(\vec{p}) = \text{entrywise-floor}(U\vec{p}/w)$

Generating Random Unit Vectors

```
def gaussian_projection_matrix(k, d):
    X = np.random.normal(size=(k, d))
    U = X / np.linalg.norm(X, axis=1)[:,None]
    return U
```

```
class NNProjectionGrid

    def __init__(self, projection_matrix, width):
        self.width = width
        self.projection_matrix = projection_matrix
        self.cells = {}

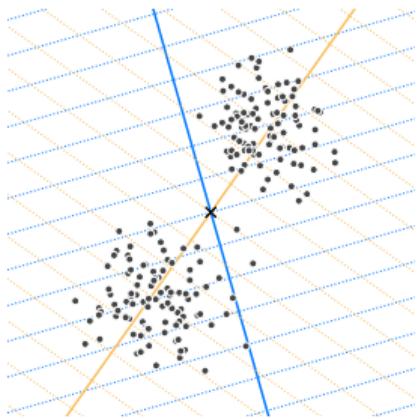
    def cell_id(self, p):
        projection = self.projection_matrix @ p
        cell_id = np.floor(projection / self.width)
        return tuple(cell_id.astype(int))

    # insert, query, points_in_cell same as for NNGrid
```

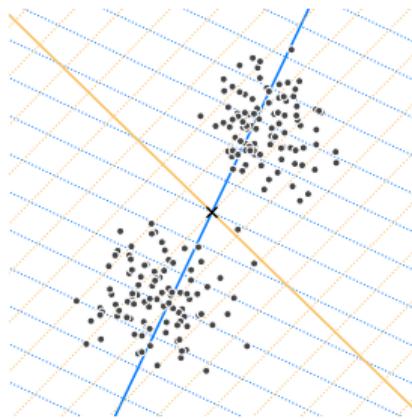
But wait...

- ▶ In high dimensions, still **very unlikely** for cell to contain >1 point.
- ▶ Idea: **banding**. Try, try again.
- ▶ Build multiple NNProjectionGrids with different random projections.
- ▶ Find points_in_cell for each, pool them together.

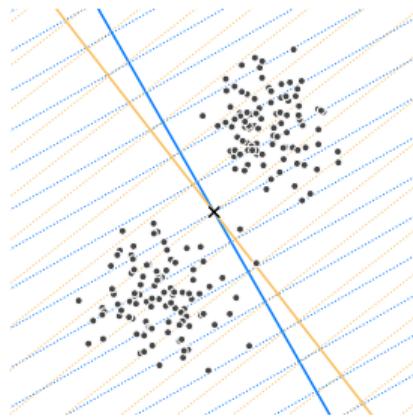
Multiple Random Projections



U_1



U_2



U_3

Locality Sensitive Hashing

- ▶ This idea (multiple random projections) is an example of **Locality Sensitive Hashing** (LSH).
- ▶ We'll explore it more in the next section.

```
class LocalitySensitiveHashing:

    def __init__(self, l, k, d, w):
        self.randomized_grids = []
        for i in range(l):
            U = gaussian_projection_matrix(k, d)
            randomized_grid = NNProjectionGrid(U, w)
            self.randomized_grids.append(randomized_grid)

    def insert(self, p):
        for randomized_grid in self.randomized_grids:
            randomized_grid.insert(p)

    ...
```

```
...  
  
def query_close(self, p):  
    nearby = []  
    for randomized_grid in self.randomized_grids:  
        points_in_cell = randomized_grid.points_in_cell(p)  
        nearby.append(points_in_cell)  
    return np.vstack(nearby)  
  
def query_nn(self, point):  
    results = self.query_close(point)  
    pool = np.vstack([r for r in results])  
    if len(pool) == 0:  
        raise ValueError('No points nearby.')  
    return brute_force_nn(pool, point)
```

Parameters

- ▶ l : number of randomized “grids”
- ▶ k : number of random directions in each “grid”
- ▶ w : bin width

Tuning Parameters

- ▶ Choose so that `.query_close` returns a small # of points.
- ▶ If # is very small (or zero), either:
 - ▶ increase w or ℓ
 - ▶ decrease k

Note

- ▶ This is an approximate NN technique!
- ▶ May not find **the** NN.
- ▶ May not return **anything**!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 7 | Part 4

Theory of Locality Sensitive Hashing

Why does LSH work?

- ▶ Two approaches to understanding LSH.
- ▶ **1) Hashing view.**
- ▶ **2) Dimensionality reduction view.**

Standard Hashing

- ▶ A **hash function** $h : \mathcal{X} \rightarrow \mathbb{Z}$ takes in an object from \mathcal{X} and returns a bucket number.

Standard Hashing

- ▶ **Collision:** two different objects have same hash.
- ▶ Usually, collisions are **bad**.
- ▶ Want similar things to have very different hashes.

Locality Sensitive Hashing

- ▶ But in NN search, we want “close” items to be in the **same bucket** (have same hash).
- ▶ “Far” items should be in **different buckets** (have different hash).

Locality Sensitive Hashing

- ▶ Let r be a distance we consider “close”.
- ▶ Let cr (with $c > 1$) be a distance we consider “far”.
- ▶ Suppose H is a **family** of hash functions.

LSH Family

- ▶ H is an **LSH family** if when h is randomly drawn from H :

$$\mathbb{P}(h(x) = h(y)) \geq p_1 \quad \text{when } d(x, y) \leq r$$

$$\mathbb{P}(h(x) = h(y)) \leq p_2 \quad \text{when } d(x, y) \geq cr$$

where $p_1 > p_2$.

Main Idea

If x and y are close, the probability that they hash to the **same** bin is not too small. If they are far, the probability is not too large.

Example: Random Projections

- ▶ We have seen one LSH family: random projections followed by binning.
- ▶ H has infinitely-many hash functions, one for each direction \vec{u} :

$$h_{\vec{u}}(\vec{p}) = \left\lfloor \frac{\vec{u} \cdot \vec{p}}{w} \right\rfloor,$$

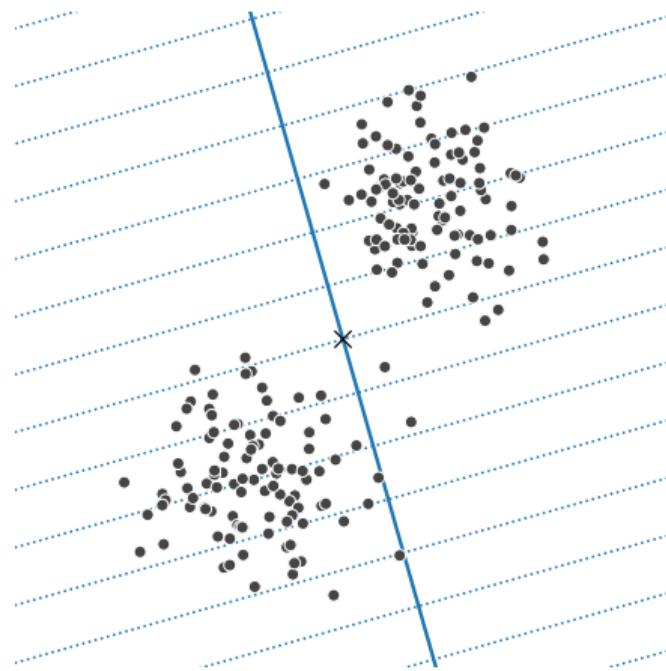
Example: Random Projections

- ▶ Suppose a random hash function h is chosen.
- ▶ Claim:

$$\mathbb{P}(h(x) = h(y)) \geq \frac{1}{2} \quad \text{when } d(x, y) \leq w/2$$

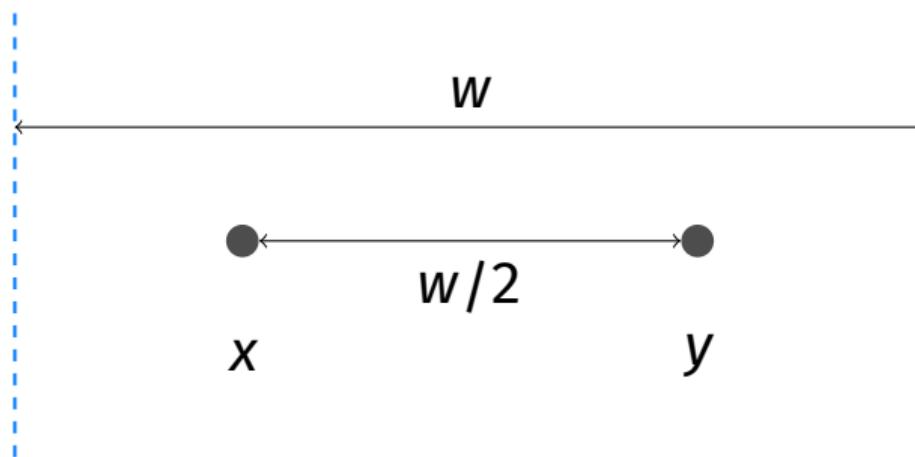
$$\mathbb{P}(h(x) = h(y)) \leq \frac{1}{3} \quad \text{when } d(x, y) \geq 2w$$

Intuition



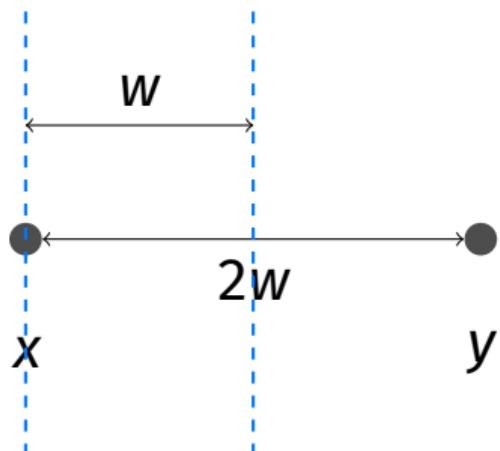
Proof: Close

- ▶ In worst case, grid is orthogonal to line between points.



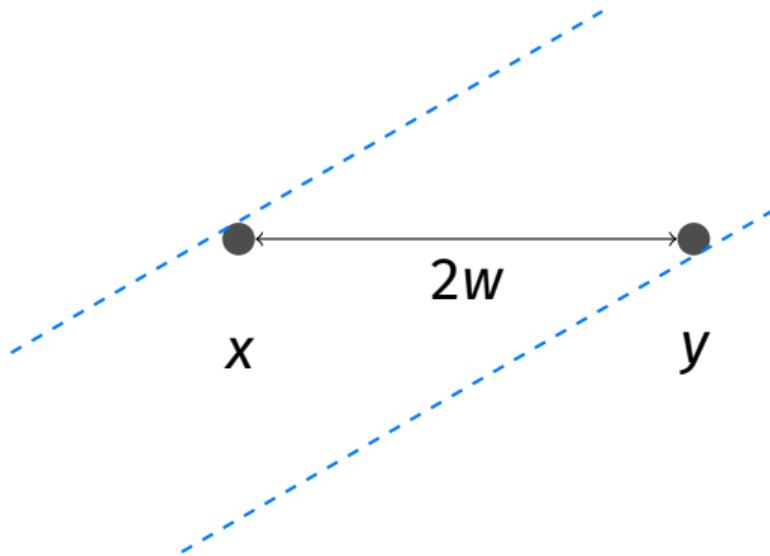
Proof: Far

- ▶ Only possible if grid is close to parallel.



Proof: Far

- ▶ Angle must be below 30° .



Amplification

- ▶ Lots of points have same hash.
- ▶ To be more selective, randomly select k hash functions for cell id.

$$\text{cell-id}(x) = (h_1(x), h_2(x), \dots, h_k(x))$$

Example: Random Projections

- ▶ In case of random projections.

$$\text{cell-id}(\vec{p}) = \left(\underbrace{\left\lfloor \frac{\vec{u}^{(1)} \cdot \vec{p}}{w} \right\rfloor}_{h_1}, \underbrace{\left\lfloor \frac{\vec{u}^{(2)} \cdot \vec{p}}{w} \right\rfloor}_{h_2}, \dots, \underbrace{\left\lfloor \frac{\vec{u}^{(k)} \cdot \vec{p}}{w} \right\rfloor}_{h_k} \right)$$

Collision Probability

- ▶ Remember:

$P(h(x) = h(y)) \geq p_1$ if close.

$P(h(x) = h(y)) \leq p_2$ if far.

- ▶ Collision occurs if $h_i(x) = h_i(y) \forall i \in \{1, \dots, k\}$.
- ▶ Probability of collision...
 - ▶ if close: $\geq p_1^k$
 - ▶ if far: $\leq p_2^k$

Choosing k

- ▶ Want prob. of far points colliding to be small.
- ▶ Say, $1/n$.
- ▶ Set $p_2^k = 1/n$. Then

$$k = \log_{p_2} \frac{1}{n} = \frac{\log n}{\log 1/p_2}$$

Main Idea

We can use $k = \Theta(\log n)$ hash functions.

Main Idea

When using random projections as hash functions, we can use $k = \Theta(\log n)$ directions. This is usually much less than d .

But wait...

- ▶ Probability of close points colliding is p_1^k .
- ▶ Let $p_1 = p_2^\rho$. We'll have $\rho < 1$, since $p_2 < p_1$.
- ▶ Since $p_2^k = \frac{1}{n}$, we have $p_1^k = \frac{1}{n^\rho}$.
- ▶ This is **very small**.

Banding

- ▶ Before: one set of k hash functions.
- ▶ With **banding**: keep ℓ sets (**bands**) of k hash functions.
- ▶ To query NN of p , find points that are in the same cell as p in *any* of the bands.

Banding

- ▶ Probability of at least one match:

$$\underbrace{\frac{1}{n^\rho}}_{\text{collision in band 1}} + \underbrace{\frac{1}{n^\rho}}_{\text{collision in band 2}} + \dots + \underbrace{\frac{1}{n^\rho}}_{\text{collision in band } \ell} = \frac{\ell}{n^\rho}$$

- ▶ Want this to be ≈ 1 , so:

$$\ell = n^\rho$$

Main Idea

We should set the number of bands to be n^ρ . ρ depends on c , and is usually not small. For random projections, $\rho \approx .63$.

Analysis

- ▶ How efficient is LSH?
- ▶ Worst case, everything hashes to same bin: $O(n)$.
- ▶ In practice, much better.
- ▶ Requires **a lot** of memory. $\Theta(ln)$.

Other Distances

- ▶ LSH works for many different similarity measures.
- ▶ Random projections are for Euclidean distances.
- ▶ But other hashing approaches work for cosine distance, Jaccard distance, etc.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 7 | Part 5

The Johnson-Lindenstrauss Lemma

Why does LSH work?

- ▶ Two approaches to understanding LSH.
- ▶ 1) Hashing view.
- ▶ **2) Dimensionality reduction view.**

Main Idea

The **Johnson-Lindenstrauss Lemma** says that, given n points in \mathbb{R}^d , you can reduce the dimensionality to $k \approx \log n$ while still preserving relative distances by randomly projecting onto a set of k unit vectors.

Claim

The **Johnson-Lindenstrauss Lemma** (Informal). Let X be a set of n points in \mathbb{R}^d . Let U be a matrix whose $k = O(\log(n)/\epsilon^2)$ rows are Gaussian random vectors in \mathbb{R}^d . Then for every $\vec{x}, \vec{y} \in X$,

$$\|\vec{x} - \vec{y}\| \leq (1 \pm \epsilon) \|U\vec{x} - U\vec{y}\|$$

LSH and J-L

- ▶ In LSH, we use $k = O(\log n)$ hash functions.
- ▶ If these hash functions are random projections, the J-L lemma tells that distances are largely preserved.

A Different View of LSH

- ▶ Given $p \in \mathbb{R}^d$, randomly project to \mathbb{R}^k with $k \approx \log n$.
- ▶ Let new coordinates be (y_1, y_2, \dots, y_k) .
- ▶ Use standard grid to assign cell id.

Main Idea

LSH (for Euclidean distances) (without banding) can be viewed as dimensionality reduction by random projections, followed by binning into a standard grid.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 7 | Part 6

NN in Practice

In Practice

- ▶ LSH is an important idea.
- ▶ Good performance in practice.
- ▶ But heuristic approaches are often faster.
- ▶ faiss and annoy, among others.

Hierarchical k-Means

Product Quantization

Navigable Small Worlds

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 1

Today's Lecture

Disjoint Sets

- ▶ Often need to keep a collection of **disjoint sets**.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ May need to union disjoint sets.
- ▶ May need to check if two items are in same set.

Use Case

- ▶ We are given a **stream** of nodes, edges.
- ▶ Want to keep track of CCs at every step.
- ▶ BFS/DFS take $\Theta(V + E)$ time; efficient to compute CCs once, but then need to recompute.

Use Cases

- ▶ Used in Kruskal's algorithm for MST.
- ▶ Used in single linkage clustering.
- ▶ Used in Tarjan's algorithm to find LCA in a tree.

Disjoint Sets, Abstractly

- ▶ A **disjoint sets** ADT represents a collection of disjoint sets.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ Supports three operations:
 - ▶ `.make_set()`, `.find_set(x)`, `.union(x, y)`
- ▶ Sometimes called a **Union-Find** data type.

Assumption

- ▶ Elements are consecutive integers.
 - ▶ Example: $\{\{4, 6, 2, 0\}, \{1, 3\}, \{5\}\}$
- ▶ Not really a limitation.
 - ▶ Keep dictionary mapping, e.g., string ids to integers.

.make_set()

- ▶ Create a new singleton set.
- ▶ Element “id” automatically inferred, returned.

```
»> ds = DisjointSet()
»> ds.make_set()
0
»> ds.make_set()
1
»> ds.make_set()
2
```

.union(x, y)

- ▶ Union sets containing x and y.
- ▶ Updates data structure in-place.

```
»> ds = DisjointSet()
»> ds.make_set()
0
»> ds.make_set()
1
»> ds.make_set()
2
»> ds.union(0, 2)
```

.find_set(x)

- ▶ Find **representative** of set containing x.
- ▶ Representative is arbitrary, but same for all items in same set.
- ▶ Used to test if two nodes in same set.
- ▶ Guaranteed to not change unless a union is performed.

```
>> # ds is [{0}, {1}, {2}]
>> ds.union(0, 2)
>> ds.find_set(0)
0
>> ds.find_set(2)
0
>> ds.union(0, 1)
>> ds.find_set(0)
1
>> ds.find_set(1)
1
>> ds.find_set(2)
1
```

Today's Lecture

- ▶ How do we implement a disjoint set?
- ▶ We'll introduce the **disjoint set forest** data structure.
- ▶ Talk about two heuristics that make it very efficient.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 2

Disjoint Set Forests

Implementing Disjoint Sets

- ▶ First idea: a `list of sets`.

`[{2, 4, 3}, {1, 5}, {}]`

- ▶ **Problem:** unioning two `sets` takes time linear in size of smaller.

Looking Ahead

- We'll design data structure so that all operations, including union, take (practically) $\Theta(1)$ time.

The Idea

- ▶ Represent collection as a forest of trees, called a **Disjoint Set Forest**.
- ▶ Example:
 $\{\{2, 4, 3, 6\}, \{1, 5\}, \{0\}\}$
- ▶ Not unique!

Tree Structure

- ▶ Each node has reference to **parent**.
- ▶ Not a binary tree!

Representing Forests

- ▶ We have several choices:
- ▶ 1) Each node is own **object** with parent attribute.
- ▶ 2) Keep a **list** containing parent of each element.

Approach #1

```
class DSFNode:  
  
    def __init__(self, parent=None):  
        self.parent = parent
```

- ▶ `make_set` becomes `DSFNode()`
- ▶ `find_set` and `union` are functions, not methods.
- ▶ They accept `DSFNode` objects.

Approach #2

```
class DisjointSetForest:

    def __init__(self):
        # self._parent[i] is
        # parent of element i
        self._parent = []

    def make_set(self):
        ...

    def find_set(self, x):
        ...

    def union(self, x, y):
        ...
```

Implementation Notes

- ▶ We'll use the second approach.
- ▶ We can use second representation because elements are consecutive integers.
- ▶ For cache locality, use numpy array, not `list`.

.make_set

```
def make_set(self):      »» dsf = DisjointSetForest()
    # infer new element's "id"   »» dsf.make_set()
    x = len(self._parent)     0
    self._parent.append(None)  »» dsf.make_set()
    return x                 1
                                »» dsf.make_set()
                                2
                                »» dsf._parent
                                [None, None, None]
```

`.find_set(x)`

- ▶ Idea: use the “root” as the representative.

.find_set

```
def find_set(self, x):
    if self._parent[x] is None:
        return x
    else:
        return self.find_set(self._parent[x])
```

`.union(x, y)`

- ▶ Idea: make one root the parent of the other.

.union(x, y)

```
def union(self, x, y):          »> # dsf is {{0}, {1}, {2}}
    x_rep = self.find_set(x)    »> dsf._parent
    y_rep = self.find_set(y)    [None, None, None]
    if x_rep != y_rep:          »> dsf.union(0, 1)
        self._parent[y_rep] = x_rep »> dsf._parent
                                    [None, 0, None]
                                    »> dsf.union(1, 2)
                                    »> dsf._parent
                                    [None, 0, 0]
```

Analysis

- ▶ `.make_set`: $\Theta(1)$ time¹
- ▶ `.union`: depends on `.find_set`
- ▶ `.find_set`: $O(h)$, where h is height of tree

¹Amortized, since we're using a dynamic array. But truly $\Theta(1)$ with an over-allocated static array or in the object representation.

Tree Height

- ▶ Trees can be very deep, with $h = O(n)$.
 - ▶ `.find_set` and `.union` can take $\Theta(n)$ time!
- ▶ Example:

```
# dsf is {{0}, {1}, {2}, {3}, {4}}
```

```
»> dsf.union(1, 0)
»> dsf.union(2, 1)
»> dsf.union(3, 2)
»> dsf.union(4, 3)
```

Tree Height

- ▶ But trees can also be shallow, with $h = O(1)$.
- ▶ Example:

```
# dsf is {{0}, {1}, {2}, {3}, {4}}
»> dsf.union(0, 1)
»> dsf.union(1, 2)
»> dsf.union(2, 3)
»> dsf.union(3, 4)
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 3

Path Compression and Union-by-Rank

The Bad News

- ▶ We saw that the tree can become very deep.
- ▶ In worst case, `.find_set` and thus `.union` take $\Theta(n)$ time.

Heuristics

- ▶ Now: two heuristics helping trees stay shallow.
- ▶ **Union-by-Rank** and **Path Compression**
- ▶ Together, these result in a massive speed up.

Path Compression

- ▶ Idea: if we find a long path during `.find_set`, “compress” it to (possibly) reduce height.

.find_set

```
def find_set(self, x):
    if self._parent[x] is None:
        return x
    else:
        root = self.find_set(self._parent[x])
        self._parent[x] = root
    return root
```

Union-by-Rank

- ▶ Should we `.union(x, y)` or `.union(y, x)`?

Union-by-Rank

- ▶ Placing deeper tree under shallower tree increases height by one.
- ▶ But placing shallower tree under deeper tree doesn't increase height.
- ▶ **Idea:** always place shallower tree under deeper.

Rank

- ▶ We need to keep track of height (**rank**) of each tree.
- ▶ Store rank attribute.
- ▶ $\text{rank}[i]$ is height² of tree rooted at node i .

²Exactly the height if path compression isn't used, but upper bound if it is.

Rank

```
class DisjointSetForest:

    def __init__(self):
        self._parent = []
        self._rank = []

    def make_set(self):
        # infer new element's "id"
        x = len(self._parent)
        self._parent.append(None)
        self._rank.append(0)
        return x
```

.union

```
def union(self, x, y):
    x_rep = self.find_set(x)
    y_rep = self.find_set(y)

    if x_rep == y_rep:
        return

    if self._rank[x_rep] > self._rank[y_rep]:
        self._parent[y_rep] = x_rep
    else:
        self._parent[x_rep] = y_rep
        if self._rank[x_rep] == self._rank[y_rep]:
            self._rank[y_rep] += 1
```

Note

- ▶ With path compression, rank is no longer *exactly* the height – it is an upper bound.
- ▶ But this is good enough.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 8 | Part 4

Analysis

Analysis of DSF

- ▶ A DSF with path compression and union-by-rank ensures trees are shallow.
- ▶ How does this affect runtime?

Answer

- ▶ Assuming union-by-rank and path compression...
- ▶ In a sequence of m operations, n of which are
 - .`make_sets`...
- ▶ Amortized cost of a single operation is $O(\alpha(n))$.
- ▶ α is the **inverse Ackermann function**, and it is essentially constant.

Inverse Ackermann

| $\alpha(n)$ | n |
|-------------|--|
| 0 | $n \in [0, 1, 2]$ |
| 1 | $n = 3$ |
| 2 | $n \in [4, \dots, 7]$ |
| 3 | $n \in [8, \dots, 2047]$ |
| 4 | $n \in [2048, \dots, 2^{2048}]$ and beyond |

Proof

- ▶ The formal analysis is quite involved.
- ▶ But we'll provide some intuition.

Union-by-rank Alone

- ▶ Union-by-rank alone ensures height is $O(\log n)$.

```
# dsf is {{0}, {1}, {2}, {3}}
»> dsf.union(0, 1)
»> dsf.union(2, 3)
»> dsf.union(0, 2)
```

Union-by-rank Alone

- ▶ Union-by-rank alone ensures `.find_set` is $O(\log n)$.

Path Compression + U-by-R

- ▶ With path compression, individual `.find_set` calls can take $O(\log n)$.
- ▶ But they massively improve subsequent calls.
 - ▶ For other nodes, too!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 1

Today's Lecture

Algorithms

- ▶ We've been studying data structures.
- ▶ We'll now move towards algorithm design.
- ▶ Data scientists do design algorithms.
- ▶ But perhaps more important to understand solutions to common problems and which problems are difficult.

Today

- ▶ We'll introduce the idea of an **optimization problem**.
- ▶ Talk about one easy strategy that sometimes works.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 2

Optimization Problems and Design Strategies

Optimization Problems

- ▶ We often want to find the **best**.
 - ▶ Shortest path between two nodes.
 - ▶ Minimum spanning tree.
 - ▶ Schedule that maximizes tasks completed.
 - ▶ Line of best fit.
- ▶ These are **optimization problems**.

Example: Regression

- ▶ Given a set of n points in \mathbb{R}^2 , find a straight line $y = mx + b$ which minimizes the Sum of Squared Errors.
- ▶ **Given:** set of n points $\{(x_i, y_i)\}$ in \mathbb{R}^2
- ▶ **Search Space:** all straight lines of form $y = mx + b$
- ▶ **Objective Function:** $\phi(m, b) = \sum_{i=1}^n (y_i - (mx_i + b))^2$

Continuous Optimization

- ▶ Here, the search space is continuous, often **infinite**.
- ▶ Methods for solving often use calculus.

Discrete Optimization

- ▶ Here, the search space is discrete, typically **finite**.
- ▶ Example: shortest path between two nodes.
- ▶ Methods for solving (usually) can't use calculus.
- ▶ We will focus on these problems.

Brute Force

- ▶ If search space is finite, can employ **brute force search.**
- ▶ Typically search space is too large to be feasible.

Design Strategies

- ▶ Focus on **design strategies** for discrete optimization.:
 - ▶ Greedy Algorithms
 - ▶ Backtracking
 - ▶ Dynamic Programming

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 3

The Greedy Approach by Example

Problem

- ▶ **Choose:** 4 numbers with largest sum.

| | | | |
|----|----|----|----|
| 95 | 83 | 80 | 77 |
| 62 | 65 | 55 | 75 |
| 85 | 91 | 70 | 74 |
| 88 | 72 | 59 | 79 |

Specification

- ▶ **Given:** A set X of n numbers and an integer k .
- ▶ **Search Space:** Subsets $S \subset X$ of size k .
- ▶ **Objective:** maximize sum of numbers in S ,

$$\phi(S) = \sum_{s \in S} s$$

Brute Force

- ▶ Brute force: try every possible subset of size k .
- ▶ How many are there?

$$\binom{n}{k} = \Theta(n^k)$$

- ▶ Time complexity is $\Theta(k \cdot n^k)$

The Greedy Approach

| | | | |
|----|----|----|----|
| 95 | 83 | 80 | 77 |
| 62 | 65 | 55 | 75 |
| 85 | 91 | 70 | 74 |
| 88 | 72 | 59 | 79 |

The Greedy Approach

- ▶ At every step, make the best decision at that moment.
- ▶ Is this optimal? Not always, but it is here.

Proof

Let $x_1 \geq \dots \geq x_k$ be the k largest numbers. Let $y_1 \geq \dots \geq y_k$ be some other solution. Since x_1, \dots, x_k are the k largest:

$$x_1 \geq y_1, \quad x_2 \geq y_2, \quad \dots, \quad x_k \geq y_k.$$

Therefore:

$$\sum_{i=1}^k x_i \geq \sum_{i=1}^k y_i$$

Since the other solution was arbitrary, this shows that the greedy solution is at least as good as anything else; therefore it is maximal.

Efficiency

- ▶ Algorithm: loop through once, find k largest numbers.
- ▶ Linear time, $\Theta(n)$.
- ▶ Much faster than $\Theta(k \cdot n^k)$!

A Variation

- ▶ Now you can only choose one number from each row.

| | | | |
|----|----|----|----|
| 95 | 83 | 80 | 77 |
| 62 | 65 | 55 | 75 |
| 85 | 91 | 70 | 74 |
| 88 | 72 | 59 | 79 |

Specification

- ▶ **Given:** An $n \times n$ matrix X of numbers and an integer k .
- ▶ **Search Space:** Subsets $S \subset X$ of size k where each element is from a different row of X .
- ▶ **Objective:** maximize sum of numbers in S .

$$\phi(S) = \sum_{s \in S} s$$

Optimality

- ▶ The greedy approach of choosing largest within each row is optimal.

Another Variation

- ▶ Now you can only choose one from each row/column.

| | | | |
|----|----|----|----|
| 95 | 83 | 80 | 77 |
| 62 | 65 | 55 | 75 |
| 85 | 91 | 70 | 74 |
| 88 | 72 | 59 | 79 |

Specification

- ▶ **Given:** An $n \times n$ matrix X of numbers and an integer k .
- ▶ **Search Space:** all subsets of entries of X of size k such that each element is in a different row/column of X .
- ▶ **Objective:** maximize sum of numbers in subset.

$$\phi(S) = \sum_{s \in S} s$$

Greedy is not Optimal

- ▶ The optimal solution is: $80 + 75 + 91 + 88 = 334$

| | | | |
|----|----|----|----|
| 95 | 83 | 80 | 77 |
| 62 | 65 | 55 | 75 |
| 85 | 91 | 70 | 74 |
| 88 | 72 | 59 | 79 |

Main Idea

For some problems, a greedy approach is guaranteed to find the optimal solution. For other problems, it is not.

Main Idea

Coming up with a greedy algorithm is usually simple – proving that it finds the optimal may not be so easy.

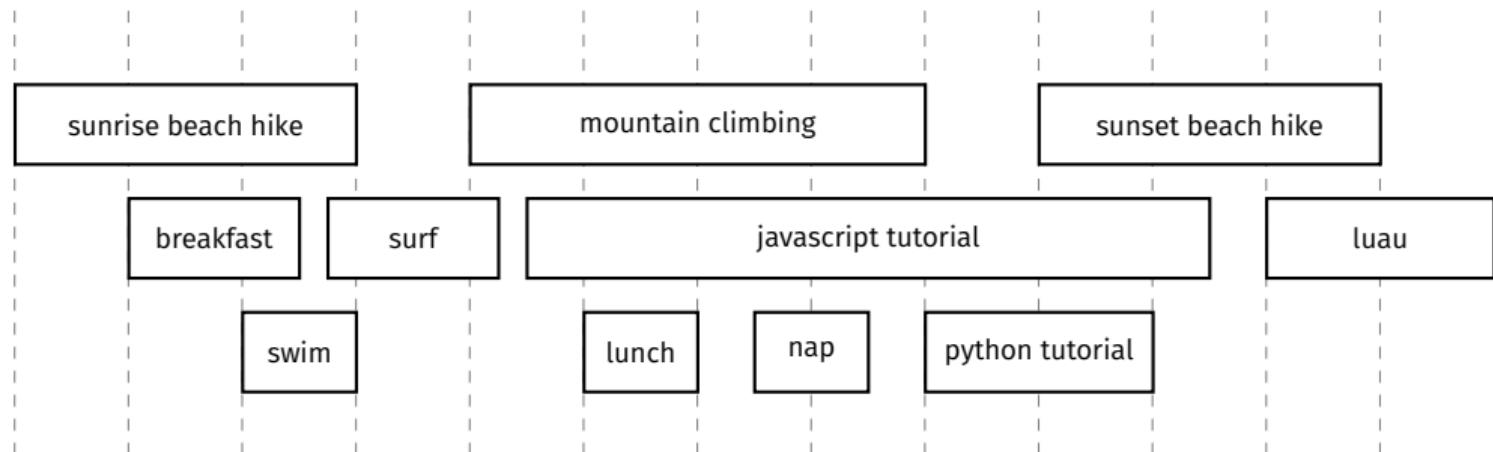
DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 4

Activity Selection Problem

Vacation Planning



Formalized

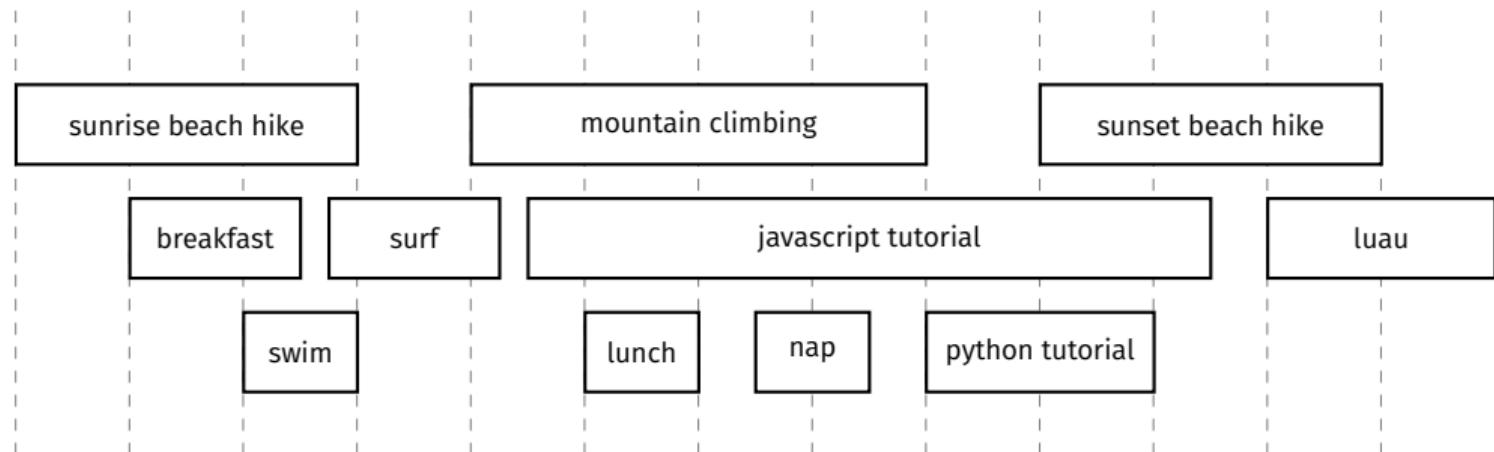
- ▶ This is called the **activity selection** problem.
- ▶ **Given:** a set of start/finish times (s_i, f_i) for n events
- ▶ **Search Space:** all schedules S with non-overlapping events
 - ▶ Format: S is a set of event indices e_1, e_2, \dots, e_k
- ▶ **Objective:** maximize $|S|$ (number of events)

$$\phi(S) = |S|$$

Greedy Strategies

- ▶ There are several strategies we might call “greedy”.
- ▶ Approach #1: in order of duration, shortest events first.

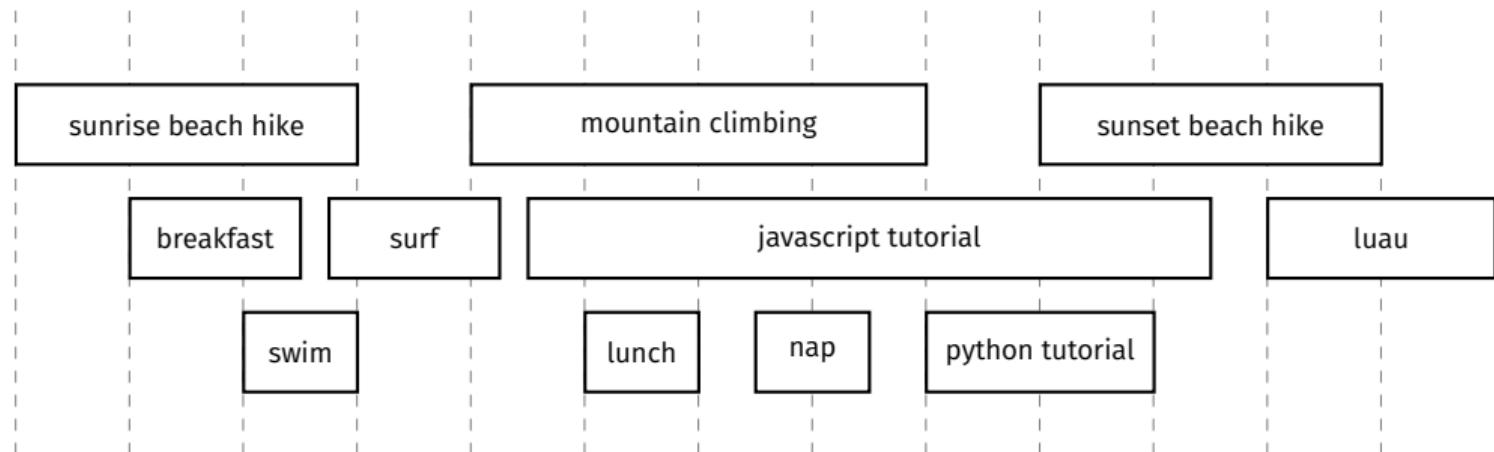
In Order of Duration



Greedy Strategies

- ▶ Approach #2: in order of start time.

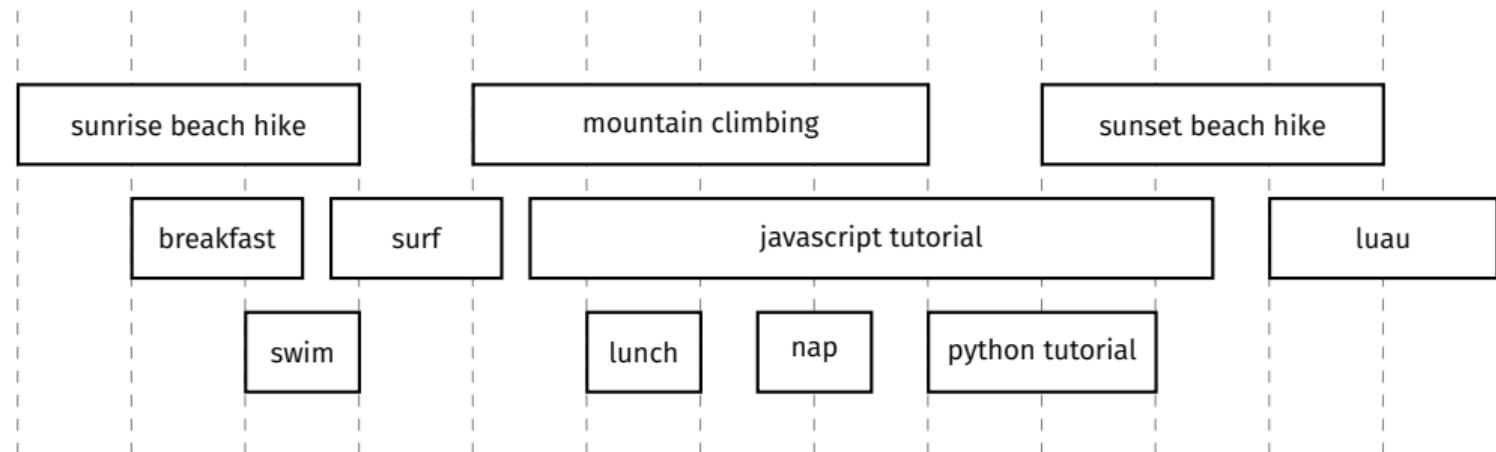
In Order of Start Time



Greedy Strategies

- ▶ Approach #3: in order of finish time.

In Order of Finish Time



In Order of Finish Time

- ▶ Choose event with earliest finish time as first event.
- ▶ Choose subsequent events in order of finish time.
 - ▶ provided that they are non-overlapping.
- ▶ This is **guaranteed** to find global optimum.
- ▶ But how do we know this?

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 5

Exchange Arguments

Convincing Yourself

- ▶ Designing a greedy algorithm is usually easy.
- ▶ It can be hard to convince yourself that it is optimal.
- ▶ Now, one proof technique: **exchange arguments**.

First: Proving Non-Optimality

- ▶ To show that a strategy is **non-optimal**, find a counterexample.

Proving Optimality

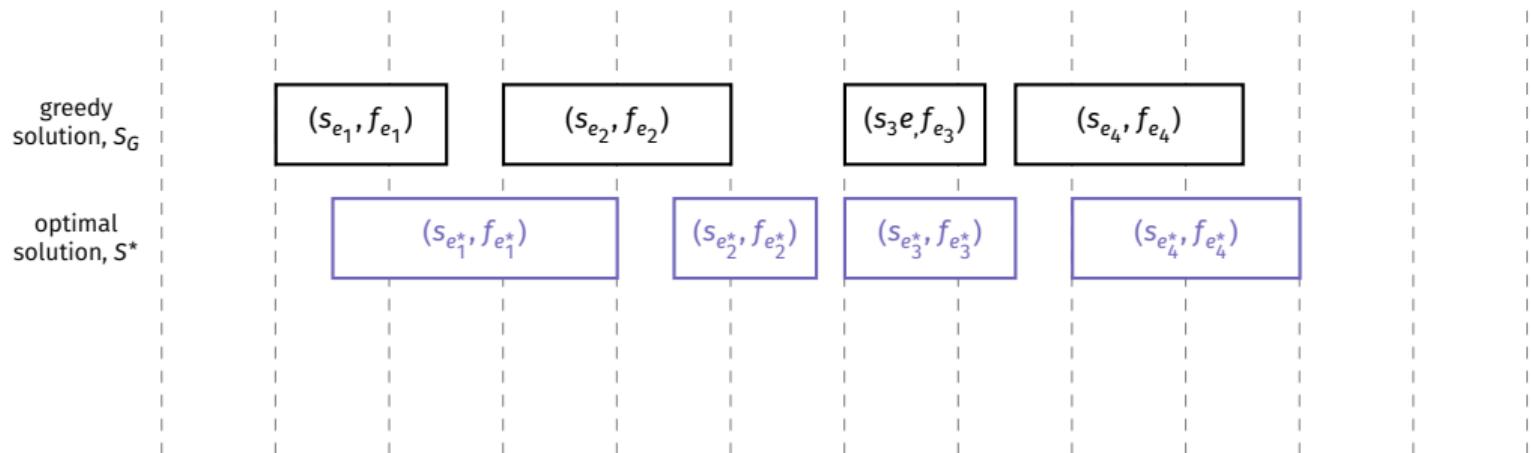
- ▶ There may be many optimal solutions – we want to show that the greedy solution S_G is always one of them.

Exchange Arguments

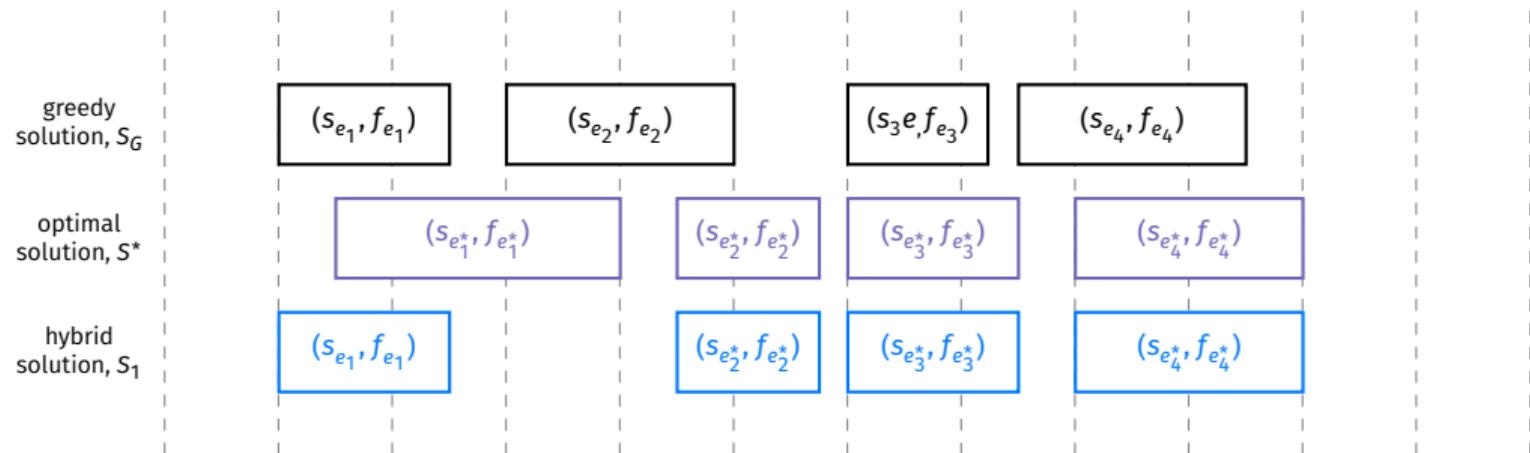
- ▶ Start with an arbitrary optimal solution, S^* .
- ▶ Make a **chain** of optimal solutions $S^*, S_1, S_2, \dots, S_G$
- ▶ At every step from S_{k-1} to S_k :
 - ▶ construct solution S_k by **exchanging** part of S_{k-1} with S_G
 - ▶ argue that S_k is **valid**¹
 - ▶ argue that S_k is **also optimal**
- ▶ Proves S_G is optimal, as
 $\phi(S^*) = \phi(S_1) = \phi(S_2) = \dots = \phi(S_G)$

¹It is part of the search space and meets all constraints.

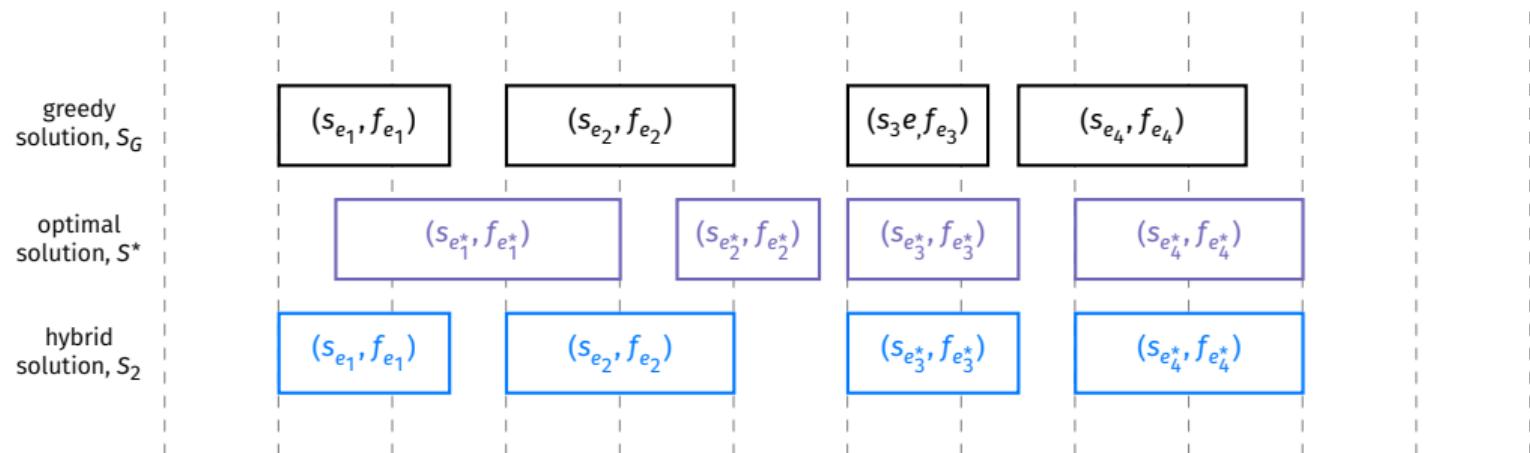
Exchange Argument for Activities



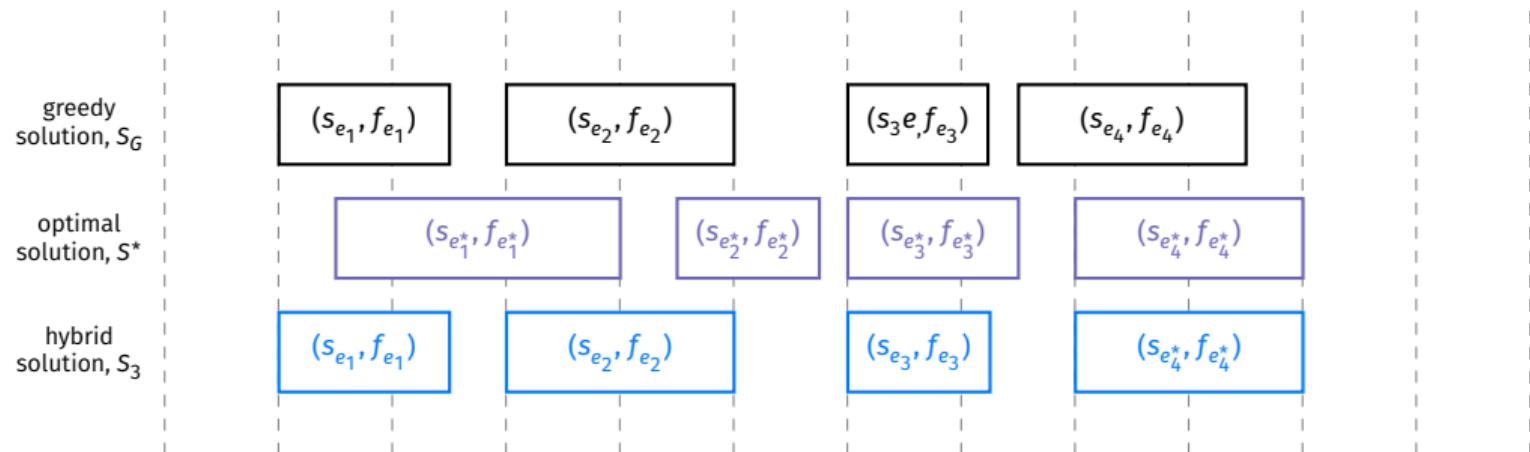
Exchange Argument for Activities



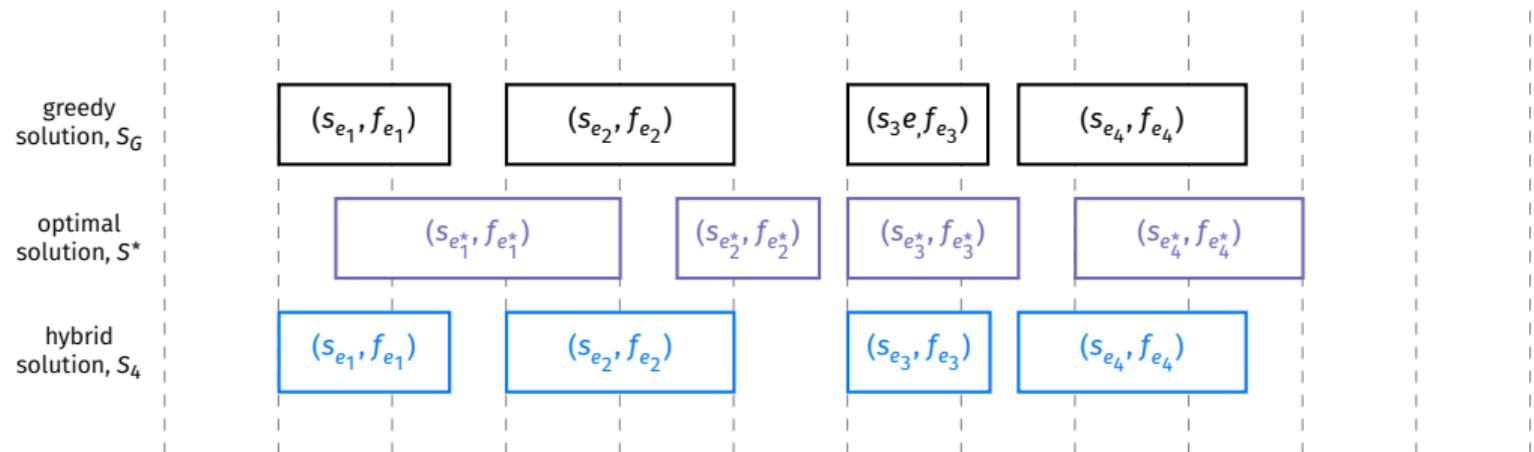
Exchange Argument for Activities



Exchange Argument for Activities



Exchange Argument for Activities



Exchange Argument for Activities

Take an arbitrary optimal solution S^* . Suppose it is different from the greedy solution, S_G (as otherwise we're done).

If it's different, it has to be different *somewhere*. Let's look at the first event in S^* that is not in S ; call this the i th event in S^* .

We'll exchange the i th event in S^* with the i th event in S_G , but we have to be a little careful: what if $|S^*| > |S_G|$, so that it's possible that S_G has no i th element? So there are two cases: $i \leq |S_G|$ and $i > |S_G|$.

Exchange Argument for Activities

First case: $i \leq |S_G|$. Then exchange the i th event in S^* with the i th event in S_G , creating a new solution S' .

This is **valid**: the event from S_G cannot overlap with any of the events in S^* , since the previous $i - 1$ events in S^* are the same as in S_G (and they didn't overlap), and the finish time of the greedy event is \leq the finish time of event it is replacing, so it cannot overlap with the remaining events.

It is also **optimal**, since $|S'| = |S^*|$.

Exchange Argument for Activities

Second case: $i > |S_G|$. This means that there is at least one “extra” event in S^* than in S_G .

But this cannot happen: this extra event does not overlap with the events in S_G (since S_G is equal to the first $i - 1$ elements of S^* , and the “extra” event doesn’t overlap with them). Its finish time is larger than any event in S_G . So the greedy approach would have included this event. Thus this case is not possible.

Exchange Argument for Activities

In either case, S' is a valid optimal schedule.

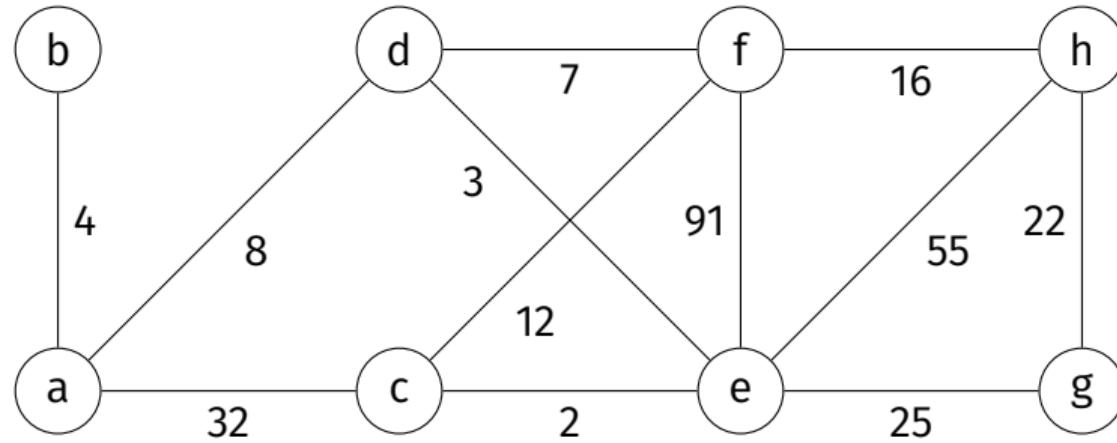
S^* and S_G can differ in only a finite number of places; therefore, repeating this procedure a finite number of times produces a chain of optimal solutions where each solution is more similar to S_G . The chain terminates when S_G is reached, which shows that S_G is optimal ($|S_G| = |S^*|$).

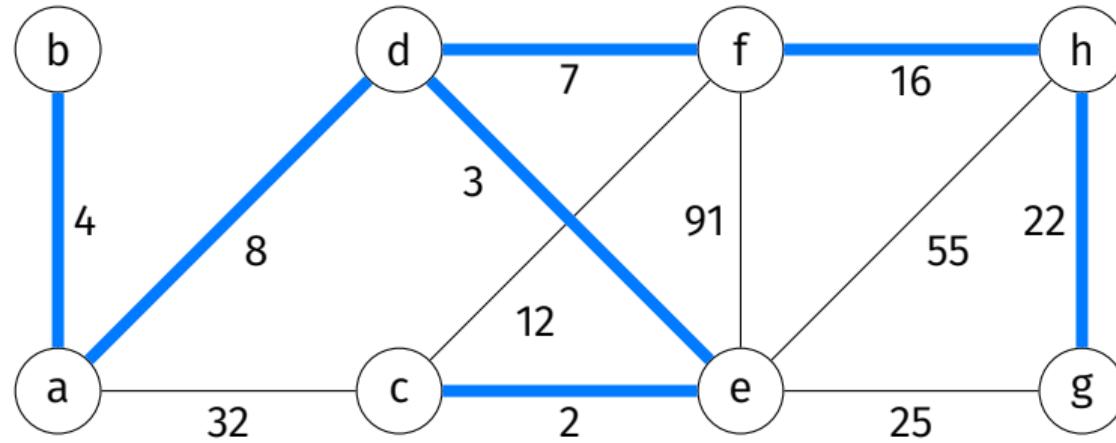
DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 6

Minimum Spanning Trees

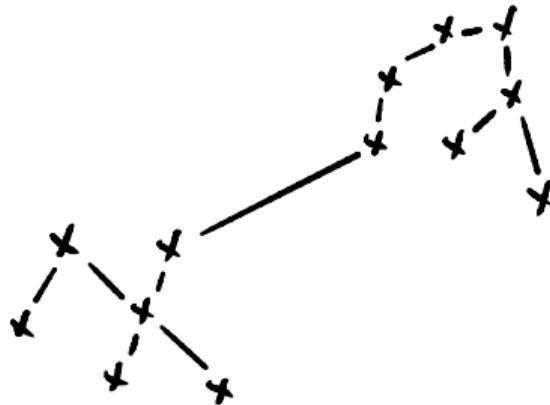




MSTs and Clustering



MSTs and Clustering



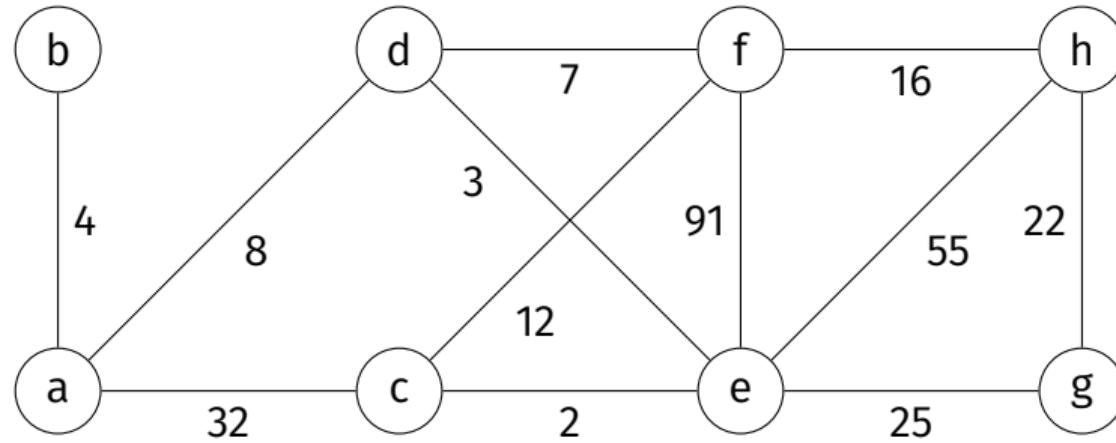
Minimum Spanning Trees

- ▶ **Given:** a weighted graph $G = (V, E, \omega)$, where $\omega : E \rightarrow \mathbb{R}$.
- ▶ **Search Space:** all **spanning trees** $T = (V, E')$, where $E' \subset E$.
- ▶ **Objective:** minimize total edge weight

$$\phi(T) = \sum_{e \in E'} \omega(e)$$

Kruskal's Algorithm

- ▶ **Kruskal's Algorithm** is a **greedy** algorithm for computing a MST.
- ▶ Idea: add edges one-by-one in order of weight.
 - ▶ But only if edge does not make a cycle!



Kruskal's Algorithm (Pseudocode)

```
def kruskals(graph, weight):
    mst = UndirectedGraph()
    edges = sorted(graph.edges, key=weight)

    for (u, v) in edges:
        if u and v are not connected in mst:
            mst.add_edge(u, v)

    return mst
```

Implementing Kruskal's Algorithm

```
def kruskals(graph, weight):
    mst = UndirectedGraph()
    edges = sorted(graph.edges, key=weight)

    dsf = DisjointSetForest()
    for i in range(len(graph.nodes)):
        dsf.make_set()

    for (u, v) in edges:
        if dsf.find_set(u) != dsf.find_set(v):
            mst.add_edge(u, v)
            dsf.union(u, v)

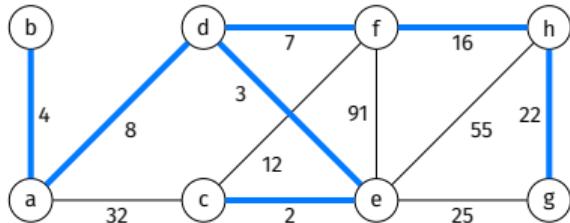
    return mst
```

Optimality

- ▶ Kruskal's Algorithm find an optimal solution.
- ▶ We can prove this with an **exchange argument**.

Notes

- ▶ The greedy approach produces a valid spanning tree.
- ▶ Any two spanning trees have same number of edges.
- ▶ Removing an edge from a MST partitions nodes in two.



Exchange Idea

- ▶ Suppose $e^* = (u, v)$ is in T^* , but not in T .
- ▶ We'll find a node e on the path from u to v in T .
- ▶ Make a new tree, T' , by taking T^* , removing e^* , replacing it with e .

Exchange Argument

Let T^* be any minimum spanning tree, and let T_G be a tree produced by Kruskal's algorithm. Suppose that T^* and T_G are different, and let $e^* = (u, v)$ be an edge in T^* that is not in T_G .

Consider the path from u to v in T_G . Adding e^* to T_G would create two different paths from (u, v) , and thus a cycle. Let (A, B) be the cut produced if e^* were removed from T^* , and let e be an edge along the cycle that crosses the cut (A, B) (there must be at least one).

We will exchange e^* in T^* for the edge e .

Exchange Argument

First, this will create a **valid** spanning tree. Removing e^* in T^* breaks the tree into two connected components with disjoint node sets A and B . Since e crosses (A, B) , adding it will re-connected the disconnected components, and thus form a spanning tree, T' .

Second, the new tree is **also optimal**. We claim that $\omega(e') \geq \omega(e)$. At the time e' was considered by Kruskal's, it was rejected because it would create a cycle. Meaning that edge e was already added, implying that $\omega(e) \leq \omega(e^*)$. As such, replacing e^* with e can only decrease or maintain² the total edge weight. Thus T' must be optimal. Repeat this process, creating a chain of trees $T^*, T_1, T_2, \dots, T_G$. Since each tree is optimal, T_G is as well.

²In fact, it must maintain. Decreasing would contradict fact that T^* is optimal.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 9 | Part 7

Designing Greedy Algorithms

Designing Algorithms

- ▶ When do we know to use a greedy algorithm?
- ▶ It isn't always obvious.

A Pattern

- ▶ Our examples have a common pattern: sort by some attribute, then loop through.
 - ▶ Number grid: take numbers in descending order.
 - ▶ Activities: take activities in increasing order of finish time.
 - ▶ MST: take edges in increasing order of weight.
- ▶ This is a new justification for value of sorting.
- ▶ Suggestion: when tackling a problem, try sorting first.

Greedy Approximations

- ▶ A greedy algorithm can be useful, even if not guaranteed to produce optimal answer.
- ▶ Especially true if exact algorithms are **slow**.
- ▶ Example: k -means clustering (Lloyd's algorithm)

k-means Problem

- ▶ **Given:** n data points X in \mathbb{R}^d , parameter k .
- ▶ **Search Space:** all clusterings $C = \{X_1, \dots, X_k\}$ of X into k disjoint sets.
- ▶ **Objective function:** minimize

$$\phi(C) = \sum_{i=1}^k \sum_{x \in X_i} (x - \text{mean}(X_i))^2$$

Greedy Algorithm

- ▶ Lloyd's algorithm (a.k.a., the “k-means algorithm”) is a greedy algorithm for minimizing the k -means objective.
- ▶ Start with k centroids, μ_1, \dots, μ_k .
- ▶ At each step, let X_i be set of points closest to μ_i , update μ_i to be $\text{mean}(X_i)$, repeat until convergence.
- ▶ Each step decreases value of objective function.

Optimality

- ▶ Lloyd's algorithm is **not** guaranteed to find optimum.
- ▶ Then again, no feasible algorithm is.
- ▶ Used in practice because it is fast and “good enough”.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 1

Today's Lecture

Beyond Greedy

- ▶ Greedy algorithms are typically **fast**, but may not find the optimal answer.
- ▶ Brute force guarantees the optimal answer, but is **slow**.
- ▶ Can we guarantee the optimal answer and be faster than brute force?

Today

- ▶ The **backtracking** idea.
- ▶ It is a useful, general algorithm design technique¹.
- ▶ And the foundation of **dynamic programming**.

¹Commonly seen in tech interviews

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 2

The 0-1 Knapsack Problem

0-1 Knapsack

- ▶ Suppose you're a thief.
- ▶ You have a knapsack (bag) that can fit 100L.
- ▶ And a list of n things to possibly steal.

| item | size (L) | price |
|---------|----------|-------|
| TV | 50 | \$400 |
| iPad | 2 | \$900 |
| Printer | 10 | \$100 |
| : | : | : |

- ▶ Goal: maximize total value of items you can fit in your knapsack.

Example

| item | size (L) | price |
|------|----------|-------|
| 1 | 50 | \$40 |
| 2 | 10 | \$25 |
| 3 | 80 | \$100 |
| 4 | 5 | \$10 |
| 5 | 20 | \$20 |
| 6 | 30 | \$6 |
| 7 | 8 | \$32 |
| 8 | 17 | \$34 |

In the bag: _____

Total value: _____

Space remaining: _____

Greedy

- ▶ Does a greedy approach find the optimal?
- ▶ What do we mean by “greedy”?
- ▶ Idea #1: take most expensive available that will fit.

Example

| item | size (L) | price |
|------|----------|-------|
| 1 | 50 | \$40 |
| 2 | 10 | \$25 |
| 3 | 80 | \$100 |
| 4 | 5 | \$10 |
| 5 | 20 | \$20 |
| 6 | 30 | \$6 |
| 7 | 8 | \$32 |
| 8 | 17 | \$34 |

In the bag: _____

Total value: _____

Space remaining: _____

Greedy, Idea #2

- ▶ We want items with high value for their size.
- ▶ Define “price density” =
`item.price / item.size`
- ▶ Idea #2: take item with highest price density.

Example

| item | size (L) | price |
|------|----------|-------|
| 1 | 50 | \$40 |
| 2 | 10 | \$25 |
| 3 | 80 | \$100 |
| 4 | 5 | \$10 |
| 5 | 20 | \$20 |
| 6 | 30 | \$6 |
| 7 | 8 | \$32 |
| 8 | 17 | \$34 |

In the bag: _____

Total value: _____

Space remaining: _____

Greedy is Not Optimal

- ▶ Claim: the best possible total value is \$157.
 - ▶ Items 2, 3, and 7.

Never Looking Back

- ▶ Once greedy makes a decision, it never looks back.
- ▶ This is why it may be suboptimal.
- ▶ **Backtracking**: go back to reconsider every previous decision.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 3

Backtracking

Backtracking

- ▶ Reconsider every decision.
- ▶ If we initially tried including x , also try *not* including x .

Backtracking

```
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x should be in bag, see what we get
    best_with = ...

    # backtrack: now assume x should not be in bag, see what we get
    best_without = ...

    return max(best_with, best_without)
```

Recursive Subproblems

- ▶ What is `BEST(items, bag_size)` if we assume that `x` **is** in the bag?
- ▶ Imagine choosing `x`.
 - ▶ Your current total value is `x.price`.
 - ▶ You have `bag_size - x.size` space left.
 - ▶ Items left to choose from: `items - x`.
- ▶ Clearly, you want the best outcome for *new* situation (subproblem).
- ▶ Answer: `x.price + BEST(items - x, bag_size - x.size)`

Recursive Subproblems

- ▶ What is $\text{BEST}(\text{items}, \text{bag_size})$ if we assume that x **is not** the bag?
- ▶ Imagine deciding x is not in the bag.
 - ▶ Your current total value is 0 .
 - ▶ You have bag_size space left.
 - ▶ Items left to choose from: $\text{items} - x$.
- ▶ Clearly, you want the best outcome for *new* situation (subproblem).
- ▶ Answer: $0 + \text{BEST}(\text{items} - x, \text{bag_size})$

Backtracking

```
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x is in the bag, see what we get
    best_with = x.price + knapsack(items - x, bag_size - x.size)

    # now assume x is not in bag, see what we get
    best_without = 0 + knapsack(items - x, bag_size)

return max(best_with, best_without)
```

Backtracking

```
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

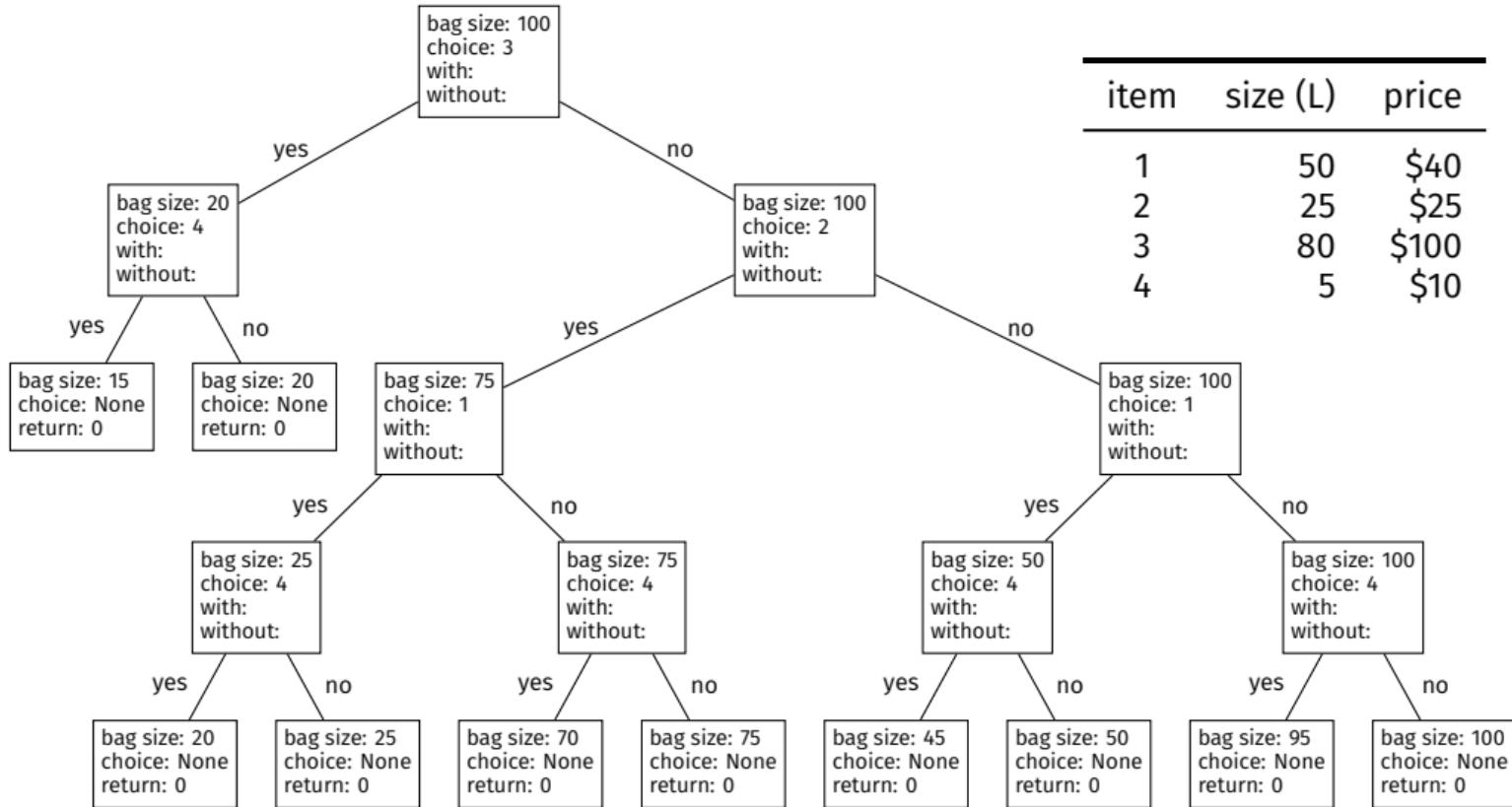
    # if None, it means there was no item that fit
    if x is None:
        return 0

    items.remove(x)
    best_with = x.price + knapsack(items, bag_size - x.size)
    best_without = knapsack(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

Backtracking

- ▶ **Backtracking**: go back to reconsider every previous decision.
- ▶ Searches the whole tree.
- ▶ Can be thought of as a DFS on implicit tree.



Exercise

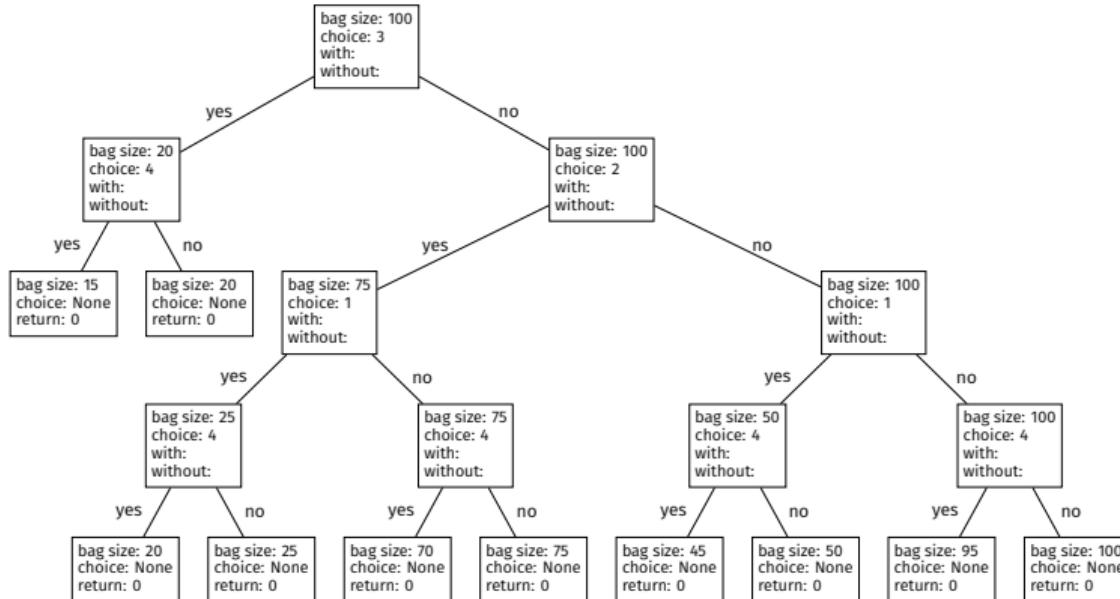
Is the backtracking solution guaranteed to find an optimal solution?

Yes!

- ▶ It tries every **valid** combination and keeps the best.
 - ▶ A combination of items is valid if they fit in the bag together.

Leaf Nodes

- Each leaf node is a different valid combination.



Exercise

Suppose instead of choosing an arbitrary node we choose most expensive. Does the answer change?

No!

- ▶ The choice of node is arbitrary.
- ▶ Call tree will change, but all valid combinations are tried.

Exercise

How does backtracking relate to the greedy approach? How would you change the code to make it greedy?

Summary

```
def knapsack_greedy(items, bag_size):
    # choose greedily
    x = items.most_valuable_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x is in the bag, see what we get
    best_with = x.price + knapsack(items - x, bag_size - x.size)

    # in the greedy approach, we don't do this
    # best_without = knapsack(items - x, bag_size)

return best_with
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 4

Efficiency Analysis

A Benchmark

- ▶ Brute force: try every **possible** combination of items.
 - ▶ Even the **invalid** ones whose total size is too big.
 - ▶ Why? Hard to know which are invalid without trying them.
- ▶ There are $\Theta(2^n)$ possible combinations.
- ▶ So brute force takes $\Omega(2^n)$ time. **Exponential** :(

Time Complexity of Backtracking

```
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    items.remove(x)
    best_with = x.price + knapsack(items, bag_size - x.size)
    best_without = knapsack(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

$$T(n) =$$

Backtracking Takes Exponential Time

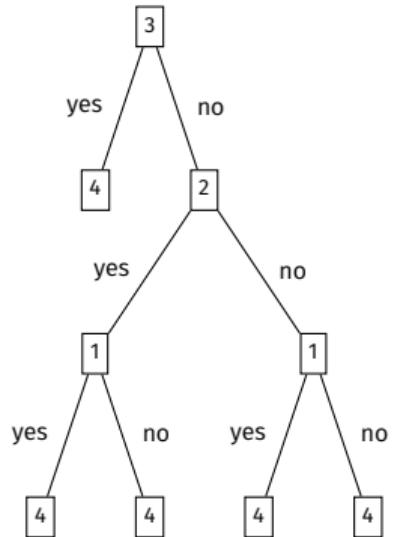
- ▶ ...in the worst case.
- ▶ This is just as bad as **brute force**.
- ▶ So why use it?
- ▶ Its worst case isn't always indicative of its practical performance.

Intuition

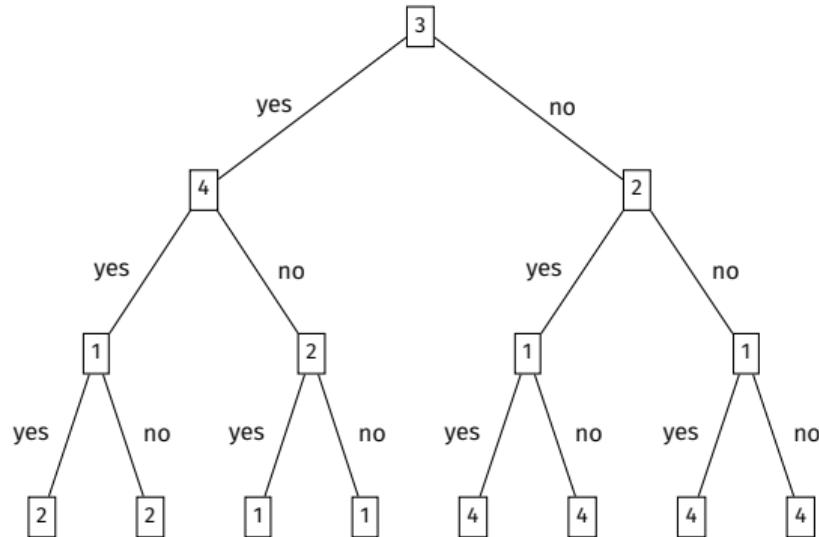
- ▶ Brute force tries all **possible** combinations.
- ▶ Backtracking tries all **valid** combinations.
- ▶ The number of valid combinations can be much less than the number of possible combinations.²

²Not always true!

Pruning



backtracking



brute force

Pruning

- ▶ Backtracking **prunes** branches that lead to invalid solutions.

Example

- ▶ 23 items with size/price chosen from $\text{Unif}([23, \dots, 46])$
- ▶ Bag size is 46
- ▶ Brute force: 52 seconds.
- ▶ Backtracking: 4 milliseconds.

Example

- ▶ 300 items with size/price chosen from $\text{Unif}([150, \dots, 300])$
- ▶ Bag size is 600
- ▶ Brute force: ? ($\approx 4.6 \times 10^{77}$ years)
- ▶ Backtracking: 30 seconds.

Backtracking Worst Case

- ▶ knapsack's **worst case** is when bag size is very large.
- ▶ All solutions are valid, aren't pruned.
- ▶ But this is actually an easy case!

```
def knapsack_2(items, bag_size):
    if sum(item.size for item in items) < bag_size:
        return sum(item.price for item in items)

    x = items.arbitrary_item(fitting_in=bag_size)

    if x is None:
        return 0

    items.remove(item)
    best_with = x.price + knapsack_2(items, bag_size - x.size)
    best_without = knapsack_2(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

Pruning

- ▶ This further prunes the tree, resulting in speedup for some inputs.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 5

Branch and Bound

Example

- ▶ Suppose you have a bag of size 100.
- ▶ One of the items is a diamond.
 - ▶ Price: \$10,000. Size: 1
- ▶ The other 49 items are coal.
 - ▶ Price: \$1. Size: 1
- ▶ Do you even consider not taking the diamond?

Idea

- ▶ Assume we take the diamond, compute best result.
- ▶ Find quick upper bound for not taking diamond.
- ▶ If upper bound is less than best for diamond, don't go down that branch.
- ▶ This is **branch and bound**; another way to prune tree.

Branch and Bound

```
def knapsack_bb(items, bag_size, find_upper_bound):
    # try to make a good first choice
    x = items.item_with_highest_price_density(fitting_in=bag_size)

    if x is None:
        return 0

    items.remove(item)
    best_with = x.price + knapsack_bb(items, bag_size - x.size)

    if find_upper_bound(items, bag_size) < best_with:
        best_without = 0
    else:
        best_without = knapsack_bb(items, bag_size)

    items.replace(x)

    return max(best_with, best_without)
```

Example

| item | size (L) | price |
|------|----------|--------|
| 1 | 50 | \$40 |
| 2 | 25 | \$25 |
| 3 | 95 | \$1000 |
| 4 | 5 | \$10 |

Upper Bounds for Knapsack

- ▶ How do we get a good upper bound?
- ▶ One idea: the solution to the *fractional knapsack* problem upper bounds that for 0/1 knapsack.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 6

Summary

Summary

- ▶ A backtracking approach is **guaranteed** to find an optimal answer.
- ▶ It is typically faster than brute force, but can still take **exponential time**.

Summary

- ▶ We can speed up backtracking by pruning:
- ▶ Three ways to prune:
 1. Prune invalid branches (default).
 2. Prune “easy” cases.
 3. Prune by branching and bounding.

Summary

- ▶ Next time: **dynamic programming.**
- ▶ We'll see it is just backtracking + memoization.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 1

Today's Lecture

Where are we?

- ▶ We've been studying algorithm design.
- ▶ **Greedy algorithms**
 - ▶ Typically fast
 - ▶ But only guaranteed to find optimal answer for a select few problems (e.g., activity scheduling)
- ▶ **Backtracking**
 - ▶ Usually have bad worst case (exponential!)
 - ▶ But are guaranteed to find optimal answer.

Today

- ▶ **Dynamic Programming**: backtracking + memoization.
- ▶ Just as general as backtracking.
- ▶ And for some problems, **massively faster**.
- ▶ A “sledgehammer” of algorithm design.¹

¹Dasgupta, Papadimitriou, Vazirani

Today

- ▶ A new problem: weighted activity scheduling.
- ▶ We'll design a **dynamic programming** solution in steps:
 1. Backtracking solution.
 2. “Nicer” backtracking with repeating subproblems.
 3. Give backtracking algorithm a short-term memory.
- ▶ We'll turn an **exponential** time algorithm to **linear** by adding 2 lines of code.

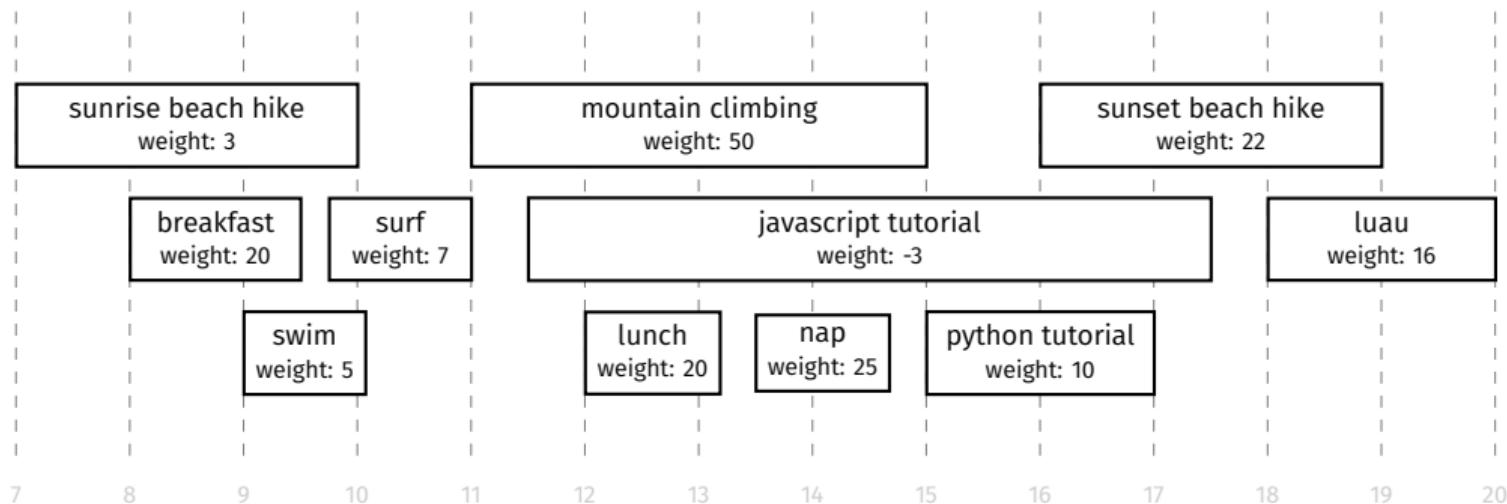
DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 2

Weighted Activity Selection Problem

Vacation Planning



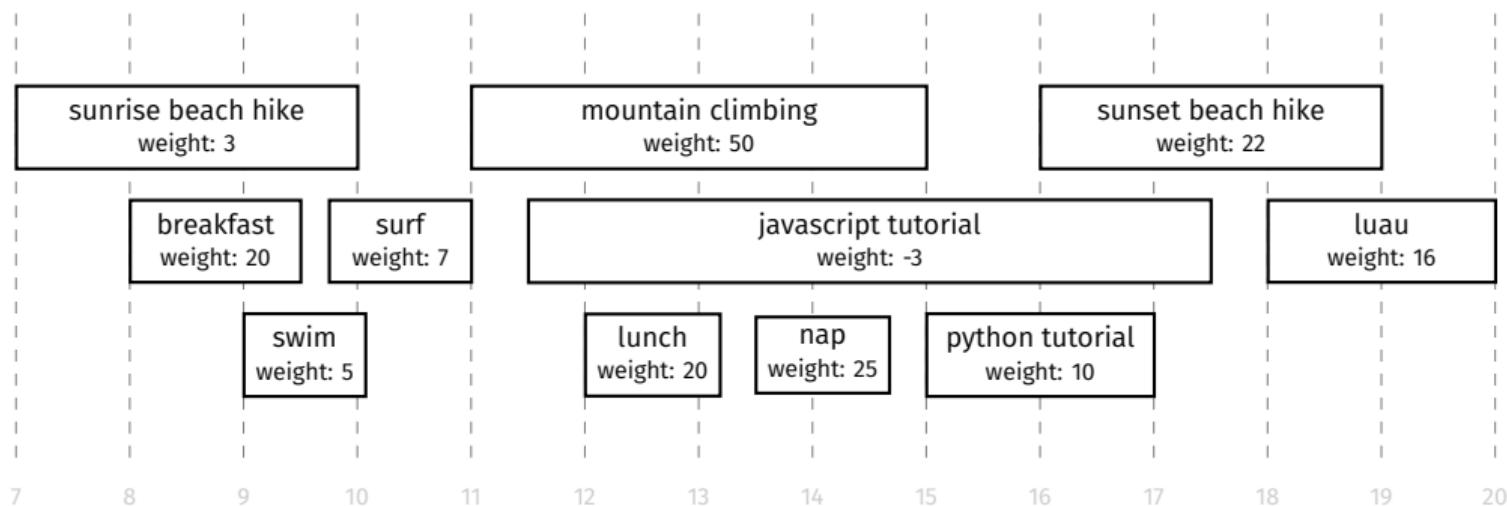
Weighted Activity Selection Problem

- ▶ **Given:** a set of activities each with start, finish, weight.
- ▶ **Goal:** Choose set of compatible activities so as to maximize total weight.

Greedy?

- ▶ Remember the *unweighted* problem: maximize total number of activities.
- ▶ Greedy solution: take compatible activity that finishes earliest, repeat.
- ▶ This was **guaranteed** to find optimal in that problem.
- ▶ It **may not** find optimal for weighted problem.

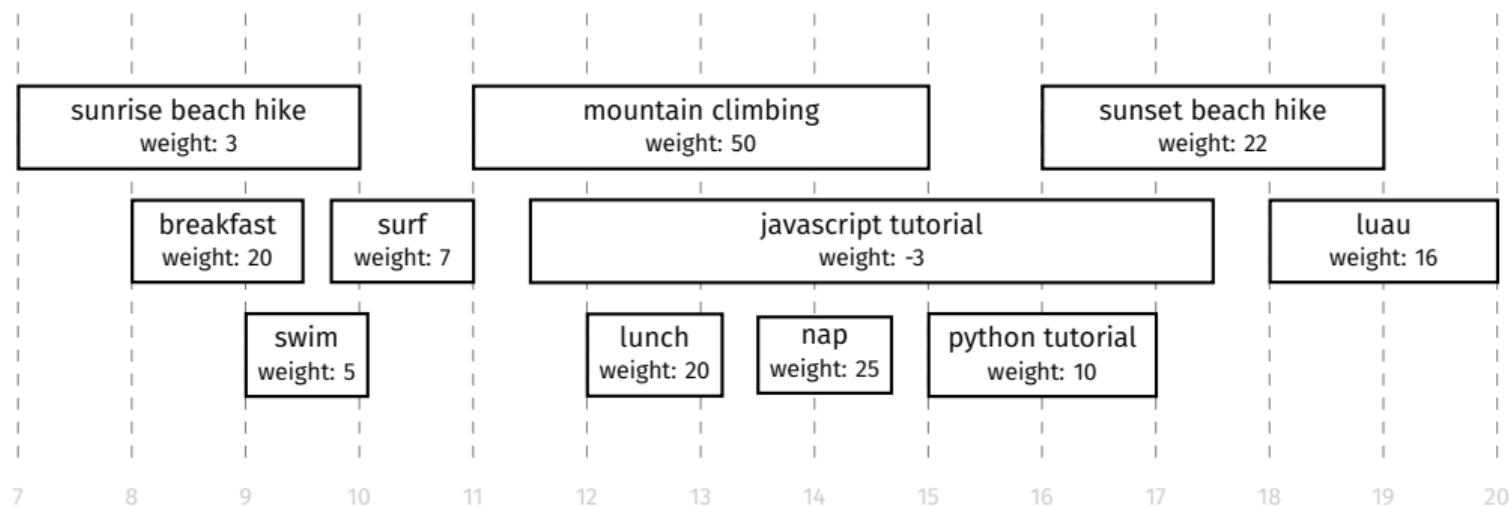
Greedy?



Greedy?

- ▶ Maybe a different greedy approach works?
- ▶ Idea: take compatible activity with **largest weight**.

Greedy?



Don't be greedy!

- ▶ The greedy approach is **not guaranteed** to find best.
- ▶ Note: you might get lucky on a particular instance!

What now?

- ▶ We'll try **backtracking**.
- ▶ It will take **exponential time**.
- ▶ But with a small change, we'll get a **linear time** algorithm that is guaranteed to find the best!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 3

Step 01: Backtracking Solution

Backtracking

- ▶ We'll build up a schedule, one activity at a time.
- ▶ Choose an arbitrary activity, x .
 - ▶ Recursively see what happens if we **do** include x .
 - ▶ Recursively see what happens if we **don't** include x .
- ▶ This will try **all valid schedules**, keep the best.

Backtracking

```
def mwsched_bt(activities):
    if not activities:
        return 0

    # choose arbitrary activity
    x = activities.choose_arbitrary()

    # best with x
    best_with = ...

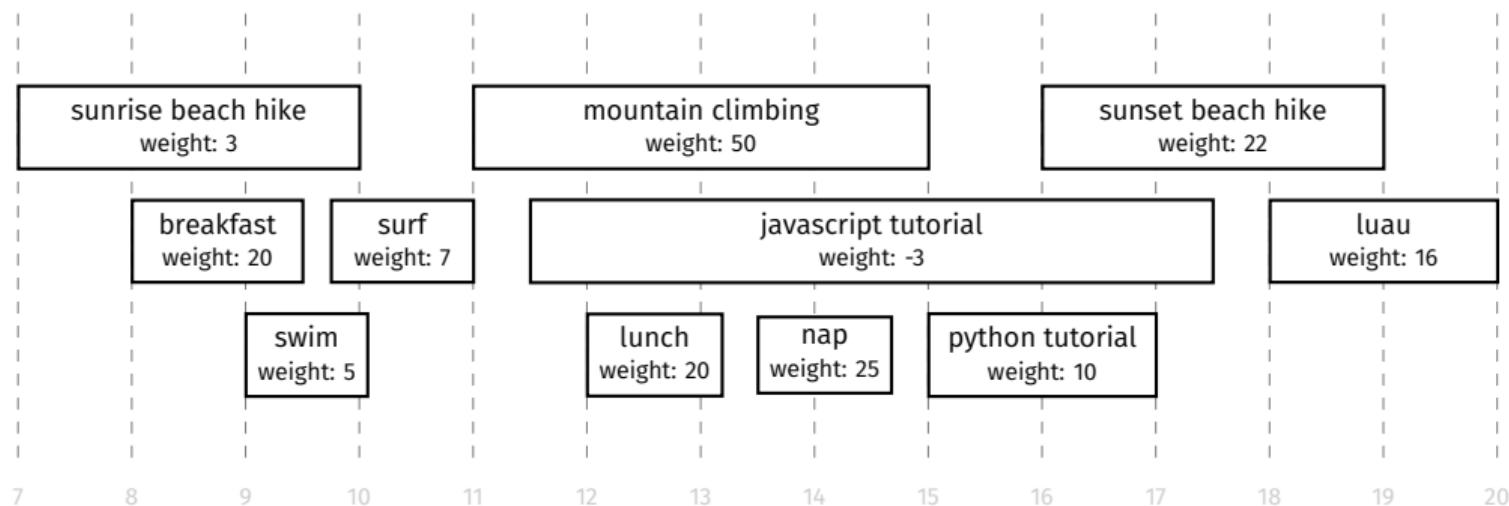
    # best without x
    best_without = ...

    return max(best_with, best_without)
```

Recursive Subproblems

- ▶ What is $\text{BEST}(\text{activities})$ if we assume that x **is** in schedule?
- ▶ Imagine choosing x .
 - ▶ Your current total weight is $x.\text{weight}$.
 - ▶ Activities left to choose from: those **compatible** with x .
- ▶ Clearly, you want the best outcome for *new situation* (subproblem).
- ▶ Answer: $x.\text{weight} + \text{BEST}(\text{activities.compatible_with}(x))$

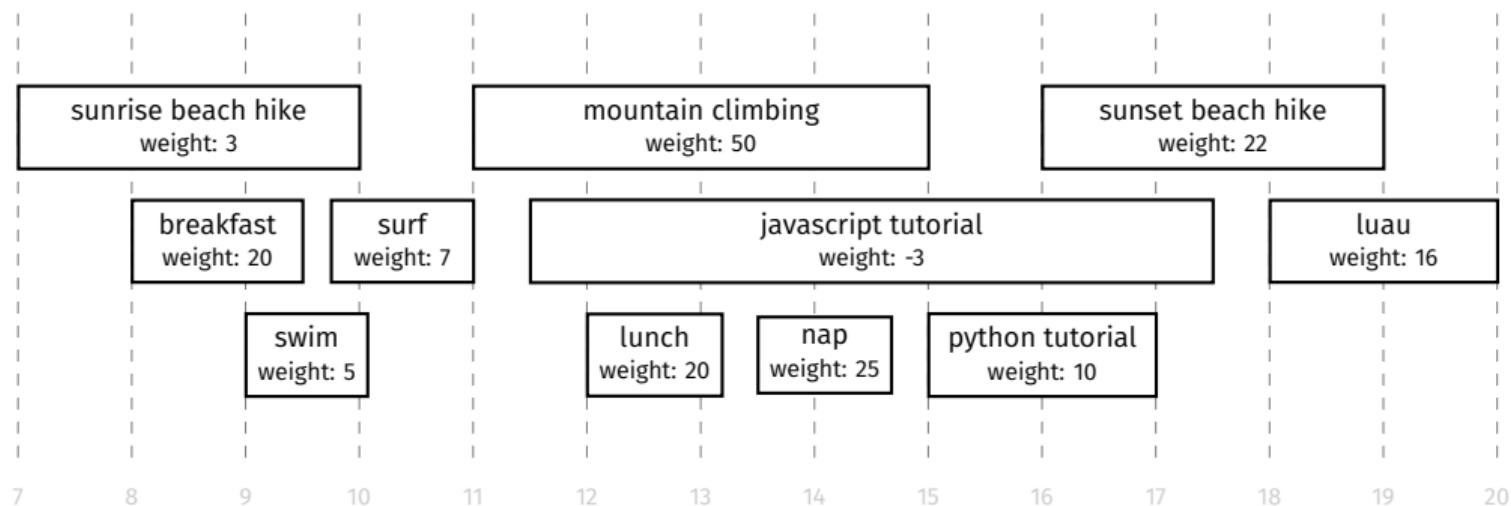
activities.compatible_with(x)



Recursive Subproblems

- ▶ What is $\text{BEST}(\text{activities})$ if we assume that x **is not** in schedule?
- ▶ Imagine not choosing x .
 - ▶ Your current total weight is 0 .
 - ▶ Activities left to choose from: all except x .
- ▶ Clearly, you want the best outcome for *new situation* (subproblem).
- ▶ Answer: $\text{BEST}(\text{activities.without}(x))$

activities.without(x)



Backtracking

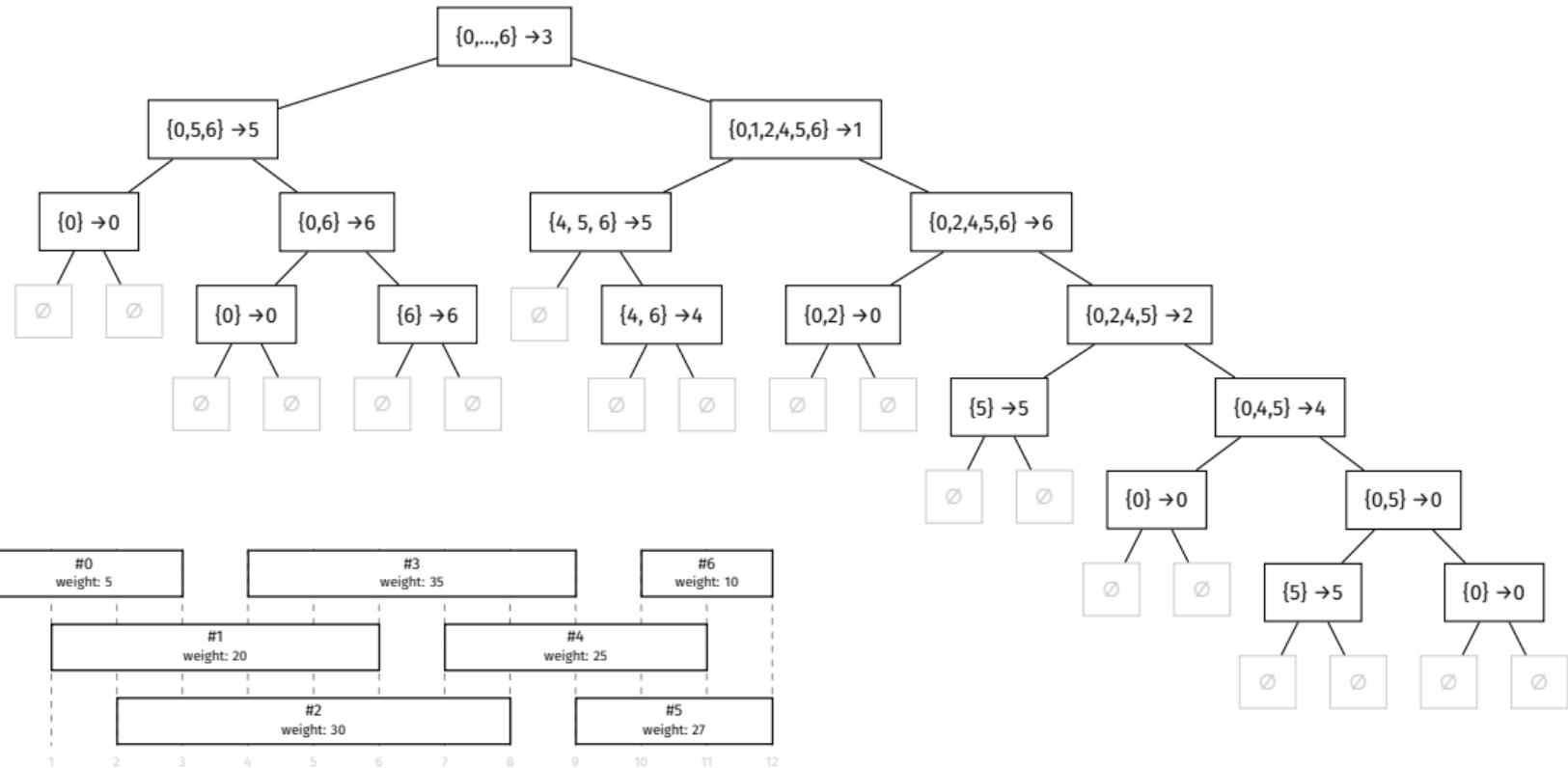
```
def mwsched_bt(activities):
    if not activities:
        return 0

    # choose arbitrary activity
    x = activities.choose_arbitrary()

    # best with x
    best_with = x.weight + mwsched_bt(activities.compatible_with(x))

    # best without x
    best_without = mwsched_bt(activities.without(x))

    return max(best_with, best_without)
```



Efficiency

- ▶ Worst case: recursive calls on problem of size $n - 1$.
- ▶ Recurrence of form $T(n) = 2T(n - 1) + \Theta(\dots)$
- ▶ **Exponential time** in worst case.
- ▶ Could prune, branch & bound, but there's a better way.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 4

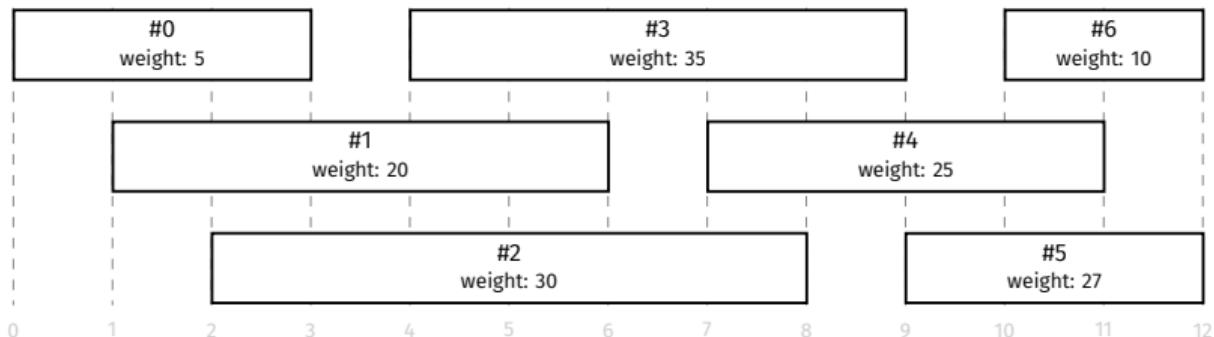
Step 02: A Nicer Backtracking Solution

Arbitrary Choices

- ▶ Our subproblems are arbitrary sets of activities.
 - ▶ E.g., {1, 3, 4, 5, 8, 11, 12}
- ▶ **Now:** If we make choice of next event more carefully, the subproblems look much nicer.
- ▶ Something great happens!

A Nicer Choice

- ▶ Instead of choosing arbitrarily, choose first.



```
def mwsched_bt_nice(activities):
    if not activities:
        return 0

    # choose first activity
    x = activities.choose_first()

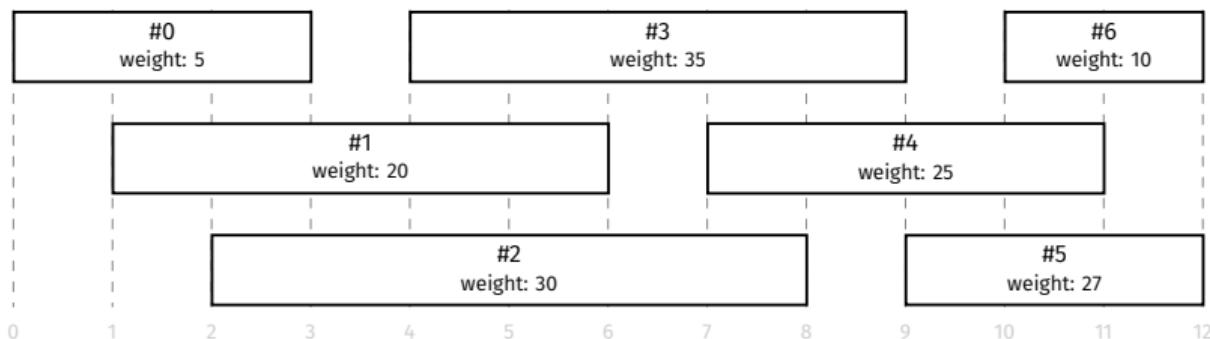
    # best with x
    best_with = x.weight + mwsched_bt(activities.compatible_with(x))

    # best without x
    best_without = mwsched_bt(activities.without(x))

    return max(best_with, best_without)
```

activities.compatible_with(x)

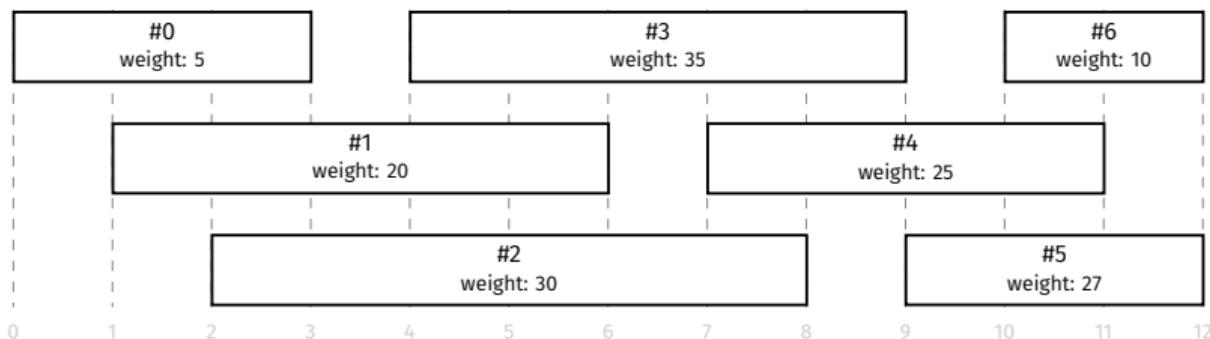
- ▶ Results in a “nice” set of the form $\{i, i + 1, \dots, n - 1\}^2$



²Assuming x is the activity with first start time.

activities.without(x)

- ▶ Results in a “nice” set of the form $\{i, i + 1, \dots, n - 1\}$ ³



³Assuming x is the activity with first start time.

Representing Remaining Activities

- ▶ Assume events are in sorted order by start time.
- ▶ Subproblems are always of form
 $\{i, i + 1, i + 2, \dots, n - 1\}$
- ▶ We can specify them with a **single number, i .**

```
def mwsched_bt_nice(activities, first: int=0):
    """Find best schedule using only events in activities[first:]
    Assumes activities sorted by start time.
    """
    if first >= len(activities):
        return 0

    # choose first event
    x = activities[first]

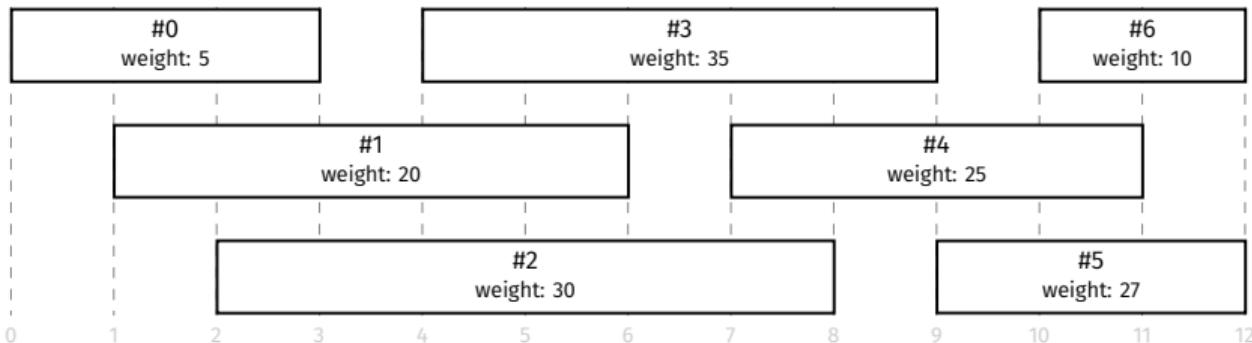
    # best with x
    next_compatible = index_of_next_compatible(activities, after=first)
    best_with = x.weight + mwsched_bt_nice(activities, next_compatible)

    # best without x
    best_without = mwsched_bt_nice(activities, first + 1)

    return max(best_with, best_without)
```

index_of_next_compatible()

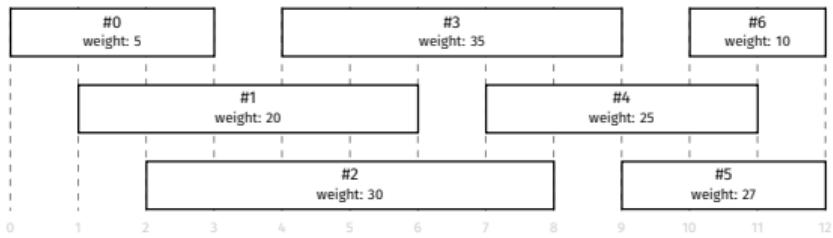
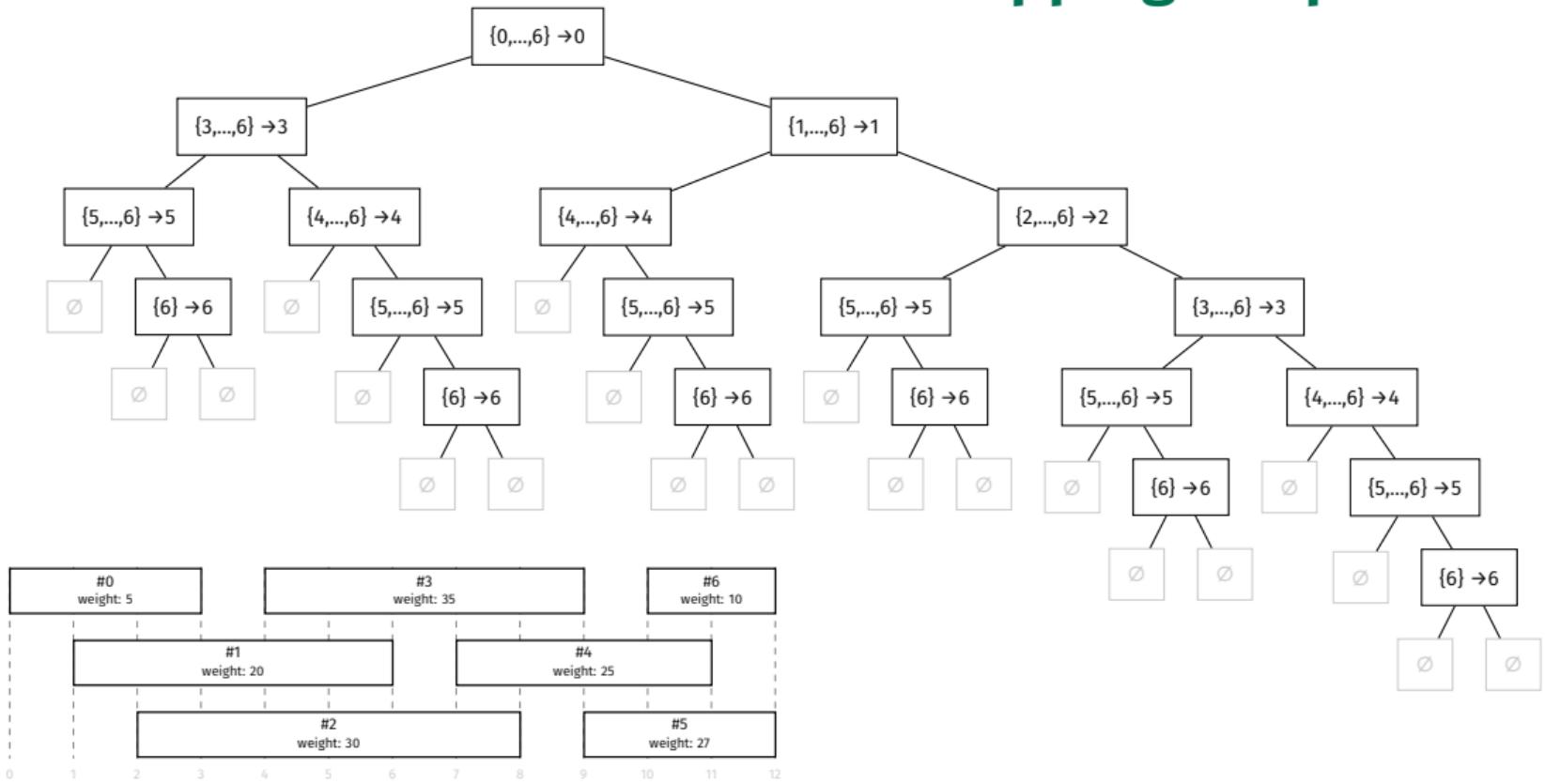
```
def index_of_next_compatible(activities, after: int):
    """Find index of first event starting after `after` ends.
    Assumes activities sorted by start time.
    """
    for j in range(after + 1, len(activities)):
        if activities[j].start >= activities[after].finish:
            return j
    return len(activities)
```



What did we gain?

- ▶ Can specify subproblems with integers instead of sets.
 - ▶ Saves memory.
- ▶ But there's an **even better** consequence!

Overlapping Subproblems

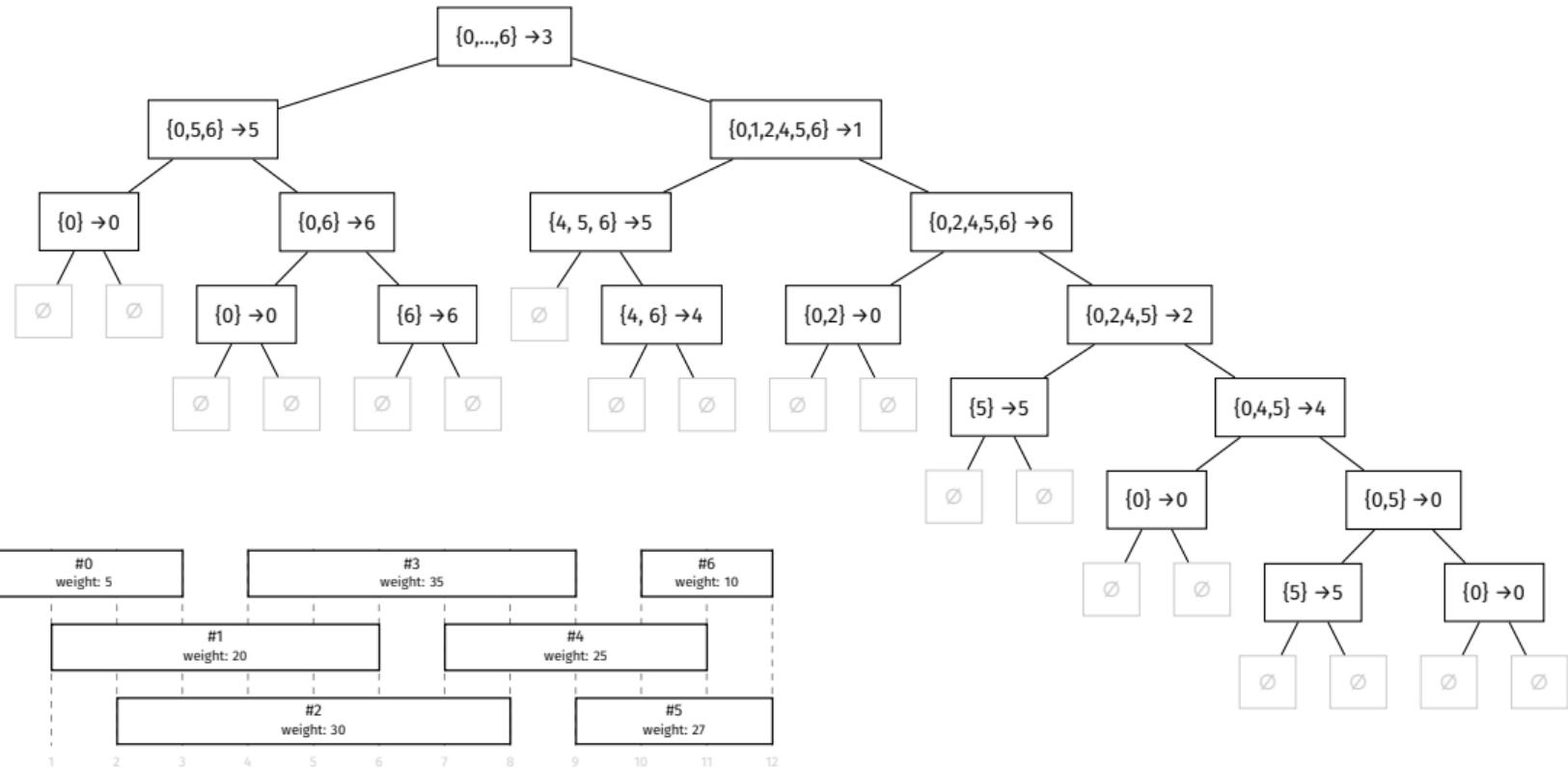


Overlapping Subproblems

- ▶ Backtracking doesn't have a memory.
- ▶ It will happily solve same subproblem over and over, getting same result each time.
- ▶ We'll speed it up by giving it a memory.

Note

- ▶ Overlapping subproblems are a consequence of this more careful choice of event.
- ▶ When we chose arbitrarily, we didn't have (as many) overlapping subproblems.



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 5

Step 03: Memoization

Backtracking + Memoization

- ▶ By making careful choices, we've found a backtracking solution with many overlapping subproblems.
- ▶ Idea: solve subproblem once, save the result!
- ▶ This is called **memoization**⁴.

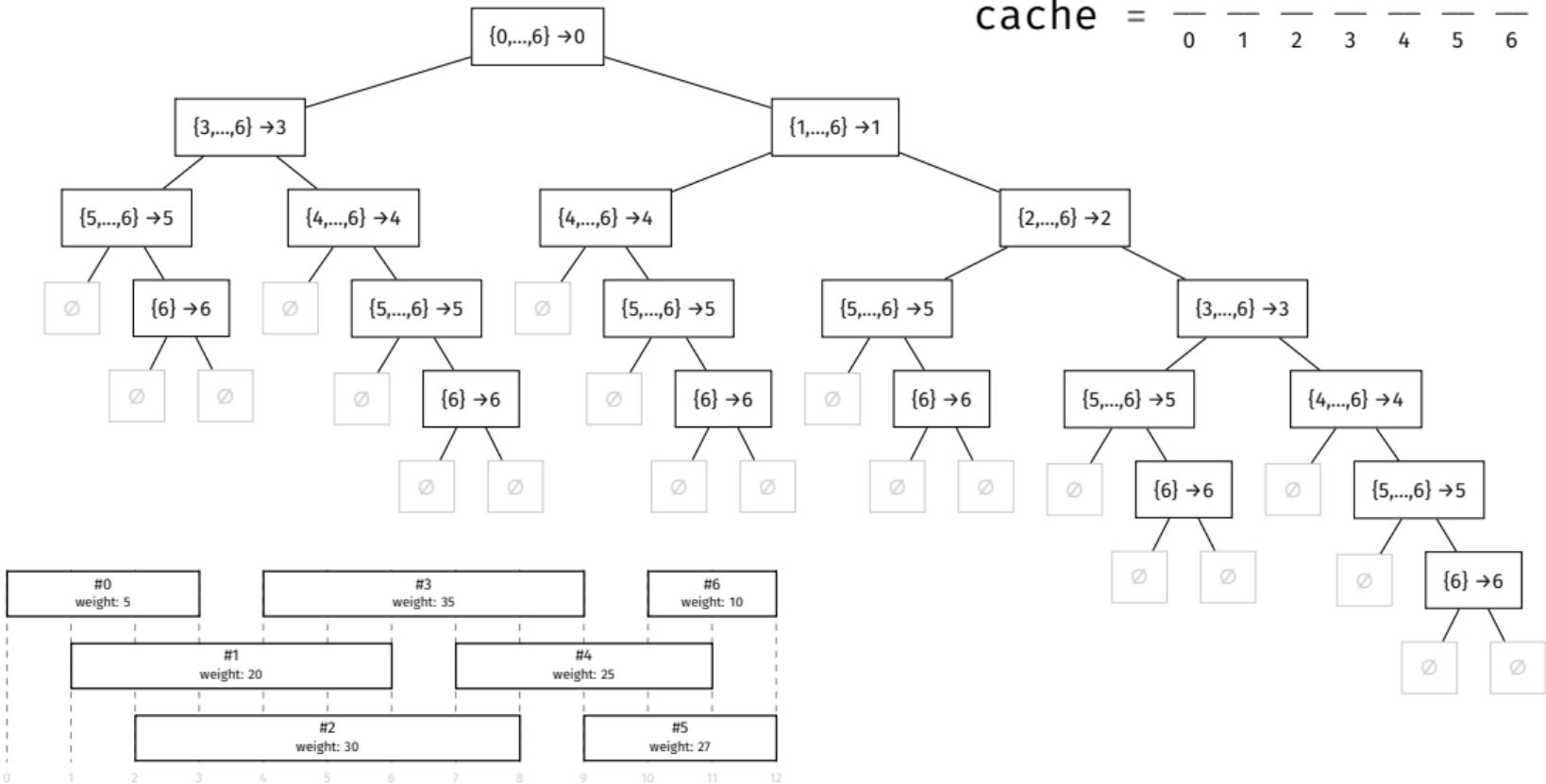
⁴Not “memorization”. That would make too much sense.

Memoization

- ▶ Keep a **cache**: dictionary or array mapping subproblems to solutions.
- ▶ Before solving a subproblem, check if already in cache.
- ▶ After solving a subproblem, save result in cache.

```
def mwsched_dp(activities, first: int=0, cache=None):
    """Find best schedule using events in activities[first:].
    Assumes activities sorted by start time."""
    if cache is None: # cache[i] is solution of activities[i:]
        cache = [None] * len(activities)
    if first >= len(activities):
        return 0
    # save some work if we've already computed this
    if cache[first] is not None:
        return cache[first]
    # choose first event
    x = activities[first]
    # best with x
    next_compatible = index_of_next_compatible(activities, after=first)
    best_with = x.weight + mwsched_dp(activities, next_compatible, cache=cache)
    # best without x
    best_without = mwsched_dp(activities, first + 1, cache=cache)
    best = max(best_with, best_without)
    # store result in cache for future reference
    cache[first] = best
    return best
```

cache = — 0 — 1 — 2 — 3 — 4 — 5 — 6



Time Complexity

- ▶ There are only n subproblems.
 - ▶ $\{0, \dots, n - 1\}, \{1, \dots, n - 1\}, \dots, \{n - 1\}$
- ▶ Solve each one once.
- ▶ The memoized solution takes $\Theta(n)$ time.

Dynamic Programming

- ▶ This approach (backtracking + memoization) is called “top-down” **dynamic programming**.
- ▶ Often reduces time from exponential to polynomial.

DSC 190

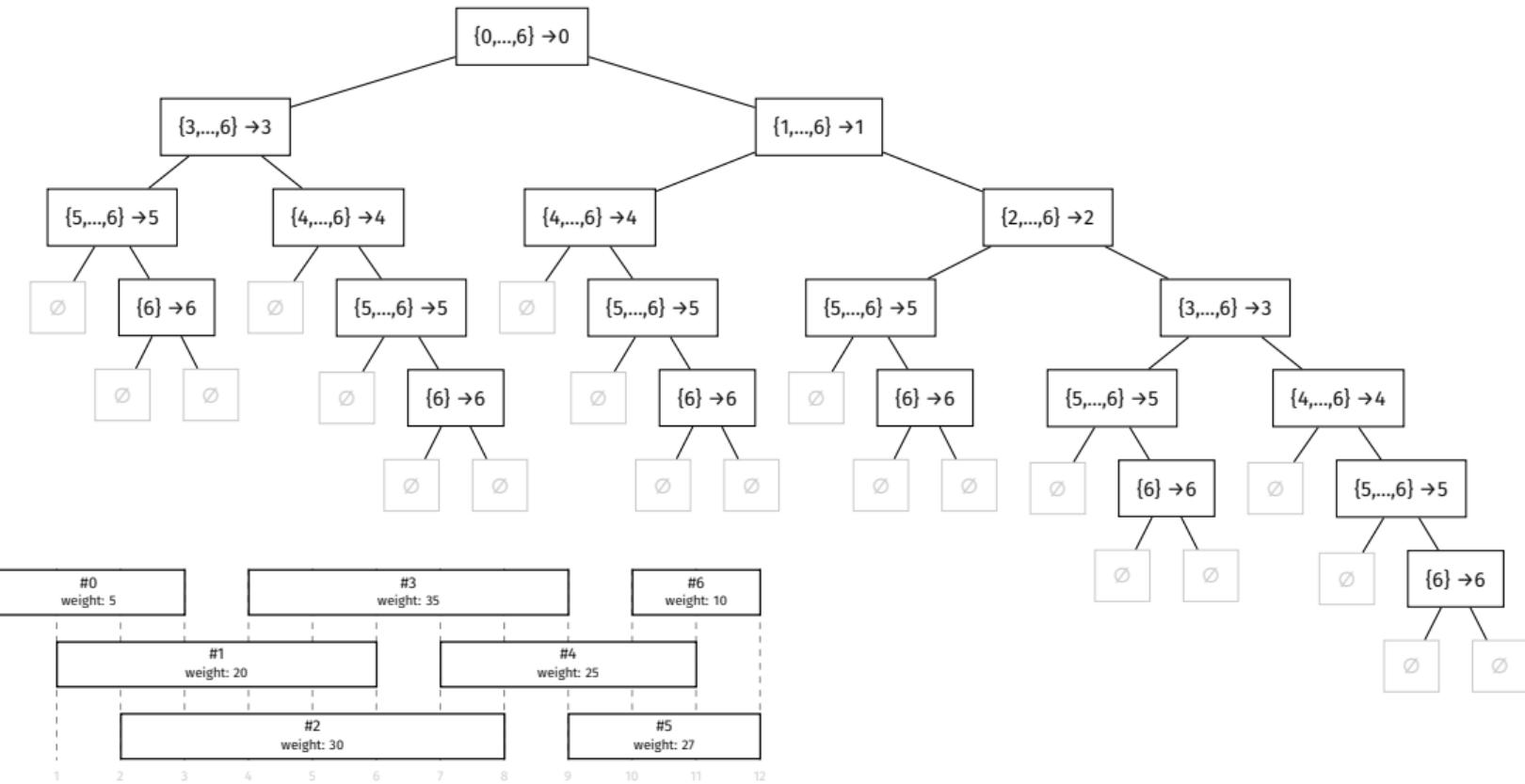
DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 6

Top-Down vs. Bottom-Up

Top-Down

- ▶ Backtracking + memoization is known as “top down” dynamic programming.
- ▶ We start at top level problem, recursively find subproblems.
- ▶ But we can start from bottom-level problems, too.



Bottom-Up

- ▶ The top-down recursive code solves problems in order:
 - ▶ $\{6\}, \{5, 6\}, \{4, \dots, 6\}, \{3, \dots, 6\}, \{2, \dots, 6\}, \{1, \dots, 6\}, \{0, \dots, 6\}$
- ▶ The bottom-up approach starts with easiest subproblem, iteratively solves harder subproblems.
- ▶ Solve $\{6\}$. Use it to solve $\{5, 6\}$. Use this to solve $\{4, \dots, 6\}$, etc.

```
def mwsched_bottom_up(activities):
    """Assumes activities sorted by start time."""
    n = len(activities)

    # best[i] is the weight of the best possible schedule that can be formed
    # using activities[i:].
    # best[n] is a dummy value; it represents the "base case"
    # solution of zero.
    # best[0] is solution to the full problem.
    best = [None] * (n + 1)
    best[n] = 0

    # solve easiest subproblem: when we have one event,
    # activities[n-1]
    best[n-1] = activities[n-1].weight

    # iteratively solve subproblems from small to big,
    # using solutions of smaller problems in solving big
    for first in reversed(range(n-1)):
        x = activities[first]

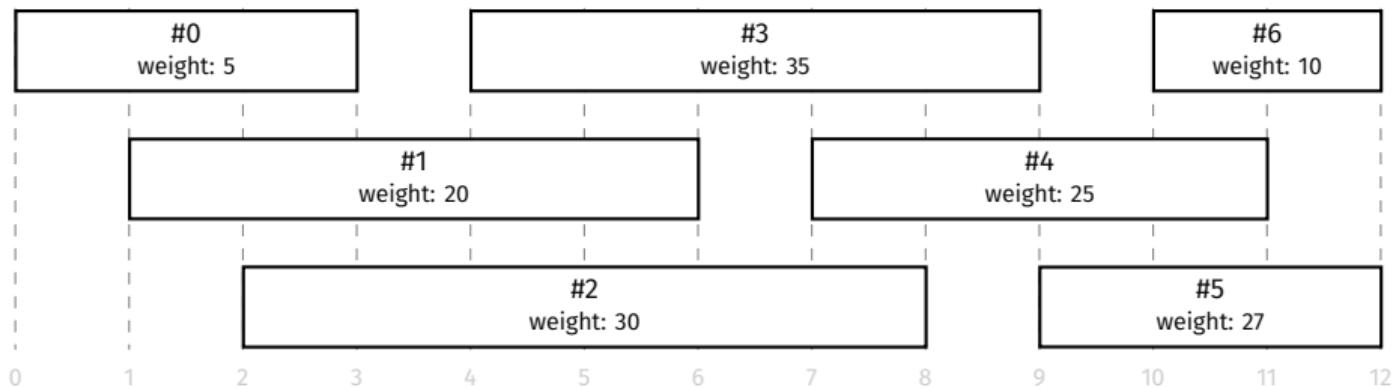
        # best with
        next_compatible = index_of_next_compatible(activities, after=first)
        best_with = x.weight + best[next_compatible]

        # best without
        best_without = best[first + 1]

        best[first] = max(best_with, best_without)

    return best[0]
```

Example



best = — — — — — — —
0 1 2 3 4 5 6 7

Which to use?

- ▶ Bottom-up and top-down will generally have same time complexity.
- ▶ Top-down arguably easier to design.
- ▶ Bottom-up avoids overhead of recursion.
- ▶ But bottom-up may solve unnecessary subproblems.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 11 | Part 7

Dynamic Programming

When can we use it?

- ▶ Memoization can be added to *any* backtracking algorithm.
- ▶ But it is only *useful* if there are **overlapping subproblems**.
- ▶ Not all problems yield overlapping subproblems.

How do we design them?

- ▶ General strategy for top-down:
 1. Write a backtracking solution.
 2. Modify backtracking solution to get overlapping subproblems that are “easy to describe”.⁵
 3. Add memoization.
- ▶ “Expert mode”: identify recursive substructure immediately.
- ▶ Can be tricky; need to be creative.

⁵Easier said than done.

How do we design them?

- ▶ General strategy for bottom-up:
 1. Write a top-down dynamic programming solution.
 2. Analyze the order in which cache is filled in.
 3. Iteratively solve subproblems in this order.

Are they guaranteed to be optimal?

- ▶ Yes! Dynamic programming *is* a form of backtracking, so it is guaranteed to find an optimal solution.

Is it at all useful for data science?

- ▶ Yes!
- ▶ Next time: the longest common subsequence problem and its applications to “fuzzy” string matching, DNA string comparison.
- ▶ Future (maybe): Hidden Markov Models, All-Pairs Shortest Paths

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 12 | Part 1

Today's Lecture

Dynamic Programming

- ▶ We've seen that dynamic programming can lead to fast algorithms that find the optimal answer.
- ▶ Today, we'll see one data science application: longest common substring.
- ▶ Used to match DNA sequences, fuzzy string comparison, etc.

The Strategy

1. Backtracking solution.
2. A “nice” backtracking solution with overlapping subproblems.
3. Memoization.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 12 | Part 2

Longest Common Subsequence

Fuzzy String Matching

- ▶ Suppose you're doing a sentiment analysis of tweets.
- ▶ How do people feel about the University of California?
- ▶ Search for: university of california
- ▶ People can't spell: uivesity of califrbia
- ▶ How do we recognize the match?

DNA String Matching

- ▶ Suppose you're analyzing a genome.
- ▶ DNA is a sequence of G,A,T,C.
- ▶ Mutations cause same gene to have slight differences.
- ▶ Person 1: GATTACAGATTACA
- ▶ Person 2: GATCACAGTTGCA

Measuring Differences

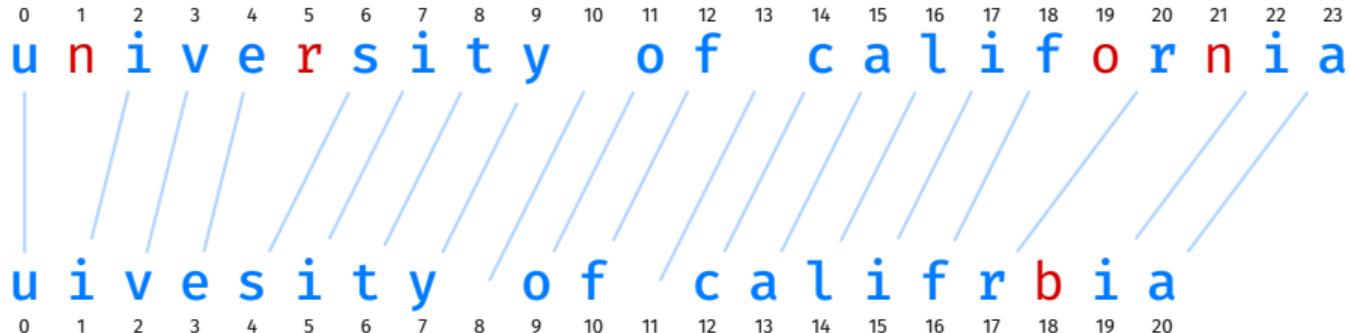
- ▶ Given two strings of (possibly) different lengths.
- ▶ Measure how similar they are.
- ▶ One approach: **longest common subsequences**.

Common Subsequences

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
u n i v e r s i t y o f c a l i f o r n i a

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
u i v e s i t y o f c a l i f r b i a

Common Subsequences



Longest Common Subsequences

- ▶ We will measure similarity by finding length of the **longest common subsequence** (LCS).
- ▶ Now: let's define the LCS..

Subsequences

s a n d i e g o

s a n d i e g o → igo

s a n d i e g o → sio

s a n d i e g o → sadego

s a n d i e g o → sandiego

Not Subsequences

s a n d i e g o

s a n d i e g o → sea

s a n d i e g o → sooo

Subsequences

- ▶ A **subsequence** of a string s of length n is determined by a strictly monotonically increasing sequence of indices with values in $\{0, 1, \dots, n - 1\}$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | a | n | d | i | e | g | o |

→

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 3 | 5 | 6 | 7 |
| s | a | d | e | g | o |

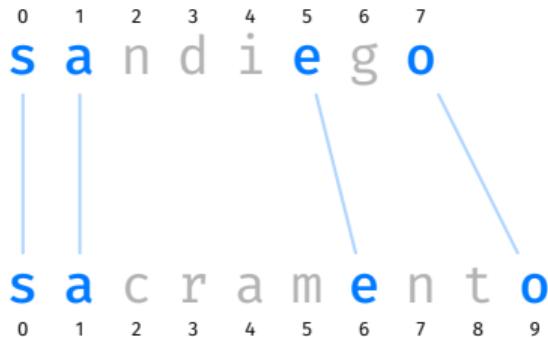
Common Subsequences

- Given two strings, a **common subsequence** is subsequence that appears in both.



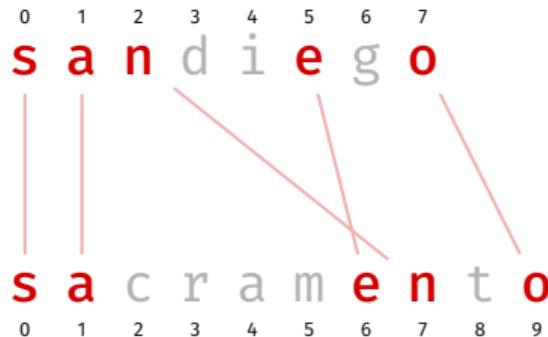
Common Subsequences

- Given two strings, a **common subsequence** is subsequence that appears in both.



Not Common Subsequences

- The lines cannot overlap.



Longest Common Subsequences

- ▶ A **longest common subsequence** (LCS) between two strings is a common subsequence that has the greatest length out of all common subsequences.



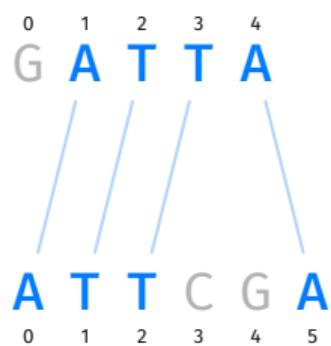
Main Idea

The longer the LCS, the more similar the two strings.

Common Subsequences, Formally

- ▶ Our backtracking solution will build a common subsequence piece by piece.
- ▶ How can we represent the idea of “lines between letters” more formally?

Matching



(0,0) (0,1) (0,2) (0,3) (0,4) (0,5)

(1,0) (1,1) (1,2) (1,3) (1,4) (1,5)

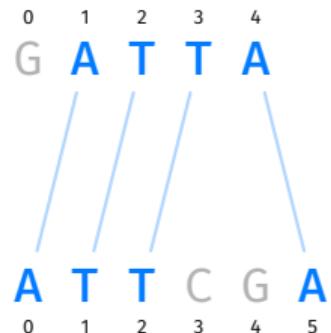
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5)

(3,0) (3,1) (3,2) (3,3) (3,4) (3,5)

(4,0) (4,1) (4,2) (4,3) (4,4) (4,5)

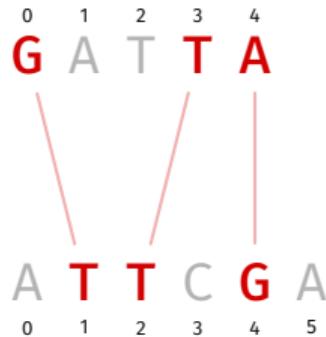
Matching

- ▶ A **matching** between strings a and b is a set of (i, j) pairs.
- ▶ Each (i, j) pair is interpreted as “ $a[i]$ is paired with $b[j]$ ”.
- ▶ Example: $\{(1, 0), (2, 1), (3, 2), (4, 5)\}$



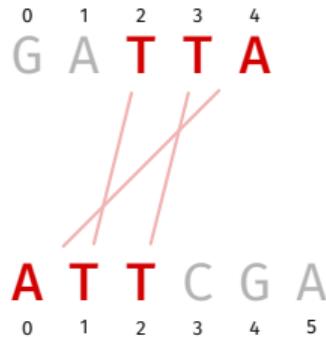
Invalid Matchings

- ▶ Not all matchings represent common subsequences!
- ▶ Example: $\{(0, 1), (3, 2), (4, 4)\}$:



Invalid Matchings

- ▶ Not all matchings represent common subsequences!
- ▶ Example: $\{(4, 0), (2, 1), (3, 2)\}$:

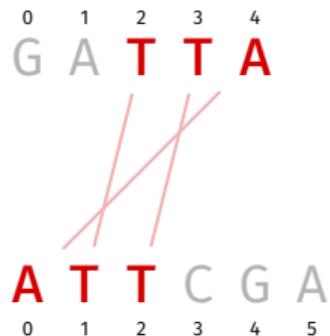


Valid Matchings

- ▶ We'll say a matching M is **valid** if:
 - ▶ $a[i] == b[j]$ for every pair (i, j) ; and
 - ▶ there are no “crossed lines”

“Crossed Lines”

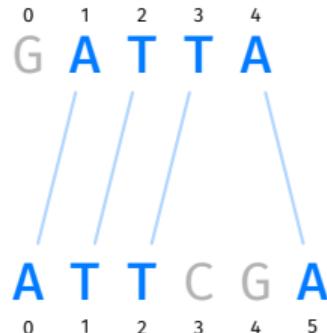
- ▶ Suppose (i, j) and (i', j') are in the matching.
- ▶ “Crossed lines” occur when either:
 - ▶ $i < i'$ but $j > j'$; or
 - ▶ $i > i'$ but $j < j'$.



Valid Matchings

- ▶ We'll say a matching M is **valid** if:
 - ▶ $a[i] == b[j]$ for every pair (i, j) ; and
 - ▶ there are no "crossed lines". that is, for every choice of distinct pairs $(i, j), (i', j') \in M$:
$$i < i' \text{ and } j < j' \quad \text{or} \quad i > i' \text{ and } j > j'$$

- ▶ Example: $\{(1, 0), (2, 1), (3, 2), (4, 5)\}$



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 12 | Part 3

Step 01: Backtracking

Road to Dynamic Programming

- ▶ We'll follow same road to a DP solution as last time.
- ▶ **Step 01: Backtracking solution.**
- ▶ Step 02: A “nice” backtracking solution with overlapping subproblems.
- ▶ Step 03: Memoization.

Backtracking

- ▶ We'll build up a matching, one pair at a time.
- ▶ Choose an arbitrary pair, (i, j) .
 - ▶ Recursively see what happens if we **do** include (i, j) .
 - ▶ Recursively see what happens if we **don't** include (i, j) .
- ▶ This will try **all valid matchings**, keep the best.

Backtracking

```
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    best_with = ...

    # best without
    best_without = ...

    return max(best_with, best_without)
```

Recursive Subproblems

- ▶ What is $\text{BEST}(a, b, \text{pairs})$ if we assume that (i, j) **is** in matching?
- ▶ If $a[i] \neq a[j]$:
 - ▶ Your current common substring is **invalid**. Length is zero.
 - ▶ Don't build matching further.
- ▶ If $a[i] == a[j]$:
 - ▶ Your current common substring has length one.
 - ▶ Pairs remaining to choose from: those **compatible** with (i, j) .
 - ▶ You find yourself in a similar situation as before.
 - ▶ Answer: $1 + \text{BEST}(\text{activities.compatible_with}(x))$

`pairs.compatible_with(x)`

| | | | | | |
|---|----------|---|----------|---|---|
| 0 | 1 | 2 | 3 | 4 | |
| G | A | T | T | A | |
| | | | | | |
| 0 | 1 | 2 | 3 | 4 | |
| A | T | T | C | G | A |

| | | | | | |
|-------|-------|-------|-------|-------|-------|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |



Backtracking

```
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    if a[i] == b[j]:
        best_with = 1 + lcs_bt(a, b, pairs.compatible_with(i, j))
    else:
        best_with = 0

    # best without
    best_without = ...

    return max(best_with, best_without)
```

Recursive Subproblems

- ▶ What is $\text{BEST}(a, b, \text{pairs})$ if we assume that (i, j) **is not** in matching?
- ▶ Imagine not choosing x .
 - ▶ Your current common substring is empty.
 - ▶ Activities left to choose from: all except (i, j) .
- ▶ You find yourself in a similar situation as before.
- ▶ Answer: $\text{BEST}(a, b, \text{pairs}.\text{without}(i, j))$

pairs.without(x)

0 1 2 3 4
G A T **T** A

A **T** T C G A
0 1 2 3 4 5

| | | | | | |
|-------|-------|-------|-------|-------|-------|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |

Backtracking

```
def lcs_bt(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.arbitrary_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    # assume (i, j) is in the LCS, but only if a[i] == b[j]
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt(a, b, pairs.compatible_with(i, j))

    # best without
    best_without = lcs_bt(a, b, pairs.without(i, j))

    return max(best_with, best_without)
```

Backtracking

- ▶ This will try all **valid** matchings.
- ▶ Guaranteed to find optimal answer.
- ▶ But takes exponential time in worst case.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 12 | Part 4

Step 02: A “Nicer” Backtracking Solution

Arbitrary Sets

- ▶ In previous backtracking solution, subproblems are arbitrary sets of pairs.
 $(0,0)$ $(0,1)$ $(0,2)$ $(0,3)$ $(0,4)$
 $(1,0)$ $(1,1)$ $(1,2)$ $(1,3)$ $(1,4)$
- ▶ Rarely see the same subproblem twice.
 $(2,0)$ $(2,1)$ $(2,2)$ $(2,3)$ $(2,4)$
- ▶ This is not good for memoization!
 $(3,0)$ $(3,1)$ $(3,2)$ $(3,3)$ $(3,4)$

Nicer Subproblems

- ▶ In backtracking, we are building a solution piece-by-piece.
- ▶ In last lecture, we saw that a careful choice of next piece led to nice subproblems.
- ▶ Let's try choosing the *last* letters from each string as the next piece of the matching.

Last Letters

0 1 2 3 4
G A T T A
0 1 2 3 4 5
A T T C G A

| | | | | | |
|-------|-------|-------|-------|-------|-------|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |

Nicer Backtracking

```
def lcs_bt_nice(a, b, pairs):
    """Solve find best matching using the pairs in `pairs`."""
    pair = pairs.last_pair()

    if pair is None:
        return 0

    i, j = pair

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice(a, b, pairs.compatible_with(i, j))

    # best without
    best_without = lcs_bt_nice(a, b, pairs.without(i, j))

    return max(best_with, best_without)
```

Subproblems

- ▶ There are two subproblems: LCS using `pairs.compatible_with(i, j)` and LCS using `pairs.without(i, j)`
- ▶ Are they “nicer”?

`pairs.compatible_with(i, j)`

0 1 2 3 4
G A T T A

(0,0) (0,1) (0,2) (0,3) (0,4) (0,5)

(1,0) (1,1) (1,2) (1,3) (1,4) (1,5)

(2,0) (2,1) (2,2) (2,3) (2,4) (2,5)

0 1 2 3 4 5
A T T C G A

(3,0) (3,1) (3,2) (3,3) (3,4) (3,5)

(4,0) (4,1) (4,2) (4,3) (4,4) (4,5)

Nicer Subproblems

- ▶ By taking (i, j) as bottom-right pair, `pairs.compatible_with(i, j)` is again rectangular.
- ▶ Easily described by its bottom-right pair, $(i - 1, j - 1)!$
- ▶ Instead of keeping set of pairs, just need to pass in i and j of last element.

```
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice_2(a, b, i-1, j-1)

    # best without
    best_without = ...

    return max(best_with, best_without)
```

`pairs.without(i, j)`

0 1 2 3 4
G A T T A

(0,0) (0,1) (0,2) (0,3) (0,4) (0,5)

(1,0) (1,1) (1,2) (1,3) (1,4) (1,5)

(2,0) (2,1) (2,2) (2,3) (2,4) (2,5)

0 1 2 3 4 5
A T T C G A

(3,0) (3,1) (3,2) (3,3) (3,4) (3,5)

(4,0) (4,1) (4,2) (4,3) (4,4) (4,5)

Problem

- ▶ `pairs.without(i, j)` is **not** rectangular.
- ▶ Cannot be described by a single pair.
- ▶ But there's a fix.

Observation

- ▶ A common substring cannot have pairs both in the last row and the last column. **Crossing lines!**

| | | | | | | | | | | | |
|---|---|---|---|---|---|-------|-------|-------|-------|-------|-------|
| | | | | | | (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| 0 | 1 | 2 | 3 | 4 | | (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| G | A | T | T | A | | (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| | | | | | | (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| A | T | T | C | G | A | (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |
| 0 | 1 | 2 | 3 | 4 | 5 | | | | | | |

Consequence

- ▶ $\text{BEST}(\text{pairs.without}(i, j)) = \max\{\text{BEST}(\text{pairs.without_row}(i)), \text{BEST}(\text{pairs.without_col}(j))\}$

| | | | | | | | | | | | |
|---|---|---|---|---|---|-------|-------|-------|-------|-------|-------|
| | | | | | | (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| 0 | 1 | 2 | 3 | 4 | | (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| G | A | T | T | A | | (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| | | | | | | (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| A | T | T | C | G | A | (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |
| 0 | 1 | 2 | 3 | 4 | 5 | | | | | | |

Observation

- ▶ `pairs.without_row(i)` represented by subprob. $(i - 1, j)$
- ▶ `pairs.without_col(j)` represented by subprob. $(i, j - 1)$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|--|-------|-------|-------|-------|-------|-------|
| | | | | | | | (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) |
| 0 | 1 | 2 | 3 | 4 | | | (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) |
| G | A | T | T | A | | | (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) |
| | | | | | | | (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) |
| A | T | T | C | G | A | | (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) |
| 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | |

“Nice” Backtracking

```
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] != b[j]:
        best_with = 0
    else:
        best_with = 1 + lcs_bt_nice_2(a, b, i-1, j-1)

    # best without
    best_without = max(
        lcs_bt_nice_2(a, b, i-1, j),
        lcs_bt_nice_2(a, b, i, j-1)
    )

    return max(best_with, best_without)
```

One More Observation

- ▶ This is fine, but we can do a little better.
- ▶ If $a[i] == b[j]$, we can assume (i, j) is in matching – don't need to consider otherwise!¹

0 1 2 3 4
G A T T A

A T T C G A
0 1 2 3 4 5

¹This is true we chose last pair; not true if choice was arbitrary.

“Nicer” Backtracking

```
def lcs_bt_nice_2(a, b, i, j):
    """Solve LCS problem for a[:i], b[:j]."""
    if i < 0 or j < 0:
        return 0

    # best with
    if a[i] == b[j]:
        # best with (i, j)
        return 1 + lcs_bt_nice_2(a, b, i-1, j-1)
    else:
        # best without (i, j)
        return max(
            lcs_bt_nice_2(a, b, i-1, j),
            lcs_bt_nice_2(a, b, i, j-1)
        )
```

Overlapping Subproblems

- ▶ Suppose a and b are of length m and n .
- ▶ There are mn possible subproblems.
- ▶ Backtracking tree has exponentially-many nodes.
- ▶ We will see many subproblems over and over again!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 12 | Part 5

Step 03: Memoization

Backtracking

- ▶ The backtracking solutions are slow.
- ▶ `a = 'CATCATCATCATCATGAAAAAAA'`
- ▶ `b = 'GATTACAGATTACAGATTACA'`
- ▶ “Nice” backtracking solution: 8 seconds.
- ▶ Memoized solution: 100 microseconds.

```
def lcs_dp(a, b, i=None, j=None, cache=None):
    """Solve LCS problem for a[:i], b[:j]."""
    if i is None:
        i = len(a) - 1

    if j is None:
        j = len(b) - 1

    if cache is None:
        cache = {}

    if i < 0 or j < 0:
        return 0

    if (i,j) in cache:
        return cache[(i, j)]

    # best with
    if a[i] == b[j]:
        # best with (i, j)
        best = 1 + lcs_dp(a, b, i-1, j-1, cache)
    else:
        # best without (i, j)
        best = max(
            lcs_dp(a, b, i-1, j, cache),
            lcs_dp(a, b, i, j-1, cache)
        )

    cache[(i, j)] = best
    return best
```

Top-Down vs. Bottom-Up

- ▶ This is the top-down dynamic programming solution.
- ▶ It takes time $\Theta(mn)$, where m and n are the string lengths.
- ▶ To find a bottom-up iterative solution, start with the easiest subproblem.
- ▶ What is it?

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 13 | Part 1

Today's Lecture

Catch-Up Lecture

- ▶ This lecture is **optional**.
- ▶ But it might help you with your homework.
- ▶ A chance to catch up.

Testing Your Code

- ▶ Testing code is **essential** (for homework and real life).
- ▶ Consider it to be part of the problem.
- ▶ How do we test Python code?

Approach #1: Run it by hand

- ▶ Write your code.
- ▶ Open up a Python interpreter.
- ▶ Type in a few examples, see if code works.
- ▶ It doesn't work. Repeat.

Downsides

- ▶ You often run the same test over and over again.
- ▶ You have to type it in every time.
- ▶ This is **annoying**.

Main Idea

If something is annoying, you'll avoid doing it.
Spend the time to make things less annoying.

Approach #2: Unit Testing Frameworks

- ▶ Create a file that only includes tests.
- ▶ Write test for each way that code will be used.
 - ▶ Example: for a stack, write test for push, pop, peek.
- ▶ Try to anticipate “corner cases”.
- ▶ Write the tests **before** you write the code.

Unit Testing in Python

- ▶ `unittest`: built-in module for unit testing
- ▶ `pytest`: nicer to use, more “modern”

```
import stack
import pytest

def test_push_then_peek():
    s = stack.Stack(10)
    s.push(1)
    s.push(5)
    s.push(3)
    assert s.peek() == 3

def test_push_then_pop():
    s = stack.Stack(10)
    s.push(1)
    s.push(5)
    s.push(3)
    assert s.pop() == 3
```

Debugging

- ▶ Testing and debugging go hand-in-hand.
- ▶ Should know how to use the Python debugger.

Unit Testing Guidelines

- ▶ Should test “public” interface, not “private” implementation details.
- ▶ Should “exercise” all of the code (coverage).
- ▶ Write the tests before the code.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 1

String Matching

Strings

- ▶ An **alphabet** is a set of possible characters.

$$\Sigma = \{G, A, T, C\}$$

- ▶ A **string** is a sequence of characters from the alphabet.

”GATTACATACGAT”

Example: Bitstrings

$$\Sigma = \{0, 1\}$$

"0110010110"

Example: Text (Latin Alphabet)

$$\Sigma = \{a, \dots, z, \text{<space>}\}$$

"this is a string"

Comparing Strings

- ▶ Suppose s and t are two strings of equal length, m .
- ▶ Checking for equality takes worst-case time $\Theta(m)$ time.

```
def strings_equal(s, t):
    if len(s) != len(t):
        return False
    for i in range(len(s)):
        if s[i] != t[i]:
            return False
    return True
```

String Matching

(Substring Search)

- ▶ **Given:** a string, s , and a pattern string p
- ▶ **Determine:** all locations of p in s
- ▶ Example:

$s = \text{"GATTACATACG"}$ $p = \text{"TAC"}$

Naïve Algorithm

- ▶ Idea: “slide” pattern p across s, check for equality at each location.

```
def naive_string_match(s, p):  
    match_locations = []  
    for i in range(len(s) - len(p) + 1):  
        if s[i:i+len(p)] == p:  
            match_locations.append(i)  
    return match_locations
```

s = "GATTACATACG" p = "TAC"

Time Complexity

```
def naive_string_match(s, p):
    match_locations = []
    for i in range(len(s) - len(p) + 1):
        if s[i:i+len(p)] == p:
            match_locations.append(i)
    return match_locations
```

Naïve Algorithm

- ▶ Worst case: $\Theta(|s| - |p| + 1) \cdot |p|)$ time¹
- ▶ Can we do better?

¹The + 1 is actually important, since if $|p| = |s|$ this should be $\Theta(1)$

Yes!

- ▶ There are numerous ways to do better.
- ▶ We'll look at one: **Rabin-Karp**.
- ▶ Under some assumptions, takes $\Theta(|s| + |p|)$ expected time.
- ▶ Not always the fastest, but easy to implement, and generalizes to other problems.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 2

Rabin-Karp

Idea

- ▶ The naïve algorithm performs $\Theta(|s|)$ comparisons of strings of length $|p|$.
- ▶ String comparison is slow: $O(|p|)$ time.
- ▶ Integer comparison is fast: $\Theta(1)$ time².
- ▶ Idea: **hash** strings into integers, compare them.

²As long as the integers are “not too big”

Recall: Hash Functions

- ▶ A **hash function** takes in an object and returns a (small) number.
- ▶ **Important:** Given the same object, returns same number.
- ▶ It may be possible for two different objects to hash to same number. This is a **collision**.

String Hashing

- ▶ A **string hash function** takes a string, returns a number.
- ▶ Given same string, returns same number.

```
»> string_hash("testing")
32
»> string_hash("something else")
7
»> string_hash("testing")
32
```

Idea

- ▶ Instead of performing $O(|p|)$ string comparison for each i :

```
s[i:i + len(p)] == p
```

- ▶ Hash, and perform $\Theta(1)$ *integer* comparison:

```
string_hash(s[i:i + len(p)]) == string_hash(p)
```

- ▶ In case of collision, need to perform full string comparison in order to ensure this isn't a false match.

Example

s = "ABBABAABBABA"
p = "BAA"

| x | string_hash(x) |
|-----|----------------|
| AAA | 2 |
| AAB | 5 |
| ABA | 3 |
| BAA | 1 |
| ABB | 4 |
| BAB | 1 |
| BBA | 3 |
| BBB | 2 |

Pseudocode

```
def string_match_with_hashing(s, p):
    match_locations = []
    for i in range(len(s) - len(p) + 1):
        if string_hash(s[i:i+len(p)]) == string_hash(p):
            # make sure this isn't a spurious match due to collision
            if s[i:i+len(p)] == p:
                match_locations.append(i)
    return match_locations
```

Time Complexity

- ▶ Comparing (small) integers takes $\Theta(1)$ time.
- ▶ But hashing a string x usually takes $\Omega(|x|)$.
- ▶ In this case, $|x| = |p|$, so overall:

$$\Omega((|s| + |p| + 1) \cdot |p|)$$

- ▶ No better than naïve!

Idea: Rolling Hashes

- ▶ We hash many strings.
- ▶ But the strings we are hashing change only a little bit.
- ▶ Example: $s = "ozymandias"$, $p = "mandi"$.

Rabin-Karp

- ▶ We'll design a special hash function.
- ▶ Instead of computing hash "from scratch", it will "update" old hash in $\Theta(1)$ time.

```
»> old_hash = rolling_hash("ozymandias", start=0, stop=5)
»> new_hash = rolling_hash("ozymandias", start=1, stop=6, update=old_hash)
```

```
def rabin_karp(s, p):
    hashed_window = string_hash(s, 0, len(p))
    hashed_pattern = string_hash(p, 0, len(p))
    match_locations = []

    if s[0:len(p)] == p:
        match_locations.append(0)

    for i in range(1, len(s) - len(p) + 1):
        # update the hash
        hashed_window = update_string_hash(s, i, i + len(p), hashed_window)

        if hashed_window == hashed_pattern:
            # make sure this isn't a false match due to collision
            if s[i:i + len(p)] == p:
                match_locations.append(i)

    return match_locations
```

Time Complexity

- ▶ $\Theta(|p|)$ time to hash pattern.
- ▶ $\Theta(1)$ to update window hash, done $\Theta(|s| - |p| + 1)$ times.
- ▶ When there is a collision, $\Theta(|p|)$ time to check.

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{c \cdot |p|}_{\text{check collisions}})$$

Worst Case

- ▶ In worst case, every position results in a collision.
- ▶ That is, there are $\Theta(|s|)$ collisions:

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{|s| \cdot |p|}_{\text{check collisions}}) \rightarrow \Theta(|s| \cdot |p|)$$

- ▶ Example: $s = \text{"aaaaaaaaaa"}$, $p = \text{"aaa"}$
- ▶ This is just as **bad** as naïve!

More Realistic Time Complexity

- ▶ Only a few valid matches and a few spurious matches.
- ▶ Number of collisions depends on hash function.
- ▶ Our hash function will reasonably have $\Theta(|s|/|p|)$ collisions.

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{c \cdot |p|}_{\text{check collisions}}) \rightarrow \Theta(|s|)$$

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 3

Rolling Hashes

The Problem

- ▶ We need to hash:
 - ▶ $s[0:0 + \text{len}(p)]$
 - ▶ $s[1:1 + \text{len}(p)]$
 - ▶ $s[2:2 + \text{len}(p)]$
 - ▶ ...
- ▶ A standard hash function takes $\Theta(|p|)$ time per call.
- ▶ But these strings overlap.
- ▶ Goal: Design hash function that takes $\Theta(1)$ time to “update” the hash.

Strings as Numbers

- ▶ Our hash function should take a string, return a number.
- ▶ Should be unlikely that two different strings have same hash.
- ▶ Idea: treat each character as a digit in a base- $|\Sigma|$ expansion.

Digression: Decimal Number System

- ▶ In the standard decimal (base-10) number system, each digit ranges from 0-9, represents a power of 10.
- ▶ Example:

$$1532_{10} = (2 \times 10^0) + (3 \times 10^1) + (5 \times 10^2) + (1 \times 10^3)$$

Digression: Binary Number System

- ▶ Computers use binary (base-2). Each digit ranges from 0-1, represents a power of 2.
- ▶ Example:

$$\begin{aligned}10110_2 &= (0 \times 2^0) + (1 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) + (1 \times 2^4) \\&= 22_{10}\end{aligned}$$

Digression: Base-256

- We can use whatever base is convenient. For instance, base-128, in which each digit ranges from 0-127, represents a power of 128.

$$\begin{aligned}12,97,199_{128} &= (101 \times 128^0) + (97 \times 128^1) + (12 \times 128^2) \\&= 209125_{10}\end{aligned}$$

What does this have to do with strings?

- ▶ We can interpret a character in alphabet Σ as a digit value in base $|\Sigma|$.
- ▶ For example, suppose $\Sigma = \{a, b\}$.
- ▶ Interpret a as 0, b as 1.
- ▶ Interpret string "**babba**" as binary string 10110_2 .
- ▶ In decimal: $10110_2 = 22_{10}$

Main Idea

We have mapped the string "**babba**" to an integer: 22. In fact, this is the *only* string over Σ that maps to 22. Interpreting a string of a and b as a binary number hashes the string!

General Strings

- ▶ What about general strings, like "I am a string."?
- ▶ Choose some encoding of characters to numbers.
- ▶ Popular (if outdated) encoding: ASCII.
- ▶ Maps Latin characters, more, to 0-127. So $|\Sigma| = 128$.

ASCII TABLE

| Decimal | Hexadecimal | Binary | Octal | Char | Decimal | Hexadecimal | Binary | Octal | Char | Decimal | Hexadecimal | Binary | Octal | Char |
|---------|-------------|--------|-------|------------------------|---------|-------------|---------|-------|------|---------|-------------|---------|-------|-------|
| 0 | 0 | 0 | 0 | [NULL] | 48 | 30 | 110000 | 60 | 0 | 96 | 60 | 1100000 | 140 | |
| 1 | 1 | 1 | 1 | [START OF HEADING] | 49 | 31 | 110001 | 61 | 1 | 97 | 61 | 1100001 | 141 | a |
| 2 | 2 | 10 | 2 | [START OF TEXT] | 50 | 32 | 110010 | 62 | 2 | 98 | 62 | 1100010 | 142 | b |
| 3 | 3 | 11 | 3 | [END OF TEXT] | 51 | 33 | 110011 | 63 | 3 | 99 | 63 | 1100011 | 143 | c |
| 4 | 4 | 100 | 4 | [END OF TRANSMISSION] | 52 | 34 | 110100 | 64 | 4 | 100 | 64 | 1100100 | 144 | d |
| 5 | 5 | 101 | 5 | [ENQUIRY] | 53 | 35 | 110101 | 65 | 5 | 101 | 65 | 1100101 | 145 | e |
| 6 | 6 | 110 | 6 | [ACKNOWLEDGE] | 54 | 36 | 110110 | 66 | 6 | 102 | 66 | 1100110 | 146 | f |
| 7 | 7 | 111 | 7 | [BELL] | 55 | 37 | 110111 | 67 | 7 | 103 | 67 | 1100111 | 147 | g |
| 8 | 8 | 1000 | 10 | [BACKSPACE] | 56 | 38 | 111000 | 70 | 8 | 104 | 68 | 1101000 | 150 | h |
| 9 | 9 | 1001 | 11 | [HORIZONTAL TAB] | 57 | 39 | 111001 | 71 | 9 | 105 | 69 | 1101001 | 151 | i |
| 10 | A | 1010 | 12 | [LINE FEED] | 58 | 3A | 111010 | 72 | : | 106 | 6A | 1101010 | 152 | j |
| 11 | B | 1011 | 13 | [VERTICAL TAB] | 59 | 3B | 111011 | 73 | : | 107 | 6B | 1101011 | 153 | k |
| 12 | C | 1100 | 14 | [FORM FEED] | 60 | 3C | 111100 | 74 | < | 108 | 6C | 1101100 | 154 | l |
| 13 | D | 1101 | 15 | [CARRIAGE RETURN] | 61 | 3D | 111101 | 75 | = | 109 | 6D | 1101101 | 155 | m |
| 14 | E | 1110 | 16 | [SHIFT OUT] | 62 | 3E | 111110 | 76 | > | 110 | 6E | 1101110 | 156 | n |
| 15 | F | 1111 | 17 | [SHIFT IN] | 63 | 3F | 111111 | 77 | ? | 111 | 6F | 1101111 | 157 | o |
| 16 | 10 | 10000 | 20 | [DATA LINK ESCAPE] | 64 | 40 | 1000000 | 100 | @ | 112 | 70 | 1110000 | 160 | p |
| 17 | 11 | 10001 | 21 | [DEVICE CONTROL 1] | 65 | 41 | 1000001 | 101 | A | 113 | 71 | 1110001 | 161 | q |
| 18 | 12 | 10010 | 22 | [DEVICE CONTROL 2] | 66 | 42 | 1000010 | 102 | B | 114 | 72 | 1110010 | 162 | r |
| 19 | 13 | 10011 | 23 | [DEVICE CONTROL 3] | 67 | 43 | 1000011 | 103 | C | 115 | 73 | 1110011 | 163 | s |
| 20 | 14 | 10100 | 24 | [DEVICE CONTROL 4] | 68 | 44 | 1000100 | 104 | D | 116 | 74 | 1110100 | 164 | t |
| 21 | 15 | 10101 | 25 | [NEGATIVE ACKNOWLEDGE] | 69 | 45 | 1000101 | 105 | E | 117 | 75 | 1110101 | 165 | u |
| 22 | 16 | 10110 | 26 | [SYNCHRONOUS IDLE] | 70 | 46 | 1000110 | 106 | F | 118 | 76 | 1110110 | 166 | v |
| 23 | 17 | 10111 | 27 | [ENG OF TRANS. BLOCK] | 71 | 47 | 1000111 | 107 | G | 119 | 77 | 1110111 | 167 | w |
| 24 | 18 | 11000 | 30 | [CANCEL] | 72 | 48 | 1001000 | 110 | H | 120 | 78 | 1111000 | 170 | x |
| 25 | 19 | 11001 | 31 | [END OF MEDIUM] | 73 | 49 | 1001001 | 111 | I | 121 | 79 | 1111001 | 171 | y |
| 26 | 1A | 11010 | 32 | [SUBSTITUTE] | 74 | 4A | 1001010 | 112 | J | 122 | 7A | 1111010 | 172 | z |
| 27 | 1B | 11011 | 33 | [ESCAPE] | 75 | 4B | 1001011 | 113 | K | 123 | 7B | 1111011 | 173 | { |
| 28 | 1C | 11100 | 34 | [FILE SEPARATOR] | 76 | 4C | 1001100 | 114 | L | 124 | 7C | 1111100 | 174 | |
| 29 | 1D | 11101 | 35 | [GROUP SEPARATOR] | 77 | 4D | 1001101 | 115 | M | 125 | 7D | 1111101 | 175 | } |
| 30 | 1E | 11110 | 36 | [RECORD SEPARATOR] | 78 | 4E | 1001110 | 116 | N | 126 | 7E | 1111110 | 176 | ~ |
| 31 | 1F | 11111 | 37 | [UNIT SEPARATOR] | 79 | 4F | 1001111 | 117 | O | 127 | 7F | 1111111 | 177 | [DEL] |
| 32 | 20 | 100000 | 40 | [SPACE] | 80 | 50 | 1010000 | 120 | P | | | | | |
| 33 | 21 | 100001 | 41 | ! | 81 | 51 | 1010001 | 121 | Q | | | | | |
| 34 | 22 | 100010 | 42 | * | 82 | 52 | 1010010 | 122 | R | | | | | |
| 35 | 23 | 100011 | 43 | # | 83 | 53 | 1010011 | 123 | S | | | | | |
| 36 | 24 | 100100 | 44 | \$ | 84 | 54 | 1010100 | 124 | T | | | | | |
| 37 | 25 | 100101 | 45 | % | 85 | 55 | 1010101 | 125 | U | | | | | |
| 38 | 26 | 100110 | 46 | & | 86 | 56 | 1010110 | 126 | V | | | | | |
| 39 | 27 | 100111 | 47 | ' | 87 | 57 | 1010111 | 127 | W | | | | | |
| 40 | 28 | 101000 | 50 | (| 88 | 58 | 1011000 | 130 | X | | | | | |
| 41 | 29 | 101001 | 51 |) | 89 | 59 | 1011001 | 131 | Y | | | | | |
| 42 | 2A | 101010 | 52 | * | 90 | 5A | 1011010 | 132 | Z | | | | | |
| 43 | 2B | 101011 | 53 | + | 91 | 5B | 1011011 | 133 | [| | | | | |
| 44 | 2C | 101100 | 54 | , | 92 | 5C | 1011100 | 134 | \ | | | | | |
| 45 | 2D | 101101 | 55 | - | 93 | 5D | 1011101 | 135 |] | | | | | |
| 46 | 2E | 101110 | 56 | , | 94 | 5E | 1011110 | 136 | ^ | | | | | |
| 47 | 2F | 101111 | 57 | / | 95 | 5F | 1011111 | 137 | - | | | | | |

In Python

```
>> ord('a')  
97  
>> ord('z')  
90  
>> ord('!')  
33
```

ASCII as Base-128

- ▶ Each character represents a number in range 0-127.
- ▶ A string is a number represented in base-128.
- ▶ Example:

$$\begin{aligned} \text{Hello}_{128} \\ = & (111 \times 128^0) \\ & + (108 \times 128^1) \\ & + (108 \times 128^2) \\ & + (101 \times 128^3) \\ & + (72 \times 128^4) \\ = & 19540948591_{10} \end{aligned}$$

| character | ASCII code |
|-----------|------------|
| H | 72 |
| e | 101 |
| l | 108 |
| o | 111 |

```
def base_128_hash(s, start, stop):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    p = 0
    total = 0
    while stop > start:
        total += ord(s[stop-1]) * 128**p
        p += 1
        stop -= 1
    return total
```

Rolling Hashes

- ▶ We can hash a string x by interpreting it as a number in a different base number system.
- ▶ But hashing takes time $\Theta(|x|)$.
- ▶ With rolling hashes, it will take time $\Theta(1)$ to “update”.

| character | ASCII code |
|-----------|------------|
| H | 72 |
| e | 101 |
| l | 108 |
| o | 111 |

Example

- ▶ Hash of “Hel” in
“**Hello**”
- ▶ Hash of “ell” in
“**Hello**”

“Updating” a Rolling Hash

- ▶ Start with old hash, subtract character to be removed.
- ▶ “Shift” by multiplying by 128.
- ▶ Add new character.
- ▶ Takes $\Theta(1)$ time.

```
def update_base_128_hash(s, start, stop, old):  
    # assumes ASCII encoding, base 128  
    length = stop - start  
    removed_char = ord(s[start - 1]) * 128**(length - 1)  
    added_char = ord(s[stop - 1])  
    return (old - removed_char) * 128 + added_char
```

```
»> base_128_hash("Hello", 0, 3)
1192684
»> base_128_hash("Hello", 1, 4)
1668716
»> update_base_128_hash("Hello", 1, 4, 1192684)
1668716
```

Note

- ▶ In this hashing strategy, there are no collisions!
- ▶ Two different string have two different hashes.
- ▶ But as we'll see... it isn't practical.

Rabin-Karp

```
def rabin_karp(s, p):
    hashed_window = base_128_hash(s, 0, len(p), q)
    hashed_pattern = base_128_hash(p, 0, len(p), q)
    match_locations = []

    if s[0:len(p)] == p:
        match_locations.append(0)

    for i in range(1, len(s) - len(p) + 1):
        # update the hash
        hashed_window = update_base_128_hash(s, i, i + len(p), hashed_window)

        # hashes are unique; no collisions
        if hashed_window == hashed_pattern:
            match_locations.append(i)

    return match_locations
```

Example

- ▶ `s = "this is a test",`
`p = "is"`
- ▶
`hashed_pattern = 13555`

| i | s[...] | hashed_window |
|----|--------|---------------|
| 0 | "th" | 14952 |
| 1 | "hi" | 13417 |
| 2 | "is" | 13555 |
| 3 | "s " | 14752 |
| 4 | " i" | 4201 |
| 5 | "is" | 13555 |
| 6 | "s " | 14752 |
| 7 | " a" | 4193 |
| 8 | "a " | 12448 |
| 9 | " t" | 4212 |
| 10 | "te" | 14949 |
| 11 | "es" | 13043 |
| 12 | "st" | 14836 |

Large Numbers

- ▶ Hashing because integer comparison takes $\Theta(1)$ time.
- ▶ Only true if integers are small enough.
- ▶ Our integers can get **very large**.

$$128^{|p|-1}$$

Example

```
»> p = "University of California"  
»> base_128_hash(p, 0, len(p))  
250986132488946228262668052010265908722774302242017
```

Large Integers

- ▶ In some languages, large integers will overflow.
- ▶ Python has arbitrary size integers.
- ▶ But comparison no longer takes $\Theta(1)$

Solution

- ▶ Use **modular arithmetic**.

- ▶ Example:

$$(4 + 7) \% 3 = 11 \% 3 = 2$$

- ▶ Results in much smaller numbers.

Idea

- ▶ Choose a random prime number $> |m|$.
- ▶ Do all arithmetic modulo this number.

```
def base_128_hash(s, start, stop, q):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    p = 0
    total = 0
    while stop > start:
        total = (total + ord(s[stop-1]) * 128**p) % q
        p += 1
        stop -= 1
    return total

def update_base_128_hash(s, start, stop, old, q):
    # assumes ASCII encoding, base 128
    length = stop - start

    removed_char = ord(s[start - 1]) * 128**(length - 1)
    added_char = ord(s[stop - 1])

    return ((old - removed_char) * 128 + added_char) % q
```

Note

- ▶ Now there can be collisions!
- ▶ Even if window hash matches pattern hash, need to verify that strings are indeed the same.

```
def rabin_karp(s, p, q):
    hashed_window = base_128_hash(s, 0, len(p), q)
    hashed_pattern = base_128_hash(p, 0, len(p), q)
    match_locations = []

    if s[0:len(p)] == p:
        match_locations.append(0)

    for i in range(1, len(s) - len(p) + 1):
        # update the hash
        hashed_window = update_base_128_hash(s, i, i + len(p), hashed_window, q)

        if hashed_window == hashed_pattern:
            # make sure this isn't a false match due to collision
            if s[i:i + len(p)] == p:
                match_locations.append(i)

    return match_locations
```

Time Complexity

- ▶ If q is prime and $> |p|$, the chance of two different strings colliding is small.
- ▶ From before: if the number of matches is small, Rabin-Karp will take $\Theta(|s| + |p|)$ expected time.
- ▶ Since $|p| \leq |s|$, this is $\Theta(s)$.
- ▶ Worst-case time: $\Theta(|s| \cdot |p|)$.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 15 | Part 1

Today's Lecture

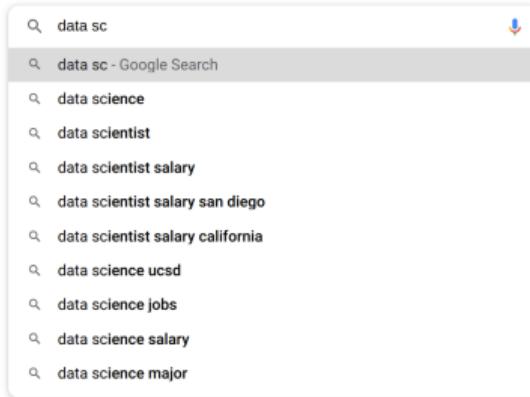
String Data Structures

- ▶ One of the themes of this quarter:
- ▶ If you're doing something once, use an algorithm.
- ▶ If you're doing it over and over, use an appropriate data structure.

String Data Structures

- ▶ Over the next two lectures, we'll look at data structures for strings.
- ▶ Today: **tries** for efficient repeated **prefix queries**.

Autocompletion



DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 15 | Part 2

Tries

Trie

- ▶ A data structure for storing strings.
- ▶ Pronounced “try”, short for “retrieval”.
- ▶ Supports fast **prefix query** and **membership query**.

Prefixes

- ▶ A **prefix** p of a string s is a contiguous slice of the form $s[0:t]$, for some t .
- ▶ Examples:
 - ▶ "test" is a prefix of "testing"
 - ▶ "te" is a prefix of "testing"
 - ▶ "sa" is a prefix of "san diego"
 - ▶ "di" is **not** a prefix of "san diego"

Prefix Query

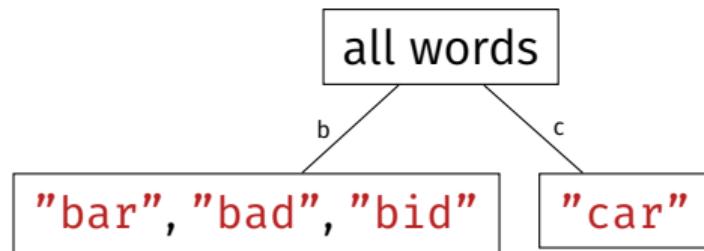
- ▶ **Given:** a collection of n strings and a prefix, p .
- ▶ **Find:** all strings in the collection for which p is a prefix.
- ▶ Example:
 - ▶ "bar", "bad", "bid", "car"
 - ▶ $p = \text{"ba"}$

Brute Force

- ▶ Loop over each of n strings, compare against prefix p .
- ▶ Worst-case time: $\Theta(n \cdot |p|)$

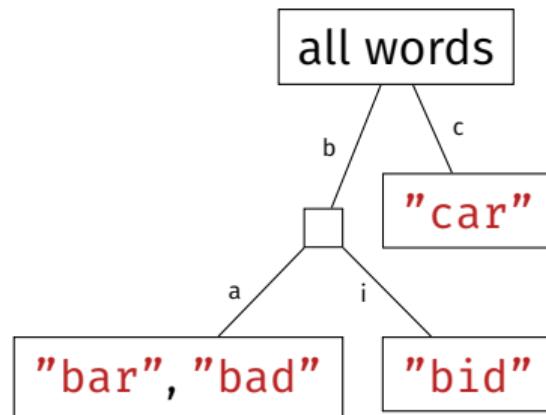
Trie: Motivation

"bar", "bad", "bid", "car"



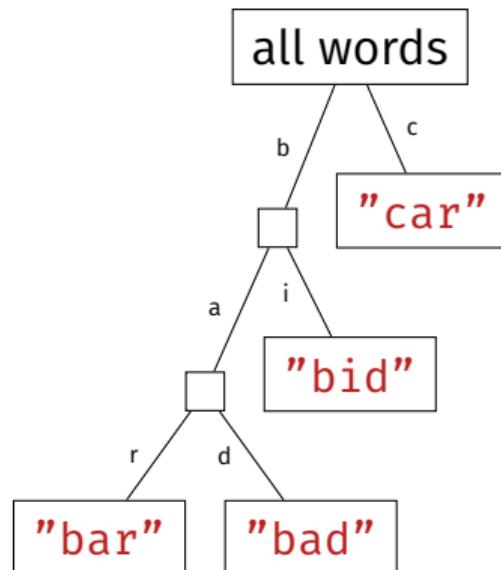
Trie: Motivation

"bar", "bad", "bid", "car"

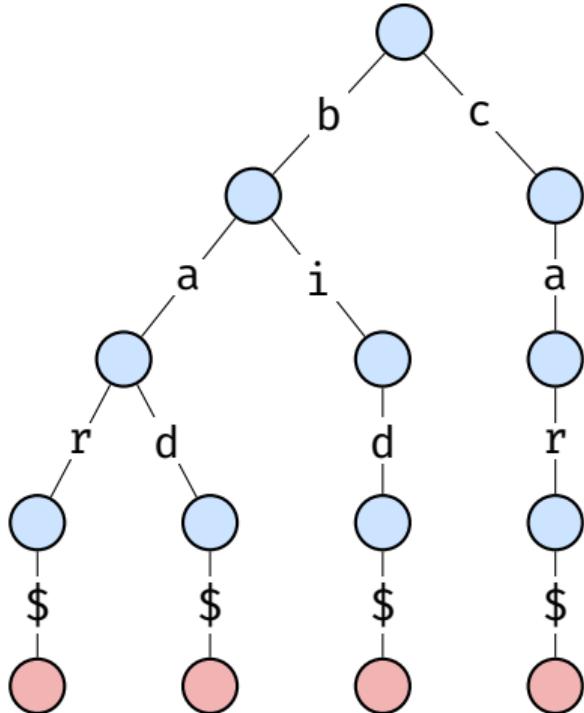


Trie: Motivation

"bar", "bad", "bid", "car"

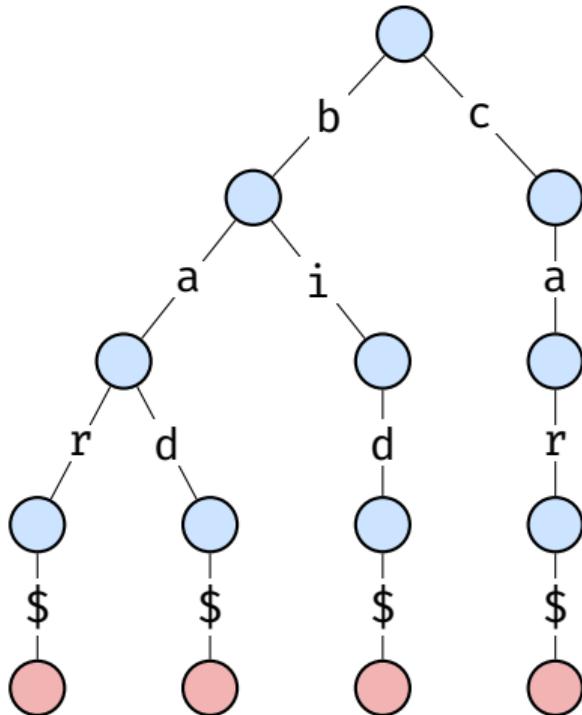


Tries



- ▶ Internal nodes represent prefixes.
- ▶ Leaf nodes represent full words.
- ▶ Edges are characters.
- ▶ Words are encoded as paths.

Sentinels



- ▶ \$ is a **sentinel**.
- ▶ It is different from the dollar sign character.
- ▶ It marks the end of a word.
- ▶ Used to show that "bar" in trie, but "ba" not.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 15 | Part 3

Implementing Tries

Representation

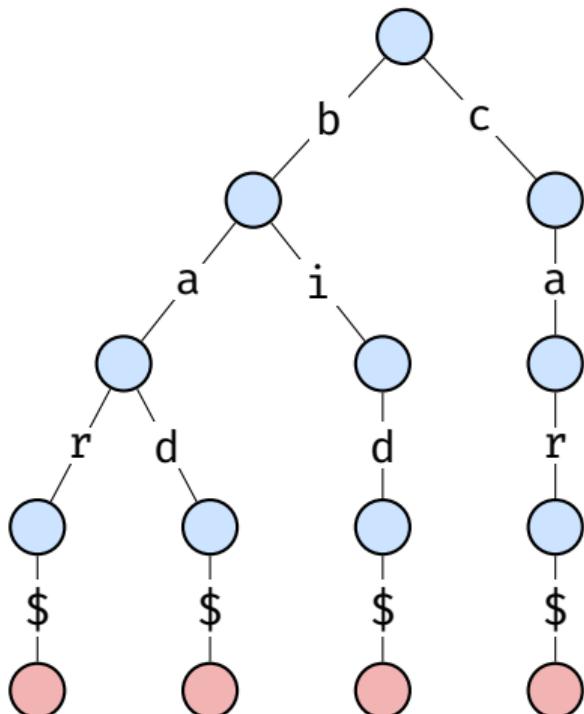
- ▶ Each node has a hash table / array mapping characters to a child nodes.
- ▶ Sentinel represented with a singleton object?

```
END_OF_STRING = object()

class TrieNode:

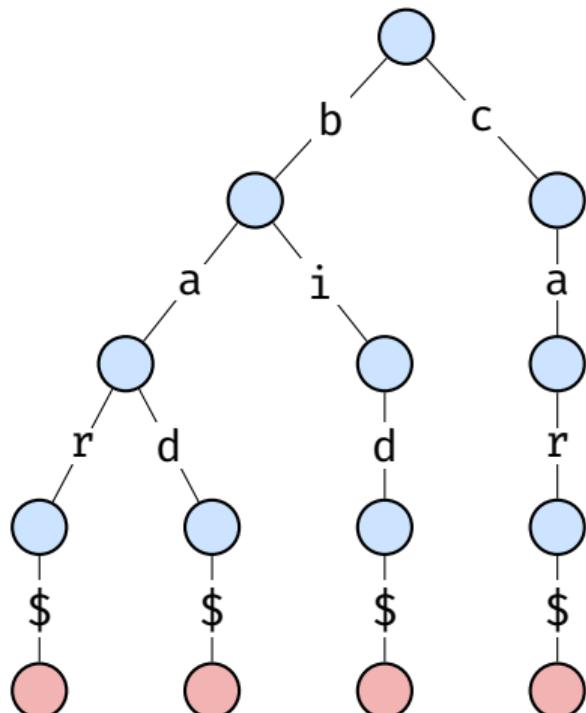
    def __init__(self):
        self.children = {}
```

Insertion



- ▶ “Walk” down tree, creating edges and nodes as necessary.
- ▶ When no more letters left, add sentinel.
- ▶ Example: insert **“cab”, “card”, “zoo”**

Insertion (Recursive)



- ▶ Suppose we `.insert(s)` on root node.
- ▶ If `s[0]` not in `self.children`, create a new node.
- ▶ Otherwise, let child be `self.children[s[0]]`.
- ▶ Recursively insert `s[1:]` into child.

```
def insert(self, s, start=0, stop=None):
    """Insert s[start:stop] into the trie."""
    if stop is None:
        stop = len(s)

    if start >= stop:
        self.children[END_OF_STRING] = TrieNode()
        return

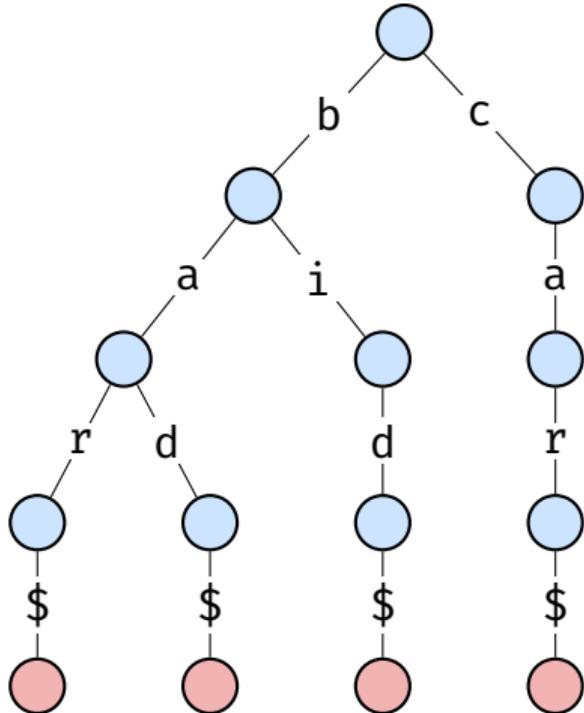
    if s[start] not in self.children:
        self.children[s[start]] = TrieNode()

    child = self.children[s[start]]
    child.insert(s, start + 1, stop)
```

Insertion Time Complexity

- ▶ $\Theta(|w|)$ time, where w is the string inserted.
- ▶ No matter how many elements in trie!

Walk



- ▶ Useful operation.
- ▶ Given a prefix, “walk” down tree.
- ▶ If we “fall off”, raise error.
- ▶ Otherwise, return last node seen.
- ▶ Examples: **“ba”**, **“bo”**

```
def walk(self, s, start=0, stop=None):
    """Walk the trie following s[start:stop].
    Raises ValueError if falls off tree.
    Returns last node encountered otherwise."""
    if stop is None:
        stop = len(s)

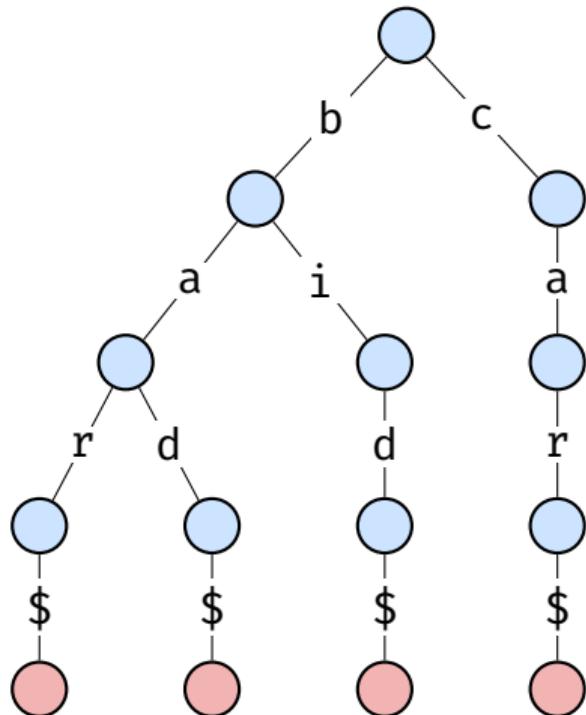
    if start >= stop:
        return self

    if s[start] not in self.children:
        raise ValueError('Fell off tree.')
    else:
        child = self.children[s[start]]
        return child.walk(s, start + 1, stop)
```

Walk Time Complexity

- ▶ Worst-case $\Theta(|p|)$ time, where p is the prefix searched.
- ▶ No matter how many elements in trie!

Membership Query



- ▶ Given p , return True/False if p in collection.
- ▶ “Walk” down tree.
- ▶ If we “fall off”, return False.
- ▶ If not, check that sentinel in children.
- ▶ Examples: “ba”, “bad”

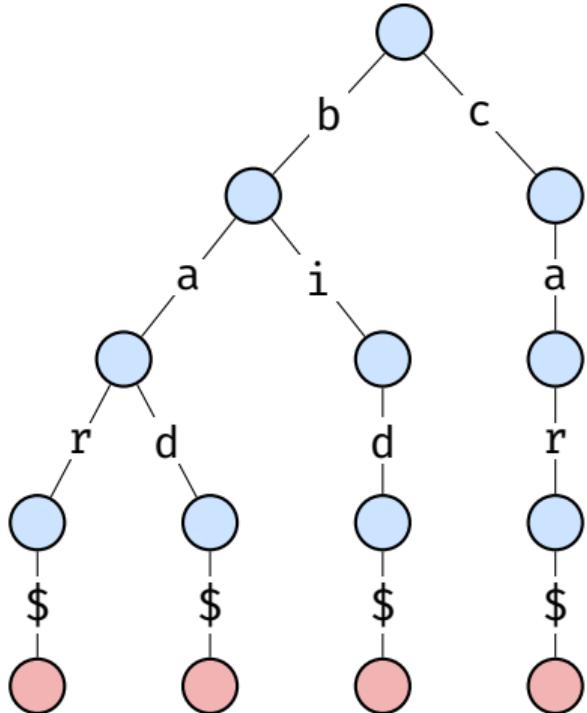
```
def membership_query(self, s, start=0, stop=None):
    """Determine if s[start:stop] is in trie."""
    try:
        node = self.walk(s, start, stop)
    except ValueError:
        return False

    return END_OF_STRING in node.children
```

Membership Query Time Complexity

- ▶ Worst-case $\Theta(|w|)$ time, where w is the prefix searched.
- ▶ No matter how many elements in trie!

Produce



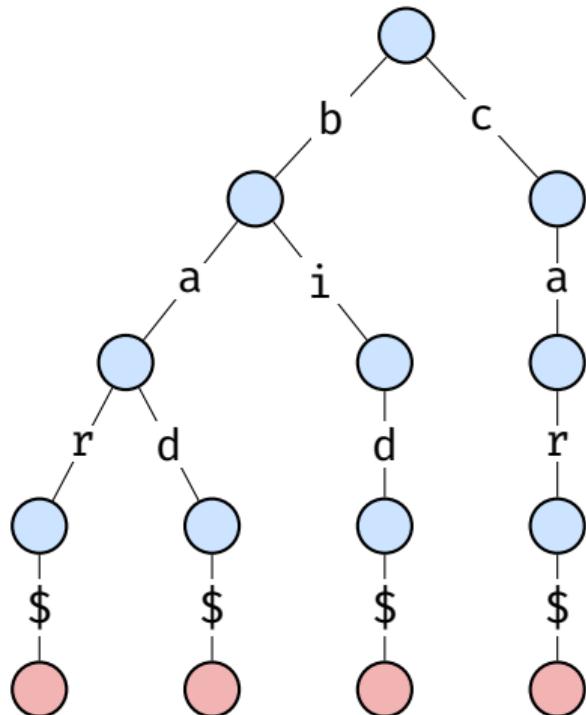
- ▶ Goal: generate all words in subtree.
- ▶ Perform a DFS, keeping track of letters along path.
- ▶ If we find a sentinel, print path.

```
def produce(self, pathchars=''):
    """Generate the words in the trie."""
    for letter, child in self.children.items():
        if letter is END_OF_STRING:
            yield pathchars
        else:
            yield from child.produce(pathchars + letter)
```

Produce Time Complexity

- ▶ Worst-case $\Theta(\ell)$ time, where ℓ is total length of all strings stored in the trie.
- ▶ If length strings is considered a constant, this is $\Theta(n)$.

Prefix Query (Complete)



- ▶ Given p , return all completions.
- ▶ “Walk” down tree.
- ▶ If we “fall off”, return empty list..
- ▶ If not, produce all nodes in subtrie.
- ▶ Examples: “ba”, “bad”

```
def complete(self, prefix):
    try:
        node = self.walk(prefix)
    except ValueError:
        return []
    return list(node.produce())
```

Prefix Query Time Complexity

- ▶ Worst-case $\Theta(|p| + \ell_p)$ time, where p is the prefix searched and ℓ_p is the total length of all matches.
- ▶ If length is considered constant, this is $\Theta(|p| + z)$, where z is number of matches.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 15 | Part 4

Demo

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 16 | Part 1

Suffix Tries and Suffix Trees

Last Time

- ▶ We have seen **tries**.
- ▶ They provide for very fast prefix searches.
- ▶ But we don't do a lot of prefix searches...

Today's Lecture

- ▶ A way of using tries for solving much more interesting problems.

String Matching

(Substring Search)

- ▶ **Given:** a string, s , and a pattern string p
- ▶ **Determine:** all locations of p in s
- ▶ Example:

$s = \text{"GATTACATACG"}$ $p = \text{"TAC"}$

Recall

- ▶ We've solved this with Rabin-Karp in $\Theta(|s| + |p|)$ expected time.
- ▶ What if we want to do *many* searches?
- ▶ Let's build a data structure for fast substring search.

Suffixes

- ▶ A **suffix** p of a string s is a contiguous slice of the form $s[t:]$, for some t .
- ▶ Examples:
 - ▶ "ing" is a suffix of "testing"
 - ▶ "ting" is a suffix of "testing"
 - ▶ "di" is **not** a suffix of "san diego"

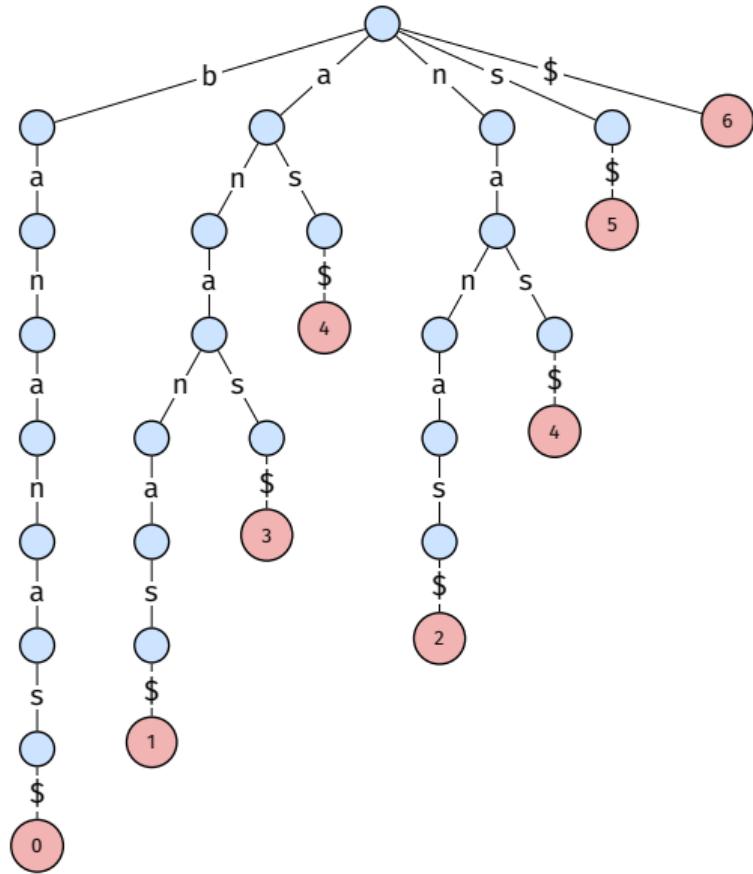
A Very Important Observation

- ▶ w is a substring of s if and only if w is a **prefix** of some **suffix** of s .

| | |
|--------------------|--------------|
| $s = "california"$ | "california" |
| $p_1 = "ifo"$ | "alifornia" |
| $p_2 = "lif"$ | "lifornia" |
| $p_3 = "flurb"$ | "ifornia" |
| | "fornia" |
| | "ornia" |
| | "rnia" |
| | "nia" |
| | "ia" |
| | "a" |
| | "" |

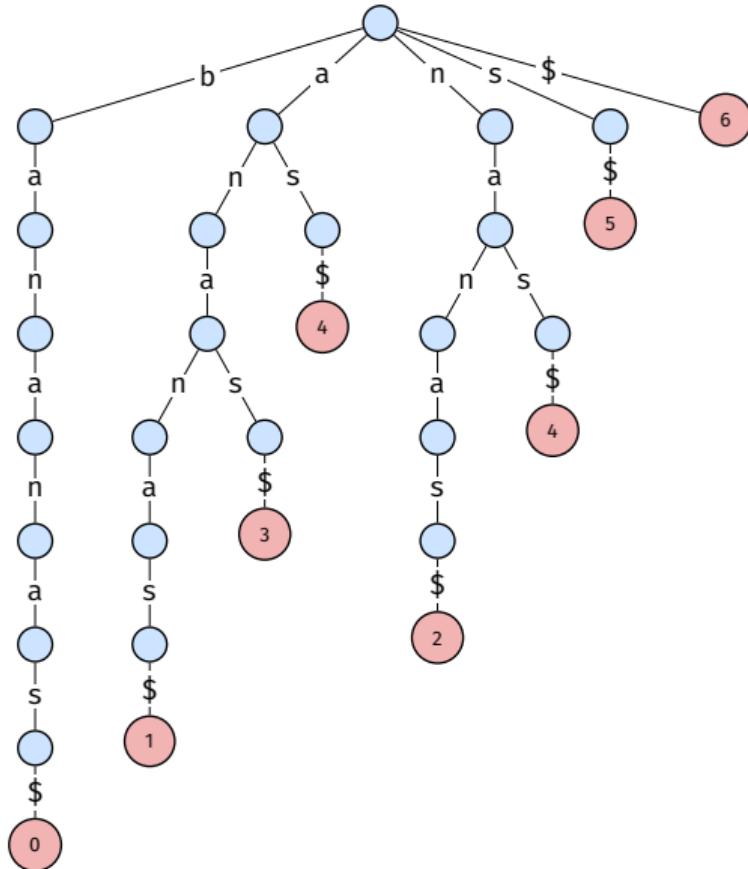
Idea

- ▶ Last time: can do fast prefix search with trie.
- ▶ Idea for fast repeated substring search of s :
 - ▶ Keep track of all suffixes of s in a trie.
 - ▶ Given a search pattern p , look up p in trie.
- ▶ A trie containing all suffixes of s is a **suffix trie** for s .



`s[0:]: "bananas"`
`s[1:]: "ananas"`
`s[2:]: "nanas"`
`s[3:]: "anas"`
`s[4:]: "nas"`
`s[5:]: "as"`
`s[6:]: "s"`
`s[7:]: ""`

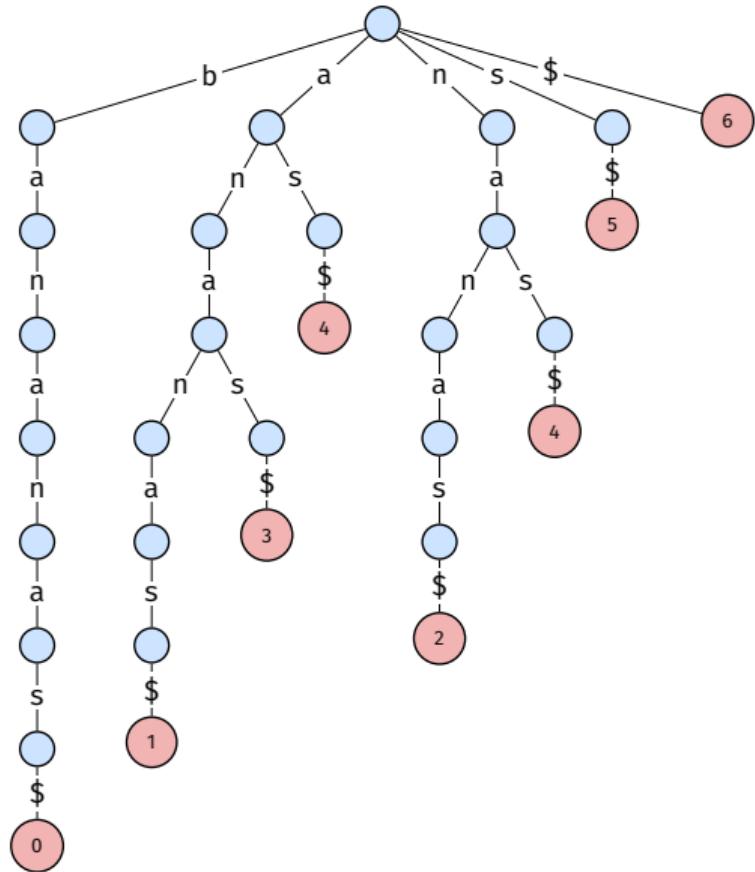
Substring Search



- ▶ Given pattern p , walk down suffix trie.
- ▶ If we fall off, return $[]$.
- ▶ Else, do a DFS of that subtree. Each leaf is a match.
- ▶ Time complexity: $\Theta(|p| + k)$, where k is number of nodes in the subtree.

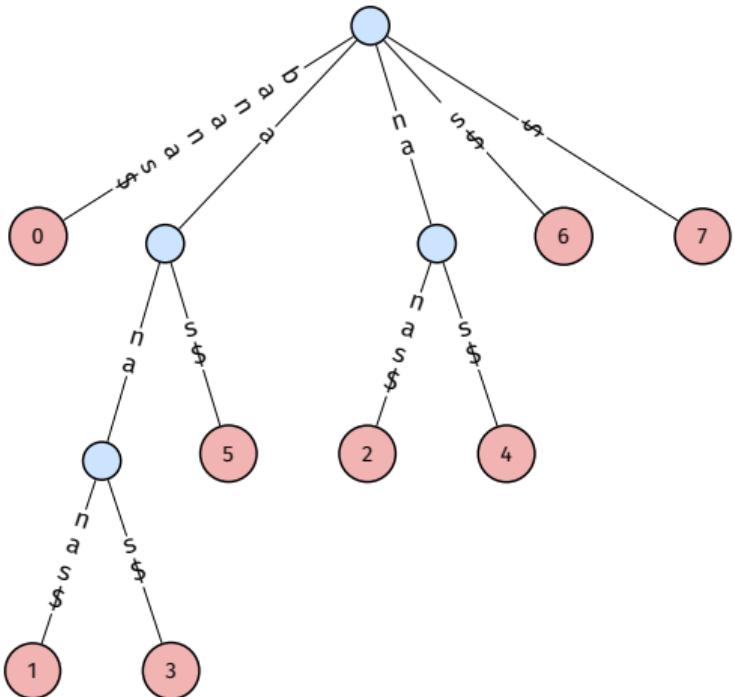
Problems

- ▶ In the worst case, a suffix tree for s has $\Theta(|s|^2)$ nodes.
 - ▶ Suffixes of length $|s|$, $|s| - 1$, $|s| - 2$, ...,
- ▶ So substring search can take $\Theta(|s|^2)$ time.
- ▶ Construction therefore takes $\Omega(|s|^2)$, too.
 - ▶ Naïve algorithm takes $\Theta(|s|^2)$ time.
- ▶ Takes $\Theta(|s|^2)$ storage.



Silly Nodes

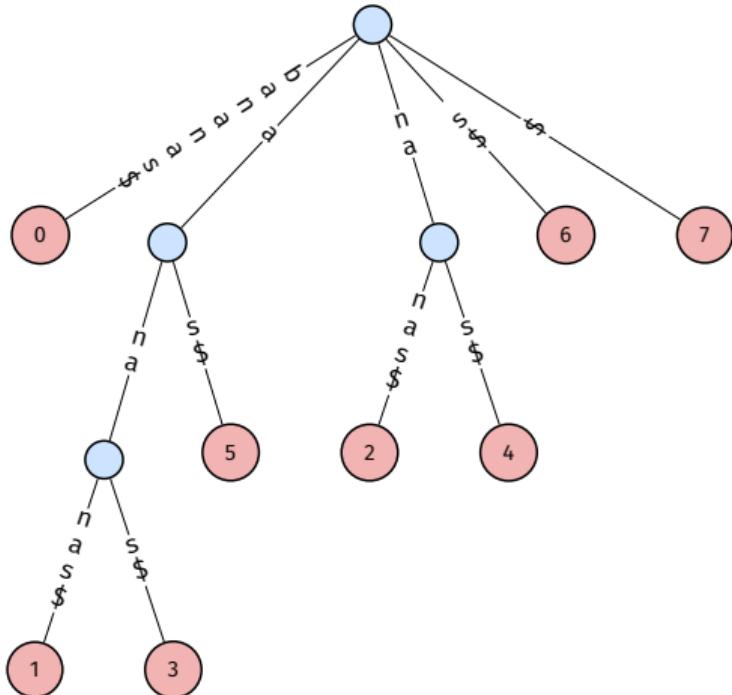
- ▶ A **silly node** has one child.
 - ▶ Fix: “Collapse” silly nodes?



“Collapsing” Silly Nodes

```
s[0:]: "bananas"  
s[1:]: "ananas"  
s[2:]: "nanas"  
s[3:]: "anas"  
s[4:]: "nas"  
s[5:]: "as"  
s[6:]: "s"  
s[7:]: ""
```

Suffix Trees



- ▶ This is a **suffix tree**^a.
- ▶ Internal nodes represent **branching words**.
- ▶ Leaf nodes represent **suffixes**.
- ▶ Leaf nodes are labeled by starting index of suffix.

^aNot to be confused with a **suffix trie**.

Branching Words

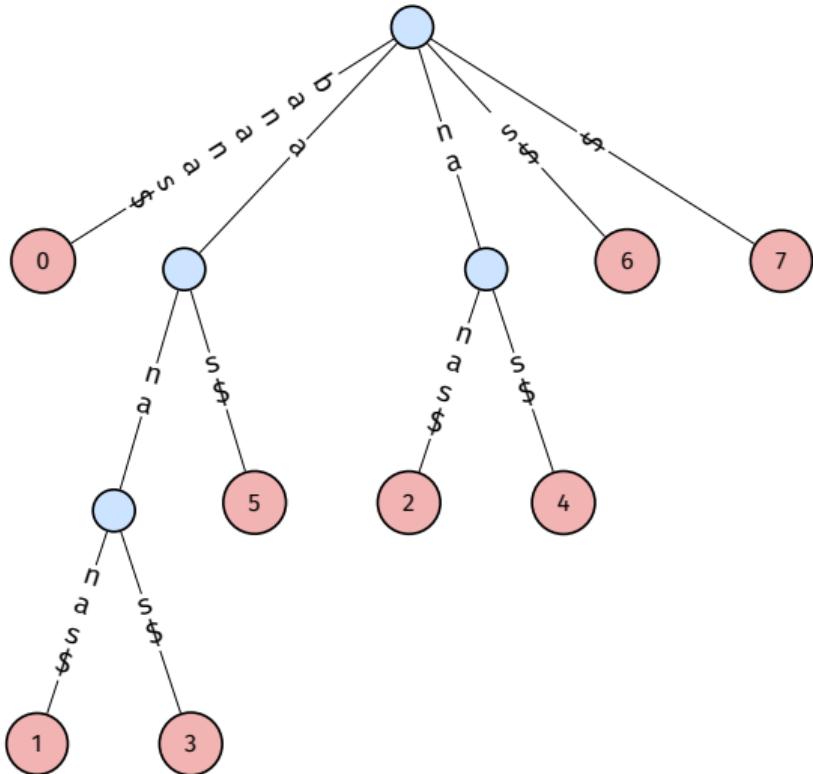
- ▶ Suppose s' is a substring of s .
- ▶ An **extension** of s' is a substring of s of the form:

$s' + \text{one more character}$

- ▶ Example: $s = \text{"bananas"},$
 - ▶ $\text{"ana"} \rightarrow \{\text{"anas"}, \text{"anan"}\}$
 - ▶ $\text{"a"} \rightarrow \{\text{"an"}, \text{"as"}\}$
 - ▶ $\text{"ban"} \rightarrow \{\text{"bana"}\}$

Branching Words

- ▶ A **branching word** is a substring of s with more than one extension.
- ▶ Example: $s = \text{"bananas"},$
 - ▶ $\text{"ana"} \rightarrow \{\text{"anas"}, \text{"anan"}\}$ (**yes**)
 - ▶ $\text{"a"} \rightarrow \{\text{"an"}, \text{"as"}\}$ (**yes**)
 - ▶ $\text{"ban"} \rightarrow \{\text{"bana"}\}$ (**no**)



Branching Words

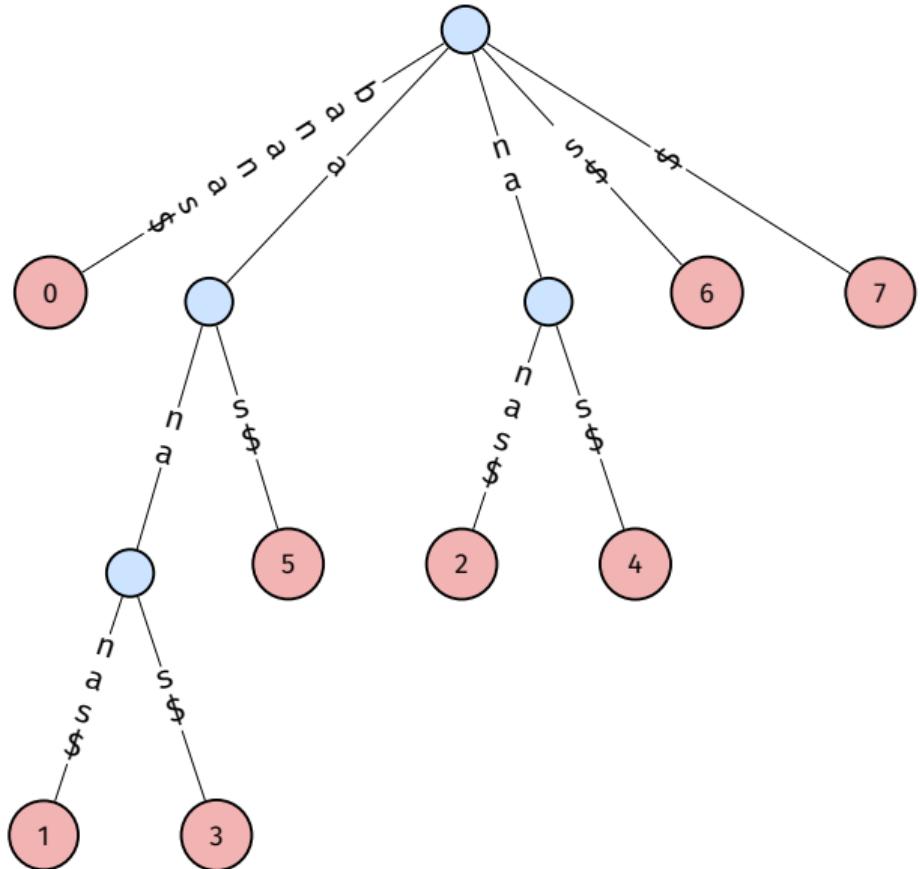
- ▶ "a", "ana", "na" are branching words in "bananas".
- ▶ Internal nodes of the suffix tree represent branching words.

Number of Branching Words

- ▶ There are $O(|s|)$ branching words.
- ▶ Proof:
 - ▶ Remove all of the internal nodes (branching words).
 - ▶ Now there are $|s|$ forests (one for each suffix).
 - ▶ Add the internal nodes back, one at a time.
 - ▶ Each addition reduces number of forests by one.
 - ▶ After adding $|s| - 1$ internal nodes, forest has one tree.
 - ▶ Therefore there are at most $|s| - 1$ internal nodes.

Size of Suffix Trees

- ▶ A suffix tree for any string s has $\Theta(|s|)$ nodes.



Substring Search

- ▶ Given pattern p , walk down suffix trie.
 - ▶ If we fall off, return $[]$.
 - ▶ Else, do a DFS of that subtree. Each leaf is a match.
 - ▶ Time complexity: $\Theta(|p| + z)$, where z is number of matches.

Naïve Construction Algorithm

- ▶ First, build a suffix trie in $\Omega(|s|^2)$ time in worst case.
 - ▶ Loop through the $|s|$ suffixes, insert each into trie.
- ▶ Then “collapse” silly nodes.
- ▶ Takes $\Omega(|s|^2)$ time. **Bad**.

Faster Construction

- ▶ There are (surprisingly) $O(|s|)$ algorithms for constructing suffix trees.
- ▶ For instance, Ukkonen's Algorithm.

Single Substring Search

Rabin-Karp

- ▶ Rolling hash of window.
- ▶ $\Theta(|s| + |p|)$ time.

Suffix Tree

- ▶ Construct suffix tree; $\Theta(|s|)$ time.
- ▶ Search it; $\Theta(|p| + z)$ time.
- ▶ Total: $\Theta(|s| + |p|)$, since $z = O(|s|)$.

Multiple Substring Search

Multiple searches of s with different patterns, $p_1, p_2,$

...

Rabin-Karp

- ▶ First search: $\Theta(|s| + |p_1|)$.
- ▶ Second search: $\Theta(|s| + |p_2|)$.

Suffix Tree

- ▶ Construct suffix tree; $\Theta(|s|)$ time.
- ▶ First search: $\Theta(|p_1| + z_1)$ time.
- ▶ Second search: $\Theta(|p_2| + z_2)$ time.
- ▶ Typically $z \ll |s|$

Suffix Trees

- ▶ Many other string problems can be solved efficiently with suffix trees!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 16 | Part 2

Longest Repeated Substring

Repeating Substrings

- ▶ A substring of s is **repeated** if it occurs more than once.
- ▶ Example: $s = \text{"bananas"}$.
 - ▶ "na"
 - ▶ "ana"

Repeating Substrings in Genomics

- ▶ A repeated substring in a DNA sequence is interesting.
- ▶ It's a “building block” of that gene.

GATTACAGTAGCGATGATTACAGGTGATTACA

Repeating Substrings in Genomics

- ▶ A repeated substring in a DNA sequence is interesting.
- ▶ It's a “building block” of that gene.

GATTACAGTAGCGATGATTACAGGTGATTACA

Longest Repeated Substrings

- ▶ The longer a repeated substring, the more interesting.
- ▶ **Given:** a string, s .
- ▶ **Find:** a repeated substring with longest length.

Brute Force

- ▶ Keep a dictionary of substring counts.
- ▶ Loop a window of size 1 over s .
- ▶ Loop a window of size 2 over s .
- ▶ Loop a window of size 3 over s , etc.
- ▶ $\Theta(|s|^2)$ time.

Suffix Trees

- We'll do this in $\Theta(|s|)$ time with a suffix tree.

Branching Words & Repeated Substrings

- ▶ Recall: a branching word is a substring with more than one extension.
- ▶ If a substring is repeated, is it a branching word?

Branching Words & Repeated Substrings

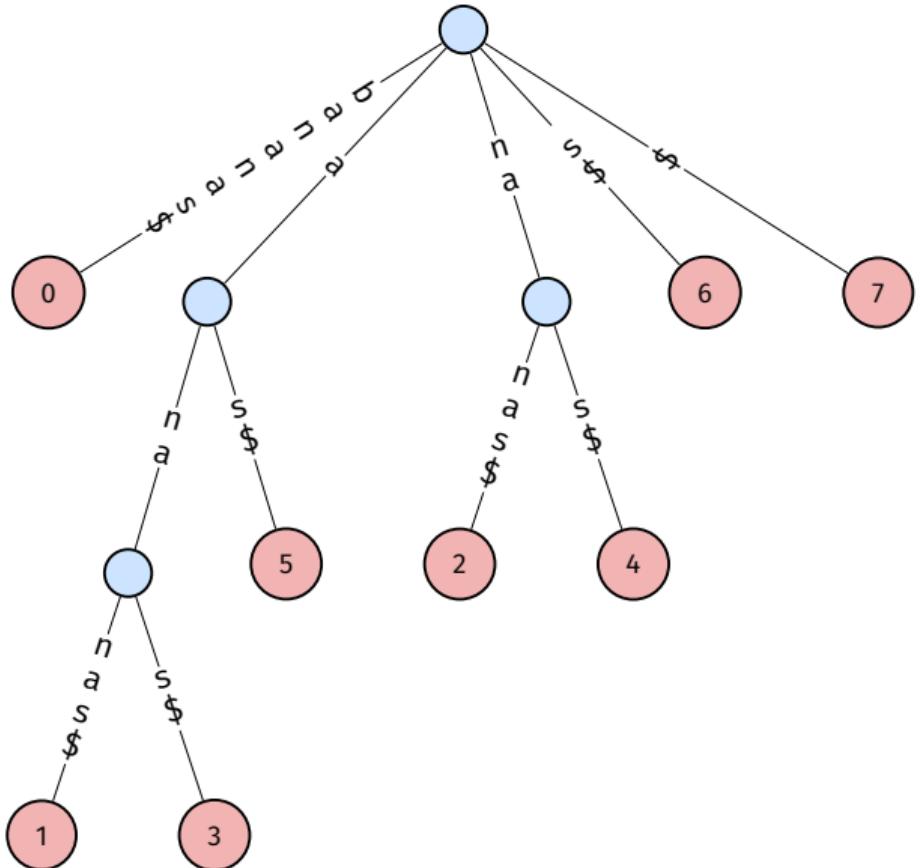
- ▶ Recall: a branching word is a substring with more than one extension.
- ▶ If a substring is repeated, is it a branching word?
- ▶ **No.** Example: "barkbark".
 - ▶ "bar" is repeated, **not** branching: {"bark"}.
 - ▶ "bark" is repeated, **is** branching:
{"barkb", "bark\$"}.

Claim

- ▶ If a substring w is repeated but not a branching word, it can't be the **longest**.
- ▶ Why? Since it isn't branching, it has one extension: w' .
- ▶ w' must also repeat, since w repeats.
- ▶ w' is longer than w , so w can't be the longest.

Claim

- ▶ A longest repeated substring must be a branching word.
- ▶ Therefore, must be an internal node of the suffix tree of s .



LRS

- ▶ Build suffix tree in $\Theta(|s|)$ time.
- ▶ Do a DFS in $\Theta(|s|)$ time.
- ▶ Keep track of “deepest” internal node. (Depth determined by number of characters.)
- ▶ This is a longest repeated substring; found in $\Theta(|s|)$ time.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 1

Today's Lecture

Massive Sets

- ▶ You've collected 1 billion tweets.¹
- ▶ **Goal:** given the text of a new tweet, is it already in the data set?

¹This is about two days of activity.

Membership Queries

- ▶ We want to perform a **membership query** on a collection of strings.
- ▶ Hash tables support $\Theta(1)$ membership queries.
- ▶ **Idea:** so let's use a hash table (Python: `set`).

Problem: Memory

- ▶ How much memory would a **set** of 1 billion strings require?
- ▶ Assume average string has 100 ASCII characters.
 $(8 \text{ bits per char}) \times (100 \text{ chars}) \times 1 \text{ billion} = 100 \text{ gigabytes}$
- ▶ That's way too large to fit in memory!

Today's Lecture

- ▶ **Goal:** fast membership queries on massive data sets.
- ▶ Today's answer: **Bloom filters**.

DSC 190

DATA STRUCTURES & ALGORITHMS

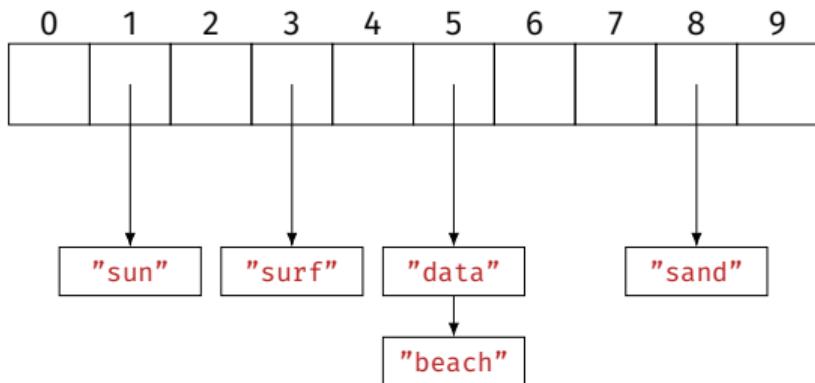
Lecture 17 | Part 2

Bit Arrays

The Challenge

- ▶ We want to perform membership queries on a massive collection (too large to fit in memory).
- ▶ We want to remember which elements are in the collection...
- ▶ ...*without* actually storing all of the elements.
- ▶ From hash tables to Bloom filters in 3 steps.

First Stop: Hash Tables



| s | hash(s) |
|---------|---------|
| "surf" | 3 |
| "sand" | 8 |
| "data" | 5 |
| "sun" | 1 |
| "beach" | 5 |

Memory Usage

- ▶ **Problem:** we're storing all of the elements.
- ▶ Why? To resolve collisions.
- ▶ **Fix:** ignore collisions.

Second Stop: Hashing Into Bit Arrays

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

| s | hash(s) |
|---------|---------|
| "surf" | 3 |
| "sand" | 8 |
| "data" | 5 |
| "sun" | 1 |
| "beach" | 5 |

- ▶ Use a bit array `arr` of size c .
- ▶ **Insertion:** Set `arr[hash(x)] = 1`.
- ▶ **Query:** Check if `arr[hash(x)] = 1`.

Second Stop: Hashing Into Bit Arrays

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

| s | hash(s) |
|---------|---------|
| "surf" | 3 |
| "sand" | 8 |
| "data" | 5 |
| "sun" | 1 |
| "beach" | 5 |

- ▶ Use a bit array arr of size c .
- ▶ **Insertion:** Set $\text{arr}[\text{hash}(x)] = 1$.
- ▶ **Query:** Check if $\text{arr}[\text{hash}(x)] = 1$.
- ▶ Can be **wrong!**

False Positives

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

| s | hash(s) |
|---------|---------|
| "surf" | 3 |
| "sand" | 8 |
| "data" | 5 |
| "sun" | 1 |
| "beach" | 5 |

- ▶ Query can return **false positives**.
- ▶ e.g.,
`hash("ucsd") == 3`
- ▶ Cannot return false negatives.

Memory Usage

- ▶ Requires c bits, where c is size of the bit array.
- ▶ False positive rate depends on c .
 - ▶ c is small \rightarrow more collisions \rightarrow more errors
 - ▶ c is large \rightarrow fewer collisions \rightarrow fewer errors
- ▶ **Tradeoff:** get more accuracy at cost of memory.

False Positive Rate

- ▶ What is the probability of a false positive?
- ▶ Suppose there are c buckets, and we've inserted n elements so far.
- ▶ We query an object x that we haven't seen before.
- ▶ False positive $\Leftrightarrow \text{arr}[\text{hash}(x)] == 1$.

False Positive Rate

- ▶ Assume `hash` assigns bucket uniformly at random.
 - ▶ If $x \neq y$ then, $\mathbb{P}(\text{hash}(x) = \text{hash}(y)) = 1/c$
- ▶ Prob. that first element does not collide with x :
 $1 - 1/c$.
- ▶ Prob. that first two do not collide: $(1 - 1/c)^2$.
- ▶ Prob. that all n elements do not collide:
 $(1 - 1/c)^n$.

False Positive Rate

- ▶ Hint: for large z , $(1 - 1/z)^z \approx \frac{1}{e}$
- ▶ So the probability of no collision is:
$$(1 - 1/c)^n = [(1 - 1/c)^c]^{n/c} \approx e^{-n/c}$$
- ▶ This is the probability of no false positive.
- ▶ Probability of false positive upon querying x :
 $\approx 1 - e^{-n/c}$

False Positive Rate

- ▶ For fixed query, probability of false positive:
 $\approx 1 - e^{-n/c}$.
 - ▶ n : number of elements stored
 - ▶ c : size of array (number of bits)
- ▶ Randomness is over choice of hash function.
 - ▶ Once hash function is fixed, the result is always the same.

Fixing False Positive Rate

- ▶ Suppose we'll tolerate false positive rate of ε .
- ▶ Assume that we'll store around n elements.
- ▶ We can choose c :

$$1 - e^{-n/c} = \varepsilon \quad \Rightarrow \quad c = -\frac{n}{\ln(1 - \varepsilon)}$$

Example

- ▶ Suppose we want $\leq 1\%$ error.
- ▶ Previous slide says our bit array needs to be 100 times larger than number of elements stored.²
- ▶ Memory when $n = 10^9$: 1 billion bits $\times 100 = 12.5$ GB.
- ▶ Can we do better?

²We could have guessed this, huh?

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 3

Bloom Filters

Wasted Space

- ▶ Suppose we want $\leq 1\%$ error.
- ▶ Our bit array needs to be 100 times larger than number of elements stored.
- ▶ That's a lot of **wasted space!**

Third Stop: Multiple Hashing

- ▶ **Idea:** use several smaller bit arrays, each with own hash function.

Third Stop: Multiple Hashing

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

| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 3 | 7 |
| "sand" | 8 | 7 |
| "data" | 5 | 4 |
| "sun" | 1 | 9 |
| "beach" | 5 | 6 |

- ▶ Use k bit arrays of size c , each with own independent hash function.

- ▶ **Insertion:** Set
 $\text{arr}_1[\text{hash}_1(x)] = 1$,
 $\text{arr}_2[\text{hash}_2(x)] = 1$,
...,
 $\text{arr}_k[\text{hash}_k(x)] = 1$.

Third Stop: Multiple Hashing

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

| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 3 | 7 |
| "sand" | 8 | 7 |
| "data" | 5 | 4 |
| "sun" | 1 | 9 |
| "beach" | 5 | 6 |

- ▶ Use k bit arrays of size c , each with own independent hash function.
- ▶ **Query:** Return **True** if **all** of
arr_1[hash_1(x)] = 1,
arr_2[hash_2(x)] = 1,
...,
arr_k[hash_k(x)] = 1.
- ▶ Example:
hash_1("hello") == 3,
hash_2("hello") == 2

Intuition

- ▶ False positive occurs only if false positive in **all** tables.
- ▶ This is pretty unlikely.
- ▶ If false positive rate in one table is small (but not tiny), probability false positive in all tables is still tiny.

More Formally

- ▶ Probability of false positive in first table:
 $\approx 1 - e^{-n/c}.$
- ▶ Probability of false positive in all k tables:
 $\approx (1 - e^{-n/c})^k.$
- ▶ Example: if $c = 4n$ and $k = 3$, error rate is $\approx 1\%$.
- ▶ Uses only $12 \times n$ bits, as opposed to $100 \times n$ from before.

Last Stop: Bloom Filters

- ▶ How many different bit arrays do we use? (What is k ?)
- ▶ How large should they be? (What is c ?)
- ▶ **Bloom filters:** use k hash functions, but only one medium-sized array.

Last Stop: Bloom Filters

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |

| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 13 | 17 |
| "sand" | 8 | 6 |
| "data" | 15 | 1 |
| "sun" | 1 | 3 |
| "beach" | 5 | 9 |

- ▶ Use one bit arrays of size c , but k hash functions.
- ▶ **Insertion:** Set $\text{arr}[\text{hash}_1(x)] = 1$, $\text{arr}[\text{hash}_2(x)] = 1$, ..., $\text{arr}[\text{hash}_k(x)] = 1$.

Last Stop: Bloom Filters

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 13 | 17 |
| "sand" | 8 | 6 |
| "data" | 15 | 1 |
| "sun" | 1 | 3 |
| "beach" | 5 | 9 |

- ▶ Use one bit arrays of size c , but k hash functions.
- ▶ **Query:** Return **True** if all of
arr[hash_1(x)] = 1,
arr[hash_2(x)] = 1,
...,
arr[hash_k(x)] = 1.
- ▶ Example:
hash_1("hello") == 3,
hash_2("hello") == 2

Intuition

- ▶ Multiple hashing allows bit arrays to be smaller.
- ▶ Even more efficient: let them share memory.
- ▶ “Overlaps” are just collisions; we can handle them.

False Positive Rate

- ▶ Consider querying new, unseen object x .
- ▶ We'll look at k bits.
 - ▶ $\text{arr}[\text{hash}_1(x)], \dots, \text{arr}[\text{hash}_k(x)]$.
- ▶ Fix one bit. What is the chance that it is already one?

False Positive Rate

- ▶ Probability of bit being zero after first element inserted: $(1 - 1/c)^k$
- ▶ After second element inserted: $(1 - 1/c)^{2k}$
- ▶ After all n elements inserted: $(1 - 1/c)^{nk}$
- ▶ And:

$$(1 - 1/c)^{nk} = [(1 - 1/c)^c]^{nk/c} \approx e^{-nk/c}$$

False Positive Rate

- ▶ Probability of bit being **one** after n elements inserted:

$$1 - e^{-nk/c}$$

- ▶ For a false positive, all k bits (for each hash function) need to be one.
- ▶ Assuming independence,³ probability of false positive:

$$(1 - e^{-nk/c})^k$$

³Only true approximately. If this bit was set, some other bit was not.

Minimizing False Positives

- ▶ For a fixed n and c , the number of hash functions k which minimizes the false positive rate is

$$k = \frac{c}{n} \ln 2$$

- ▶ Plugging this into the error rate:

$$\varepsilon = (1 - e^{-nk/c})^k \implies \ln \varepsilon = -\frac{c}{n} (\ln 2)^2$$

- ▶ If we fix ε , then $c = -n \ln \varepsilon / (\ln 2)^2$

Summary: Designing Bloom Filters

- ▶ Suppose we wish to store n elements with ϵ false positive rate.
- ▶ Allocate a bit array with $c = -n \ln \epsilon / (\ln 2)^2$ bits.
- ▶ Pick $k = \frac{c}{n} \ln 2$ hash functions.

Example

- ▶ Let $n = 10^9$, $\varepsilon = 0.01$.
- ▶ We need $c \approx 9.5n \rightarrow 10n$ bits = 1.25 GB.
- ▶ We choose $k = \frac{9.5n}{n} \ln 2 \rightarrow 7$ hash functions.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 4

Bloom Filters in Practice

Applications

- ▶ A cool data structure.
- ▶ Most useful when data is huge or memory is small.

Application #1

- ▶ De-duplicate 1 billion strings, each about 100 bytes.
- ▶ Memory required for `set`: 100 gigabytes.
- ▶ Instead:
 - ▶ Loop through data, reading one string at a time.
 - ▶ If not in Bloom filter, write it to file.
- ▶ With 1% error rate, takes 1.25 GB.

Application #2

- ▶ A ***k*-mer** is a substring of length *k* in a DNA sequence:

"GATTACATATAAGGTGTCGA"

- ▶ Useful: does a long string have a given *k*-mer?
- ▶ There are a *massive* number of possible *k*-mers.
 - ▶ 4^k , to be precise.
 - ▶ Example: there are over 10^{18} 30-mers.
- ▶ Slide window of size *k* over sequence, store each substring in Bloom filter.

Application #2

- ▶ Human genome is a 725 Megabyte string, 2.9 billion characters.
- ▶ To store all k -mers, each character stored k times.
- ▶ Storing 30-mers in `set` would take $30 \times 725 \text{ MB} \approx 22 \text{ GB}$.
- ▶ By “forgetting” the actual strings, Bloom filter (1% false positive) takes
2.9 billion bits $\approx 360 \text{ megabytes}$

Application #3

- ▶ Suppose you have a massive database on disk.
- ▶ Querying the database will take a while, since it has to go to disk.
- ▶ Build a Bloom filter, keep in memory.
 - ▶ If Bloom filter says x not in database, don't perform query.
 - ▶ Otherwise, perform DB query.
- ▶ Speeds up time of “misses”.

Limits

- ▶ Bloom filters are useful in certain circumstances.
- ▶ But they have disadvantages:
 - ▶ Need good idea of size, n , ahead of time.
 - ▶ There are false positives.
 - ▶ The elements are not stored (can't iterate over them).
- ▶ Often a **set** does just fine, with some care.

Example

- ▶ Suppose you have 1 billion tweets.
- ▶ Want to de-duplicate them by **tweet ID** (64 bit number).
- ▶ Total size: 8 gigabytes.
- ▶ I have 4 GB RAM. Should I use a Bloom filter?

De-duplication Strategy

- ▶ Design a hash function that maps each tweet ID to $\{1, \dots, 8\}$.

- ▶ Loop through tweet IDs one-at-a-time, hash, write to file:

`hash(x) == 3 → write to data_3.txt`

- ▶ Read in each file, one-at-a-time, de-duplicate with `set`, write to `output.txt`

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 18 | Part 1

The Count-Min Sketch

Last Time: Membership Queries

- ▶ You've collected 1 billion tweets.¹
- ▶ **Goal:** given the text of a new tweet, is it already in the data set?
- ▶ Data set is too large to fit into memory.
- ▶ Our solution: **Bloom filters**.

¹This is about two days of activity.

Today: Frequencies

- ▶ You've collected 1 billion tweets.
- ▶ **Goal:** given the text of a tweet, how many times have we seen it?
- ▶ Data set is too large to fit into memory.
- ▶ Today's solution: the **Count-Min Sketch**.

Frequency Counts

- ▶ **Given:** a collection $X = \{x_1, x_2, \dots, x_n\}$.
- ▶ **Support:**
 - ▶ `.count(x)`: Number of times x appears.
 - ▶ `.increment(x)`: Increment count of x

Simple Solution

- ▶ Use hash tables: dictionary of counts.

```
class SetCounts:

    def __init__(self):
        self.counts = {}

    def increment(self, x):
        if x not in self.counts:
            self.counts[x] = 1
        else:
            self.counts[x] += 1

    def count(self, x):
        try:
            return self.counts[x]
        except KeyError:
            return 0
```

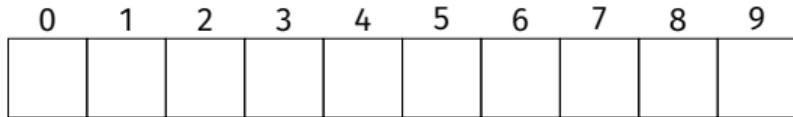
Problem: Memory Usage

- ▶ Requires storing the keys.
- ▶ Example: store approximately 1 billion tweets (100 GB).
- ▶ Can't fit the dictionary in memory.

A Fix

- ▶ Why do we store all of the keys?
- ▶ To resolve collisions.
- ▶ What if we ignore collisions?

Hashing Into Counters



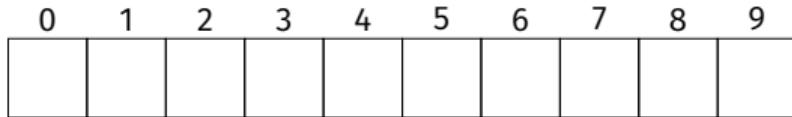
| s | hash(s) | |
|---------|---------|---------|
| "surf" | 3 | "data" |
| "sand" | 8 | "surf" |
| "data" | 5 | "sand" |
| "sun" | 1 | "surf" |
| "beach" | 5 | "surf" |
| | | "surf" |
| | | "beach" |
| | | "data" |
| | | "beach" |
| | | "surf" |
| | | "sun" |

- ▶ Use a size c ($c \ll n$) array of integers (counts).
- ▶

```
.increment(x):  
arr[hash(x)] += 1
```
- ▶

```
.count(x):  
return arr[hash(x)]
```

Hashing Into Counters



| s | hash(s) | |
|---------|---------|---------|
| "surf" | 3 | "data" |
| "sand" | 8 | "surf" |
| "data" | 5 | "sand" |
| "sun" | 1 | "surf" |
| "beach" | 5 | "surf" |
| | | "beach" |
| | | "data" |
| | | "data" |
| | | "beach" |
| | | "surf" |
| | | "sun" |

- ▶ Use a size c ($c \ll n$) array of integers (counts).
- ▶

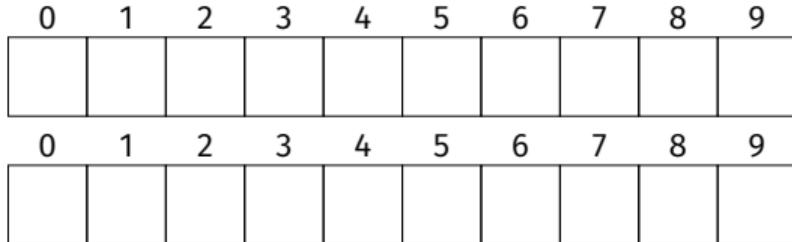
```
.increment(x):  
arr[hash(x)] += 1
```
- ▶

```
.count(x):  
return arr[hash(x)]
```
- ▶ Can be **wrong!**

Biased Estimate

- ▶ The count returned from this approach is **biased high**.
- ▶ Can we do better?
- ▶ **Idea:** multiple hashing. Perform previous k times.
- ▶ This is the **count-min sketch**.

Count-Min Sketch



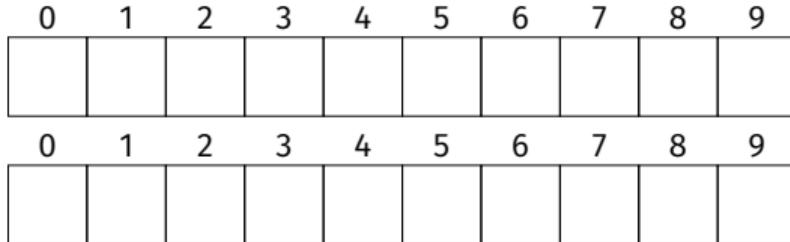
| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 3 | 7 |
| "sand" | 8 | 7 |
| "data" | 5 | 4 |
| "sun" | 1 | 9 |
| "beach" | 5 | 6 |

"data"
"surf"
"sand"
"surf"
"surf"
"beach"
"data"
"beach"
"surf"
"sun"

- ▶ Use k arrays of counts, each with own independent hash functions.
- ▶

```
.increment(x): Set arr_1[hash_1(x)] += 1,  
arr_2[hash_2(x)] += 1,  
...,  
arr_k[hash_k(x)] += 1.
```

Count-Min Sketch



| s | hash_1(s) | hash_2(s) |
|---------|-----------|-----------|
| "surf" | 3 | 7 |
| "sand" | 8 | 7 |
| "data" | 5 | 4 |
| "sun" | 1 | 9 |
| "beach" | 5 | 6 |

"data"
"surf"
"sand"
"surf"
"surf"
"beach"
"data"
"beach"
"surf"
"sun"

- ▶ Use k arrays of counts, each with own independent hash functions.
- ▶ `.count(x)`: Return the **minimum** of `arr_1[hash_1(x)]`, `arr_2[hash_2(x)]`, ..., `arr_k[hash_k(x)]`.

Returning the Minimum Count

- ▶ The count is still biased high.
- ▶ But by returning the minimum, bias is reduced.

Memory Usage

- ▶ Each counter cell stores an integer (64 bits).
- ▶ Total size:
$$64 \times c \cdot k \text{ bits}$$
- ▶ c and k should be chosen to match prescribed level of error.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 18 | Part 2

Designing a Count-Min Sketch

Error Rate

- ▶ Count-min sketch is a probabilistic data structure.
 - ▶ Returns the wrong answer sometimes.
- ▶ How wrong is it, probably?
- ▶ And how does this depend on c and k ?

Notation

- ▶ We see n items, record frequencies in count-min sketch.
- ▶ For any item x , let f_x be its true frequency.
- ▶ $\hat{f}_x^{(i)} \equiv \text{arr_i}[\text{hash_i}(x)]$ is estimated frequency of x according to row i . \hat{f}_x is aggregate estimate: $\hat{f}_x = \min_i \hat{f}_x^{(i)}$.
- ▶ Note: $\hat{f}_x^{(i)} \geq f_x$

Absolute and Relative Error

- ▶ Absolute error: $\hat{f}_x - f_x$
 - ▶ This will grow as collection size $n \rightarrow \infty$.
- ▶ Relative error: $(\hat{f}_x - f_x)/f_x$
 - ▶ We're more interested in this. Want it to be small.
 - ▶ If $f_x = \Theta(n)$, we want:

$$(\hat{f}_x - f_x)/n < \varepsilon \implies \hat{f}_x - f_x < \varepsilon n$$

Analyses

- ▶ We'll first look at the expected value of the estimate in a single row.
- ▶ Then, we'll compute the probability that the aggregate estimate is much larger than the true value.

Expected Value

- ▶ Fix an object, x , and a row i .

$$\mathbb{E}[\hat{f}_x^{(i)}] = \text{expected count in } x\text{'s bin}$$

$$= f_x + \mathbb{E}[\text{tot. frequency of colliding items } y \neq x]$$

$$= f_x + \sum_{y \neq x} f_y \cdot \mathbb{P}(\text{hash}(y) == \text{hash}(x))$$

$$= f_x + \frac{1}{c} \sum_{y \neq x} f_y \leq f_x + \frac{n}{c}$$

Expected Value

- ▶ We found: $\mathbb{E}[\hat{f}_x^{(i)}] \leq f_x + \frac{n}{c}$.
- ▶ Is this good or bad?
 - ▶ Suppose $f_x = p_x n$, where $p_x \in [0, 1]$.
 - ▶ Absolute error is $\Theta(n)$.
 - ▶ But **relative** error is $\frac{1}{pc}$.
 - ▶ Independent of n !

Extreme Values

- ▶ Goal: show unlikely for $\hat{f}_x^{(i)}$ to be much larger than f_x
- ▶ Let's find α s.t. $\mathbb{P}(\hat{f}_x^{(i)} - f_x > \alpha) < 1/2$. Then:

$$\begin{aligned}\mathbb{E}[\hat{f}_x^{(i)}] &\geq f_x + \alpha \cdot P(\hat{f}_x^{(i)} - f_x > \alpha) \\ &= f_x + \alpha/2\end{aligned}$$

- ▶ We know $\mathbb{E}[\hat{f}_x^{(i)}] \leq f_x + \frac{n}{c}$, so $\alpha < 2n/c$.

Extreme Values

- ▶ We've shown that $\mathbb{P}(\hat{f}_x^{(i)} - f_x > 2n/c) < 1/2$.
- ▶ This is just for the i th row.
- ▶ Minimum is $> 2n/c$ only if *every* row is $> 2n/c$.
- ▶ Probability of this happening:

$$\prod_{i=1}^k \mathbb{P}(\hat{f}_x^{(i)} - f_x > 2n/c) \leq \left(\frac{1}{2}\right)^k$$

Extreme Values

- ▶ Let \hat{f}_x be the aggregate estimate. We have shown:

$$\mathbb{P}(\hat{f}_x - f_x > 2n/c) < \left(\frac{1}{2}\right)^k$$

- ▶ Want $\hat{f}_x - f_x < \varepsilon$. Set $c = 2/\varepsilon$.
- ▶ To ensure that an over-estimate larger than ε occurs with probability δ , set

$$\left(\frac{1}{2}\right)^k = \delta \quad \implies \quad k = \log_2 \frac{1}{\delta}$$

Designing a Count-Min Sketch

- ▶ Pick your ϵ and δ : “I want overestimates to be smaller than ϵn at least $1 - \delta$ percent of the time.”
- ▶ Set number of buckets to $c = 2/\epsilon$
- ▶ Set number of rows/hash functions to $k = \log_2 1/\delta$.

Example

- ▶ We have 1 billion tweets, want to count number of occurrences for each.
- ▶ Assume each tweet requires 800 bits.
- ▶ `dict`: around 100 gigabytes, assuming \approx 1 billion unique

Example

- ▶ Instead, use a count-min sketch. Say, $\varepsilon = .001$ and $\delta = .01$.
- ▶ $c = 2/\varepsilon = 2000$
- ▶ $k = \log_2 1/\delta \approx 7$.
- ▶ Memory: $7 \times 2000 \times 64 \text{ bits} = 112 \text{kilobytes}$

Example

- ▶ Now supposed you have 42 quadrillion tweets.
- ▶ `dict`: 4.2 exabytes
- ▶ count-min sketch: 112 kilobytes

How?

- ▶ The relative error ε of a count-min sketch does not depend on n !
- ▶ The n is “hidden” inside the relative error:

$$\hat{f}_x - f_x < \varepsilon n$$

Count-Min Sketch and Bloom Filters

- ▶ The Count-Min Sketch and Bloom Filters are both probabilistic data structures.
- ▶ Both make use of multiple hashing.
- ▶ Why does CMS take much less memory?

Less Memory

- ▶ Why does a CMS use less memory than a Bloom filter?
- ▶ The problem it is solving is easier.
- ▶ Bloom filter: big difference between seeing an element once and never seeing it.
- ▶ Count-Min sketch: essentially no difference.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 1

Complexity Theory

The quest for efficient algorithms is about finding clever ways to avoid taking exponential time. So far we have seen the most brilliant successes of this quest; now we meet the quest's most embarrassing and persistent failures.

- paraphrased from *Algorithms* by Dasupta, Papadimitriou, Vazirani

Exponential to Polynomial

- ▶ Many problems have brute force solutions which take exponential time.
- ▶ Example: clustering to maximize separation
- ▶ The challenge of algorithm design: find a more efficient solution.

Polynomial Time

- ▶ If an algorithm's worst case time complexity is $O(n^k)$ for some k , we say that it runs in **polynomial time**.
 - ▶ Example: $\Theta(n \log n)$, since $n \log n = O(n^2)$.
- ▶ Polynomial is much faster than exponential for big n .
 - ▶ But not necessarily for small n .
 - ▶ Example: n^{100} vs 1.0001^n .
- ▶ We therefore think of polynomial as “efficient”.

Question

- ▶ Is every problem solvable in polynomial time?

Question

- ▶ Is every problem solvable in polynomial time?
- ▶ **No!** Problem: print all permutations of n numbers.

Question

- ▶ Is every problem solvable in polynomial time?
- ▶ **No!** Problem: print all permutations of n numbers.
- ▶ **No!** Problem: given $n \times n$ checkerboard and current pieces, determine if red can force a win.

Ok, then...

- ▶ What problems can be solved in polynomial time?
- ▶ What problems can't?
- ▶ How can I tell if I have a hard problem?

Ok, then...

- ▶ What problems can be solved in polynomial time?
- ▶ What problems can't?
- ▶ How can I tell if I have a hard problem?
- ▶ Core questions in **computational complexity theory**.

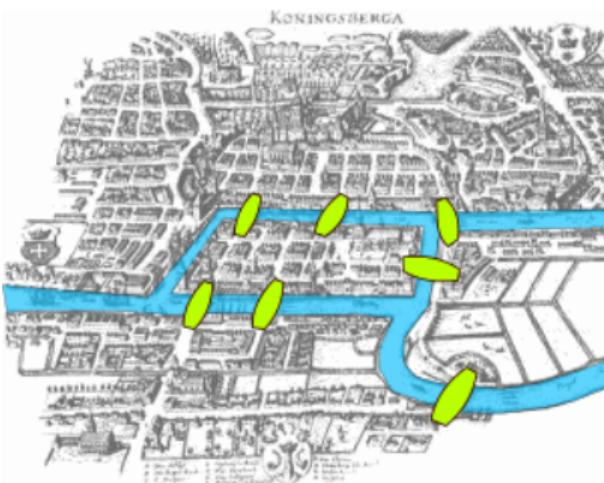
DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 2

Eulerian and Hamiltonian Cycles

Example: Bridges of Königsberg



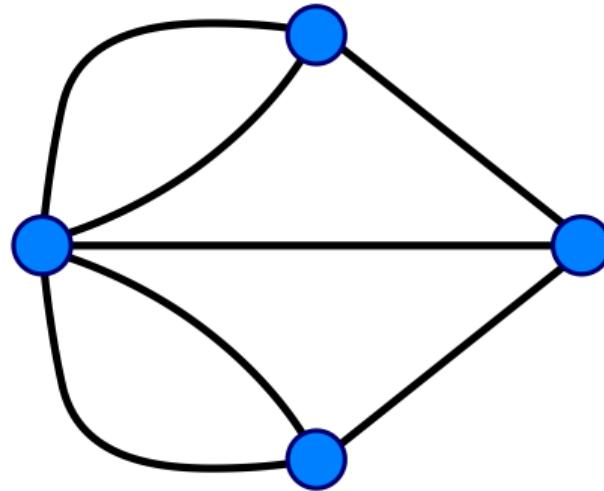
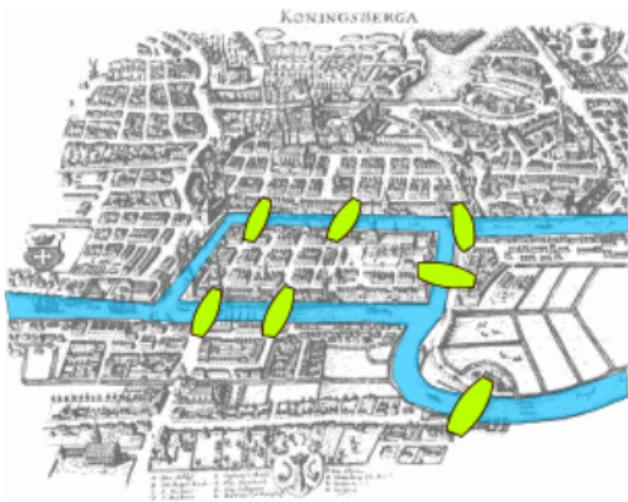
- ▶ **Problem:** Is it possible to start and end at same point while crossing each bridge exactly once?

Leonhard Euler



1707 - 1783

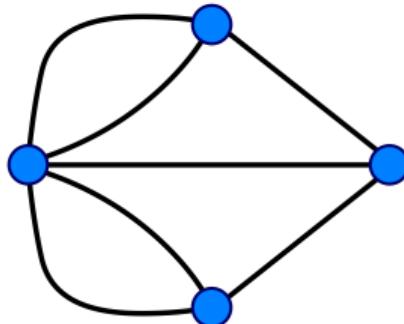
Eulerian Cycle



Is there a cycle which uses each edge exactly once?

Necessary conditions

- ▶ Graph must be connected.
- ▶ Each node must have even degree.
- ▶ Answer for Königsberg answer: it is **impossible**.



In General...

- ▶ These conditions are **necessary** and **sufficient**.
- ▶ A graph has a Eulerian cycle **if and only if**:
 - ▶ it is connected;
 - ▶ each node has even degree.

Exercise

Can we determine if a graph has an Eulerian cycle in time that is polynomial in the number of nodes?

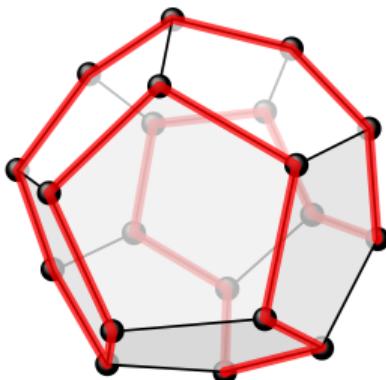
Answer

- ▶ We can check if it is connected in $\Theta(V + E)$ time.
- ▶ Compute every node's degree in $\Theta(V)$ time with adjacency list.
- ▶ Total: $\Theta(V + E) = O(V^2)$. **Yes!**

Gaming in the 19th Century

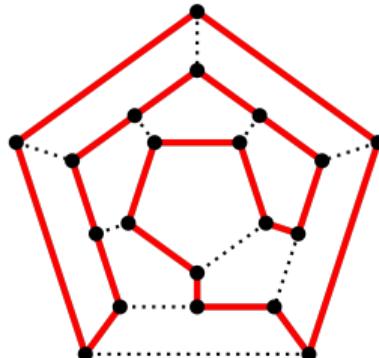
I have found that some young persons have been much amused by trying a new mathematical game which the Icosian furnishes [...]

- W.R. Hamilton, 1856



Hamiltonian Cycles

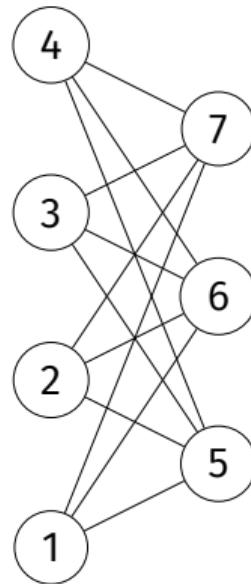
- ▶ A **Hamiltonian cycle** is a cycle which visits each *node* exactly once (except the starting node).
- ▶ Game: find a Hamiltonian cycle on the graph below:



Exercise

Can we determine whether a general graph has a Hamiltonian cycle in polynomial time?

Some cases are easy



In General

- ▶ Could brute-force.
- ▶ How many possible cycles are there?

Hamiltonian Cycles are Difficult

- ▶ This is a **very difficult** problem.
- ▶ No polynomial algorithm is known for general graphs.
- ▶ In special cases, there may be a fast solution.
But in general, worst case is hard.

Note

- ▶ Determining if a graph has a Hamiltonian cycle is **hard**.
- ▶ But if we're given a "hint" (i.e., (v_1, v_2, \dots, v_n) is possibly a Hamiltonian cycle), we can check it very quickly!
- ▶ Hard to solve; but easy to verify "hints".

Similar Problems

- ▶ Eulerian: polynomial algorithm, “**easy**”.
- ▶ Hamiltonian: no polynomial algorithm known, “**hard**”.

Main Idea

Computer science is littered with pairs of similar problems where one easy and the other very hard.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 3

Shortest and Longest Paths

Problem: SHORTPATH

- ▶ **Input:** Graph¹ G , source u , dest. v , number k .
- ▶ **Problem:** is there a path from u to v of length $\leq k$?
- ▶ **Solution:** BFS or Dijkstra/Bellman-Ford in polynomial time.
- ▶ **Easy!**

¹Weighted with no negative cycles, or unweighted.

Problem: LONGPATH

- ▶ **Input:** Graph² G , source u , dest. v , number k .
- ▶ **Problem:** is there a **simple** path from u to v of length $\geq k$?
- ▶ Naïve solution: try all $V!$ path candidates.

²Weighted or unweighted.

Long Paths

- ▶ There is no known polynomial algorithm for this problem.
- ▶ It is a **hard problem**.
- ▶ But given a “hint” (a possible long path), we can verify it very quickly!

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 4

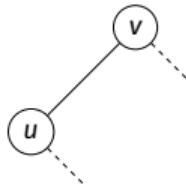
Reductions

Reductions

- ▶ HAMILTONIAN and LONGPATH are related.
- ▶ We can “convert” HAMILTONIAN into LONGPATH in polynomial time.
- ▶ We say that HAMILTONIAN **reduces** to LONGPATH.

Reduction

- ▶ Suppose we have an algorithm for LONGPATH.
- ▶ We can use it to solve HAMILTONIAN as follows:
 - ▶ Pick arbitrary node u .
 - ▶ For each neighbor v of u :
 - ▶ Create graph G' by copying G , deleting (u, v)
 - ▶ Use algorithm to check if a simple path of length $\geq |V| - 1$ from u to v exists in G' .
 - ▶ If yes, then there is a Hamiltonian cycle.



Reductions

- ▶ If Problem A reduces³ to Problem B, it means “we can solve A by solving B”.
- ▶ Best possible time for A \leq best possible time for B + polynomial
- ▶ “A is no harder than B”
- ▶ “B is at least as hard as A”

³We'll assume reduction takes polynomial time.

Relative Difficulty

- ▶ If Problem A reduces to Problem B, we say B is **at least as hard** as A.
- ▶ Example: HAMILTONIAN reduces to LONGPATH.
LONGPATH is at least as hard as HAMILTONIAN.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 5

P $\stackrel{?}{=}$ NP

Decision Problems

- ▶ All of today's problems are **decision problems**.
 - ▶ Output: yes or no.
 - ▶ Example: Does the graph have an Euler cycle?

P

- ▶ Some problems have polynomial time algorithms.
 - ▶ SHORTPATH, EULER
- ▶ The set of decision problems that can be solved in polynomial time is called P.
- ▶ Example: SHORTPATH and EULER are in P.

NP

- ▶ The set of decision problems with “hints” that can be verified in polynomial time is called **NP**.
- ▶ All of today’s problems are in NP.
 - ▶ All problems in P are also in NP.
- ▶ Example: SHORTPATH, EULER, HAMILTONIAN, LONGPATH are all in NP.

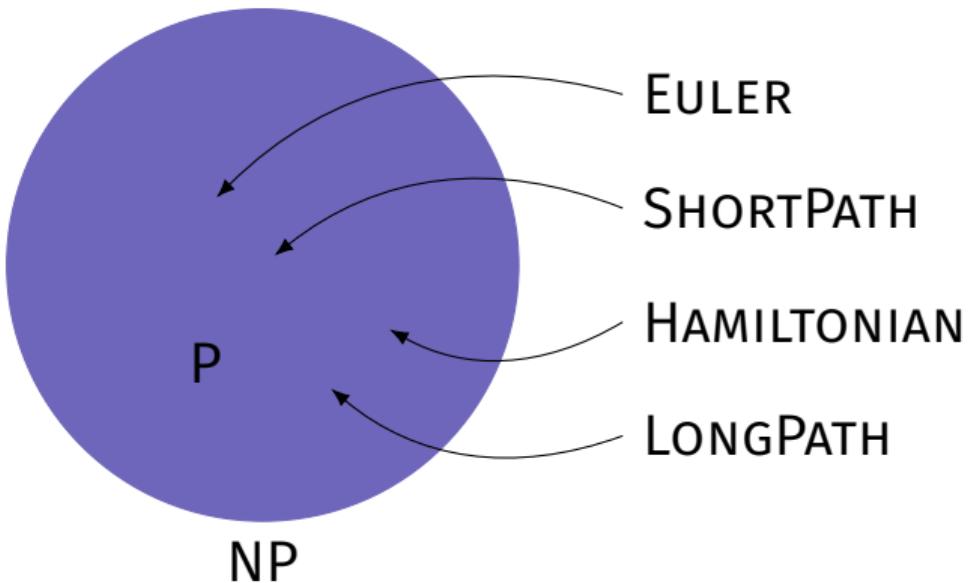
$P \subset NP$

- ▶ P is a subset of NP .
- ▶ It seems like some problems in NP aren't in P .
 - ▶ Example: HAMILTONIAN, LONGPATH.
- ▶ We don't know polynomial time algorithms for these problems.
- ▶ But that doesn't such an algorithm is impossible!

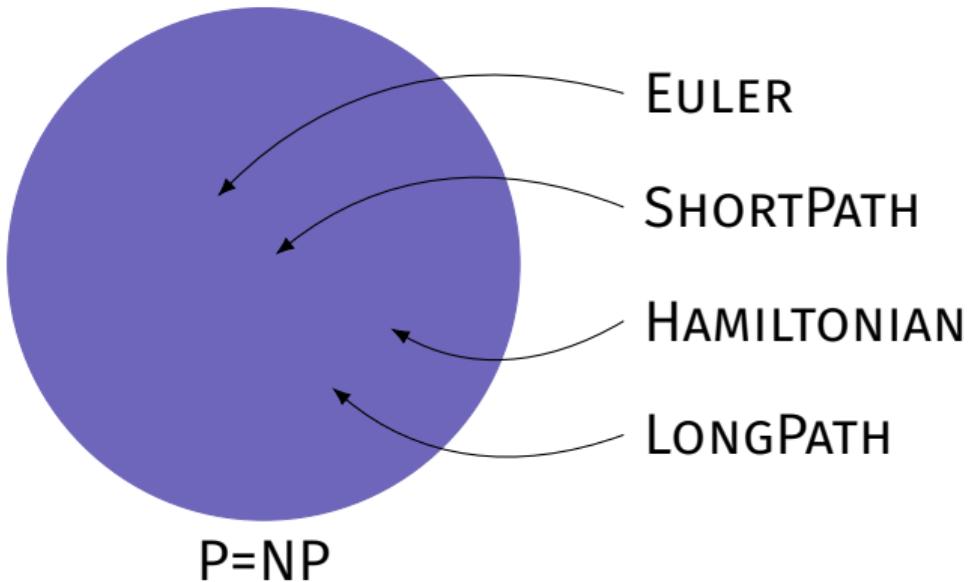
P = NP?

- ▶ Are there problems in NP that aren't in P?
 - ▶ That is, is $P \neq NP$?
- ▶ Or is any problem in NP also in P?
 - ▶ That is, is $P = NP$?

P \neq NP



P = NP



P = NP?

- ▶ Is P = NP?

⁴If you solve it, you'll be rich and famous.

P = NP?

- ▶ Is P = NP?
- ▶ **No one knows!**
- ▶ Biggest open problem in Math/CS.⁴
- ▶ Most think P \neq NP.

⁴If you solve it, you'll be rich and famous.

What if P = NP?

- ▶ Possibly Earth-shattering.
 - ▶ Almost all cryptography instantly becomes obsolete;
 - ▶ Logistical problems solved exactly, quickly;
 - ▶ *Mathematicians* become obsolete.
- ▶ But maybe not...
 - ▶ Proof could be non-constructive.
 - ▶ Or, constructive but really inefficient. E.g., $\Theta(n^{10000})$

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 6

NP-Completeness

Problem: 3-SAT

- ▶ Suppose x_1, \dots, x_n are boolean variables
(True, False)
- ▶ A **3-clause** is a combination made by **or-ing** and possibly negating three variables:
 - ▶ $x_1 \text{ or } x_5 \text{ or } (\text{not } x_7)$
 - ▶ $(\text{not } x_1) \text{ or } (\text{not } x_2) \text{ or } (\text{not } x_4)$

Problem: 3-SAT

- ▶ **Given:** m clauses over n boolean variables.
- ▶ **Problem:** Is there an assignment of x_1, \dots, x_n which makes all clauses true simultaneously?
- ▶ No polynomial time algorithm is known.
- ▶ But it is easy to verify a solution, given a hint.
 - ▶ 3-SAT is in NP.

Cook's Theorem

Every problem in NP is polynomial-time reducible to 3-SAT.

- ▶ ...including Hamiltonian, long path, etc.
- ▶ 3-SAT is at least as hard as every problem in NP.
- ▶ “hardest problem in NP”

Cook's Theorem (Corollary)

- ▶ If 3-SAT is solvable in polynomial time, then all problems in NP are solvable in polynomial time.
 - ▶ ...including Hamiltonian, long path, etc.

NP-Completeness

- ▶ We say that a problem is **NP-complete** if:
 - ▶ it is in NP;
 - ▶ every problem in NP is reducible to it.
- ▶ HAMILTONIAN, LONGPATH, 3-SAT are all NP-complete.
- ▶ NP-complete problems are the “hardest” in NP.

Equivalence

- ▶ In some sense, NP-complete problems are equivalent to one another.
- ▶ E.g., a fast algorithm for HAMILTONIAN gives a fast algorithm for 3-SAT, LONGPATH, and all problems in NP.

Who cares?

- ▶ Complexity theory is a fascinating piece of science.
- ▶ But it's practically useful, too, for recognizing hard problems when you stumble upon them.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 19 | Part 7

Hard Optimization Problems

Hard Optimization problems

- ▶ NP-completeness refers to **decision problems**.
- ▶ What about optimization problems?
- ▶ We can typically state a similar decision problem.
- ▶ If that decision problem is hard, then optimization is at least as hard.

Problem: bin packing

- ▶ Optimization problem:
 - ▶ **Given:** bin size B , n objects of size $\alpha_1, \dots, \alpha_n$..
 - ▶ **Problem:** find minimum number of bins k that can contain all n objects.
- ▶ Decision problem version:
 - ▶ **Given:** bin size B , n objects of size $\alpha_1, \dots, \alpha_n$, integer k .
 - ▶ **Problem:** is it possible to pack all n objects into k bins?
- ▶ Decision problem is NP-complete, reduces to optimization problem.

Example: traveling salesperson

- ▶ Optimization problem:
 - ▶ **Given:** set of n cities, distances between each.
 - ▶ **Problem:** find shortest Hamiltonian cycle.
- ▶ Decision problem:
 - ▶ **Given:** set of n cities, distance between each, length ℓ .
 - ▶ **Problem:** is there a Hamiltonian cycle of length $\leq \ell$?
- ▶ Decision problem is NP-complete, reduces to optimization problem.

NP-complete problems in machine learning

- ▶ Many machine learning problems are NP-complete.
- ▶ Examples:
 - ▶ Finding a linear decision boundary to minimize misclassifications in non-separable regime.
 - ▶ Minimizing k -means objective.

So now what?

- ▶ Just because a problem is NP-Hard, doesn't mean you should give up.
- ▶ Usually, an approximation algorithm is fast, "good enough".
- ▶ Some problems are even hard to *approximate*.

Summary

- ▶ Not every problem can be solved efficiently.
- ▶ Computer scientists are able to categorize these problems.

DSC 190

DATA STRUCTURES & ALGORITHMS

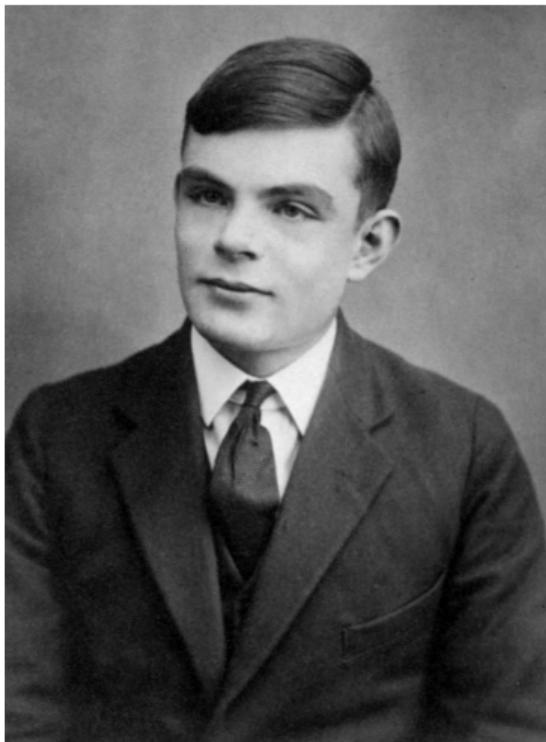
Lecture 19 | Part 8

The Halting Problem

Really hard problems

- ▶ Some decision problems are harder than others.
- ▶ That is, it takes more time to solve them.
- ▶ Given enough time, all decision problems can be solved, right?

Alan Turing



1912-1954

Turing's Halting Problem

- ▶ **Given:** a function f and an input x .
- ▶ **Problem:** does $f(x)$ halt, or run forever?
- ▶ Algorithm must work for all functions/inputs!

Turing's Argument

- ▶ Turing says: no such algorithm can exist.
- ▶ Suppose there is a function `halts(f, x)`:
 - ▶ Returns **True** if $f(x)$ halts.
 - ▶ Returns **False** if $f(x)$ loops forever.

Turing's Argument

- ▶ Consider `evil_function`.
 - ▶ If it halts, it doesn't.
 - ▶ If it doesn't halt, it does.
- ▶ Contradicts claim that `halt` works.

```
def evil_function(f):  
    if halts(f, f):  
        # loop forever  
    else:  
        return
```

Undecidability

- ▶ The halting problem is **undecidable**.
- ▶ Fact of the universe: there can be no algorithm for solving it which works on all functions/inputs.
- ▶ All of these problems are undecidable:
 - ▶ Does the program terminate?
 - ▶ Does this line of code ever run?
 - ▶ Does this function compute what its specification says?

The End