Midterm Exam review

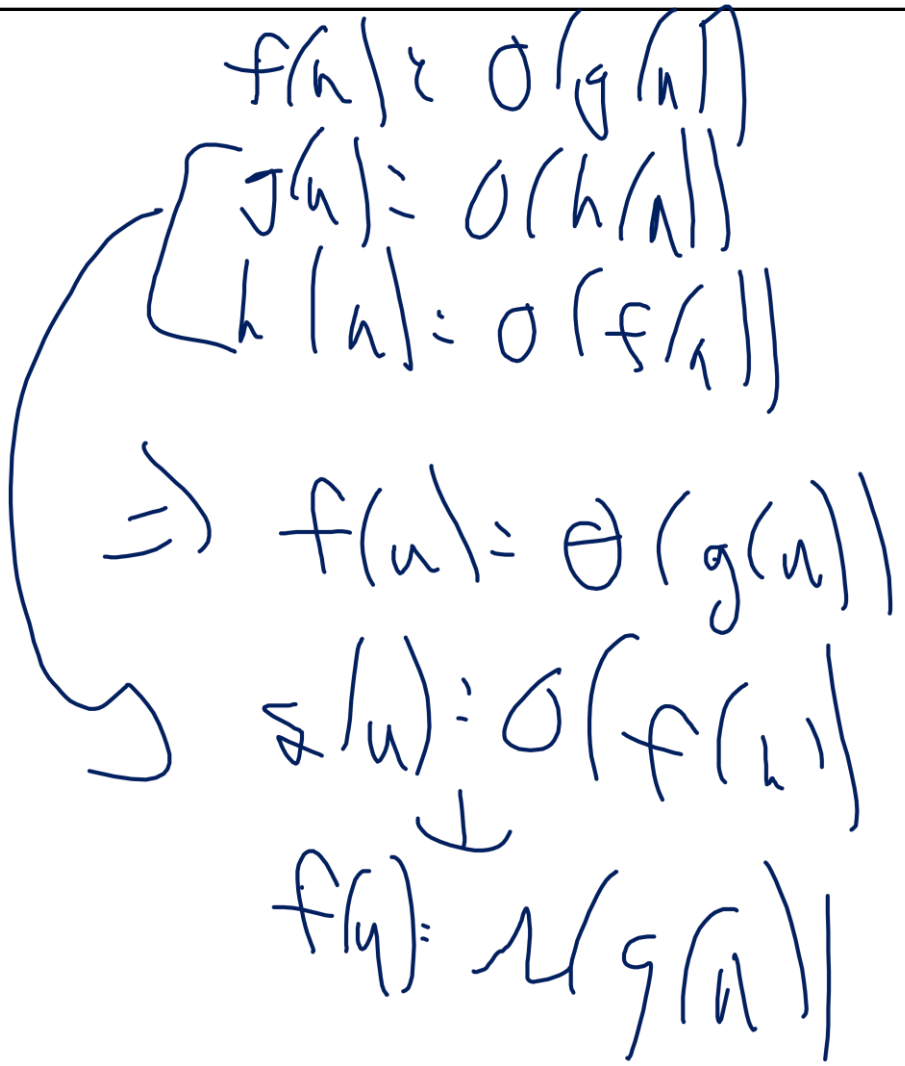
William Hendrix

Lecture 7

Midterm Exam review

Potential topics:

- Big-Oh formal definitions
  - ~~Properties~~
- Algorithm analysis
  - Iterative analysis
  - Recursive analysis
  - Master Theorem
- Data structures
  - Stacks and queues
  - AVL trees
  - Hash tables
  - Sets
  - Maps
  - Priority Queues/Heaps
  - Union-Find
  - Algorithm design
- Sorting



Summary: complexity and Big-Oh

- RAM model of computation
- Useful approximation of real-world behavior:
  - Basic instructions take same amount of time
  - Memory access is instantaneous
- Key aspect of complexity: asymptotic growth
  - How fast does the function grow?
  - Constant, logarithmic, linear, etc.?
- Big-Oh: classify functions according to growth rate

$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- *Analogy:* O, Ω, and Θ "act like" $\leq$, $\geq$, and $=$

Big-Oh properties

- **Interrelationships:** O and Ω are "*opposite*", Θ is "*composite*"
  $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Leftrightarrow f(n) = \Theta(g(n))$
- **Reflexive**
  $f(n) = O(f(n))$, for any function f
- *Θ only*: **Symmetric**
  $f(n) = \Theta(g(n)) \rightarrow g(n) = \Theta(f(n))$
- **Transitive**
  $f(n) = O(g(n))$ and $g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$
- **Ignore constant coefficients**
  $\forall x > 0, xf(n) = O(f(n))$
- **Ignore small terms**
  $f(n) = O(g(n)) \rightarrow \Theta(f(n) + g(n)) = \Theta(g(n))$
- **Envelopment** (+ and *)
  $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  $O(f(n))O(g(n)) = O(f(n)g(n))$

Big-Oh exercise

- Prove the envelopment property of addition of Big-Oh:
  If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
  then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Big-Oh exercise

- Prove the envelopment property of addition of Big-Oh:
  If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
  then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

  Proof. Since $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, there exists positive constants c_1, c_2, n_1, and n_2 such that $f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$ and $f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. If $n \geq \max(n_1, n_2)$, both of these inequalities will be true. If we add both sides of the inequalities, we see that $f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$ for all $n \geq \max(n_1, n_2)$. If we let $c_3 = \max(c_1, c_2)$ and $n_3 = \max(n_1, n_2)$, we see that $f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_3(g_1(n) + g_2(n))$ for all $n \geq n_3$. Therefore, $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$, by the formal definition of Big-Oh. $\square$

Summary: algorithm analysis

- Identify loops and function calls
  - Everything else is *Θ(1)*
- *For loops*:
  - Estimate number of iterations
    - Incrementing by *c*: divide range by *c* to get iterations
    - Multiplying by *c:* take $\log_c$ of the end/start ratio
  - Estimate loop body running time
    - Might depend on iteration #
  - If iterations don't depend on i: # iterations * time per iteration
  - Otherwise: sum up all iterations

$$\sum_{i=1}^{x} i = \Theta(x^2) \qquad \sum_{i=1}^{x} \frac{1}{i} = \Theta(\lg x) \qquad \sum_{i=1}^{x} 2^i = \Theta(2^x) \qquad \sum_{i=1}^{x} \frac{1}{2^i} = \Theta(1)$$

- *For functions:*
  - Analyze other functions separately
  - Recursive functions: set up a recurrence and solve
- Overall complexity: largest loop or function call complexity

Analysis exercise

- Find the worst-case complexity of the algorithm below:

```
   Input: list: linked list with n nodes
   Input: n: length of list
 1 node = list.head
 2 while node ≠ nullptr do
 3 |   min = node
 4 |   test = node.next
 5 |   while test ≠ nullptr do
 6 |   |   if test.value < min.value then
 7 |   |   |   min = test
 8 |   |   end
 9 |   |   test = test.next
10 |   end
11 |   Swap node.value with min.value
12 |   node = node.next
13 end
```

Analysis exercise

- Find the worst-case complexity of the algorithm below:

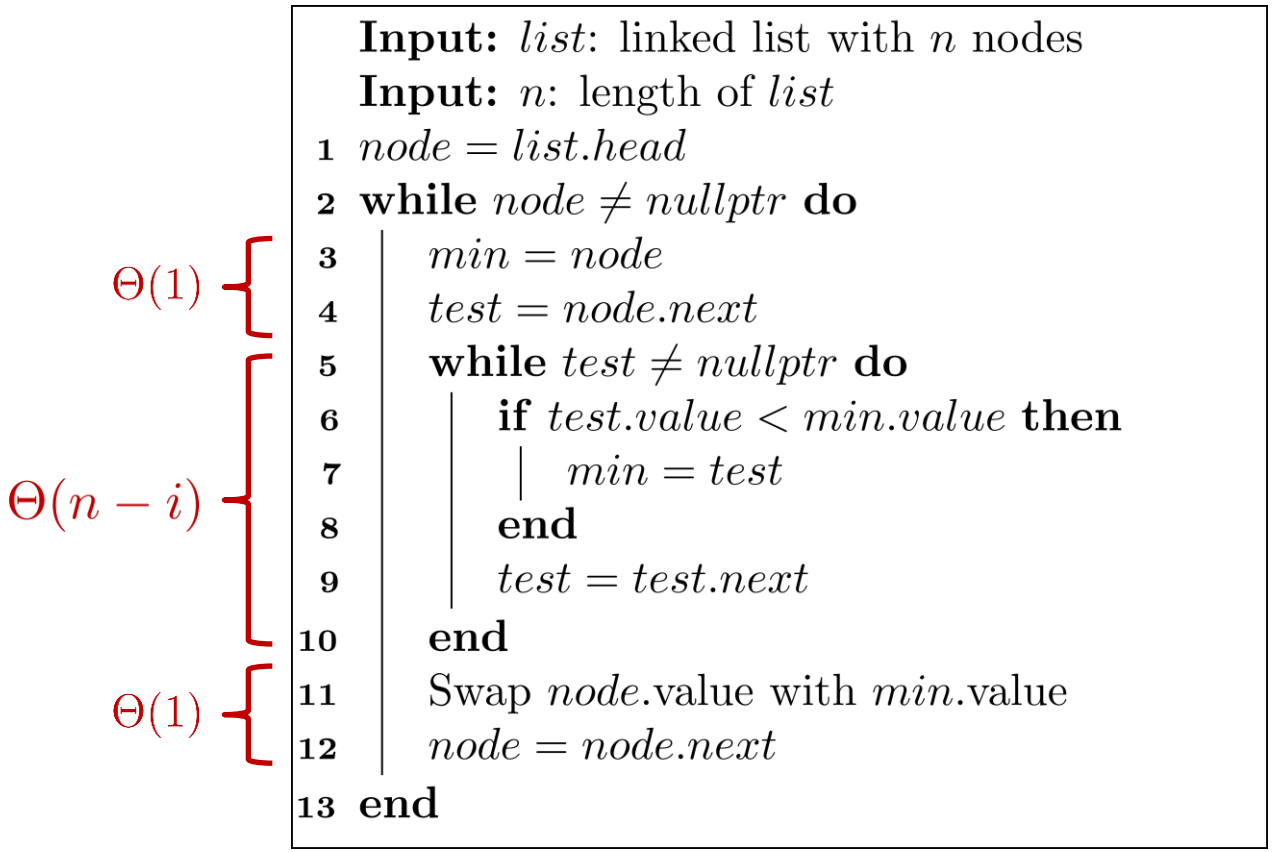
$O(n)$ or $\Theta(n-i)$ iters

$\Theta(1)$

```
   Input: list: linked list with n nodes
   Input: n: length of list
 1 node = list.head
 2 while node ≠ nullptr do
 3 |   min = node
 4 |   test = node.next
 5 |   while test ≠ nullptr do
 6 |   |   if test.value < min.value then
 7 |   |   |   min = test
 8 |   |   end
 9 |   |   test = test.next
10 |   end
11 |   Swap node.value with min.value
12 |   node = node.next
13 end
```

Analysis exercise

- Find the worst-case complexity of the algorithm below:



Analysis exercise

- Find the worst-case complexity of the algorithm below:

$\Theta(1)$

$\Theta(n)$ iters

$\Theta(n-i)$ / iter

```
   Input: list: linked list with n nodes
   Input: n: length of list
 1 node = list.head
 2 while node ≠ nullptr do
 3 |   min = node
 4 |   test = node.next
 5 |   while test ≠ nullptr do
 6 |   |   if test.value < min.value then
 7 |   |   |   min = test
 8 |   |   end
 9 |   |   test = test.next
10 |   end
11 |   Swap node.value with min.value
12 |   node = node.next
13 end
```

Analysis exercise

- Find the worst-case complexity of the algorithm below:

```
   Input: list: linked list with n nodes
   Input: n: length of list
 1 node = list.head
 2 while node ≠ nullptr do
 3 |   min = node
 4 |   test = node.next
 5 |   while test ≠ nullptr do
 6 |   |   if test.value < min.value then
 7 |   |   |   min = test
 8 |   |   end
 9 |   |   test = test.next
10 |   end
11 |   Swap node.value with min.value
12 |   node = node.next
13 end
```

$$\sum_{i=1}^{n} n - i$$

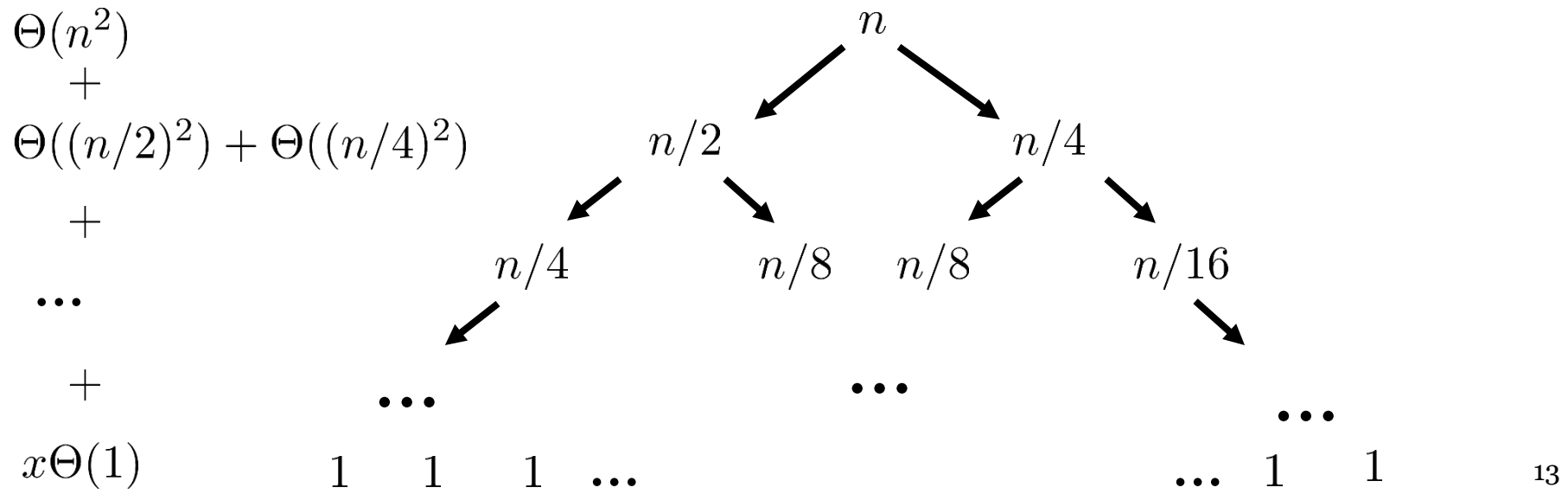
$$= \sum_{i=1}^{n-1} i$$

$$= \Theta(n^2)$$

Summary: using recursion trees

1. Construct a recurrence
2. Start with n
3. Split n into the recursive cases
4. Repeat down to the base cases
5. Add up complexity on each level
6. Add up complexity for every level

$$T(n) = \underbrace{T(n/2) + T(n/4)}_{\#\text{ and size of recursive calls}} + \underbrace{\Theta(n^2)}_{\text{Complexity of non-recursive code}}$$



Recursive analysis exercise

1. Give a recurrence that describes the worst-case time complexity for the algorithm below:

Input: A: array of n objects
Input: B: array of n objects
Input: n: number of elements in A and B
Output: minimum distance between two points in A and B according to function *dist*

```
1  Algorithm: MinDist
2  if n = 1 then
3  |   return dist(A[1], B[1])
4  else
5  |   mid = ⌊n/2⌋
6  |   A1 = A[1..mid]
7  |   A2 = A[mid + 1..n]
8  |   B1 = B[1..mid]
9  |   B2 = B[mid + 1..n]
10 |   d11 = MinDist(A1, B1)
11 |   d12 = MinDist(A1, B2)
12 |   d21 = MinDist(A2, B1)
13 |   d22 = MinDist(A2, B2)
14 |   return min{d11, d12, d21, d22}
15 end
```

Assume *dist* takes $\Theta(1)$ time

Back

Recursive analysis exercise

- All lines other than 10-13 (and possibly 6-9) are $\Theta(1)$
- Lines 6-9 could take $\Theta(n)$ if you are making copies instead of using pointers
- Lines 10-13 all take $T(n/2)$
- If you assume subarrays are defined with pointers:
  - $T(n) = 4T(n/2) + \Theta(1)$
- If you assume subarrays are copies:
  - $T(n) = 4T(n/2) + \Theta(n)$

Recursive analysis exercise

2. Sketch a recursion tree for your chosen recurrence

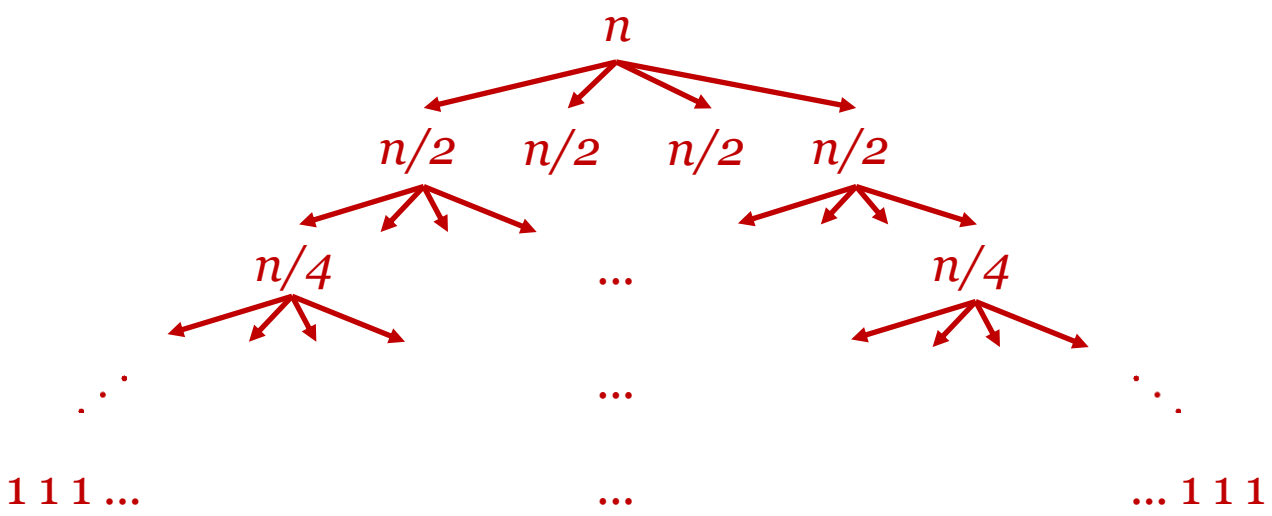
   $T(n) = 4T(n/2) + \Theta(1)$ or $4T(n/2) + \Theta(n)$

3. Analyze the complexity of this recursion tree

   Hint: $4^{\lg n} = n^2$

Recursive analysis exercise

Tree



Complexity

Θ(1)	Θ(n)
Θ(1)	Θ(n)
4Θ(1)	4Θ(n/2) = 2Θ(n)
16Θ(1)	16Θ(n/4) = 4Θ(n)
Both:	$4^{\lg n}\Theta(1) = \Theta(n^2)$

Total (both): $\Theta(n^2)$

Master Theorem

- Powerful theorem for proving complexity of divide-and-conquer algorithms

Master Theorem. If $T(n) = aT(n/b) + f(n)$,

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(f(n) \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \quad \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

Four steps to solve:

1. Identify *a*, *b*, and *f(n)*
2. Calculate $c = \log_b(a)$
3. Decide case: *f(n)* vs. n^c: $O(n^{c-\varepsilon})$, $\Theta(n^c)$, $\Omega(n^{c+\varepsilon})$
4. Apply Master Theorem (test regularity if case 3)

Master Theorem exercises

- The table below describes information about five recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

- Please fill in the missing values in the table.

#	a	b	f(n)	T(n)
1.	3	3	$\Theta(n)$	
2.	8	2	$\Theta(n^2 \lg n)$	
3.		3	$\Theta(n^2)$	$O(n^2)$
4.	4		$\Theta(1)$	$O(\sqrt{n})$
5.	9	3		$\Omega(n^2)$

Master Theorem exercises

- The table below describes information about five recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

- Please fill in the missing values in the table.

#	*a*	*b*	*f*(*n*)	*T*(*n*)
1.	3	3	$\Theta(n)$	$\Theta(n \lg n)$
2.	8	2	$\Theta(n^2 \lg n)$	$\Theta(n^3)$
3.	≤ 8	3	$\Theta(n^2)$	$O(n^2)$
4.	4	≥ 16	$\Theta(1)$	$O(\sqrt{n})$
5.	9	3	anything	$\Omega(n^2)$

Stacks and queues

- **Stacks**
  - Support *push()* and *pop()* operations
  - Last-In, First-Out (LIFO) order

- **Queues**
  - Support *enqueue()* and *dequeue()* operations
  - First-In, First-Out (FIFO) order

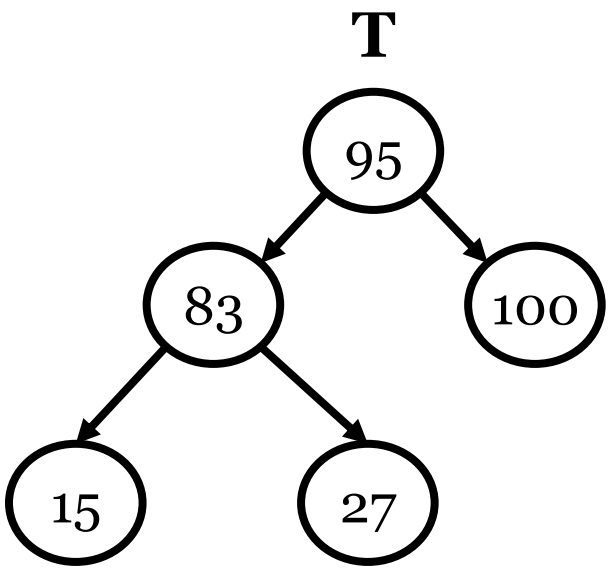
- **Deques** ("decks")
  - Support all 4 operations

- All three implemented using dynamic arrays
- All operations $\Theta(1)$
  - *enqueue()* and *push()* $\Theta(1)$ *amortized* time

Stack/queue exercise

- Consider the following algorithm for iterating through the elements of a Binary Search Tree:

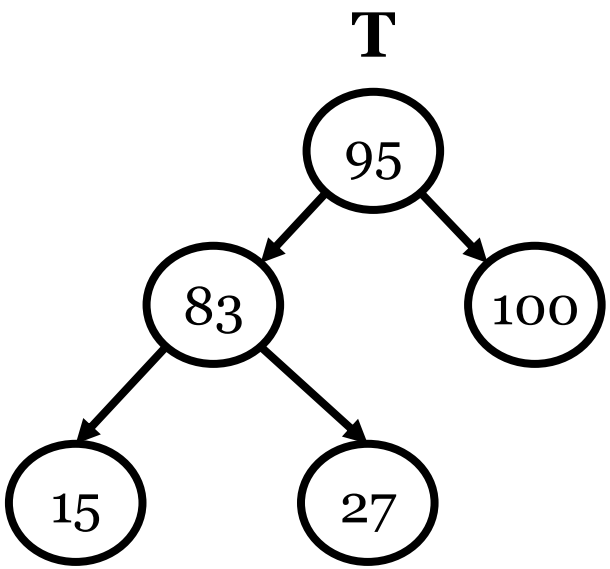
```
   Input: tree: a BST
1  Algorithm: Iterate
2  nodes = {}
3  Add tree.root to nodes
4  while nodes ≠ ∅ do
5  |   Print all the elements of nodes
6  |   t = next element of nodes
7  |   Add t.left to nodes, unless it's NIL
8  |   Add t.right to nodes, unless it's NIL
9  end
```



- Assume that Line 5 prints the node values in the order they would be popped or dequeued

1. What is printed by Iterate(T) if nodes is a stack?
2. What is printed by Iterate(T) if nodes is a queue?

Stack/queue exercise



Iteration	Stack (top-bot)	Queue
1	95	95
2	100, 83	83, 100
3	83	100, 15, 25
4	27, 15	15, 27
5	15	27

Output

Stack: 95, 100, 83, 27, 15

Queue: 95, 83, 100, 15, 27

AVL trees

- Self-balancing BST
- **AVL tree property:** node balance must be 0, +1, or -1

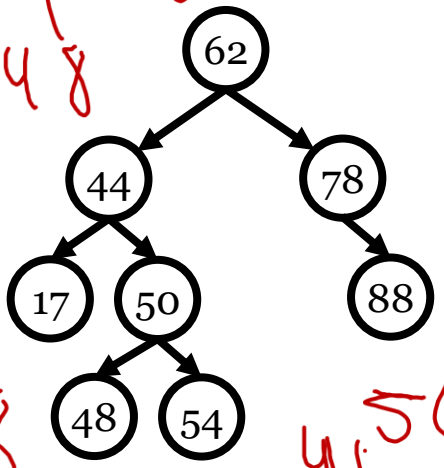
- Insert(x):
  - Normal BST insert
  - Update parent balance (±1)
  - Balance 0: stop
  - Balance ±1: continue
  - Balance ±2: rotate then stop
    - LL: rotate left child right
    - RR: rotate right child left
    - LR: rotate grandchild left then right
    - RL: rotate grandchild right then left

- Delete(x):
  - Normal BST deletion
  - Update parent balance (∓ 1)
  - Balance 0: continue
  - Balance ±1: stop
  - Balance ±2: rotate then continue
    - LL: rotate left child right
    - RR: rotate right child left
    - LR: rotate grandchild left then right
    - RL: rotate grandchild right then left

AVL tree exercise

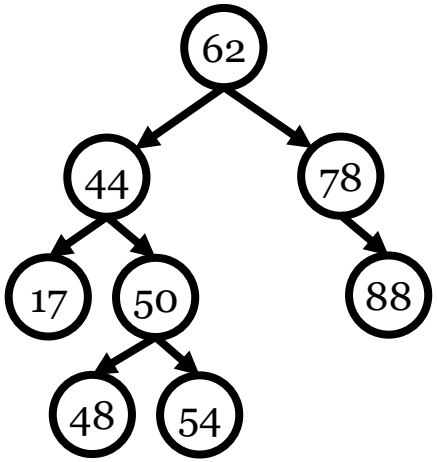
Back

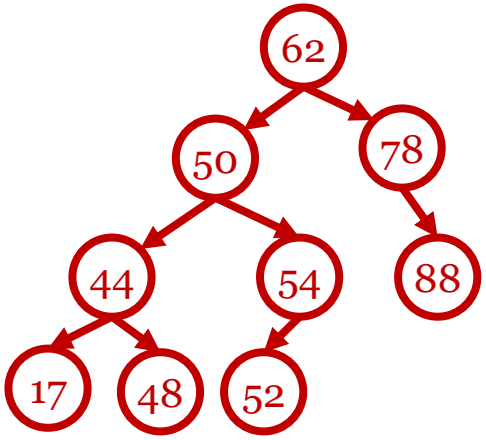
1. Draw the AVL tree resulting from the insertion of an entry with key 52 into the AVL tree to the right
2. Draw the AVL tree resulting from the removal of the entry with key 62 from the AVL tree to the right



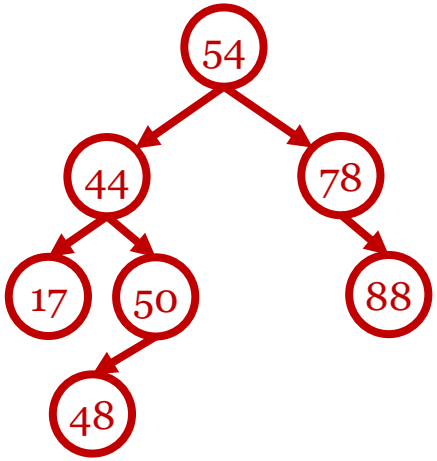
AVL tree exercise

1. Draw the AVL tree resulting from the insertion of an entry with key 52 into the AVL tree to the right
2. Draw the AVL tree resulting from the removal of the entry with key 62 from the AVL tree to the right





1



2

Hash table

- Sparse array-based structure
- **Collisions:** more than one element to insert with same hash value
- **Separate chaining**
  - Array of singly-linked lists
  - Prepend when inserting
  - Linked list deletion
  - O(1) worst case for insert
  - Easier to get good performance
- **Open addressing**
  - Insert into "next" available space
  - Mark "absent" when deleting
  - More space efficient
- Both Θ(1) expected case complexity for all operations
  - Assuming few collisions

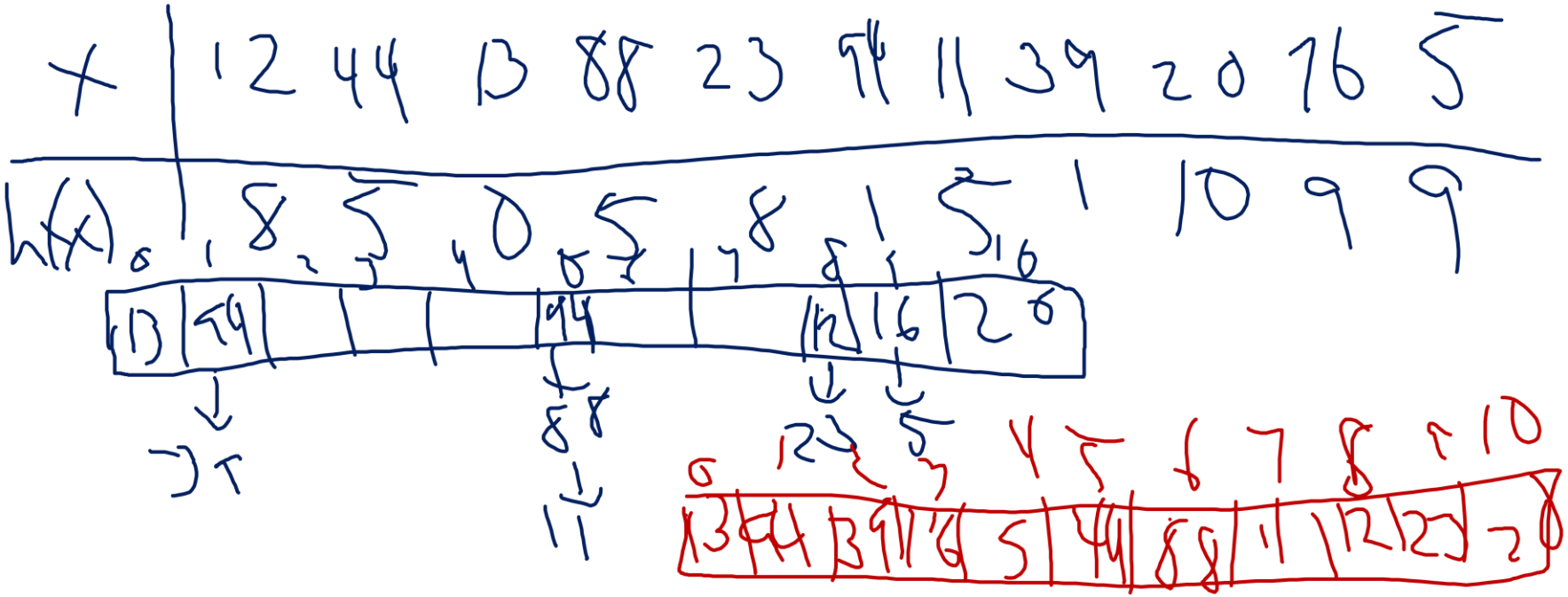
Hashing

- **Probing:** scanning strategy for open addressing
  - **Linear probing:** scan linearly
  - **Quadratic probing:** scan quadratically
  - **Double hashing:** scan with secondary hash function
- **Rehashing:** expand and reinsert when load factor exceeds threshold
  - **Load factor:** `size / capacity`
- 3 steps in a good hash function

1. For multi-byte inputs: combine bits together
   - Multiply and add or rotate and xor
2. Scatter inputs
   - Multiply or rotate
3. Modulus

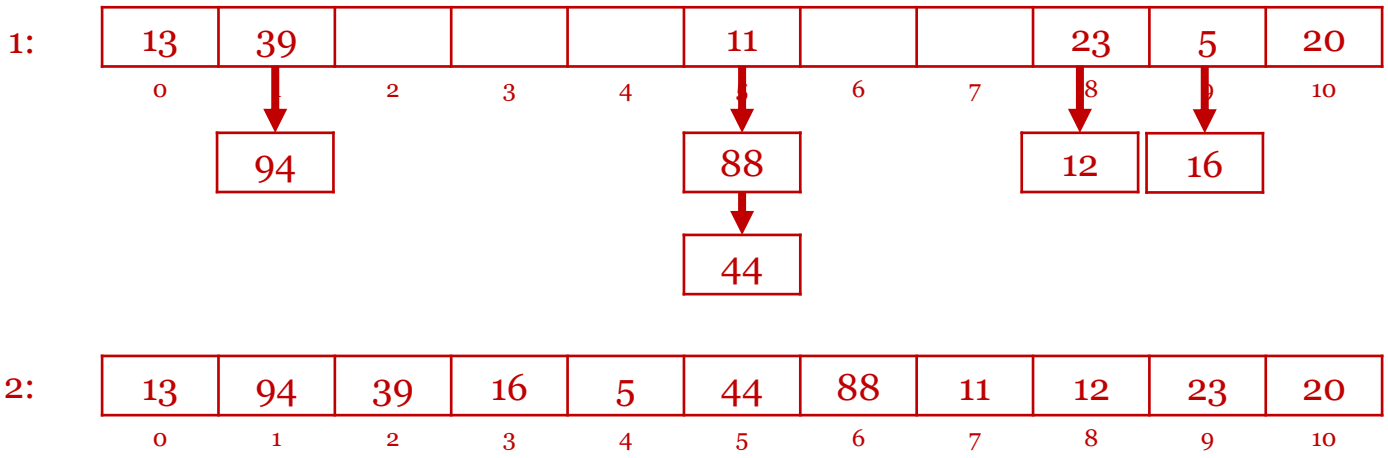
Hash table exercise

1. Draw the 11-entry hash table that results from using the hash function $h(i) = (3i + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
2. What is the result of the previous question if collisions are handled by linear probing?



Hash table exercise

1. Draw the 11-entry hash table that results from using the hash function $h(i) = (3i + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
2. What is the result of the previous question if collisions are handled by linear probing?



Sets

- 3 main operations: Insert(x), Delete(x), Search(x)
- 4/5 secondary operations: Max(), Min(), Successor(x), Predecessor(x)

- Two main implementations with various pros/cons
  - Balanced binary search tree
  - Hash table (expected case)
  - Time complexity, time coefficient (e.g., caching), space (e.g., links vs. no links, empty cells)

- Hash tables: separate chaining vs. open addressing
  - *Load factor:* size of table relative to # of elements

Set complexity

Operation	BBST	Hash table
Search(x)	$O(\lg n)$	$O(1)^{\dagger}$
Delete(x)	$O(\lg n)$	$O(1)^{\dagger}$
Insert(x)	$O(\lg n)$	$O(1)^{*\dagger}$
Min()	$O(\lg n)$	$O(n)^{\dagger}$
Max()	$O(\lg n)$	$O(n)^{\dagger}$
Pred(x)	$O(\lg n)$	$O(n)^{\dagger}$
Succ(x)	$O(\lg n)$	$O(n)^{\dagger}$

* Amortized time
† Expected case

Set exercise

1. Create a table with the complexity of the algorithm below using 4 different set implementations:
   - Sorted and unsorted array, balanced BST, hash table (expected)
   - Assume A, B, and C are *unsorted arrays*

```
   Input: A, B: arrays of integers
   Input: m, n: length of A and B, respectively
   Output: intersection of A and B
 1 Algorithm: Intersect
 2 S = Set()
 3 C = {}
 4 for i = 1 to n do
 5 |   S.Insert(B[i])
 6 end
 7 for i = 1 to m do
 8 |   if S.Search(A[i]) ≠ NIL then
 9 |   |   Add A[i] to C
10 |   end
11 end
12 return C
```

2. Would the runtime change if we swapped A and B?

Back

Set exercise solution

1. Intersect will perform:
   - n calls to Insert() (size of B)
   - m calls to Search() (size of A)
   - $\Theta(m+n)$ other operations

Implementation	n insertions	m searches	Total time
Unsorted array	$\Theta(n)$	$\Theta(mn)$	$\Theta(mn)$
Sorted array	$\Theta(n^2)$	$\Theta(m \lg n)$	$\Theta(m \lg n + n^2)$
Balanced BST	$\Theta(n \lg n)$	$\Theta(m \lg n)$	$\Theta((m+n) \lg n)$
Hash table	$\Theta(n)^\dagger$	$\Theta(m)^\dagger$	$\Theta(m + n)^\dagger$

2. Doesn't matter when using a hash table or unsorted array

- Sorted array: n affects complexity more—faster if B is smaller
- Balanced BST: n affects complexity more—slightly faster if B is smaller

Maps

- Associates some set off values to another
  - Very useful as look-up table for computations
- Two implementations
- Array-based map
  - *f(i)* stored at *arr[i]*
  - All operations other than initialization take $\Theta(1)$ time
  - Space determined by domain size
- Set-based map (hash map)
  - Set of ordered pairs
  - Complexity determined by set implementation
    - BBST or hash table
  - Space depends on number of values stored
  - Not as efficient if most values in domain appear in map

Priority queues and heaps

- **Priority Queue**
  - Abstract data structure that supports extracting max/min element
  - Main operations (max): Max(), DeleteMax(), Insert(x)
- **Heap:** primary implementation for Priority Queue
  - Array-based complete BST
  - *Heap property*: all children are smaller (larger) than their parent
  - Parent of i is at $i/2$, children are at $2i$ and $2i+1$
  - Helper operations: PercolateUp(i), PercolateDown(i)
    - Shift a value up or down in the tree to satisfy heap property
    - Both: *O(lg n)*

Operation	Heap
Insert(x)	*O(lg n)*
Max()	*O(1)*
DeleteMax()	*O(lg n)*
Heapify	*O(n)*

- No Fibonacci heaps on exam!

Heap exercise

- The algorithm below takes in an index for an element in a heap and reduces its value, then rebalances the heap:

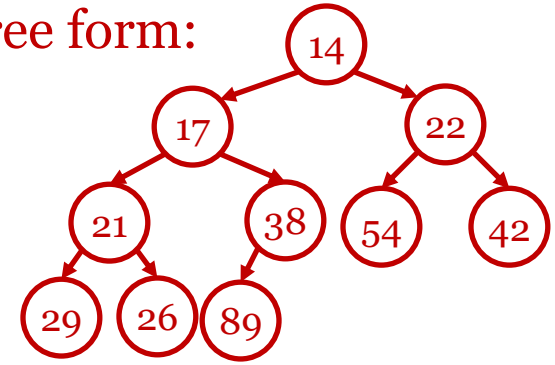
```
   Input: heap: heap to modify (min heap)
   Input: i: index of value in heap to change
   Input: v: new value for heap[i]
1  Algorithm: heap.ReduceKey(i, v)
2  heap[i] = v
3  p = ⌊i/2⌋
4  while i > 1 and heap[i] < heap[p] do
5  |   Swap heap[i] and heap[p]
6  |   i = p
7  |   p = ⌊i/2⌋
8  end
```

1. What is the result of calling ReduceKey(9, 17) on the min heap [14, 21, 22, 26, 38, 54, 42, 29, 75, 89]?
2. What is the worst-case time complexity of ReduceKey(i, j) on a heap with n values? Justify your answer.

Heap exercise

1. Array form: [14, 17, 22, 21, 38, 54, 42, 29, 26, 89]

   Tree form:



2.

Complexity

	Input: *heap*: heap to modify (min heap)
	Input: i: index of value in *heap* to change
	Input: v: new value for $heap[i]$
	1 **Algorithm:** *heap*.ReduceKey(i, v)
$\Theta(1)$	2 $heap[i] = v$
$\Theta(1)$	3 $p = \lfloor i/2 \rfloor$
$O(\lg i)$ iter.	4 **while** $i > 1$ and $heap[i] < heap[p]$ **do**
$\Theta(1)$	5 Swap $heap[i]$ and $heap[p]$
$\Theta(1)$	6 $i = p$
$\Theta(1)$	7 $p = \lfloor i/2 \rfloor$
	8 **end**

Total: $O(\lg i)$

Since $i = O(n)$, $O(\lg n)$ is also a valid answer.

Back

Union-Find operations

- **Initialize(n)**
  - Assigns every element to its own partition
  - $O(n)$

```
1 Algorithm: UnionFind(n)
2 unionfind = Array(n)
3 for i = 1 to n do
4 |  unionfind[i] = i
5 end
6 Let size be an array of n ones
7 return unionfind
```

- **Find(x)**
  - Follow links to partition ID (root)
  - Recursively point to root
  - $O(\alpha(n))$
  - Generally less than 5 for conceivable n

```
1 Algorithm: Find(x)
2 if unionfind[x] ≠ x then
3 |  id = Find(unionfind[x])
4 |  unionfind[x] = id
5 end
6 return unionfind[x]
```

- **Union(a, b)**
  - Find root of both sides
  - Point smaller tree to larger
  - $O(\alpha(n))$

```
1 Algorithm: Union(a, b)
2 ra = Find(a)
3 rb = Find(b)
4 if size[ra] > size[rb] then
5 |  Swap ra and rb
6 end
7 unionfind[ra] = rb
8 size[rb] = size[rb] + size[ra]
```

Union-Find exercise

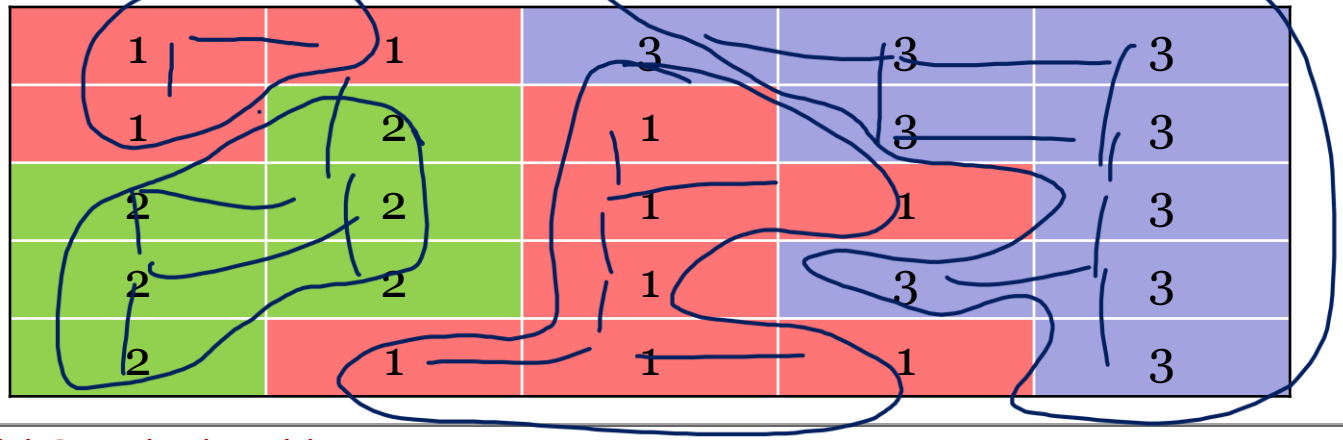
- **Problem:** blob counting
- **Input:** an n by n matrix of integers 1-4
- **Output:** number of contiguous regions of the same integer
  - Contiguous: cells adjacent horizontally or vertically
- **Example:** $n = 5$, 4 blobs

1	1	3	3	3
1	2	1	3	3
2	2	1	1	3
2	2	1	3	3
2	1	1	1	3

1. Design an algorithm to count blobs
2. Analyze its complexity

Union-Find exercise

- Blob counting algorithm and complexity
- **Output:** number of contiguous regions of the same integer




```
blobCounting(matrix):
  uf = UnionFind(n^2) //O(n^2)
  for each cell in matrix: //O(n^2) iters @ O(alpha(n^2)) / iter => O(n^2alpha(n^2))
    look at 4 neighbors //O(1)
    if cell has the same number as a neighbor: //O(1)
        union them //O(alpha(n^2))

  count = 0 // O(1)
  for i = 1 to n^2: //O(n^2) iters @ O(1) / iter => O(n^2)
    if uf[i] = i: //O(1)
      count++ //O(1)
  return count //O(1)
Total:  O(n^2alpha(n^2))
```

Back

Union-Find exercise solution

- **Main idea:** use Union-Find to keep track of blobs
- **Pseudocode**
  - Initialize Union-Find
  - Iterate through all n^2 cells
  - Union with cells above, below, left, and right if they have same color
    - More clever: just check right and down (or up and left)
  - Afterwards, count number of distinct partition IDs
    - More clever: if they were distinct before, Union reduces the number of blobs by 1
    - Count backwards from n^2
- **Analysis**
  - Initialize: $O(n^2)$
  - First loop: n^2 iterations, $O(\alpha(n^2))$ time -> $O(n^2\alpha(n^2))$
  - Second loop: $O(n^2\alpha(n^2))$ if using hash table
  - Total: $O(n^2\alpha(n^2)) = O(n^2\alpha(n))$

Union-Find algorithm

Input: n: size of input matrix
Input: A: $n \times n$ matrix in which to count blobs
Output: the number of blobs in A
Algorithm: CleverBlobCount

```
uf = UnionFind(n^2)
blobs = n^2
for r = 0 to n − 1 do
    for c = 0 to n − 1 do
        x = rn + c
        if r < n − 1 then
            right = rn + c + 1
            if A[r, c] = A[r, c + 1] and uf.Find(x) ≠ uf.Find(right) then
                uf.Union(x, right)
                blobs = blobs − 1
        if c < n − 1 then
            down = (r + 1)n + c
            if A[r, c] = A[r + 1, c] and uf.Find(x) ≠ uf.Find(down) then
                uf.Union(x, down)
                blobs = blobs − 1
    end
end
return blobs
```

Complexity: $O(n^2\alpha(n^2)) = O(n^2\alpha(n))$

Algorithm design example

- What data structure is useful for the following solution to the *knapsack problem*?

- **Problem:** knapsack problem (a.k.a., subset sum)
  - **Input:** various chemicals with *weight* and *value*, and weight limit w
  - **Output:** amount of each chemical to take with weight at most w and maximum value (fractional values allowed)

- **Algorithm strategy**
  - Add the most expensive chemical
  - Repeat until weight exceeds w
  - Take as much of the last chemical to hit the weight limit

Algorithm design example

MaxHeap is useful because we are trying to find the max value/weight
We can use a map to represent how much of each chemical we take

```
   Input: chem: array of chemicals
   Input: n: number of chemicals
   Input: w: max weight
   Output: amount of each chemical to take to maximize value
 1 Algorithm: Knapsack
 2 heap = MaxHeap()
 3 for i = 1 to n do
 4 |   Insert chem[i] into heap according to chem[i].value / chem[i].weight
 5 end
 6 takechem = Array(n)
 7 Initialize takechem to 0
 8 weight = 0
 9 while weight < w do
10 |   c = heap.DeleteMax()
11 |   takechem[c] = 1
12 |   weight = weight + chem[c].weight
13 end
14 takechem[c] = takechem[c] − (weight − w)/chem[c].weight
15 return takechem
```

Sorting

- Three quadratic algorithms
  - SelectionSort, BubbleSort, InsertionSort
- Three $n \lg n$ algorithms
  - HeapSort, MergeSort, QuickSort
  - Average case for QuickSort: $\Theta(n \lg n)$
- Three non-comparison-based algorithms
  - CountingSort, BucketSort, RadixSort
  - All can sort in $\Theta(n)$ in some cases
- Be familiar with:
  - Pseudocode, complexity (best, worst, average case), advantages and disadvantages
- Lower bound for comparison-based sorting
  - $\Omega(n \lg n)$
  - Proof based on counting execution paths

Sorting exercise

1. Analyze the worst-case complexity of the algorithm below.
   - Binary search on an array of size x takes $\Theta(\lg x)$ time
2. How does it compare to InsertionSort?

Input: *data*: array of integers
Input: n: size of *data*
Output: permutation of *data* such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$

```
1 Algorithm: BinaryInsertionSort
2 for i = 2 to n do
3 |   ins = data[i]
4 |   Use binary search to find where to insert ins into
  |    data[1..i − 1]
5 |   Insert ins into data[1..i − 1] at that location
6 end
7 return data
```

Sorting exercise

1. Analyze the worst-case complexity of the algorithm below.
   - Binary search on an array of size x takes $\Theta(\lg x)$ time
2. How does it compare to InsertionSort?

Input: *data*: array of integers
Input: n: size of *data*
Output: permutation of *data* such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$

```
1 Algorithm: BinaryInsertionSort
2 for i = 2 to n do
3 |   ins = data[i]
4 |   Use binary search to find where to insert ins into
  |    data[1..i − 1]
5 |   Insert ins into data[1..i − 1] at that location
6 end
7 return data
```

$\Theta(n)$ iter
$\Theta(1)$
$\Theta(\lg i)$
$O(i)$
$\Theta(1)$

$$\sum_{i=2}^{n} O(i) = O(n^2)$$

$O(n^2)$ complexity overall: line 4 complexity is dominated by line 5

It might be slightly faster than InsertionSort on large data, but the best case is no longer linear due to binary search. It's still much worse than MergeSort or QuickSort on large data.

Coming up

- **Midterm Exam** will be next week
  - Single page of notes
    - Double-sided, standard paper size
  - Submit **Feedback Form** for bonus homework points
  - **Practice exam** to be posted
  - *Optional review session* Monday at 6pm via Teams

- *Post-exam:* algorithm design