

COMP 251

Algorithms & Data Structures (Winter 2022)

Comp250 - Review

School of Computer Science
McGill University

Based on slides from (Langer,2012), (CRLS, 2009),
(Waldispuhl,20'21'), (Sora,2015), Hatami, Bailey, Stepp, Alberini,
Snoeyink & Blanchette.

Announcements

- Software
 - Ed-Discussion: In MyCourses => Content/Information/Ed-Discussion
 - Email: cs251-winter@cs.mcgill.ca
 - Ed-Lessons: In MyCourses => Content/Information/Ed-Discussion
- Lecture.
 - <https://docs.google.com/document/d/15y0MH4cR3yMjOmN4WfyawG5iLuijAFJZ1GioSWG8hTc/edit?usp=sharing>

Zoom Link:
<https://mcgill.zoom.us/j/84851396315>

Index of Lectures:
[Lecture January 11](#) (Comp250 review)

■ Lecture January 11

1. Data Structures.

Q: What does ADT mean? [slide 6]

A: Abstract data structure.

A: (David) It is also known in the CS field as **Abstract Data Type**.

Q: What are segment trees?[slide 8]

A: It's a special type of binary search tree that performs a specific operation (say sum or maximum) over a range of array values and stores that in the tree instead of the array elements. Useful when you need to query for a certain operation (eg. sum/max/min) over a subset of elements.

2. Induction and Recursion.

3. Tools for analysis of algorithms

Announcements

- Office Hours (more slots in "busy" weeks).

OFFICE HOURS.

	Monday	Tuesday	Wednesday	Thursday	Friday
9am - 10am					
10am - 11am		Shishir-OH	Rui-OH	Jennifer-OH	T.A meeting
11am - 12pm	David-OH	Shishir-OH	Yaxuan Li-OH		Zedian Xiao-OH
12pm - 1pm				Parham-OH	
1pm - 2pm		Itai-OH			Ade-OH
2pm - 3pm		Lecture		Lecture	James-OH
3pm - 4pm	James-OH	Lecture	David-OH	Lecture	
4pm - 5pm		Shael-OH		Alvin-OH	
5pm - 6pm					
6pm - 7pm					
7pm - 8pm					
8pm - 9pm					
9 pm - 10pm					

What should you already know?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

What I would need for Comp251?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

Data Type VS Data Structure

Examples

ADT	Data Structure
list	array, linked list
stack	
queue	
priority queue	heap, sorted array,
map	hash table,
⋮	⋮

abstract data type

Figure taken from Comp251 - 2014

- Sometimes the boundary between DT and DS is unclear and arbitrary. In Comp251, we will use many partially implemented DS (this will not happen during the implementation) during the analysis of algorithms, .
 - Enough details to discuss the complexity of the algorithm.

Data Type VS Data Structure

Examples

ADT	Data Structure	
list stack queue	array, linked list	Linear
priority queue	heap, sorted array,	
map	hash table,	Non-Linear
⋮	⋮	

Data Structures Classification

- Basic data structures (built-in libraries).
 - Linear DS.
 - Non-Linear DS.
- Data structures (Own libraries).
 - Graphs.
 - Union-Find Structures.
 - Segment Tree.

Data Structures Classification

- Basic data structures (built-in libraries).
 - Linear DS (ordering the elements sequentially).
 - Static Array (Array in C/C++ and in Java).
 - Resizable array (C++ STL<vector> and Java ArrayList).
 - Linked List: (C++ STL<list> and Java LinkedList).
 - Stack (C++ STL<stack> and Java Stack).
 - Queue (C++ STL <queue> and Java Queue).

Data Structures Classification

- Basic data structures (built-in libraries).
 - Non-Linear DS. *non-consecutive indexes*
 - Balanced Binary Search Tree (C++ STL `<map>/<set>` and in Java `TreeMap/TreeSet`).
 - AVL and Red-Black Trees = Balanced BST
 - `<map>` stores (key -> data) VS `<set>` only stores the key
 - Heap (C++ STL `<queue>:priority_queue` and Java `PriorityQueue`).
 - BST complete.
 - Heap property VS BST property.
 - Hash Table (Java `HashMap/HashSet/HashTable`).
 - Non synchronized vs synchronized.
 - Null vs non-nulls
 - Predictable iteration (using `LinkedHashMap`) vs non predictable.

Deciding the Order of the Tasks

- Returns the newest task (stack)
- Returns the oldest task (queue)
- Returns the most urgent task (priority queue)
- Returns the easiest task (priority queue)

Stack

- Last in, first out (Last In First Out)
- Stacks model piles of objects (such as dinner plates)
- Supports three constant-time operations
 - Push(x): inserts x into the stack
 - Pop(): removes the newest item
 - Top(): returns the newest item
- Very easy to implement using an array
- Useful for:
 - Processing nested formulas
 - Depth-first graph traversal
 - Data storage in recursive algorithms
 - Reverse a sequence
 - Matching brackets
 - And a lot more

return newest task



last in, first out

Queue

- First in, first out (FIFO)
- Supports three constant-time operations
 - Enqueue(x): inserts x into the queue
 - Dequeue(): removes the oldest item
 - Front(): returns the oldest item
- Implementation is similar to that of stack
- Useful for:
 - implementing buffers
 - simulating waiting lists
 - shuffling cards
 - And a lot more

return oldest task

Priority Queue

- Each element in a PQ has a priority value
- Three operations:
 - Insert(x, p): inserts x into the PQ, whose priority is p
 - RemoveTop(): removes the element with the highest priority
 - Top(): returns the element with the highest priority
- All operations can be done quickly if implemented using a heap (if not use a sorted array)
- priority_queue (C++), PriorityQueue (Java)
- Useful for
 - Maintaining schedules / calendars
 - Simulating events
 - Breadth-first search in a graph
 - Sweep line geometric algorithms

returns most urgent task

important

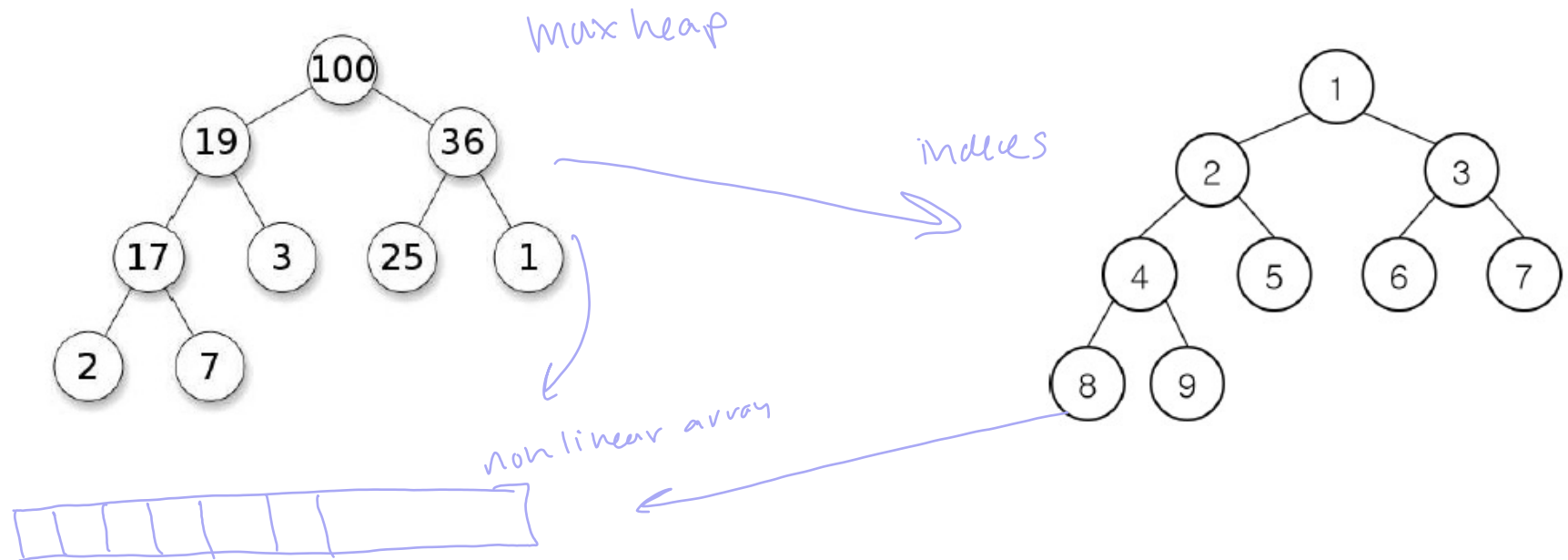
can be made with a heap

Heap

- Complete binary tree with the heap property:
 - The value of a node \geq values of its children
 - What is the difference between full vs complete?
- The root node has the maximum value
 - Constant-time top() operation
- Inserting/removing a node can be done in $O(\log n)$ time without breaking the heap property
 - May need rearrangement of some nodes

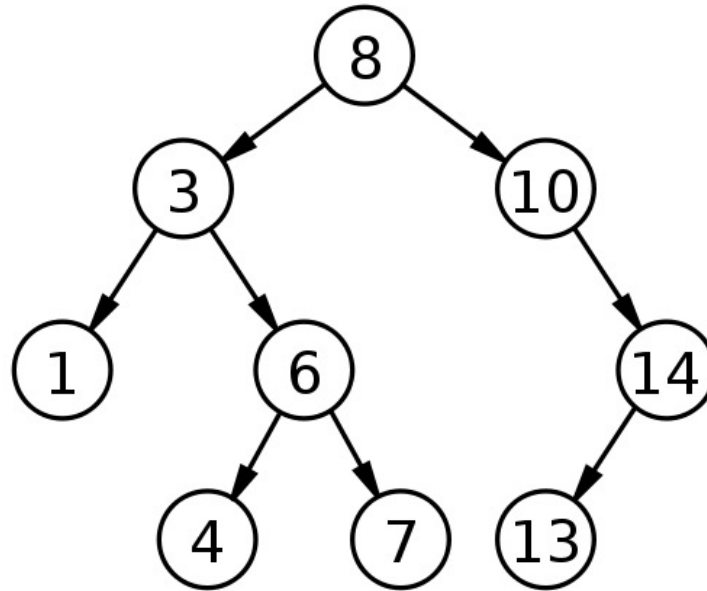
*maintains
restores heap property*

Heap



- Start from the root, number the nodes 1, 2, . . . from left to right
- Given a node k easy to compute the indices of its parent and children
 - Parent node: $\text{floor}(k/2)$
 - Children: $2k$, $2k + 1$

BST Binary Search Tree



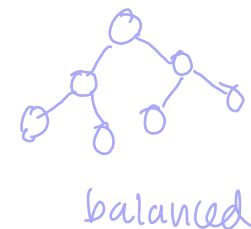
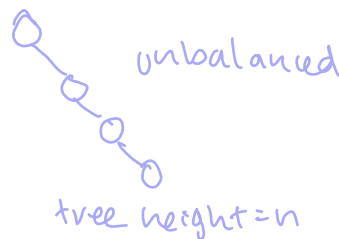
- The idea behind is that each node has, at most, two children
- A binary tree with the following property: for each node v ,
 - value of $v \geq$ values in v 's left subtree
 - value of $v <$ values in v 's right subtree

BST Binary Search Tree

- Supports three operations
 - Insert(x) : inserts a node with value x
 - Delete(x) : deletes a node with value x , if there is any
 - Find(x) : returns the node with value x , if there is any
- Many extensions are possible
 - Count(x) : counts the number of nodes with value less than or equal to x
 - GetNext(x) : returns the smallest node with value $\geq x$
 - Curious about the extensions? Please register Comp321 next term :P

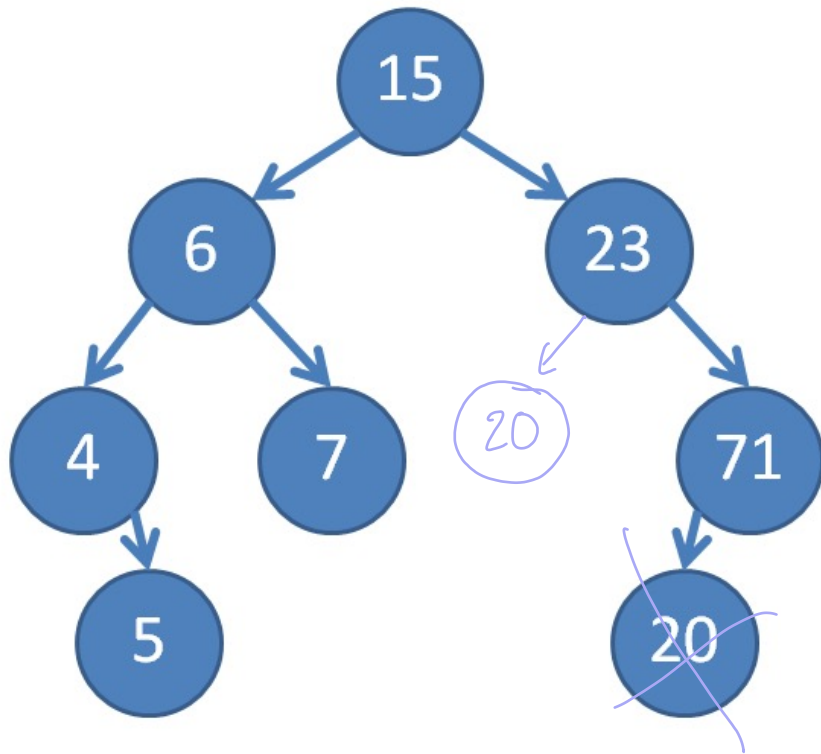
BST Binary Search Tree

- Simple implementation cannot guarantee efficiency
 - In worst case, tree height becomes n (which makes BST useless)
- Guaranteeing $O(\log n)$ running time per operation requires balancing of the tree (hard to implement).
 - For example AVL and Red-Black trees
 - We will skip the details of these balanced trees because we will fully cover them during the next lectures.
 - What does balanced mean??
 - The heights of the two child subtrees of any node differ by at most one.

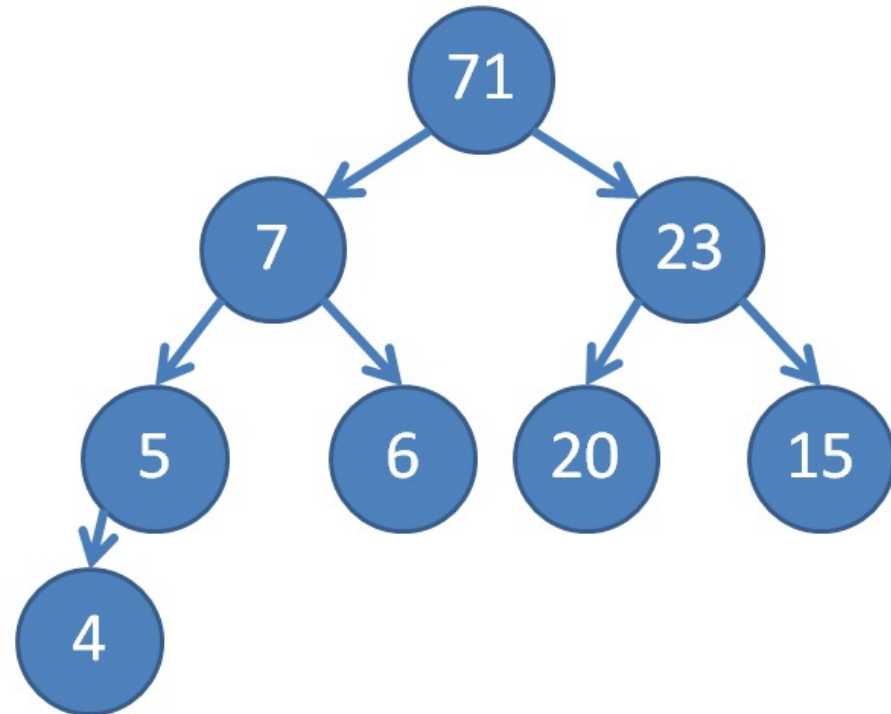


Question for you

- Basic data structures (built-in libraries).



BST

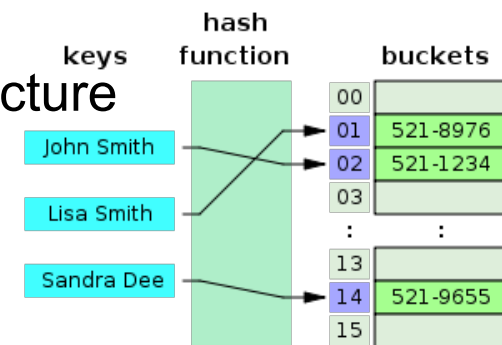


HEAP

Do you recognize the problem?

Hash Tables

- A key is used as an index to locate the associated value.
 - Content-based retrieval, unlike position-based retrieval.
 - Hashing is the process of generating a key value.
 - An ideal algorithm must distribute evenly the hash values => the buckets will tend to fill up evenly = fast search.
 - A hash bucket containing more than one value is known as a “collision”.
 - Open addressing => A simple rule to decide where to put a new item when the desired space is already occupied.
 - Chaining => We associate a linked list with each table location.
 - Hash tables are excellent dictionary data structures.
 - We will cover all the details about hash tables next lecture



Data Structures

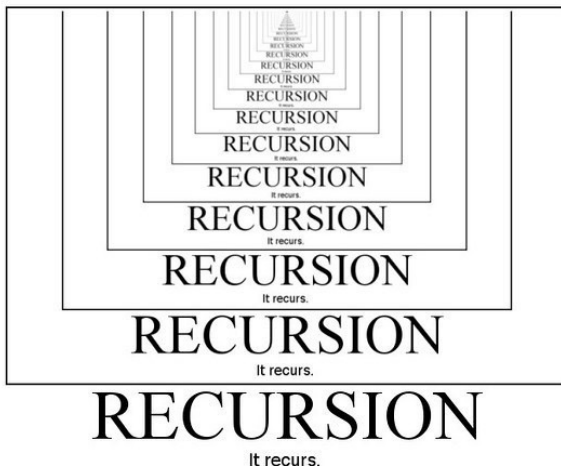
- Data structures (Own Libraries).
 - Graphs.
 - We will talk a lot about them in the last part of the course.
 - Union-Find Disjoint Sets
 - We will cover it a lot during the first part of the course, starting in two lectures.
 - Segment tree.
 - We will not cover it, but if you are curious register Comp321 next term ;)

What have we covered so far?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

Recursion - Definitions

- **Recursive definition:**
 - A definition that is defined in terms of **itself**.
- **Recursive method:**
 - A method that calls **itself** (directly or indirectly).
- **Recursive programming:**
 - Writing methods that call **themselves** to solve problems recursively.



Did you watch a movie called 'Inception'?

Recursion - Example

$c(x)$ = total number of credits required to complete course x

$c(\text{COMP462}) = ?$

= 3 credits + **#credits for prerequisites**

#COMP462 has 2 prerequisites: COMP251 & MATH323

= 3 credits + **$c(\text{COMP251}) + c(\text{MATH323})$**

The function c calls itself twice

$c(\text{COMP251}) = ?$

$c(\text{MATH323}) = ?$

$c(\text{COMP251}) = 3 \text{ credits} + c(\text{COMP250})$ #COMP250 is a prerequisite

- Substitute $c(\text{COMP251})$ into the formula:
- $c(\text{COMP462}) = 3 \text{ credits} + 3 \text{ credits} + c(\text{COMP250}) + c(\text{MATH323})$
- $c(\text{COMP462}) = 6 \text{ credits} + c(\text{COMP250}) + c(\text{MATH323})$

Recursion - Example

$$c(\text{COMP462}) = 6 \text{ credits} + c(\text{COMP250}) + c(\text{MATH323})$$

$$c(\text{COMP250}) = ? \quad c(\text{MATH323}) = ?$$

$$c(\text{COMP250}) = 3 \text{ credits} \# \text{ no prerequisite}$$

$$c(\text{COMP462}) = 6 \text{ credits} + 3 \text{ credits} + c(\text{MATH323})$$

$$c(\text{MATH323}) = ?$$

$$c(\text{MATH323}) = 3 \text{ credits} + c(\text{MATH141})$$

$$c(\text{COMP462}) = 9 \text{ credits} + 3 \text{ credits} + c(\text{MATH141})$$

$$c(\text{MATH141}) = ?$$

$$c(\text{MATH141}) = 4 \text{ credits} \# \text{ no prerequisite}$$

$$c(\text{COMP462}) = 12 \text{ credits} + 4 \text{ credits} = 16 \text{ credits}$$

Recursion – Algorithm Structure

- Every recursive algorithm involves at least 2 cases:
 - **base case**: A simple occurrence that can be answered directly.
 - **recursive case**: A more complex occurrence of the problem that cannot be directly answered but can instead be described in terms of smaller occurrences of the same problem.
- Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
- **A crucial part of recursive programming is identifying these cases.**

Recursion – Algorithm Example

- **Algorithm** binarySearch(array, start, stop, key)
 - **Input:**
 - A sorted array
 - the region start...stop (inclusively) to be searched
 - the key to be found
 - **Output:** returns the index at which the key has been found, or returns -1 if the key is not in array[start...stop].

- **Example:** Does the following **sorted** array A contains the number 6?

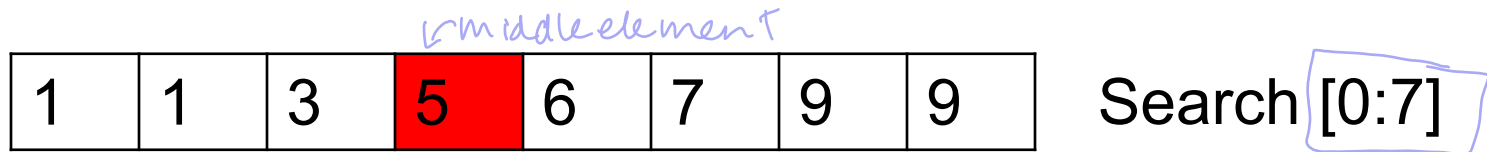
A =

1	1	3	5	6	7	9	9
---	---	---	---	---	---	---	---

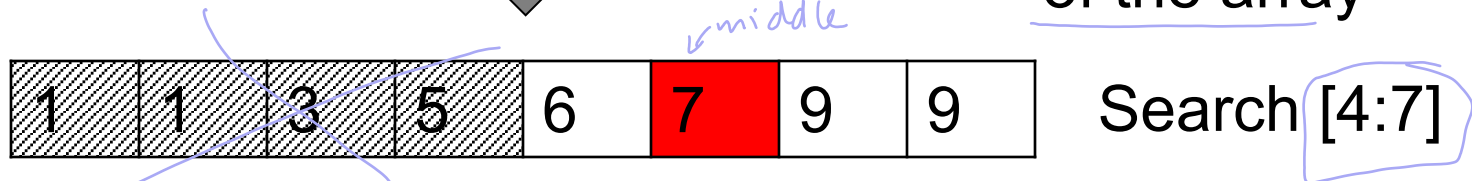
(Handwritten blue annotations: indices 0-7 written below the array, and a long arrow above the array pointing from index 0 to index 7)

Call: `binarySearch(A, 0, 7, 6)`

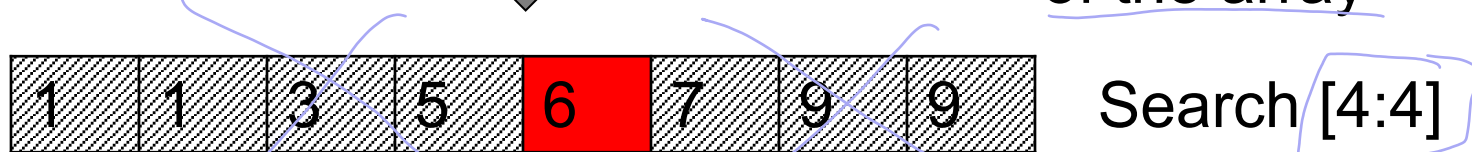
Recursion – Algorithm Example



$5 < 6 \Rightarrow$ look into right half
of the array



$7 > 6 \Rightarrow$ look into left half
of the array



6 is found. Return 4 (index)

Recursion – Algorithm Example

```
int bsearch(int[] A, int i, int j, int x) {  
    if (i <= j) { // the region to search is non-empty  
        int e =  $\lfloor (i+j)/2 \rfloor$ ; compute middle point  
        if (A[e] > x) {  
            return bsearch(A, i, e-1, x);  
        } else if (A[e] < x) {  
            return bsearch(A, e+1, j, x);  
        } else {  
            return e; - found  
        }  
    } else { return -1; } // value not found  
}
```

*which part
of array
are we
interested*

What have we covered so far?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

Recursion(CS) VS Recurrence(Math)

- A **recurrence** is a function that is defined in terms of
 - one or more base cases, and
 - itself, with smaller arguments.
- Examples:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2 \cdot n}{3}\right) + n & \text{if } n > 1 \end{cases}$$

- Many technical issues:
 - Floors and ceilings
 - Exact vs. *asymptotic* functions
 - Boundary conditions
- We usually express both the recurrence and its solution using *asymptotic* notation.

Algorithm Analysis

- **Q:** How to estimate the running time of a recursive algorithm?
- **A:**
 1. Define a function $T(n)$ representing the time spent by your algorithm to execute an entry of size n
 2. Write a recursive formula computing $T(n)$
 3. Solve the recurrence
- **Notes:**
 - n can be anything that characterizes accurately the size of the input (e.g. size of the array, number of bits)
 - We count the number of elementary operations (e.g. addition, shift) to estimate the running time.
 - We often aim to compute an upper bound rather than an exact count.

Algorithm Analysis (binary search)

```
int bsearch(int[] A, int i, int j, int x) {  
    if (i <= j) { // the region to search is non-empty  
        int e =  $\lfloor (i+j)/2 \rfloor$ ;  
        if (A[e] > x) { return bsearch(A, i, e-1, x);  
        } else if (A[e] < x) { return bsearch(A, e+1, j, x);  
        } else { return e; }  
    } else { return -1; } // value not found  
}
```

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + c' & \text{if } n > 1 \end{cases}$$

base case
recursive calls

- n is the size of the array

Algorithm Analysis

- **Q:** How to estimate the running time of a recursive algorithm?
- **A:**
 1. Define a function $T(n)$ representing the time spent by your algorithm to execute an entry of size n
 2. Write a recursive formula computing $T(n)$
 3. **Solve the recurrence**
- **Notes:**
 - n can be anything that characterizes accurately the size of the input (e.g. size of the array, number of bits)
 - We count the number of elementary operations (e.g. addition, shift) to estimate the running time.
 - We often aim to compute an upper bound rather than an exact count.

Substitution method

- **How to solve a recursive equation?**

1. Guess the solution.
2. Use **induction** to find the constants and show that the solution works.

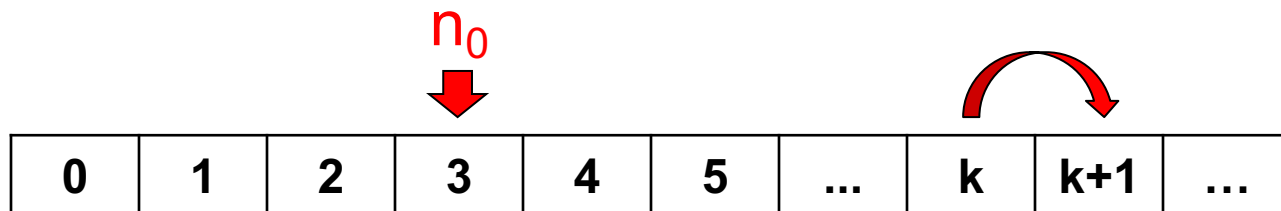
Mathematical Induction

- Many statement of the form “for all $n \geq n_0$, $P(n)$ ” can be proven with a logical argument call *mathematical induction*.
- The proof has two components:
 - Base case: $P(n_0)$
 - Induction step: for any $n \geq n_0$, if $P(n)$ then $P(n+1)$

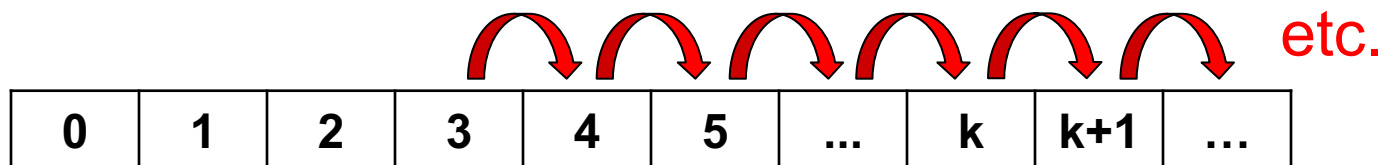


Mathematical Induction

- Many statement of the form “for all $n \geq n_0$, $P(n)$ ” can be proven with a logical argument call *mathematical induction*.



Implies



Mathematical Induction – Example 1

Claim: *for any* $n \geq 1$, $1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2}$

Proof:

- Base case: $n = 1$ $1 = \frac{1 \cdot 2}{2}$ ✓

- Induction step:

for any $k \geq 1$, *if* $1 + 2 + 3 + 4 + \cdots + k = \frac{k \cdot (k+1)}{2}$

then $1 + 2 + 3 + 4 + \cdots + k + (k+1) = \frac{(k+1) \cdot (k+2)}{2}$ *sub in k+1*

Mathematical Induction – Example 1

Assume $1 + 2 + 3 + 4 + \dots + k = \frac{k \cdot (k + 1)}{2}$

then $1 + 2 + 3 + 4 + \dots + k + (k + 1)$

$$= \frac{k \cdot (k + 1)}{2} + (k + 1)$$

$$= \frac{k \cdot (k + 1) + 2 \cdot (k + 1)}{2}$$

$$= \frac{(k + 2) \cdot (k + 1)}{2}$$

Induction hypothesis

Mathematical Induction – Example 2

Claim: *for any* $n \geq 1$, $1 + 3 + 5 + 7 + \cdots + (2 \cdot n - 1) = n^2$

Proof:

- Base case: $n = 1$ $1 = 1^2$ ✓


- Induction step:

for any $k \geq 1$, *if* $1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) = k^2$

$k+1$ *then* $1 + 3 + 5 + 7 + \cdots + (2 \cdot (k + 1) - 1) = (k + 1)^2$

Mathematical Induction – Example 2

Assume $1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) = k^2$

then $1 + 3 + 5 + 7 + \cdots + (2 \cdot k - 1) + (2 \cdot (k + 1) - 1)$  Induction hypothesis

$$= k^2 + 2 \cdot (k + 1) - 1$$

$$= k^2 + 2 \cdot k + 1$$

$$= (k + 1)^2$$

Substitution method

- **How to solve a recursive equation?**

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.
 1. Now we have the background to do this.

Substitution method – Binary Search

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + 1 & \text{if } n > 1 \end{cases}$$

Note: set the constant $c=0$ and $c'=1$

Guess: $T(n) = \log_2 n$

Base case: $T(1) = \log_2 1 = 0$ ✓

Inductive case:

Assume $T(n/2) = \log_2(n/2)$

$T(n) = T(n/2) + 1 = \log_2(n/2) + 1$

$= \log_2(n) - \log_2 2 + 1 = \log_2 n$ ✓

Induction
hypothesis can be
anything $< n$

Back Substitution method – Binary Search

SOLVING THE RECURRENCE USING BACK SUBSTITUTION

To simplify our analysis let's assume n is a power of 2. This will not affect the order of growth of the solution.

$$\begin{aligned}T(n) &= c + T(n/2) \\&= c + c + T(n/4) \\&= c + c + c + T(n/8) \\&\dots \\&= c \cdot k + T(n/2^k) \\&= c \cdot \log_2 n + T(1), \text{ when } k = \log_2 n \\&= c \cdot \log_2 n + b\end{aligned}$$

which is $\Theta(\log_2 n)$.

Taken from Prof. Giulia Alberini's lecture notes Comp250

What have we covered so far?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

Running time – Binary Search

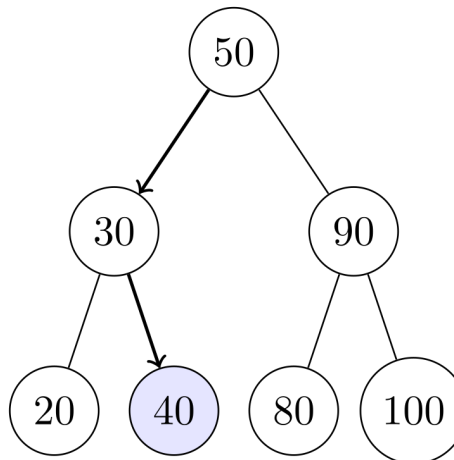
Best case: The value is exactly in the middle of the array.

$\Rightarrow \Omega(1)$

*eg 50
constant time*

Worst case: You recursively search until you reach an array of size 1
(Note: It does not matter if you find the key or not).

$\Rightarrow O(\log_2 n)$

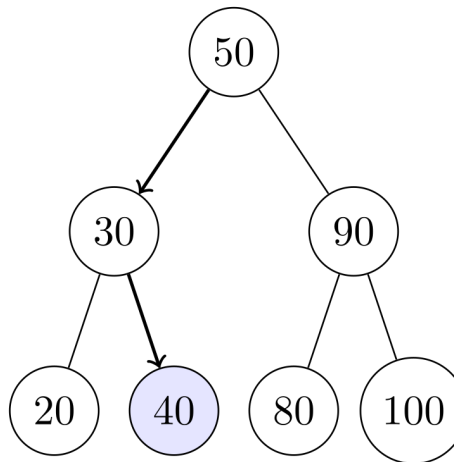


A tree representing binary search. The array being searched here is [20, 30, 40, 50, 80, 90, 100]
(from Wikipedia)

Running time – Binary Search

The time it takes for an algorithm to run depends on:

- constant factors (often implementation dependent)
- the size n of the input
- the values of the input, including arguments if applicable...



A tree representing binary search. The array being searched here is [20, 30, 40, 50, 80, 90, 100]
(from Wikipedia)

Running 'time' – Measure

- Goal: Analyze an algorithm written in pseudocode and describe its running time as a function $f(n)$ of the input size n
 - Without having to write code
 - Experimental VS Theoretical analysis *code* *no code*
 - In a way that is independent of the computer used
- To achieve that, we need to
 - Make simplifying assumptions about the running time of each basic (primitive) operations
 - Study how the number of primitive operations depends on the size of the problem solved

Running 'time' – Primitive Operations

- Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- **Assigning a value to a variable:** $x \leftarrow 1$
- **Calling a method:** `Expos.addWin()`
 - Note: doesn't include the time to execute the method
- **Returning from a method:** `return x;`
- **Arithmetic operations on primitive types**
 - $x + y$, $r * 3.1416$, x/y , etc.
- **Comparisons on primitive types:** $x == y$
- **Conditionals:** `if (...) then.. else...`
- **Indexing into an array:** `A[i]`
- **Following object reference:** `Expos.losses`

T_{assign}

T_{call}

T_{return}

T_{arith}

T_{comp}

T_{cond}

T_{index}

T_{ref}

- **Note:** Multiplying two Large Integers is *not* a primitive operation, because the running time depends on the size of the numbers multiplied.

Recursion – Algorithm Example

```
int bsearch(int[] A, int i, int j, int x) {  
    if (i<=j) {  
        int e = l(i+j)/2;  
        if (A[e] > x) {  
            return bsearch(A,i,e-1,x);  
        } else if (A[e] < x) {  
            return bsearch(A,e+1,j,x);  
        } else {  
            return e;  
        }  
    }  
    else {  
        return -1; }  
}
```

$T_{\text{cond}} + T_{\text{comp}}$
 $T_{\text{arith}} + T_{\text{assign}}$
 $T_{\text{cond}} + T_{\text{comp}} + T_{\text{index}}$
 $T_{\text{return}} + T_{\text{call}}$
 $T_{\text{cond}} + T_{\text{comp}} + T_{\text{index}}$
 $T_{\text{return}} + T_{\text{call}}$
 T_{cond}
 T_{return}
 T_{cond}
 T_{return}

repeated $\text{Log}(|A|)$
times in the worst
case scenario

Running 'time' – Primitive Operations

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$$T_{\text{assign}} \approx T_{\text{comp}} \approx T_{\text{arith}} \approx \dots \approx T_{\text{index}} = \boxed{1 \text{ primitive operation}}$$

assume constant time same for all (Δ)

- We are only interested in the **number** of primitive operations performed *which ones does not matter – just how many*
- Worst-case running time for binary search becomes:

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T\left(\frac{n}{2}\right) + c' & \text{if } n > 1 \end{cases}$$

- When n is large, $T(n)$ grows approximately like $\log(n)$. Then, we will write $T(n)$ is $O(\log(n)) \Rightarrow T(n)$ is big O of $\log(n)$

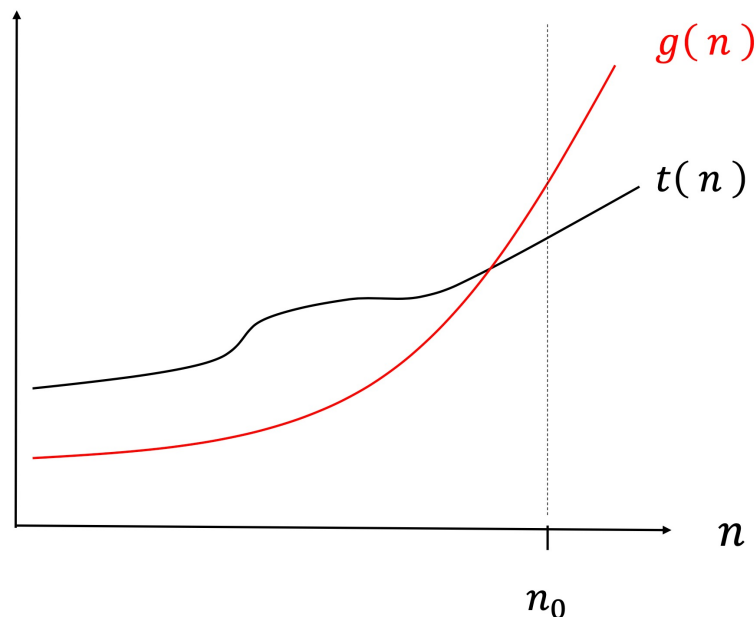
Towards a formal definition of big O

- Let $t(n)$ be a function that describes the time it takes for some algorithm on input size n .
- We would like to express how $t(n)$ grows with n , as n becomes large i.e. asymptotic behavior.
- *Unlike with limits*, we want to say that $t(n)$ grows like certain simpler functions such as $\sqrt{n}, \log_2 n, n, n^2, 2^n \dots$

Towards a formal definition of big O

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. We say $t(n)$ is *asymptotically bounded above* by $g(n)$ if there exists n_0 such that, for all $n \geq n_0$,

$$t(n) \leq g(n)$$



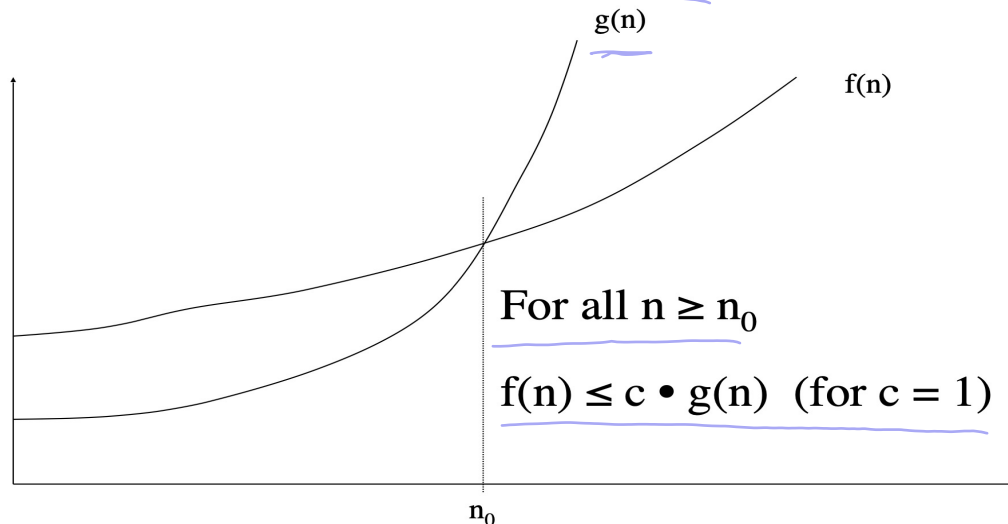
Towards a formal definition of big O

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$.
We say $t(n)$ is $O(g(n))$ if there exists two positive constants n_0 and c such that, for all $n \geq n_0$,

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

- Intuition

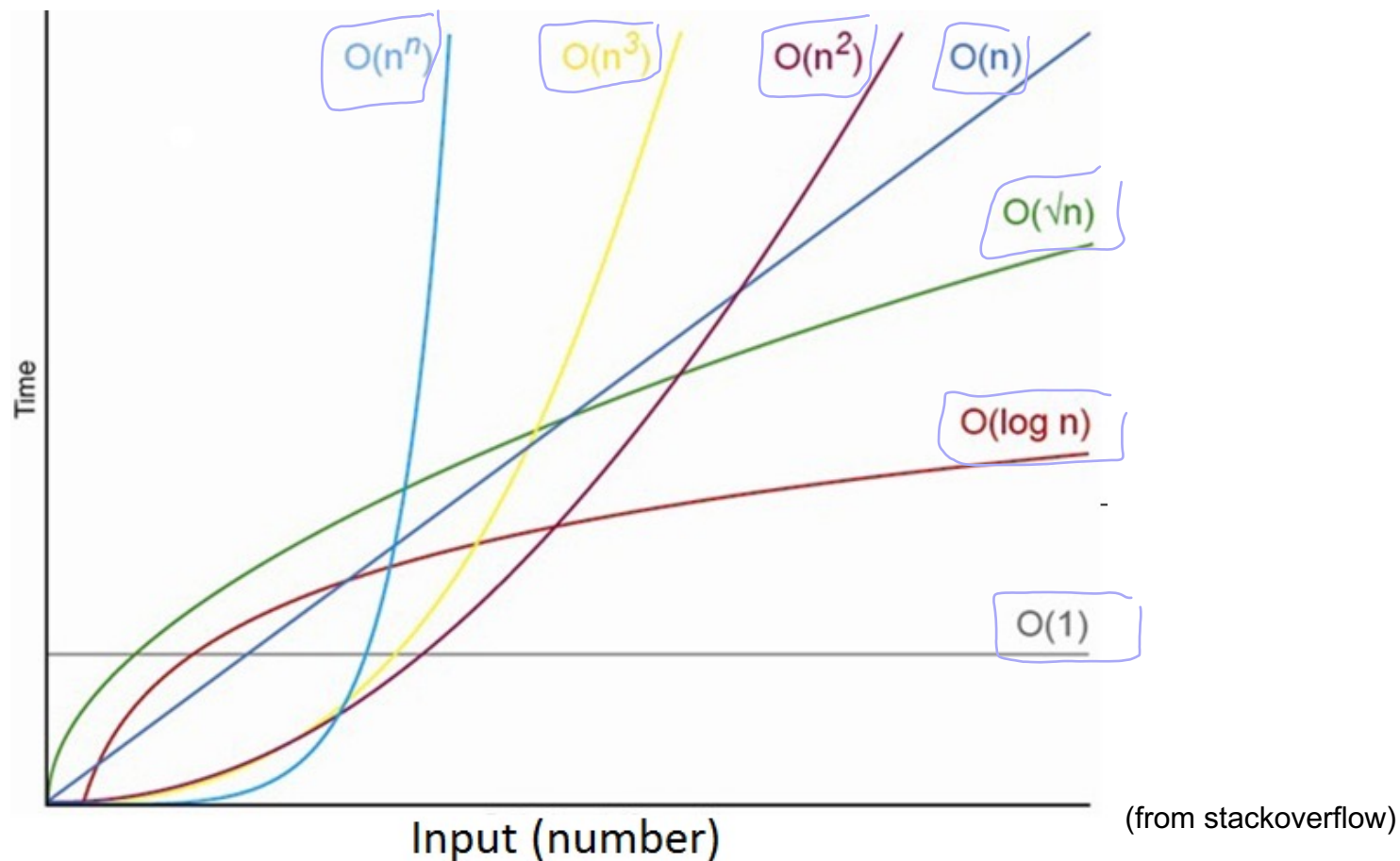
- $\text{"}f(n) \text{ is } O(g(n))\text{"}$ if and only if there exists a point n_0 beyond which $f(n)$ is less than some fixed constant times $g(n)$.



Towards a formal definition of big O

- Tips.
 - Never write $O(3n)$, $O(5 \log_2 n)$, *etc.*
 - Instead, write $O(n)$, $O(\log_2 n)$, *etc.*
 - **Why?** The point of the big O notation is to avoid dealing with constant factors. It's technically correct but we don't do it...
 - n_0 and c are not uniquely defined. For a given n_0 and c that satisfies $O()$, we can increase one or both to again satisfy the definition. **There is not “better” choice of constants.**
 - **However**, we generally want a “tight” upper bound (asymptotically), so functions in the big O gives us more information (Note: This is not the same as smaller n_0 or c). For instance, $f(n)$ that is $O(n)$ is also $O(n^2)$ and $O(2^n)$. But $O(n)$ is more informative.

Growth of functions



Tip: It is helpful to memorize the relationship between basic functions.

Growth of functions

	<i>constant</i>	<i>logarithmic</i>	<i>linear</i>	<i>N-log-N</i>	<i>quadratic</i>	<i>cubic</i>	<i>exponential</i>
<i>n</i>	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



If the unit is in seconds, this would make $\sim 10^{11}$ years...

Growth of functions

By Pictures

s **Big-Oh** (most commonly used)

□ bounded above

s **Big-Omega**

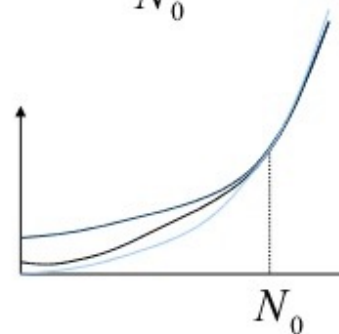
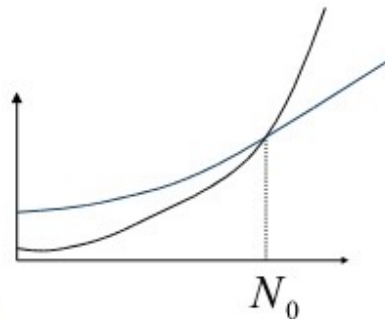
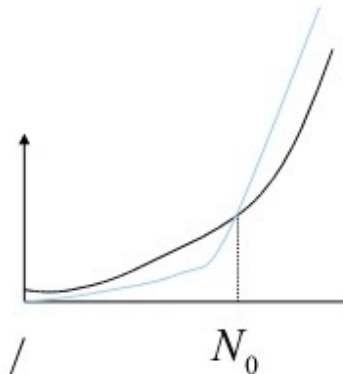
□ bounded below

s **Big-Theta**

□ exactly

s **Small-o**

□ not as expensive as ...



$f(n) = O(g(n))$	is like	$a \leq b$,
$f(n) = \Omega(g(n))$	is like	$a \geq b$,
$f(n) = \Theta(g(n))$	is like	$a = b$,
$f(n) = o(g(n))$	is like	$a < b$,
$f(n) = \omega(g(n))$	is like	$a > b$.

Towards a formal definition of big Ω

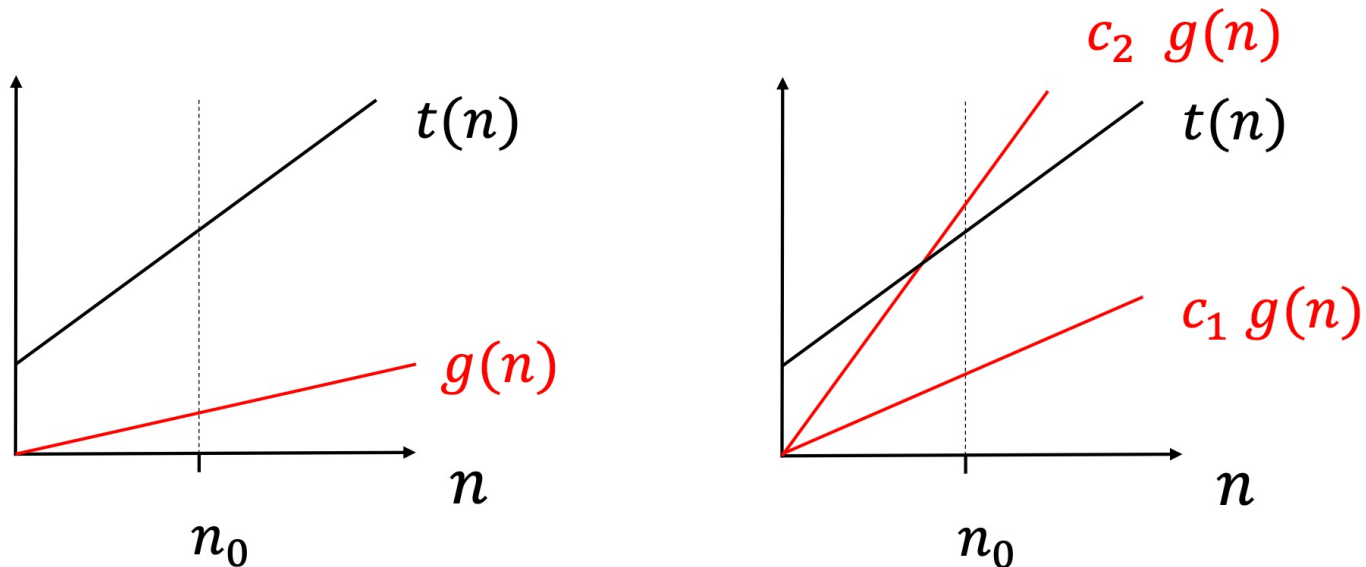
- Let $t(n)$ and $g(n)$ be two functions with $n \geq 0$.
- We say $t(n)$ is $\Omega(g(n))$, if there exists two positives constants n_0 and c such that, for all $n \geq n_0$,
- $t(n) \geq c \cdot g(n)$
- Note: This is the opposite of the big O notation. The function g is now used as a "lower bound".

Towards a formal definition of big *Theta*

- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$.

We say $t(n)$ is $\Theta(g(n))$ if there exists three positive constants n_0 and c_1, c_2 such that, for all $n \geq n_0$,

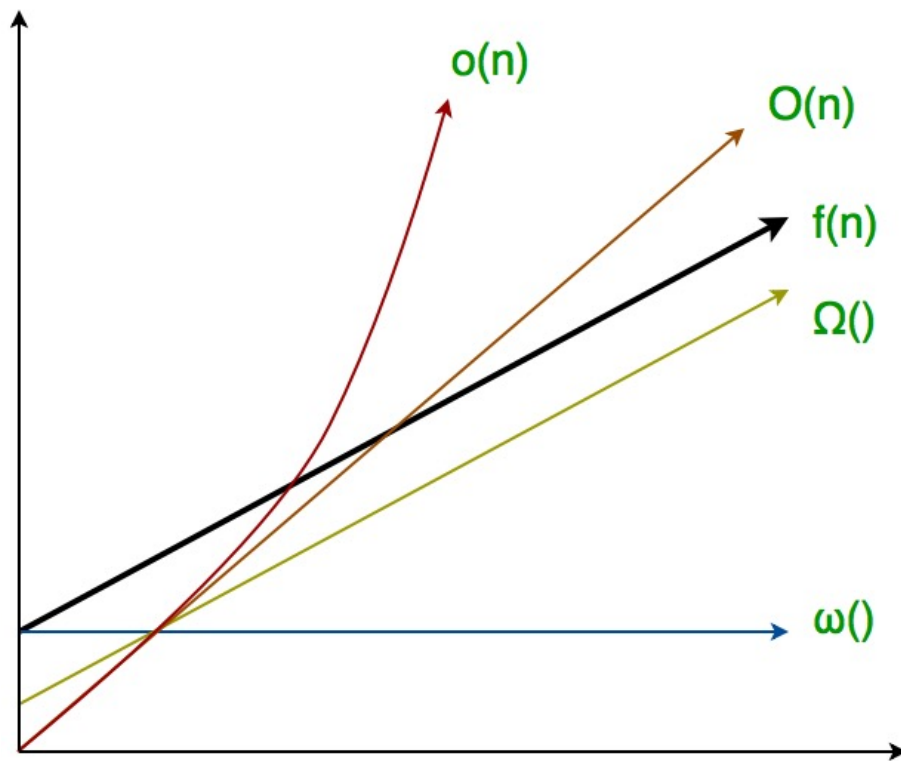
$$c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n)$$



Note: if $t(n)$ is $\Theta(g(n))$. Then, it is also $O(g(n))$ and $\Omega(g(n))$.

Big VS little

- The big O (resp. big Ω) denotes a tight upper (resp. lower) bounds, while the little o (resp. little ω) denotes a loose upper (resp. lower) bounds.



(from [geekforgeek.org](http://www.geekforgeek.org))

What have we covered so far?

- Linear Data Structures.
 - Array lists, singly and doubly linked lists, stacks, queues.
- Induction and Recursion
- Tools for analysis of algorithms.
 - Recurrences.
 - Asymptotic notation (big O, big Omega, big Theta)
- Basics in non-linear data structures.
 - Trees, heaps, maps, graphs.

