COMP 251

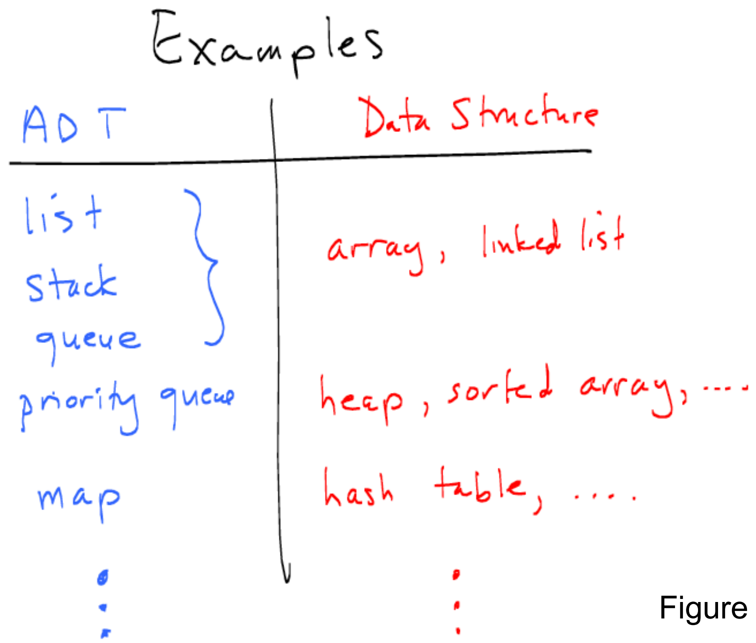
Algorithms & Data Structures (Winter 2021)

Comp250 - Review

School of Computer Science
McGill University

Based on slides from (Langer,2012), (CRLS, 2009), (Waldispuhl,2020), (Sora,2015), Hatami, Bailey, Stepp, Snoeyink & Blanchette.

Data Type VS Data Structure



Examples

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

Figure taken from Comp251 - 2014

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

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

Priority Queue

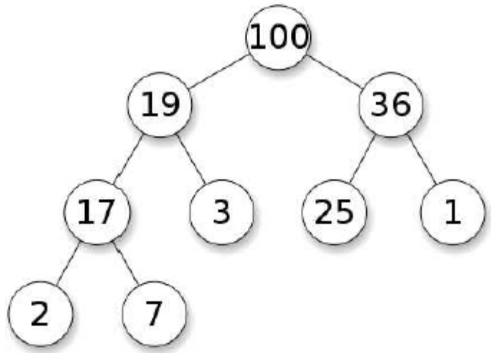
- Each element in a PQ has a priority value
- Three operations:
  - Insert(x, p) : inserts x into the PQ, whose priority is p
  - RemoveTop() : removes the element with the highest priority
  - Top() : returns the element with the highest priority
- All operations can be done quickly if implemented using 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
  - Sweepline geometric algorithms

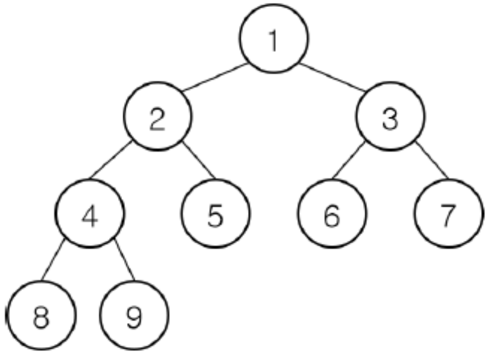
Heap

- Complete binary tree with the heap property:
  - The value of a node ≥ 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

binary tree of height h
every level less than h is full
all nodes at level h are
as far to the left as possible.

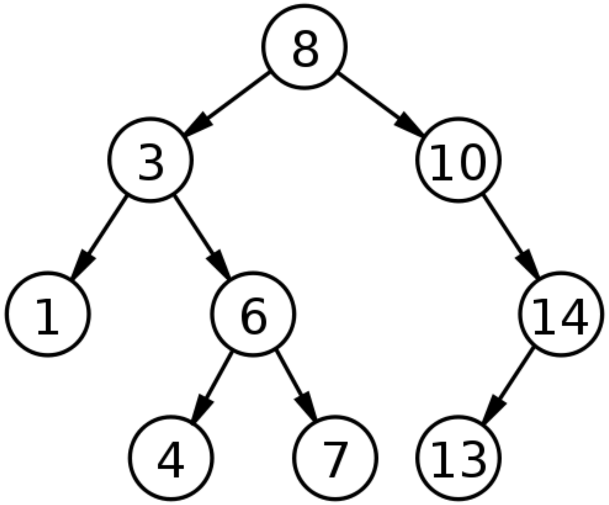
Heap





- 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: 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 ≥ values in v ’s left subtree
  - value of v < values in v ’s right subtree

BST Binary Search Tree

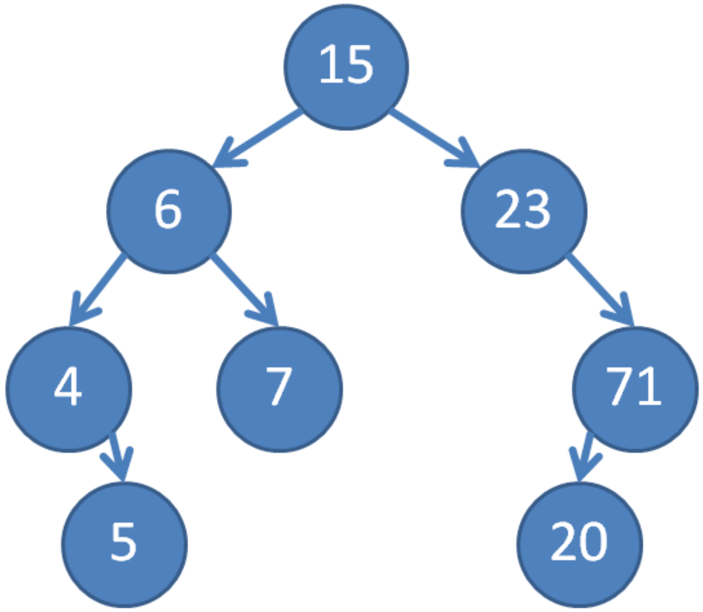
- Supports three operations
  - Insert(x) : inserts a node with value x
  - Delete(x) : deletes a node with value x , if there is any
  - Find(x) : returns the node with value x , if there is any
- Many extensions are possible
  - Count(x) : counts the number of nodes with value less than or equal to x
  - GetNext(x) : returns the smallest node with value ≥ x

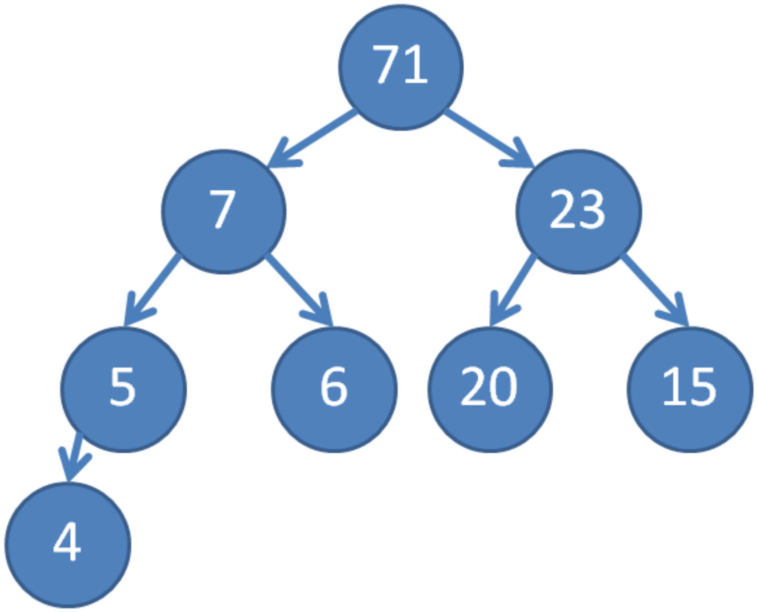
BST Binary Search Tree

- 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).





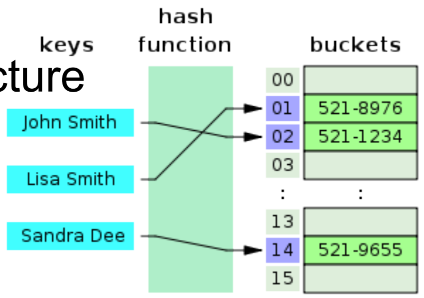
BST

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

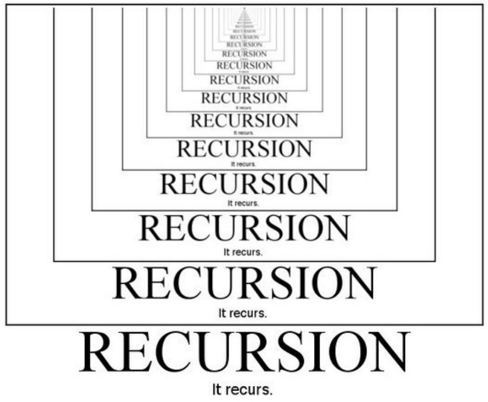
- Data structures (Own Libraries).
  - Graphs.
    - We will talk a lot about them in the last part of the course.
  - Union-Find Disjoint Sets
    - We will cover it a lot during the first part of the course, starting in two lectures.
  - Segment tree.
    - We will not cover it, but if you are curious 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(COMP462)   = ?

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

#COMP462 has 2 prerequisites: COMP251 & MATH323

$$= \underbrace{3\text{ credits} \mathbf{+ c(COMP251) + c(MATH323)}}_{\textit{The function c calls itself twice}}$$

c(COMP251) = ?   c(MATH323) = ?

c(COMP251) = 3 credits + c(COMP250) #COMP250 is a prerequisite

- Substitute c(COMP251) into the formula:
- c(COMP462) = 3 credits + 3 credits + c(COMP250) + c(MATH323)
- c(COMP462) = 6 credits + c(COMP250) + c(MATH323)

Recursion - Example

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)

c(COMP250) = ?   c(MATH323) = ?

c(COMP250) = 3 credits # no prerequisite

c(COMP462) = 6 credits + 3 credits + c(MATH323)

c(MATH323) = ?

c(MATH323) = 3 credits + c(MATH141)

c(COMP462) = 9 credits + 3 credits + c(MATH141)

c(MATH141) = ?

c(MATH141) = 4 credits # no prerequisite

c(COMP462) = 12 credits + 4 credits = 16 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]`.

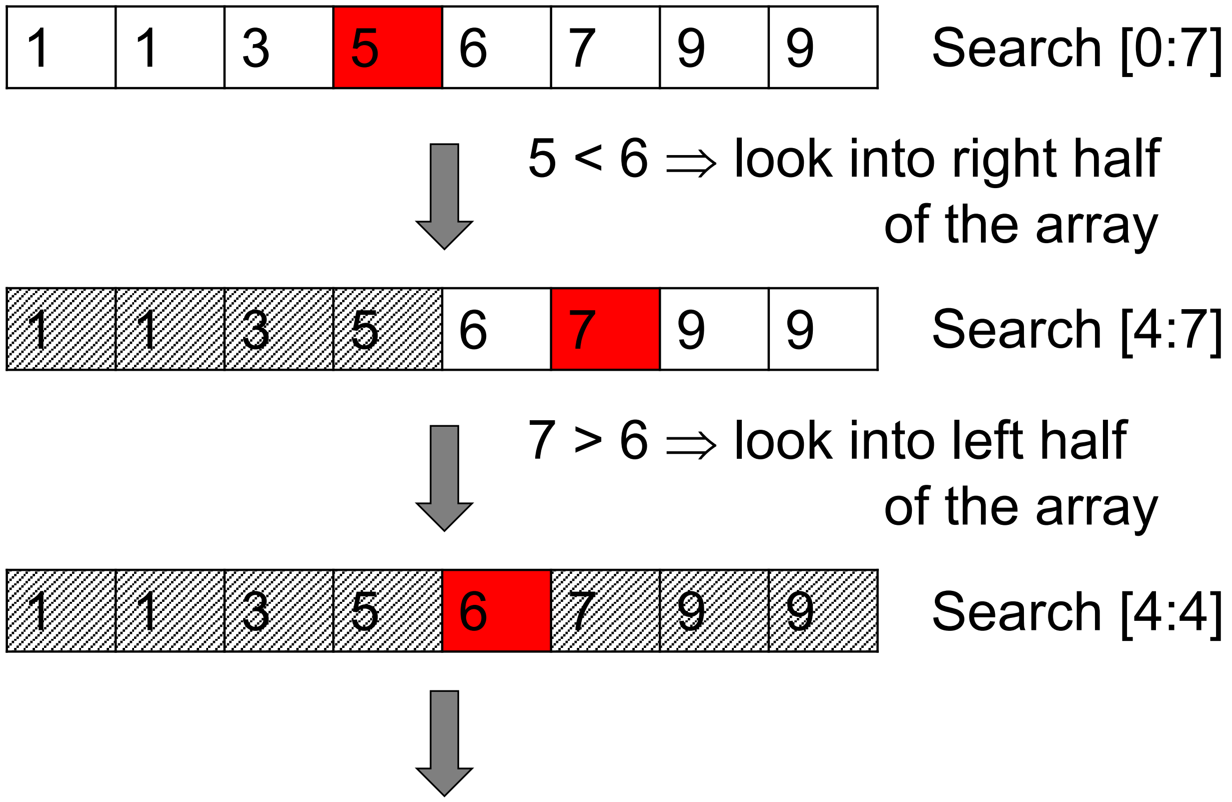
- **Example:** Does the following **sorted** array A contains the number 6?

A =

1	1	3	5	6	7	9	9

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

Recursion – Algorithm Example


1 1 3 5 6 7 9 9
Search [0:7]
5 < 6 ⇒ look into right half of the array
1 1 3 5 6 7 9 9
Search [4:7]
7 > 6 ⇒ look into left half of the array
1 1 3 5 6 7 9 9
Search [4:4]

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 = ⌊(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; } // value not found
}
```

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:

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

- 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 = ⌊(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; } // value not found
}
```

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

- *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

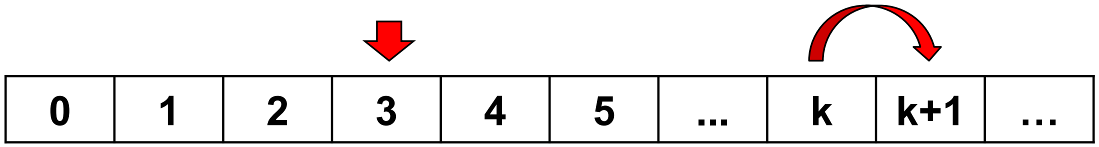
- **How to solve a recursive equation?**

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

Mathematical Induction

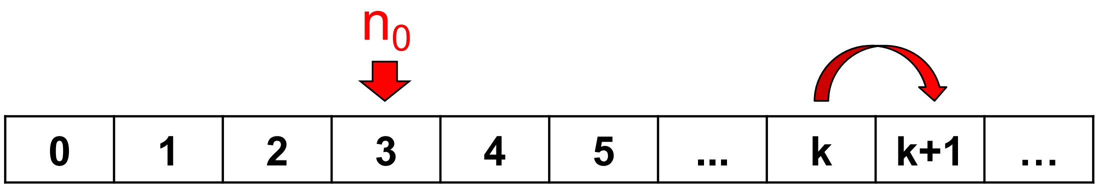
- Many statement of the form “*for all* $n \geq n_0$ *,* $P(n)$” can be proven with a logical argument call *mathematical induction.*

- The proof has two components:
- **Base case**: $P(n_0)$
- **Induction step**: for any $n \geq n_0$, if $P(n)$ then $P(n+1)$

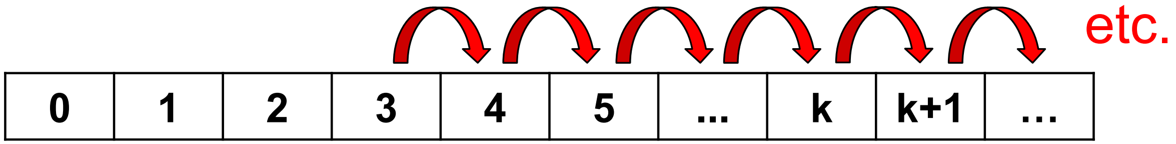


Mathematical Induction

- 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: $\text{for any } n \geq 1, \quad 1+2+3+4+\cdots+n = \dfrac{n\cdot(n+1)}{2}$

Proof:

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

- Induction step:

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

$$\text{then} \quad 1+2+3+4+\cdots+k+(k+1) = \frac{(k+1)\cdot(k+2)}{2}$$

Mathematical Induction – Example 1

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

$$\textit{then}\quad 1+2+3+4+\cdots+k+(k+1)$$

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

Induction hypothesis

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

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

Mathematical Induction – Example 2

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

Proof:

- Base case: $n=1 \qquad 1=1^2$ ✓

- Induction step:

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

Mathematical Induction – Example 2

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

$$\textit{then} \quad 1+3+5+7+\cdots+(2\cdot k-1)+(2\cdot(k+1)-1)$$

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

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

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

Induction hypothesis

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

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

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

Induction hypothesis can be anything < n

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

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.

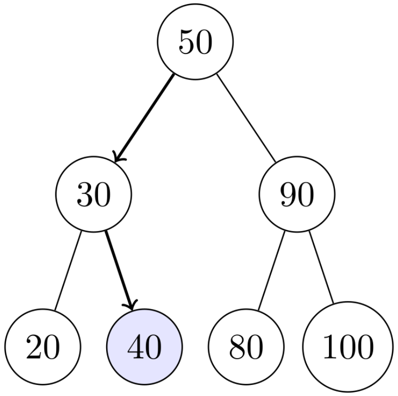
Running time – Binary Search

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

$\Rightarrow \Omega(1)$

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).

$\Rightarrow O(\log_2 n)$

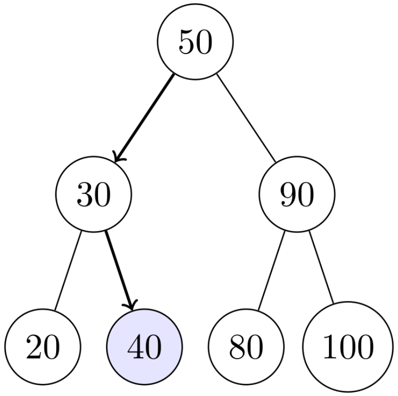


A tree representing binary search. The array being searched here is [20, 30, 40, 50, 80, 90, 100]
(from Wikipedia)

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
  - 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 ←1 T_{assign}
  - **Calling a method:** Expos.addWin() T_{call}
    - Note: doesn’t include the time to execute the method
  - **Returning from a method**: return x; T_{return}
  - **Arithmetic operations on primitive types** T_{arith}
    - x + y, r*3.1416, x/y, etc.
  - **Comparisons on primitive types:** x==y T_{comp}
  - **Conditionals:** if (...) then.. else... T_{cond}
  - **Indexing into an array:** A[i] T_{index}
  - **Following object reference:** Expos.losses 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 = ⌊(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; }

}
```

`if (i<=j) {`	$T_{cond} + T_{comp}$
`int e = ⌊(i+j)/2⌋;`	$T_{arith} + T_{assign}$
`if (A[e] > x) {`	$T_{cond} + T_{comp} + T_{index}$
`return bsearch(A,i,e-1,x);`	$T_{return} + T_{call}$
`} else if (A[e] < x) {`	$T_{cond} + T_{comp} + T_{index}$
`return bsearch(A,e+1,j,x);`	$T_{return} + T_{call}$
`} else {`	T_{cond}
`return e;`	T_{return}
`} else {`	T_{cond}
`return -1; }`	T_{return}

$\}$ repeated 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:

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

- We are only interested in the **number** of primitive operations performed
- Worst-case running time for binary search becomes:

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

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

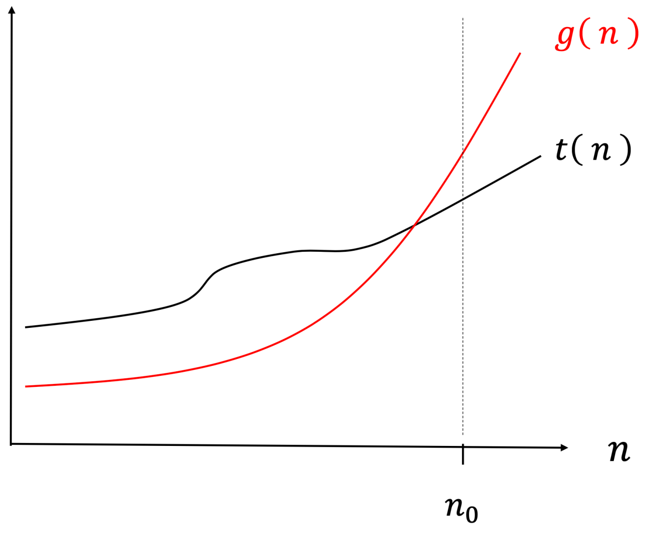
Towards a formal definition of big O

- 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 $\sqrt{n}, \log_2 n, n, n^2, 2^n$ ...

Towards a formal definition of big O

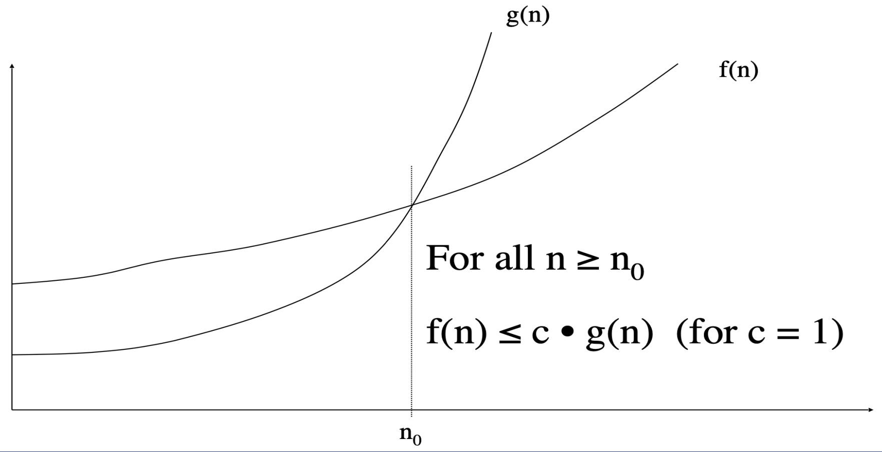
- Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. We say $t(n)$ is *asymptotically bounded above* by $g(n)$ if there exists n_0 such that, for all $n \geq n_0$,

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



Towards a formal definition of big O

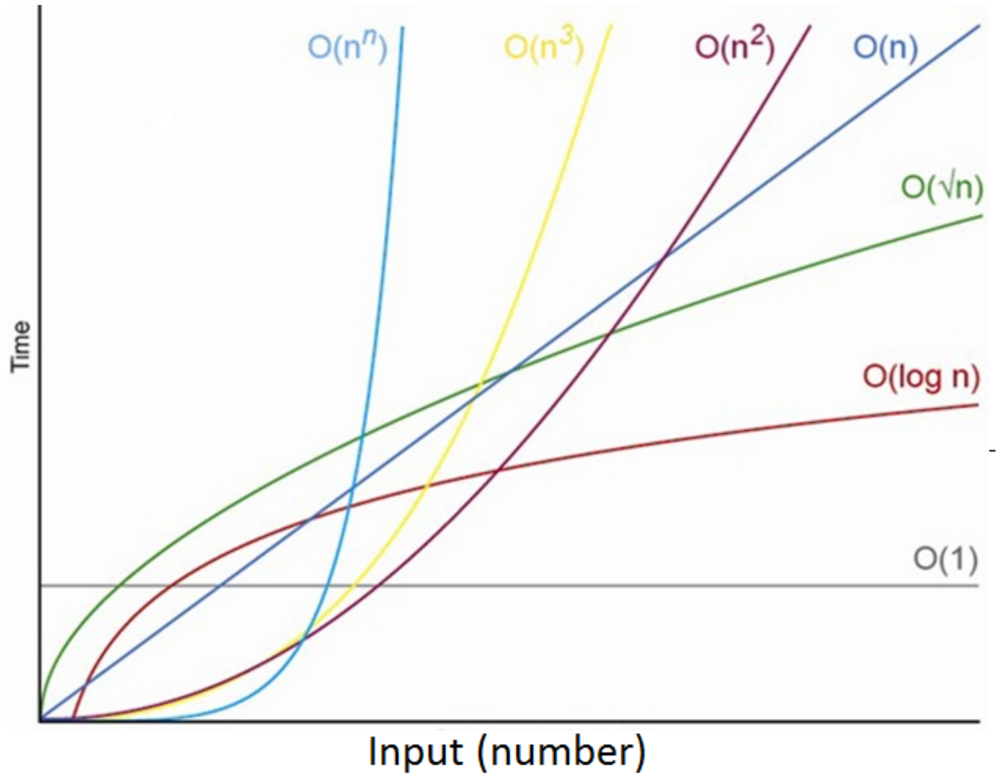
- Let $t\ n$ and $g\ n$ be two functions, where $n \geq 0$.
  We say $t(n)$ is $O(g(n))$ if there exists two positive constants n_0 and c such that, for all $n \geq n_0$,
  $$t(n) \leq c \cdot g(n)$$
- Intuition
  - "$f(n)\ is\ O(g(n))$" if and only if there exists a point n_0 beyond which $f(n)$ is 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.

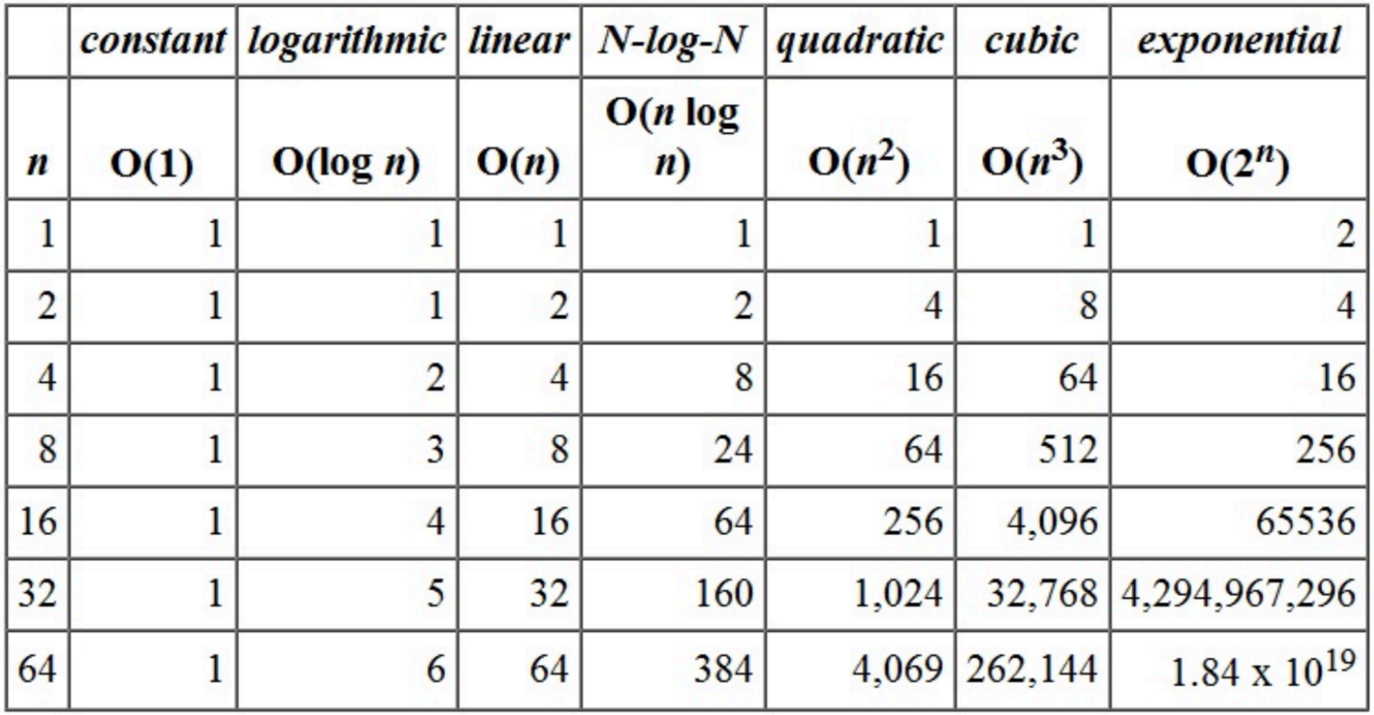
Growth of functions



(from stackoverflow)

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

Growth of functions



	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**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 ~10^{11} years…

Towards a formal definition of big Ω

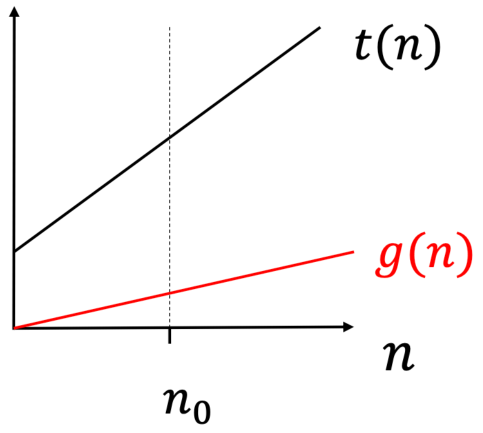
- Let $t(n)$ and $g(n)$ be two functions with $n \geq 0$.
- We say $t(n)$ is $\Omega(g(n))$, if there exists two positives constants n_0 and c such that, for all $n \geq n_0$,
- $t(n) \geq c \cdot g(n)$
- Note: This is the opposite of the big O notation. The function g is now used as a "*lower* bound”.

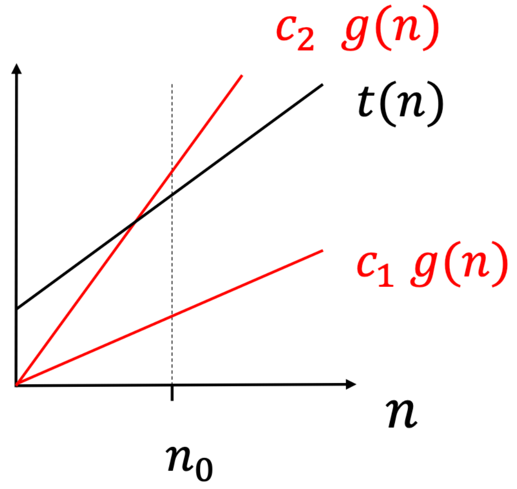
Towards a formal definition of big *Theta*

- 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 $n \geq n_0$,

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

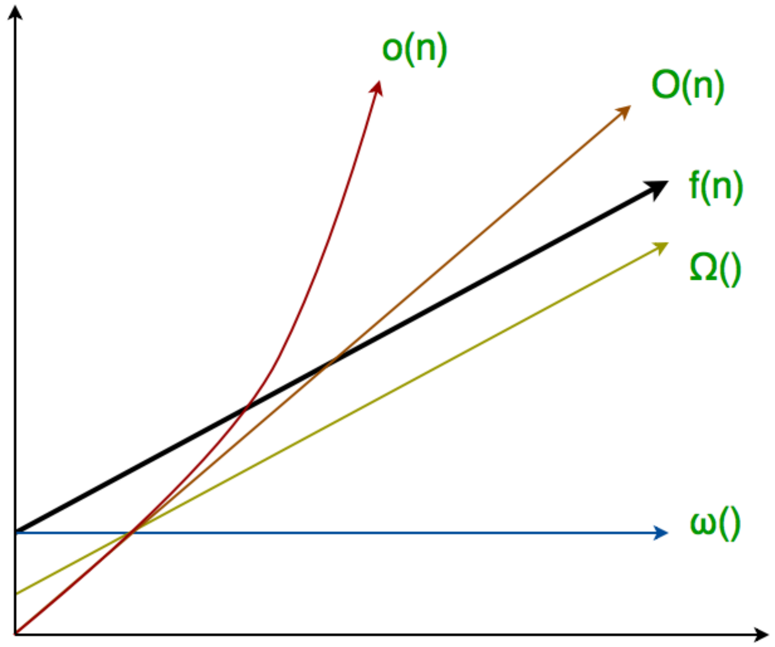




Note: if $t(n)$ is $\Theta(g(n))$. Then, it is also $O(g(n))$ and $\Omega(g(n))$.

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 lose upper (resp. lower) bounds.



(from 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.

time for questions