COMP20007 Workshop Week 10

	Counting & Radix sort: Questions 10.1-10.3 Heap: Questions 10.4-10.5
LAB	Revision : complexity, recurrences, master theorem or: previous weeks' lab

Counting Sort: Intro

Given:

```
    input array: {0,1,2,0,0,1,2,1,1,0,0,0}
```

How to sort the array efficiently?

Discuss limitations.

Counting Sort: Intro

`input array: {0,1,2,0,0,1,2,1,1,0,0,0}, k= 3`

→ The array can be sorted in **θ**(n+k) time, using **θ**(k) additional memory for frequencies.

But that not applied when each data is a records just like in real life, for example:

weight	**19**	**200**	**124**	**9**	**8**	**23**	**16**	**29**	**76**	**54**	**20**	**10**
color	green	black	red	grey	white	blue	violet	black	cyan	blue	green	white
key	0	1	2	0	0	1	2	1	1	0	0	0

normally `{0,1,2,0,0,1,2,1,1,0,0,0}` only represent keys, hiding other components of the data records. Now we can't overwrite the input array, you need to copy the elements to a new output array.

➔ Still **θ**(n+k) time, **θ**(k) memory for frequencies, but also need an array of size n for the output sorted array

➔ **θ**(n+k) memory, and the algorithm is *NOT-inplace*

Also note: Counting Sort can be implemented as *stable*

Counting Sort: how

Conditions: keys are integers in a small range (small in comparison with n), for example: array of positive integers, each ≤ 2:

`input array: {0,1,2,0,0,1,2,1,1,0,0,0}`

value	**0**	**1**	**2**
frequency table	6	4	2
end-index in the sorted array (easily transformed from freq table)	6 − 1 = **5**	5 + 4 = **9**	9 + 2 = **11**

Sorted array: `{0,0,0,0,0,0,` `1,1,1,1,` `2,2}`

keys 0 from index 5 backward

keys 1 from index 9 backward

keys 2 from index 11 backward

Note: to have stable sort, iterate the original from right to left when copying to sorted array

Counting Sort: summary

- Can be applied when A[i] in range min..max, where k= max-min+1 is small
- Time complexity: $\Theta(n+k)$ (or $\Theta(n)$ if k could be considered as a small constant)
- In-place: NO, using additional memory: $\Theta(n+k)$, (or $\Theta(n)$ if k small)
    - why additional memory?
        - $\Theta(k)$ for table of frequencies (and then that becomes the table of ending positions)
        - $\Theta(n)$ for building the output sorted array
- Stable: YES, but with careful implementation

Radix Sort

Applied when all keys can be represented as *same-size strings* over a small alphabet σ. Examples:

{1, 12, 7, 10, 6, 9, 8, 3}

→ {0001, 1100, 0111, 1010, 0110, 1001, 1000, 0011} σ= {0,1} using 4-digit binary

{1,22,17,167,26,19,28,173,…}

→ {001, 022, 017, 167, 026, 019, 028, 173,…} σ= {0,1,…,9} using 3-digit decimal

→ {01, 16, 11, A7, 1A, 13, 1C, AD…} σ= {0,1,…,9,A,B,..F} using 2-digit hexadecimal

Radix Sort:

From **rightmost to leftmost position** of strings: sort the string by that position by:

- using any *stable* sorting method, and
- counting sort is often the choice

Complexity for n strings of length k: ???

Q10.1 - Counting Sort: Use counting sort to sort the following array of characters:

[a, b, a, a, c, d, a, a, f, c, b]

How much space is required if the array has n characters and our alphabet has k possible letters.

Q10.2 - Radix Sort: Use radix sort to sort the following strings:

```
abc bab cba ccc bbb aac abb bac bcc cab aba
```

As a reminder radix sort works on strings of length k by doing k passes of some other (stable) sorting algorithm, each pass sorting by the next most significant element in the string. For example in this case you would first sort by the 3rd character, then the 2nd character and then the 1st character.

Q10.3: Which property is required to use counting sort to sort an array of tuples by only the first element, leaving the original order for tuples with the same first element. For example, the input may be:

(8, campbell), (6, tal), (3, keir), . . . (6, gus), (0, nick), (8, tom)

Discuss how you would ensure that counting sort satisfies this property. Can you achieve this using only arrays? How about using auxiliarry linked data structures?

Check your answer: Q10.2

Radix Sort: Use radix sort to sort the following strings:

abc bab cba ccc bbb aac abb bac bcc cab aba

First, sort (using *stable* counting sort) by the last letters:

ab**c** ba**b** cb**a** cc**c** bb**b** aa**c** ab**b** ba**c** bc**c** ca**b** ab**a**

→cb**a** ab**a** ba**b** bb**b** ab**b** ca**b** ab**c** cc**c** aa**c** ba**c** bc**c**

Next, do the same for the middle letter :

c**b**a a**b**a b**a**b b**b**b a**b**b c**a**b a**b**c c**c**c a**a**c b**a**c b**c**c

→bab cab aac bac cba aba bbb abb abc ccc bcc

Last, do the same for the first letter:

bab **c**ab **a**ac **b**ac **c**ba **a**ba **b**bb **a**bb **a**bc **c**cc **b**cc

→ aac aba abb abc bab bac bbb bcc cab cba ccc

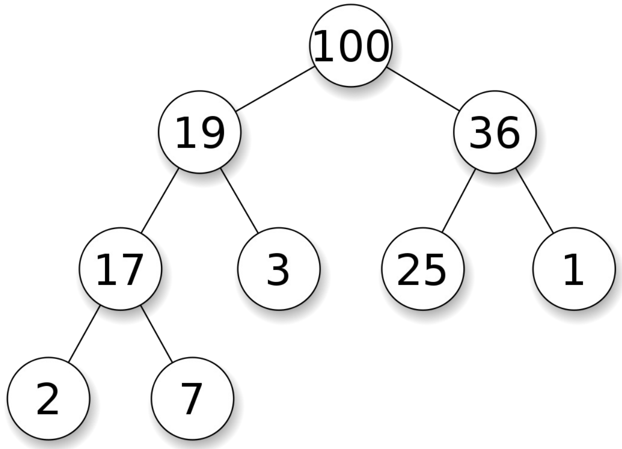
Note: the sorting method is required to be stable, why?

An Implementation for Priority Queue: Binary Heap

Binary Heap as a *concrete data type* (implementation) for PQ.

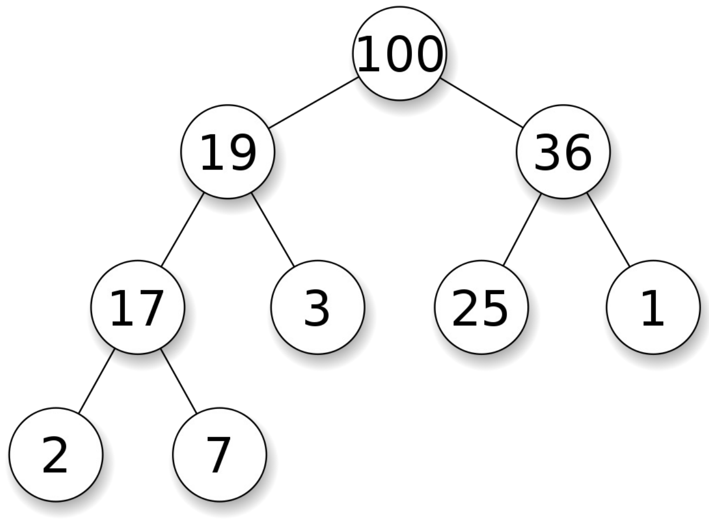
Simple priority: higher, lower ➔ max heap, min heap

What is a, say, max heap?
How is it implemented?



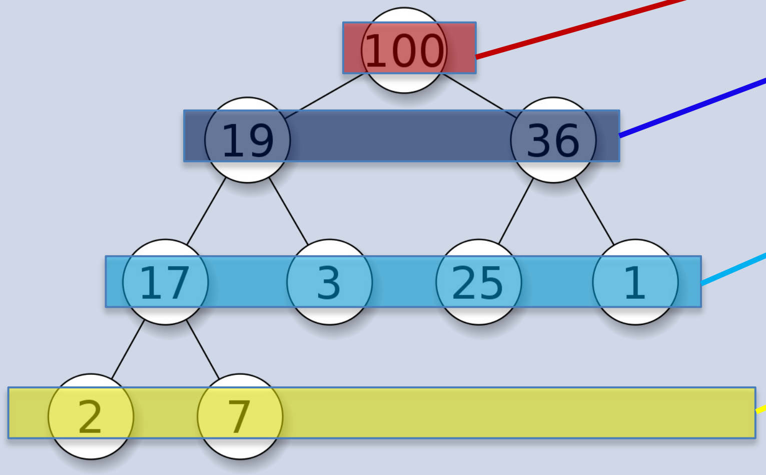
'Can I borrow your baby?...'

Binary Heap: conceptually, is a binary tree satisfying 2 conditions

Example	**Conditions**
	1. The tree is *complete*: • all levels, except for the last, are full • the last level is filled from left to right 2. The *heap property*: each node has a higher priority (here, is larger) than any of its descendants (or equivalently, just its children).

Binary Heap: is implemented as an array!

Visualisation: as a complete binary tree



Implementation: using arrays

idx	0	1	2	3	4	5	6	7	8	9
val	×	100	19	36	17	3	25	1	2	7

note: `H[1]` is for the root
`H[0]` not used (or used for a sentinel)

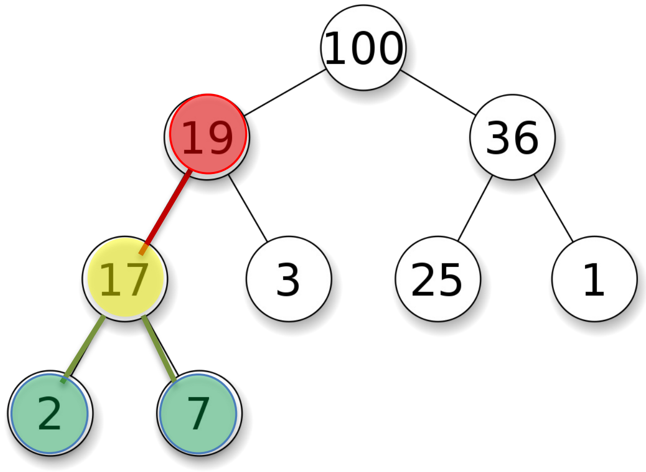
Heap is `H[1..n]`

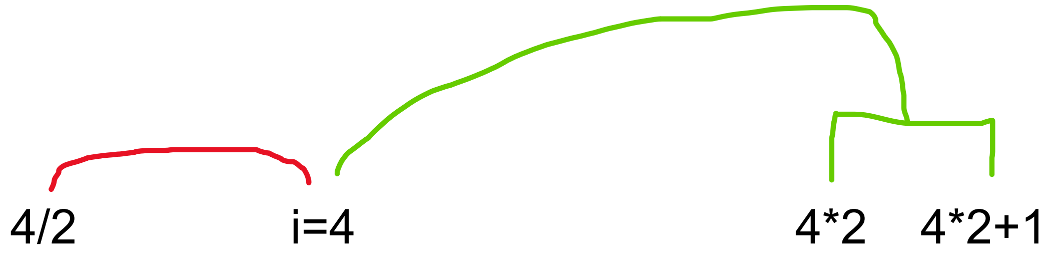
- level `i` occupies 2^i cells in array `H[1..n]` (except for the last level)
- if root is level 1, then level `i` starts from `H[`2^{i-1}`]`

Binary Heap: parent and children of a node

- left child of `H[i]` is `H[2*i]`
- right child of `H[i]` is `H[2*i+1]`

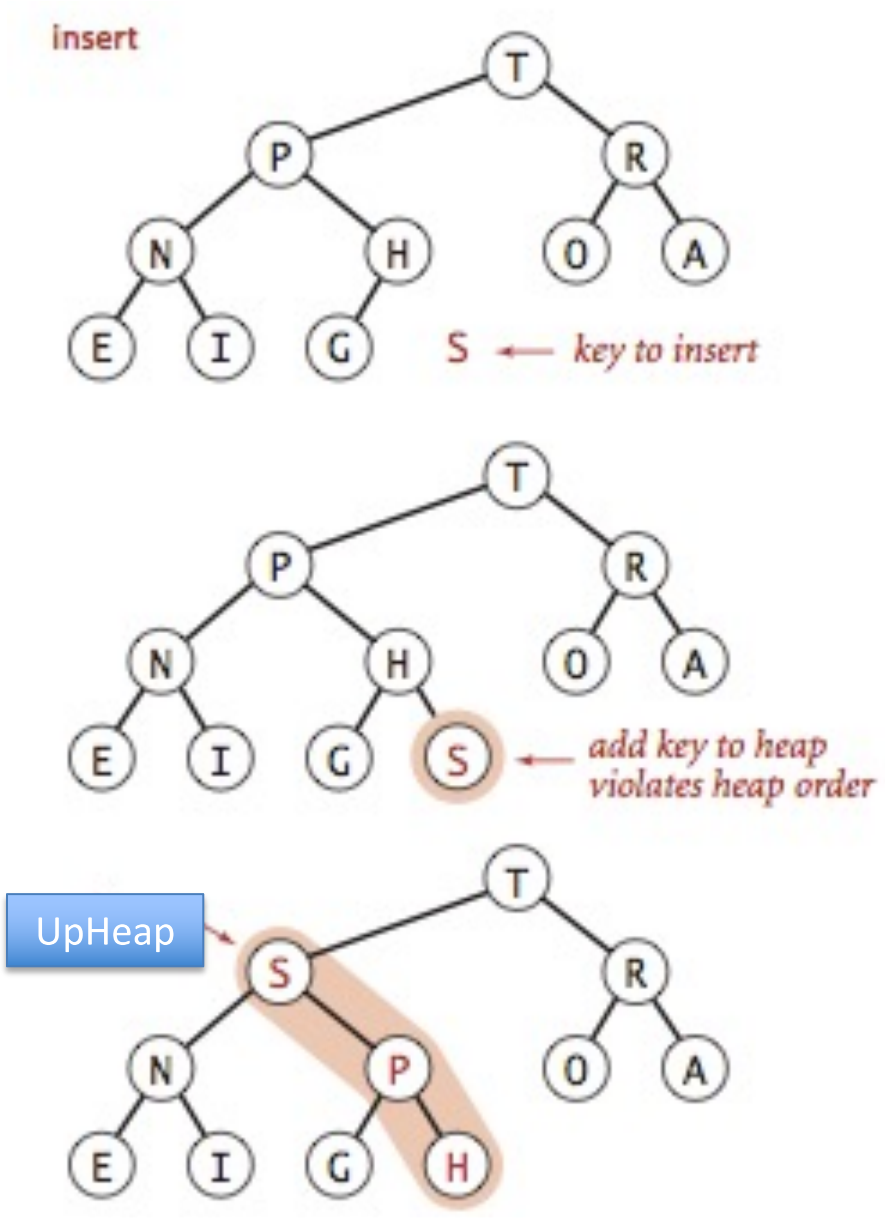
parent of `H[i]` is `H[i/2]`





idx	0	1	2	3	4	5	6	7	8	9
val	×	100	19	36	17	3	25	1	2	7

inject = enPQ = Insert a new elem into a heap using UpHeap



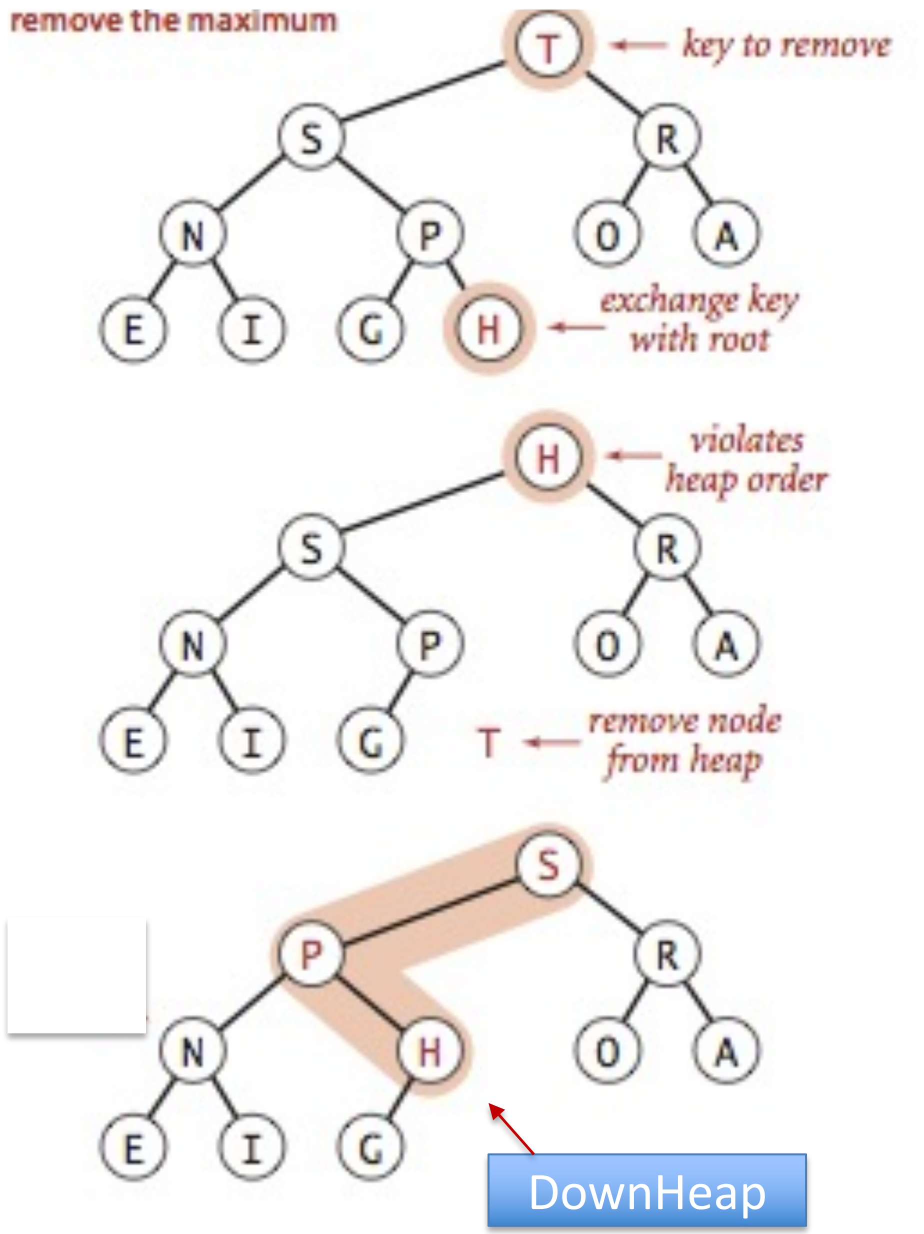
Notes:
Procedure:

Complexity:

UpHeap (aka. SiftUp)

while (has parent and parent has lower priority): swap up with the parent

eject = delete (and returns the heaviest) and repair using DownHeap



Notes:
Procedure:

Complexity:

`DownHeap (aka. SiftDown):`

while (has children and the heavier child has higher priority): swap down with the *heavier* child

Heapify: Turning an array H[1..n] into a heap

```
function Heapify(H[1..n])
  for i ← n/2 downto 1 do
     downheap(H, i)
```

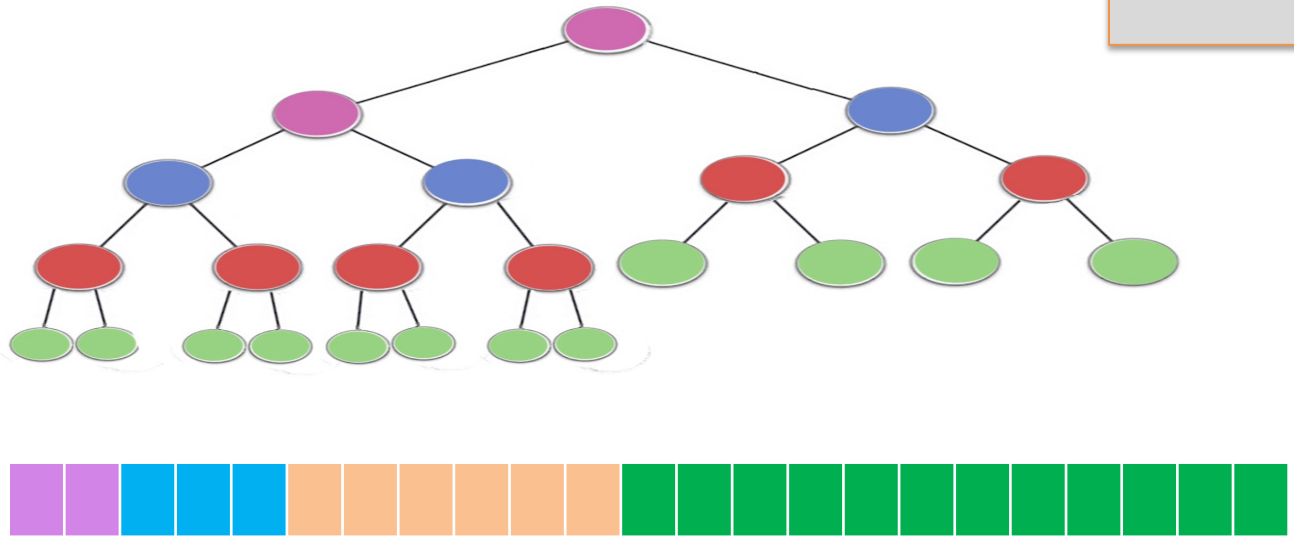
= **Θ(n)** (see lectures and/or ask Google for a proof)

The operation is aka. **Heapify**/Makeheap/ Bottom-Up Heap Construction

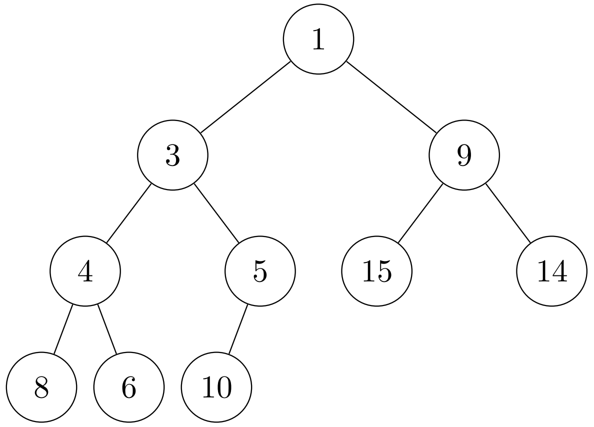
Example: build maxheap for keys E X A M P

Notes:

Complexity:



Q 10.4



a)Show how this heap would be stored in an array as discussed in lectures (root is at index `1`; node at index `i` has children at indices `2i` and `2i+1`)

b)Run the `RemoveRootFromHeap (eject)` algorithm from lectures on this heap by hand (i.e., swap the root and the "last"element and remove it. To maintain the heap property we then SiftDown from the root).

c)Run the InsertIntoHeap `(inject)` algorithm and insert the value 2 into the heap

Your answers:

a) array is: [???]

b) Run the RemoveRootFromHeap:

c) Run the InsertIntoHeap(2):

NOTES ON HEAPSORT:

Q 10.5 [opt]: k-smallest using min-heap

The *k-th smallest* problem:

- Given an array `A[]` of `n` elements, and an integer `k`
- Find the `k`-th smallest value (suppose that `k` is zero-origin, that is, `k` can be any of 0, 1, 2, ..., n-1)

How can we use a min-heap data structure to solve the kth-smallest element problem? What is the time-complexity of this algorithm?

Your answer:

Algorithm	Complexity
`function HeapkthSmallest(A[0..n−1],k)`	

Revision 1: Complexity Analysis – Lec W2, Workshop W3; W4- recurrences,

$1 \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n}$ where $0 < \varepsilon < 1 < c$

$O(\, f(n) + g(n)\,) = O(\, \max\{f(n), g(n)\}\,)$ note: these 3 also applied to big-$\boldsymbol{\Theta}$

$O(c\, f(n)) = O(\, f(n)\,)$

$O(\, f(n) \times g(n)\,) = O(f(n)) \times O(g(n))$

$1+2+ \ldots + n = n(n+1)/2 = \boldsymbol{\Theta}(n^2)$

$1^2 + 2^2 + \ldots + n^2 = n(n+1)(2n+1)/6 = \boldsymbol{\Theta}(n^3)$

$1 + x + x^2 + \ldots + x^n = (x^{n+1}-1)/(x-1) \quad (x \neq 1)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f(n) = O(g(n)) \\ c & f(n) = \boldsymbol{\Theta}(\,g(n)\,) \\ \infty & f(n) = \boldsymbol{\Omega}(g(n)) \end{cases}$$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Revision: Complexity – Recurrences : Q 10.6 and more

Q 10.6+: For each of the following cases, indicate whether *f(n)* is *O(g(n))*, or ***Ω****(g(n))*, or both (that is, ***Θ****(g(n))*)

(a) $f(n) = (n^3 + 1)^6$ and $g(n) = (n^6 + 1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

(c) $f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

(d) $f(n) = 2\log_2\{(n + 50)^5\}$ and $g(n) = (\log_e(n))^3$,

(e) $f(n) = (n^2 + 3)!$ and $g(n) = (2n + 3)!$,

(f) $f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

Q 10.6: Solve the following recurrence relations. Give both a closed form expression in terms of n and a Big-Theta bound.

a) $T(n)= T(n/2)+1,\ T(1)= 1$

b) $T(n)= T(n-1) + n/5,\ T(0)= 0$

Other exercises: review complexity exercises for Workshops Week 3, 4, 7; Levitin C2

Revision: Complexity – Recurrences : Q 10.6 and more

Q 10.6+: For each of the following cases, indicate whether *f(n)* is

- Choose the simplest logic:
  - drop free constants & lower members
  - use lim
- Normally, you should show your work-out
- Feel free to add other questions, including the ones in MST

(a) $f(n) = (n^3+1)^6$ and $g(n) = (n^6+1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

$f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

$f(n) = 2\log_2\{(n+50)^5\}$ and $g(n) = (\log_e(n))^3$,

$f(n) = (n^2+3)!$ and $g(n) = (2n+3)!$,

$f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

Q 10.6: Solve the following recurrence relations in terms of n and a Big-Theta bound.

a) $T(n) = T(n/2) + 1$, $T(1) = 1$

b) $T(n) = T(n-1) + n/5$, $T(0) = 0$

The Master Theorem is convenient but:

- constants a, b, d not always available
- not for finding closed form expression

Other exercises: review complexity exercises for Workshops Week 3, 4, 7; Levitin C2

Lab

- work other lab questions from earlier weeks, or
- continue with reviewing: complexity, recurrences, divide-and-conquer, DP

Some exercises from Levitin's C2, Skiena's C2

5. List the following functions according to their order of growth from the lowest to the highest:

$$(n-2)!,\quad 5\lg(n+100)^{10},\quad 2^{2n},\quad 0.001n^4+3n^3+1,\quad \ln^2 n,\quad \sqrt[3]{n},\quad 3^n.$$

2-26. *[5]* Place the following functions into increasing asymptotic order.
$f_1(n) = n^2\log_2 n$, $f_2(n) = n(\log_2 n)^2$, $f_3(n) = \sum_{i=0}^{n} 2^i$, $f_4(n) = \log_2(\sum_{i=0}^{n} 2^i)$.

2-31. *[5]* For each pair of expressions (A, B) below, indicate whether A is O, o, Ω, ω, or Θ of B. Note that zero, one or more of these relations may hold for a given pair; list all correct ones.

	A	B
(a)	n^{100}	2^n
(b)	$(\lg n)^{12}$	$\sqrt{n}$
(c)	$\sqrt{n}$	$n^{\cos(\pi n/8)}$
(d)	10^n	100^n
(e)	$n^{\lg n}$	$(\lg n)^n$
(f)	$\lg(n!)$	$n\lg n$

4. Consider the following recursive algorithm.

```
ALGORITHM  Q(n)
    //Input: A positive integer n
    if n = 1 return 1
    else return Q(n − 1) + 2 ∗ n − 1
```

a. Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.

b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.