Program Structure and Algorithms

Sid Nath

Lecture 2

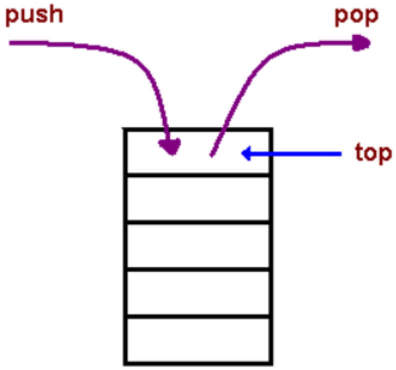
Agenda

- Administrative

- Lecture

Stacks

- A collection based on the principle of adding elements and retrieving them in the opposite order
  - Last-in, first-out (LIFO)
  - Elements are stored in the order of insertion
  - Caller can only add/remove/examine the topmost element
- Operations
  - Add/Push(item) O(1) w/o resizing; O(n) w/ resize
  - Pop() O(1)
  - Peek() // examine the top element without removing it O(1)
  - Size() O(1)
  - IsEmpty() O(1)
- Can be implemented with either array or linked lists

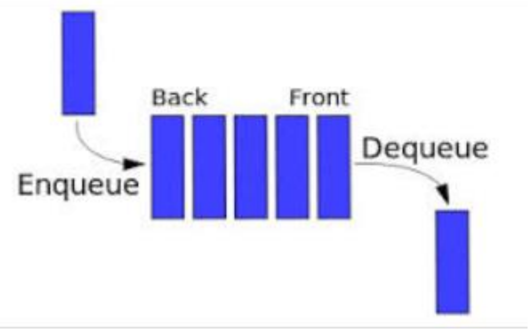


Stack Applications

- Depth first search in graphs, tree traversals (implicit program stack)
- Reversing a string
- Backtracking, undo functionality in editors, etc., application history

Queues

- Like a stack BUT retrieves elements in the order they are added
  – First in, first out (FIFO)
  – Caller adds to the end of the queue, examine/remove from front of the queue
- Operations
  – Add(item) O(n) w/ resize else O(1) (enqueue())
  – Remove() O(1) (dequeue())
  – Peek() O(1)
  – Size() O(1)
  – IsEmpty() O(1)



- Can be implemented with either array or linked lists

Queues Applications

- Task scheduling in OS
- Managing network packets
- Breadth first search in graphs, preorder traversal in trees (flattening a binary tree)

Order of growth

- Time required to solve a problem depends on the number of steps it uses
- Growth functions estimate the #steps an algorithm uses as its input grows
- Compare efficiencies of algorithms
  - Look only at the leading term of the formula for running time.
    - drop lower-order terms.
    - ignore the constant coefficient in the leading term.
- The worst-case running time
  - Meaning: It grows like n^2 does not equal n^2

How do we compare growth of functions?

- Formal definitions

$\boxed{O}$	$\approx$	$\leq$
Ω	$\approx$	$\geq$
Θ	$\approx$	$=$
o	$\approx$	$<$
ω	$\approx$	$>$

O-notations

- Characterizes an ***upper bound*** on the asympototic behavior of a function: function grows ***no faster*** than a certain rate based on the highest order term.
- **For example:**
  - $f(n) = 7n^3 + 100n^2 - 20n + 6$ is $O(n^3)$, since the highest order term is $7n^3$, and therefore the function grows no faster than n^3.
  - The function $f(n)$ is also $O(n^5)$, $O(n^6)$, and $O(n^c)$ for any constant $c \geq 3$.

Example

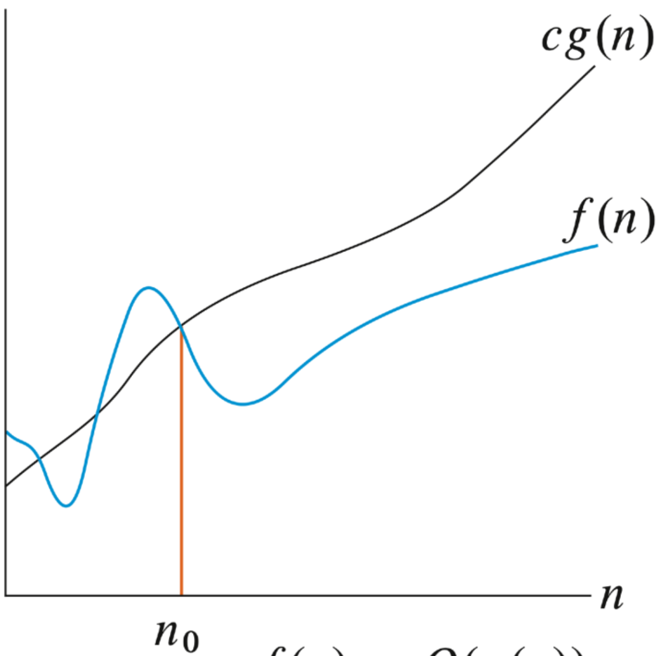
$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$

Example

$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$



$f(n) = O(g(n))$

Example

$$What\ is\ O(.)\ for\ f(n) = n^2 + 2n + 1?$$

$$Let\ g(n) = n^2$$

$$|f(n) \le c|n^2|\ \ \forall n > n_0$$

$$|n^2 + 2n + 1| \le c|n^2|$$

$$n^2 + 2n + 1 \le n^2 + 2n^2 + n^2\ \ \forall n > 1$$

$$n^2 + 2n + 1 \le 4\mathrm{n}^2\ \ \ \forall n > 1$$

$$Therefore, n_0 = 1\ and\ c = 4, or$$

$$f(n) grows\ as\ O(n^2)$$

Ω-notation

- Characterizes a ***lower bound*** on the asymptotic behavior of a function.
- **For example:**
  - $f(n) = 7n^3 + 100n^2 - 20n + 6$ is $\Omega(n^3)$, since the highest-order term, n^3, grows at least as fast as n^3.
  - The function $f(n)$ is also $\Omega(n^2)$, $\Omega(n)$ and $\Omega(nc)$ for any constant $c \leq 3$.

Example

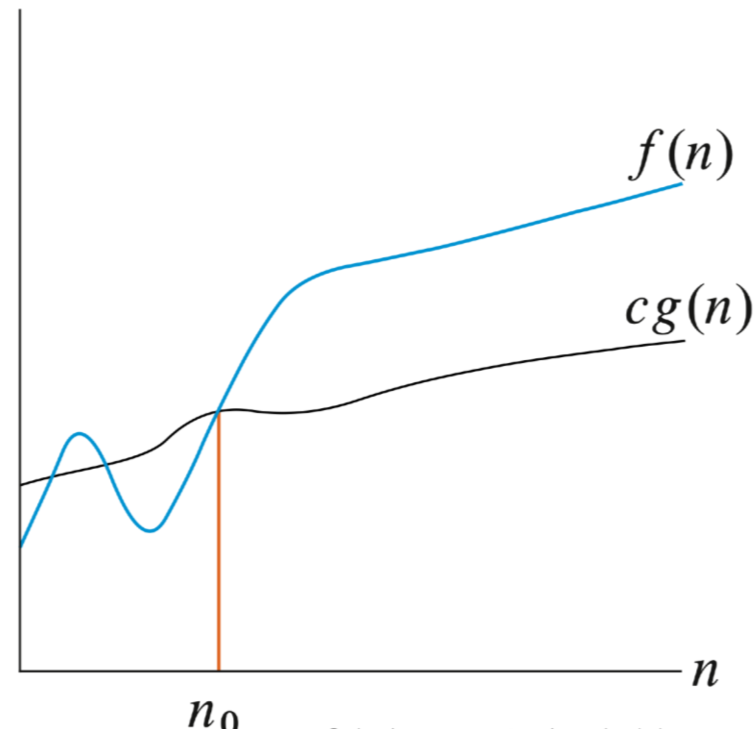
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

Example

$\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}



$f(n) = \Omega(g(n))$

Example

$$What\ is\ \Omega(.)\ for\ f(n) = \log(n^4) + 2^n?$$

$$Let\ g(n) = \log n$$
$$\log(n^4) + 2^n = 4\log n + 2^n$$
$$4\log n + 2^n \geq cg(n) \geq c\log n\ \ \forall n > 1$$
$$Therefore, n_0 = 1\ and\ c = 1, or$$
$$f(n)\ is\ lower - bounded\ as\ \Omega(\log n)$$

Θ-notation

- Characterizes a ***tight bound*** on the asympototic behavior of a function: function grows ***precisely*** at a certain rate, again based on the highest-order term.
- If a function is is both $O(f(n))$ and $\Omega(f(n))$, then a function is $\Theta(f(n))$.

Example

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$

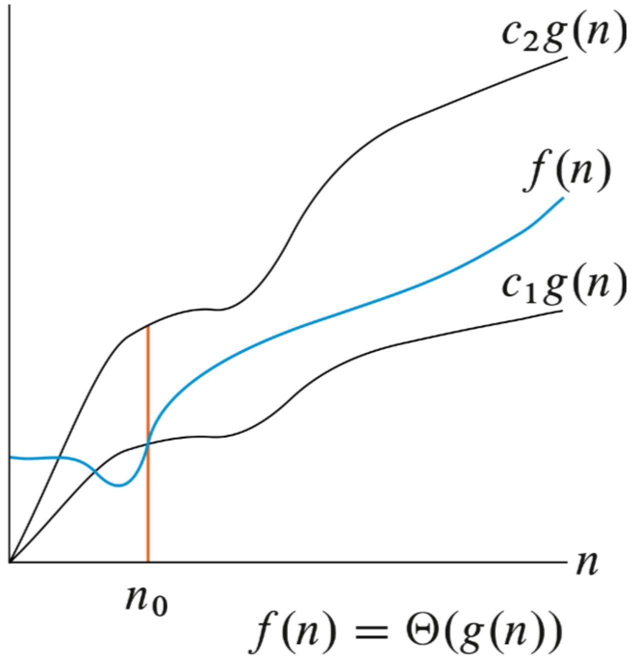
Example

$n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem

$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Leading constants and low-order terms don't matter.



Example

$$What\ is\ \Theta(.)\ for\ f(n) = n^2 + 5n\log n\ ?$$

$$Let\ g(n) = n^2$$

$$5n\ logn \le 5n^2\ \forall n > 1, c = 5$$

$$n^2 + 5n\log n \le 6n^2, \forall n > 1 \Rightarrow n_0 = 1; c_2 = 6$$

$$f(n)\ is\ upper - bounded\ by\ O(n^2)$$

Also, $n^2 + \ 5n\log n \ge n^2, \forall n > 1 \Rightarrow n_0 = 1; c_1 = 1$

$$f(n)\ is\ lower - bounded\ as\ \Omega(n\text{^}2)$$

$$Thus, f(n) \in \Theta(n^2)$$

The sorting problem

- **Input:** A sequence of *n* numbers $a_1, a_2, ..., a_n$
- **Output:** A permutation (reordering) $a_1', a_2', ..., a_n'$ of the input sequence such that

$$a_1' \leq a_2' \leq ... \leq a_n'$$

  - increasing order
  - each element: *key*

Sorting

- Foundational computation task widely used in all domains

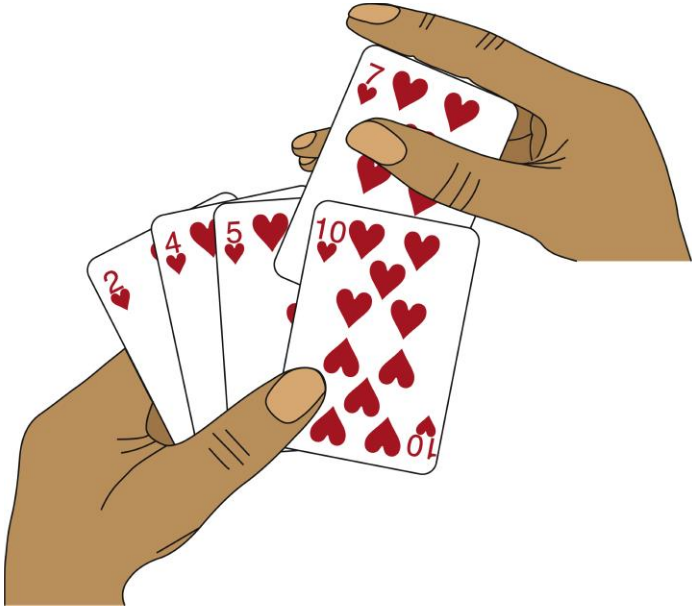
- Comparison-based sorting
  - Comparison operation determines which of two elements should occur first in the final sorted order
  - Examples: insertion sort, quicksort, heapsort, mergesort, selection sort, bubble sort, ….

- Non-comparison-based sorting
  - Integer sort, counting sort, radix sort

Insertion Sort

- Input: Array of size n, $A[1{:}\,n]$

- English description
  - assume A[1:j-1] are sorted
  - for A[j]: compared against A[1:j-1] & inserted in right place

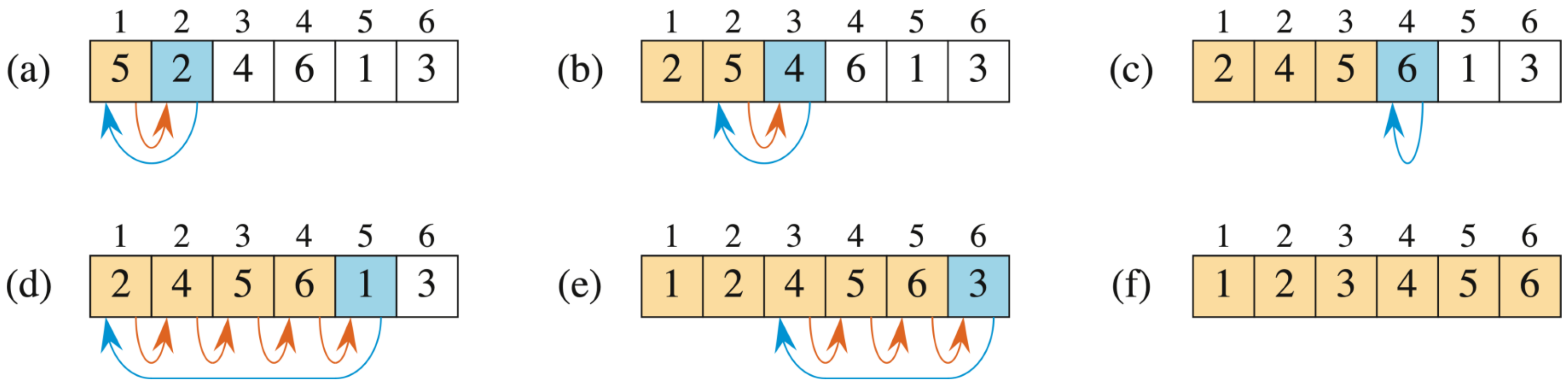


Express insertion sorting in pseudo code

```
INSERTION-SORT (A)
for j = 2 to A.length
        key = A[ j ]
        // Insert A[ j ] into the sorted sequence A[1 ..j -1]
          i  =  j -1
          while i >0 and A[i ] >key
               A[i +1] = A[i ]
                i =  i -1
          A[i +1]  = key
```

Example

- A = {5, 2, 4, 6, 1, 3 }
- |A| = n = 6



Algorithm analysis: running time (time complexity)

- Running time (time complexity) is measured by the number of an operation that take constant time such as
  - multiplication, division, comparison, ...

- Problem (input) size: = *n* in sorting problem
  - parameter that describes the size of input data.

- For the comparison-based sorting algorithms, the time complexity is measured by the number of elements in the input

Complexity (running time) analysis of insertion sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i - 1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\qquad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$\qquad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$\quad A[j+1] = key$	c_8	$n-1$

Complexity analysis-- insertion sort

- t_j # comparisons **while** loop executes in iteration j.

- $T(n)$ = running time of INSERTION-SORT = # *comparisons*

- t_j : varies depending on input
- Best case: $\theta(n)$ (array is sorted)
- Worst case: $\theta(n^2)$ (array is reverse sorted)
- Average case: $\theta(n^2)$

Sorting Algorithms

- Insertion sort: incremental
- Quick sort: recursive
- Non-comparison-based linear-time sorting
  - Counting sort
  - Radix sort

Lecture 2 summary

- Stacks, Queues
- Analysis of asymptotic running times
- Insertion sort – incremental
  - Analysis of running time