Computer science fundamentals

Miler T. Lee

16 February 2005
revised 20 February 2005, 26 May 2005

1 Programming languages

- all programming languages are **Turing complete**, meaning they can represent the same class of problems; instead, the differences arise from how things are implemented
- **strict operator**s: all operands must be defined; in C, && is not strict since you can evaluate the first argument as false and not need to check the second argument; any arithmetic operator would be strict
- **Garbage collection**: unused memory is freed and can be reallocated later. This is a manual process in C, C++, but is automated in Java and other languages. Strategies to do automated garbage collection include
  - **reference counting**, where a piece of memory has a counter that stores how many things point to it, reclaimed when the counter is 0
  - **mark and sweep**: follow all links and mark, sweep unmarked
- **Type unsafety**: type casts, pointer arithmetic, explicit C-style deallocation (creates dangling pointer); none of these are allowed in a type-safe language such as Java
- **Type checking**: strongly typed languages usually require both static and dynamic checks. Java is strongly typed. C is moderate-to-low typed. Perl throws caution to the wind.
  - run-time checks: e.g., Java type exceptions; high cost is a disadvantage
  - compile-time: catches errors earlier, faster at run-time; conservative
- **Polymorphism**: same code (e.g., a function) that can be used with different types
  - parametric: function may be applied to any arguments whose types match a type expression
  - explicit: C++ templates, at link-time, instantiates the function with appropriate type, so produces multiple copies and larger code, but will run faster
  - subtype: if parameter must be type A, can also accept a subtype (e.g., Java subclass) of A
- **Object-Oriented Programming**: objects are the basic units of abstraction and allow for the modularity inherent in OOP. The "noun" becomes the focus rather than the "verb" (as in an imperative language like C) - Square.area() rather than area(Square)
  - **polymorphism** achieved through dynamic lookup: method is selected at runtime according to the implementation of the object that receives the message. So a "Square" object and a "Circle" object could both respond to a method that's called "area()" but different versions would be called
  - **abstraction**, **encapsulation**: implementation can be hidden (e.g., Java private key word) without affecting functionality, thereby adding a layer of security (client can't change
  - **inheritance**: new classes defined from existing ones

2 Computational complexity

Growth of functions

- Big-Theta

$$\Theta(g(n)) = \Big\{ f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \Big\} \tag{1}$$

  - g(n) is a **tight asymptotic bound** for f(n)
  - E.g., $x^2 + 3x + 4 = \Theta(x^2)$, but $x^2 \neq \Theta(x^3)$ and $x^3 \neq \Theta(x^2)$

- Big-O

$$O(g(n)) = \Big\{ f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 \Big\} \tag{2}$$

  - g(n) is an **asymptotic upper bound** for f(n), not necessarily tight
  - E.g., $x^2 + 3x + 4 = O(x^2)$, and $x^2 = O(x^3)$ but $x^3 \neq O(x^2)$

- Little-O

$$o(g(n)) = \Big\{ f(n) : \forall c, n_0 > 0 \text{ such that } 0 \leq f(n) < c g(n) \text{ for all } n \geq n_0 \Big\} \tag{3}$$

  - g(n) is a **non-tight asymptotic upper bound** for f(n)
  - E.g., $x^2 = o(x^3)$ but $x^2 \neq o(x^2)$

- Big-Omega

$$\Omega(g(n)) = \Big\{ f(n) : \exists c, n_0 > 0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 \Big\} \tag{4}$$

  - g(n) is an **asymptotic lower bound** for f(n), not necessarily tight
  - E.g., $x^2 + 3x + 4 = \Omega(x^2)$, and $x^3 = \Omega(x^2)$ but $x^2 \neq \Omega(x^3)$

- Little-Omega

$$\omega(g(n)) = \Big\{ f(n) : \forall c, n_0 > 0 \text{ such that } 0 \leq c g(n) < f(n) \text{ for all } n \geq n_0 \Big\} \tag{5}$$

  - g(n) is a **non-tight asymptotic lower bound** for f(n)
  - E.g., $x^3 = \omega(x^2)$ but $x^2 \neq \omega(x^2)$

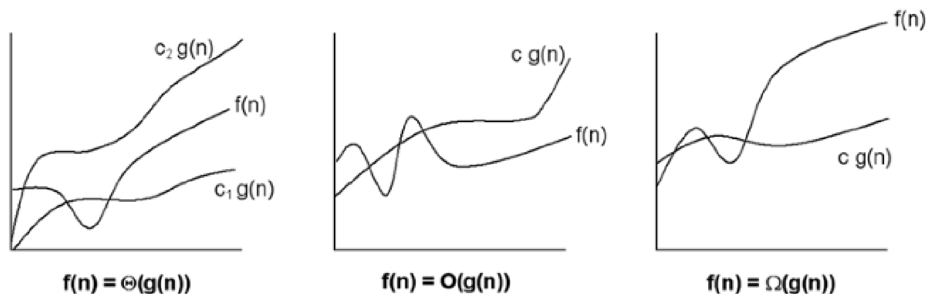

Figure 1: Growth of functions (adapted from CLRS)

1
$\log\log n$
$\log n$
$\sqrt{n}$
$\log^2 n$
$2^{\log n}$
n
$n\log n$, $\log(n!)$
n^2
$n^{\log n}$
2^n
$n!$
n^n
2^{2^n}

Table 1: Functions in increasing order

3 Algorithms

3.1 Strategies

- **Divide-and-conquer**: solve subproblems and put them together to solve the larger problem
- **Greedy algorithm**: obtain globally optimal solution for a problem by making the locally best choice at every step; problems that exhibit this structure are said to satisfy the **greedy-choice property**; top-down approach, **optimal substructure** of the problem is exploited, since the greedy steps lead to optimal solutions to subproblems, which are assumed to lead to an optimal solution to the global problem
  - Example: Greedy activity selector: given a set of activities with start/end time intervals, schedule the maximum number of mutually compatible events
    1. add earliest-finishing activity to set
    2. add next activity with start time after finish time
- **Dynamic programming**: optimal substructure, overlapping subproblems, but the choices at each step depend on possibly future solutions to subproblems; generally a bottom-up approach
- **Recursion / mathematical induction**: consists of a base case and a recursive (inductive) step
  - E.g., recursive definition: The set of natural numbers
    - i 0 is a natural number (base case)
    - ii if n is a natural number, then n+1 is a natural number
    - iii the set of natural numbers is the smallest set that satisfies these conditions
  - E.g., recursive function: Factorial

```
Factorial(n)
      if  n = 0 , return 1
      else return n * Factorial(n-1)
```

  - E.g., mathematical induction to prove a statement: $0+1+2+\ +n = n(n+1)/2$
    - i **base case**: the property holds for n = 0, $0(0+1)/2 = 0$
    - ii **inductive step**: assume that the property holds for $n = m$; we show it holds for $n = m+1$
      $0+1+2+\ +m = m(m+1)/2$ by the inductive hypothesis
      $0+1+2+\ +m+(m+1) = m(m+1)/2+m+1 = (m+1)(m+1+1)/2$
      thus, since the property holds for $m+1$, we conclude it holds for all n

3.2 Sorting

- **Bubble sort**: for each pair of elements, if in wrong order, swap them; iterate through the list repeatedly until no more swaps are done. O(n^2)
- **Insertion sort**: for each element i, place properly in order relative to 1...i-1. O(n^2)
- **Shell sort**: insertion sort over every ith element for decreasing i. Technically this is $\Theta(n^{1.5})$.
- **Selection sort**: iterate through original array, find smallest, insert in order into second array (or alternatively, swap it with the ith element from the beginning of the list). O(n^2)
- **Merge sort**: divide-and-conquer - split your list in two, each sublist into two, etc, until each list has only one member; merge lists together into sorted order in worst case O(n) time; there will be O(lg n) merge operations. Hence, O(nlgn) total time.
- **Heap sort**: relies on heap data structure, which can be visualized as a binary tree (or an equivalent array), with the property that each child stores a value less than or equal to its parent node; this is the max-heap property, A[parent(i)] $\geq$ A[i]
  - **Heapify**: for a given node i, assume that the left and right subtrees satisfy the max-heap property; A[i] may be smaller than its children (a violation of the max-heap property), so float the node down until the heap property is satified; O(lg n)
  - **Build-heap**: runs heapify bottom-up on all interior nodes; O(n) * O(lg n)
  - **Heapsort**: call build-heap; the root is the largest value, so remove it and put it in your sorted array; this leaves a "hole" in your tree at the root position, so fill it with a leaf node; now the heap is one smaller and the heap property is violated, so call heapify; repeat until the heap is empty and your sorted array is full; O(n lg n)
- **Quicksort**: divide-and-conquer approach - pick an element to be the pivot and divide the array into two subarrays such that each element in the left subarray is less than or equal to the pivot and each element in the right subarray is greater than or equal to the pivot. Put the pivot between the two subarrays. Recursively run quicksort on the two subarrays.

  Implementation:

  i choose the rightmost element as the pivot

  ii use two pointers i, j; repeat (O(n) per iteration):

  move j to the left until it hits an element $\geq$ pivot
  move i to the left until it hits an element $\leq$ pivot
  if i ¡ j, swap elements

  iii exchange the left most element in the right subarray with the pivot

- **Counting sort**: stable, linear time; assumes each element is an integer

  i create a count array s.t. each index is how many elements have value $\leq$ index (O(n))

  ii iterate backward in original array, populate a new array:

  place each element in the index corresponding to the count indicated by the count array
  decrement the count in the count array at that index

- **Radix sort**: successively sort (using a stable sort) based on each digit from least significant to most; if you assume the number of digits is a constant, this ends up being linear as well.
- **Bucket sort**: designed to be linear for input drawn from a uniform distribution over [0, 1); create n buckets, distribute the elements, sort the buckets, concat the buckets

- **Linear search**: search one by one until you find what you're looking for
- **Binary search**: depends on having a sorted array
  compare your target value with the middle element in the array
  - if equal, you're done
  - if less than, do binary search on left subarray
  - if greater than, do binary search on right subarray

	Worst case	Best case	notes
Bubble sort	O(n^2)	O(n)	in-place
Insertion sort	O(n^2)	O(n)	in-place
Selection sort	O(n^2)	O(n^2)	second array is populated in-order
Merge sort	O($n \lg n$)	O($n \lg n$)	
Heapsort	O($n \lg n$)	O($n \lg n$)	
Quicksort	O(n^2)	O($n \lg n$)	in-place
Counting sort	O($n + k$)	O($n + k$)	stable, k = size of max element
Radix sort	O($d(n + k)$)	O($d(n + k)$)	d = number of digits; using counting sort
Bucket sort	O(n^2)	O(n)	linear on average
Shell sort	O($n^{1.5}$)		empirical bound; "fast" quadratic on average; unstable
Linear search	O(n)	O(1)	
Binary search	O($\lg n$)	O(1)	

Table 2: Summary of runtime properties of sort/search algorithms

4 Data structures

- **Stack**: dynamic set observing a last-in, first-out (LIFO) policy; operations:
  - **push** inserts an element to the end (top) of the list
  - **pop** removes an element from the end of the list
- **Queue**: dynamic set observing a first-in, first-out (FIFO) policy; operations:
  - **enqueue** inserts an element to the end (tail) of the list
  - **dequeue** removes an element from the beginning (head) of the list
- **Linked list:** a linear-ordered dynamic list where each element also stores a pointer to at most one successor element
  - can be circular, such that the last element points to the first element
  - can be doubly-linked, such that each element also stores a pointer to the predecessor element
  - insertion and removal of an element is efficient compared to an array
- **Priority queue**: a sorted queue based on a priority criterion for the elements, such that the element with the highest priority is dequeued first
  - common implementation is based on the heap data structure
  - Operations:
    * **Extract-max**: identical to one iteration of Heapsort
    * **Heap-insert**: insert new node at leaf and float upward

- **Rooted tree**: trees have a **depth** (the maximum number of nodes on the path from the root to a leaf) and a **branching factor**(the maximum number of children that any node has); a tree with a fixed branching factor of 2 is a binary tree, 3 ternary, etc.
  - There are $\frac{1}{n+1}\binom{2n}{n}$ different binary trees with n nodes
  - Representing an n-ary tree: each element stores pointers to all children; this is inefficient as n becomes large
  - Representing a tree with arbitrary numbers of children using a binary tree (O(n) space)): each element points to its leftmost child and its sibling immediately to the right - i.e., every element needs only store two pointers (left-child, right-sibling representation)
  - the nodes in a binary tree can be returned in a particular order
    - **infix order** returns the nodes in a left subtree followed by the root followed by the right
    - **prefix order** returns the root first, then the left, then the right subtrees
    - **postfix order** returns the left subtree, then the right, then the root
- Searching through a tree can be done using two general strategies
  - **Breadth-first search**: searches all nodes at a given depth before searching nodes further down
    - incurs a very high storage cost, which at any point will be O(branching factor$^{\text{depth}}$)
    - can be implemented using a queue: enqueue the root into the queue; for every element dequeued, if it is not the target element, enqueue all of its children into the queue
  - **Depth-first search**: traverses to the bottom of the tree to start searching, systematically searches children before parents
    - storage cost at any point is O(branching factor * depth)
    - with infinite-depth trees, it is therefore possible to get stuck in a particular branch and never terminate the algorithm - this means that DFS is neither optimal (guaranteed to find the best solution) or complete (if there's a solution, it will find it)
    - DFS can be depth-limited, for example, to overcome this problem
    - can be implemented using a stack: push the root into the stack; for every element popped, if it is not the target element, push all of its children onto the stack
- **Binary search trees**
  - all nodes in left subtree of a subroot are $\leq$ subroot
  - all nodes in right subtree of a subroot are $\geq$ subroot
  - Successor of a node, O(depth):
    - case 1: if there is a right subtree, successor is leftmost node in right subtree
    - case 2: successor is the parent if node is left child of parent
    - case 3: successor is the parent of lowest ancestor that is left child of its parent otherwise
  - Insert: begin at root, trace downward until empty slot is found
  - Delete:
    - case 1: node to delete is childless, make parent have NIL child
    - case 2: single child, splice together parent and child of deleted node
    - case 3: two children, splice out node's successor (which has no left child) and replace the deleted node with the successor node
- Other trees
  - **Red-black trees**: balanced binary tree with height at most $2\log(n+1)$
  - **B trees**: r-b trees with unlimited branching factor

- **AVL tree**: balanced binary search tree, guarantee O(lg n) search time
- **Binomial trees**: 2k nodes, height is k, root has degree k, $\binom{k}{i}$ nodes at depth i

- **Hash tables**: storage of elements based on keys; extracting an element using its key is an idealized O(1) operation
  - each element in the universe needs to be mapped onto a key - elements may share a key, but no element may have more than one key; this entails the use of a **hash function** based on a randomized algorithm to ensure no bias in key selection, thus keeping the average number of elements with a particular key uniformly low (load factor)
  - The most common implementation is to use **chaining** to resolve multiple elements per key
    - create an array with one slot per key
    - for each element that hashes to a particular key, append the element to a linked list associated with the key
    - returning a target element with a particular key entails searching the appropriate linked list, which is a constant-time operation if there is an appropriate load factor