Algorithms and NP

Discrete Structures (CS 173) Fall 2016 Lecture B

Gul Agha

Slides based on Derek Hoiem, University of Illinois

This class

- Running time of algorithm continued
  - Towers of Hanoi
  - Computing factorial series
  - Multiplying large numbers
- The master theorem
- Algorithmic complexity
- P vs. NP

Selecting the base case

- *Algebraic proofs:* usually the smallest value(s)
  - Example: Prove $S_n = \sum_{i=1}^{n} i = n(n+1)/2$ for any positive integer n. *Base*: n=1.
- *Puzzles:* often the number of game pieces or the initial configuration
  - Example: The tower of Hanoi puzzle can be solved for any number of disks. *Base*: it works for one and two disks.
- *Graphs:* a single node graph or an empty graph
  - Example: Any fully connected graph with n nodes has $n(n-1)/2$ edges. *Base*: A graph with one node has 0 edges.
- *Trees:* a tree of height 0
  - Example: Any full and complete binary tree has $2^{h+1} - 1$ nodes, where h is the height of the tree. *Base:* A tree of height 0 has one node.

Induction strategy

- *Algebraic proofs*: rewrite the equation for $n = k + 1$ in terms of the equation for $n = k$.
- *Puzzles*: figure out how solving the puzzle for $k + 1$ pieces involves solving the puzzle for k pieces.
- *Graphs*: Start with a graph with $k + 1$ nodes. Apply inductive hypothesis to a graph with one node removed, and note the difference caused by removing the node.
- *Trees*: Start with a tree of height $k + 1$. Use the fact that the root's subtrees have height of k or less to show that the claim holds for the full tree.

Legend

“An Indian temple in Kashi Vishwanath contains three time-worn posts with 64 golden disks that once were in order of size on a single post. Brahmins have been moving these disks since the beginning of time following an immutable law of Brahma: no larger disk may be placed on a smaller disk. When the last move of the puzzle will be completed, the universe will end.”

The Towers of Hanoi (also called Towers of Brahma) puzzle was invented by the French mathematician Édouard Lucas in 1883. (version from Wikipedia)

Towers of Hanoi

Algorithms

- Practice analyzing runtime based on pseudocode

```
closestpair(p_1, ..., p_n) : array of 2D points)
        best1 = p_1
        best2 = p_2
        bestdist = dist(p_1,p_2)
        for i = 1 to n
                for j = 1 to n
                        newdist = dist(p_i,p_j)
                        if (i ≠ j and newdist < bestdist)
                                best1 = p_i
                                best2 = p_j
                                bestdist = newdist
        return (best1, best2)
```

Key concept: instructions in the loops dominate

```
procedure bubbleSort(A : list of sortable items)
  repeat
    swapped = false
      for i = 1 to length(A) - 1 inclusive do:
        /* if this pair is out of order */
        if A[i-1] > A[i] then
          /* swap them and remember something changed */
          swap( A[i-1], A[i] )
          swapped = true
        end if
      end for
   until not swapped
end procedure
```

Key concept: if the number of iterations of the loop is uncertain, determine the worst case

Note: can also do best-case or average-case analysis

http://www.youtube.com/watch?v=lyZQPjUT5B4

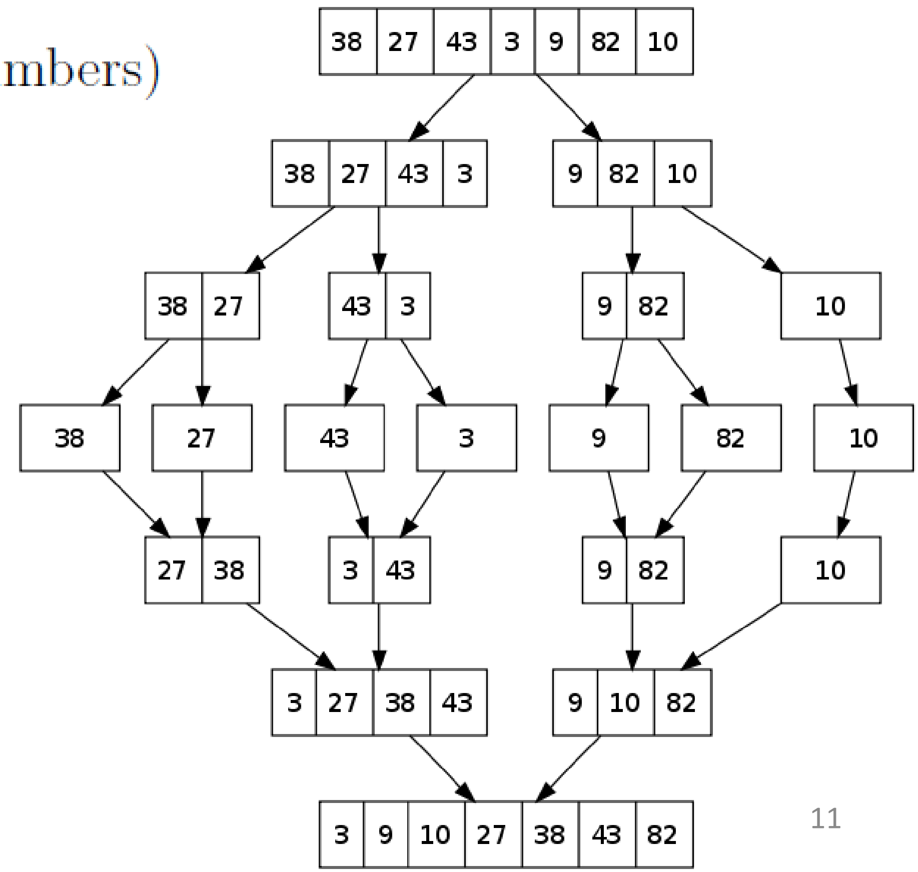
```
merge(L1,L2: sorted lists of real numbers)
        O = emptylist
        while (L1 is not empty or L2 is not empty)
                if (L1 is empty)
                        move head(L2) to the tail of O
                else if (L2 is empty)
                        move head(L1) to the tail of O
                else if (head(L1) <= head(L2))
                        move head(L1) to the tail of O
                else move head(L2) to the tail of O
        return O
```

Key concept: runtime can be in terms of multiple input parameters

```
mergesort(L = a_1, a_2, ..., a_n: list of real numbers)
    if (n = 1) then return L
    else
        m = ⌊n/2⌋
        L_1 = (a_1, a_2, ..., a_m)
        L_2 = (a_{m+1}, a_{m+2}, ..., a_n)
        return merge(mergesort(L_1),mergesort(L_2))
```

merge(L_1,L_2: sorted lists of real numbers)

Key concept: n/2 recursion



Comparison of sorting algorithms

http://www.sorting-algorithms.com/random-initial-order

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends	Partitioning	Quicksort is usually done in place with O(log(*n*)) stack space.[citation needed] Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(*n*) space array to store the partition.[citation needed]
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends; worst case is n	Yes	Merging	Highly parallelizable (up to O(log(*n*)) using the Three Hungarian's Algorithm or more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place Merge sort	—	—	$n(\log n)^2$	1	Yes	Merging	Implemented in Standard Template Library (STL);[2] can be implemented as a stable sort based on stable in-place merging.[3]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(*n* + *d*), where *d* is the number of inversions
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists.[4] Used to sort this table in Safari or other Webkit web browser.[5]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	n comparisons when the data is already sorted or reverse sorted.
Shell sort	n	$n(\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes	Insertion	When using a self-balancing binary search tree
Cycle sort	—	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes
Library sort	—	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	—	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences within O(*n* log *n*)

Key concept: Be aware of best, worse, and average case hanoi(A,B,C: pegs, $d_1, d_2 \ldots d_n$: disks)

    if ($n = 1$) move $d_1 = d_n$ from A to B.

    else

        hanoi(A,C,B,$d_1, d_2, \ldots d_{n-1}$)

        move d_n from A to B.

        hanoi(C,B,A,$d_1, d_2, \ldots d_{n-1}$)

http://introcs.cs.princeton.edu/java/23recursion/hanoi/

Example: Fibonacci

Lesson 1: Be careful of implementation details that subtly affect computational complexity

Lesson 2: Knowing complexity of algorithm can help find major implementation flaws

See code

Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b \geq 1$$

Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b \geq 1$$

Leaf term dominates (*hyper expansion*)

If $f(n) = \Theta(n^c)$ with $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$

Each level costs the same (*balanced expansion*)

If $f(n) = \Theta\left(n^c \log^k n\right)$ with $\mathrm{k} \geq 0$, $c = \log_b a$, then $T(n) = \Theta(n^c \log^{\mathrm{k}+1} n\ )$

Top node dominates (*slow expansion*)

If $f(n) = \Theta(n^c)$ with $c > \log_b a$, then $T(n) = \Theta(n^c)$

Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b \geq 1$$

If $f(n) = \Theta(n^c)$ with $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$

Example: $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$

Example algorithm: multiplying large numbers

Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b \geq 1$$

If $f(n) = \Theta\left(n^c \log^k n\right)$ with $\mathrm{k} \geq 0,\ c = \log_b a,$ then $T(n) = \Theta(n^c \log^{\mathrm{k}+1} n\ )$

Example: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Example algorithm: sorting

Master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b \geq 1$$

If $f(n) = \Theta(n^c)$ with $c > \log_b a$, then $T(n) = \Theta(n^c)$

Example: $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

Example: multiplying large numbers

Multiplying small numbers in binary

$$\begin{array}{r} 101 \\ \times\, 011 \end{array}$$

Complexity:

Multiplying large numbers

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$
$$xy = (x_1 2^m + x_0)(y_1 2^m + y_0)$$
$$= x_1 y_1 2^{2m} + (x_0 y_1 + y_0 x_1) 2^m + x_0 y_0$$

Complexity:

Example: multiplying large numbers

Multiplying large numbers

$$x = x_1 2^m + x_0$$
$$y = y_1 2^m + y_0$$
$$xy = (x_1 2^m + x_0)(y_1 2^m + y_0)$$
$$= x_1 y_1 2^{2m} + (x_0 y_1 + y_0 x_1) 2^m + x_0 y_0$$

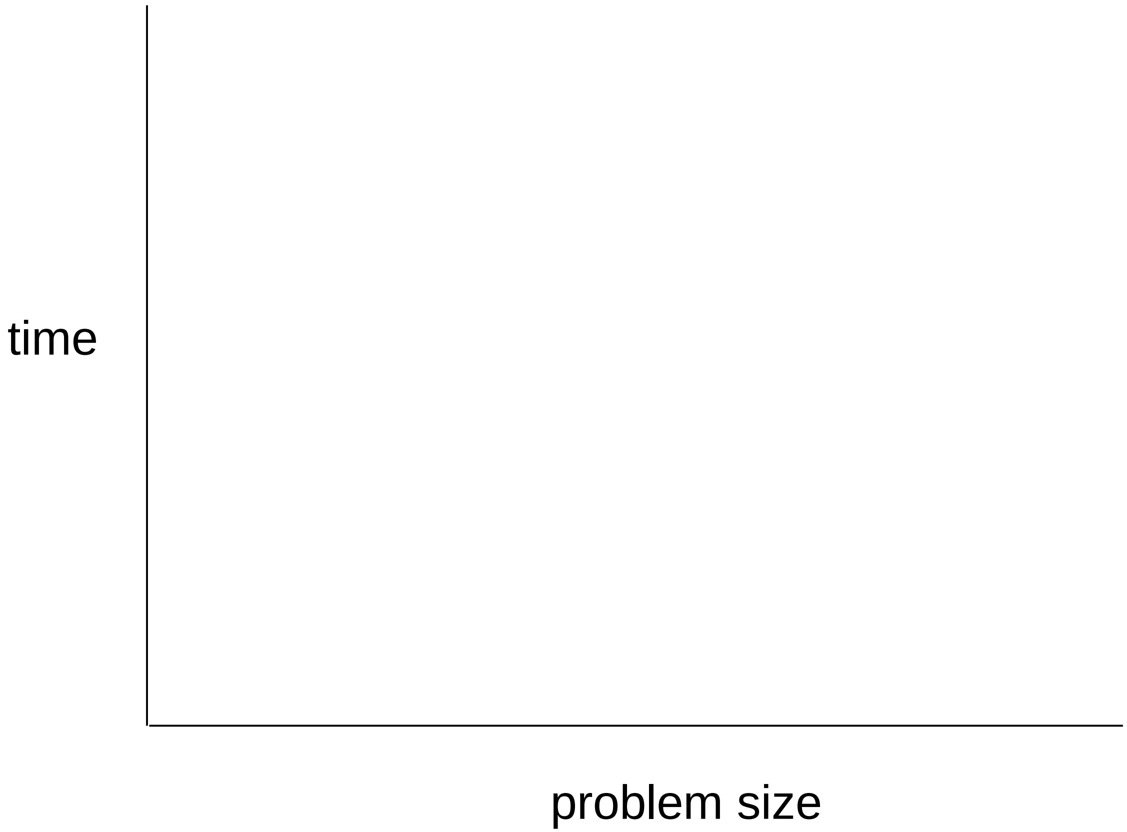
Trick by Anatolii Karatsuba

$$(x_0 y_1 + y_0 x_1) = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

Complexity:

Algorithm complexity

constant, sublinear, linear, linearithmic, quadratic, cubic, exponential, factorial

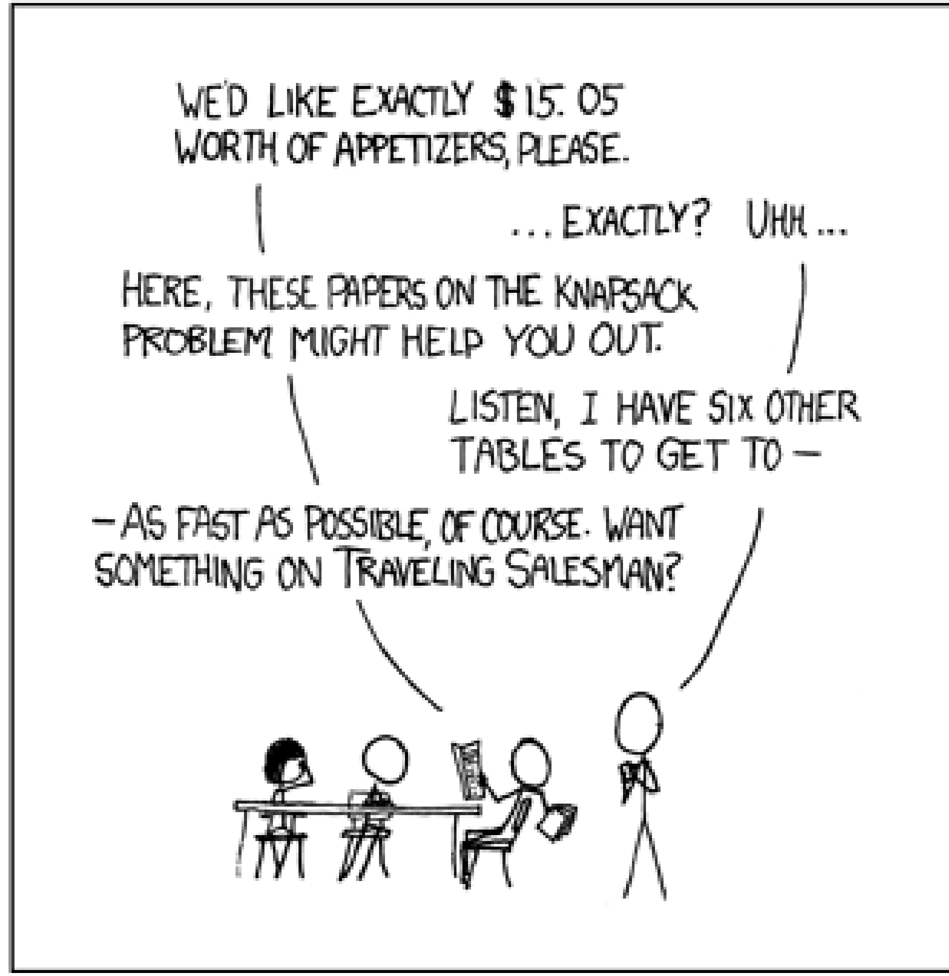


http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

Average-case vs. worst-case complexity

- Sometimes worst case is unlikely or avoidable
  - E.g., quicksort
- Average-case complexity describes behavior for a typical case

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



http://xkcd.com/287/

Things to remember

- Be able to analyze code for computational cost
  - Tools: finding loops and recursive calls, using recursion trees
  - Sometimes need to know inner-workings of a library to determine (e.g., factorialSeries)

- Be able to convert to big-O or big-Theta and be familiar with basic complexity terms
  - E.g., linear, $n \log n$, polynomial, exponential