Big O and Algorithms (Part 1)

by Wolfdog1

Discrete Structures (CS 173) Lecture B

Gul Agha

Slides based on Derek Hoiem, University of Illinois

Today

- How do we characterize the computational cost of an algorithm?
- How do we compare the speed of two algorithms?
- How do we compute cost based on code or pseudocode?

What affects an algorithm's runtime

- Computer architecture
- Programming language and compiler
- Other processes running on the computer
- Parameters (input) to the algorithm
- Design of the algorithm

What affects an algorithm's runtime

- Computer architecture
- Programming language and compiler
- Other processes running on the computer

External factors

- Parameters (input) to the algorithm
- Design of the algorithm

Design factors

How should we think about an algorithm's computational cost?

- Time taken?
- Number of instructions called?
- Expected time as a function of the inputs?
- **How quickly the cost grows as a function of the problem size (number of inputs, dimensions, etc)**

Example: Finding smallest m values

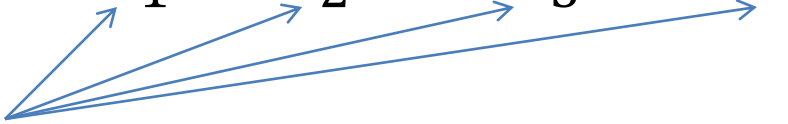
```
output = findmin(input, m) % returns m smallest inputs

for each ith output (there are m of these)
    for each jth input (there are n of these)
       if j is not used  and input(j) < output(i)
          output(i) = input(j);
           j_out = j;
       end
    end
    mark that j_out has been used as an output
end
return output
```

See findmin.m

Example: Finding smallest m values

```
minvals = findmin(vals, m)
```

$$\text{Cost} = k_1 + k_2 n + k_3 m + k_4 mn$$



Constants that depend on implementation details, architecture, etc.

Dominant factor

$$\text{Cost is } O(mn)$$

Key ideas in runtime analysis

1. Model how algorithm speed depends on the size of the input/parameters
   - Ignore factors that depend on architecture or details of compiled code

2. We care mainly about *asymptotic* performance
   - If the input size is small, we're not usually worried about speed anyway

3. We care mainly about dominant terms
   - An algorithm that takes n^2 time takes almost as long (as a ratio) as one that takes $30 + 4n + n^2$ time if n is large

Asymptotic relationships

(asymptotically equivalent) $f(n) \sim g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

(asymptotically smaller) $f(n) \ll g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$\boldsymbol{f(n)}$	$\boldsymbol{g(n)}$
$3 + n^2 + sqrt(n)$	n^2
$2 + 2n^2 + n$	n^2
$1000 + 100n$	n^2
$2^n + n$	$n!$
n^5	2^n
nm	$n \log n$

If $f(n) \ll g(n)$, then $f(n)h(n) \ll g(n)h(n)$ for non-zero $h(n)$

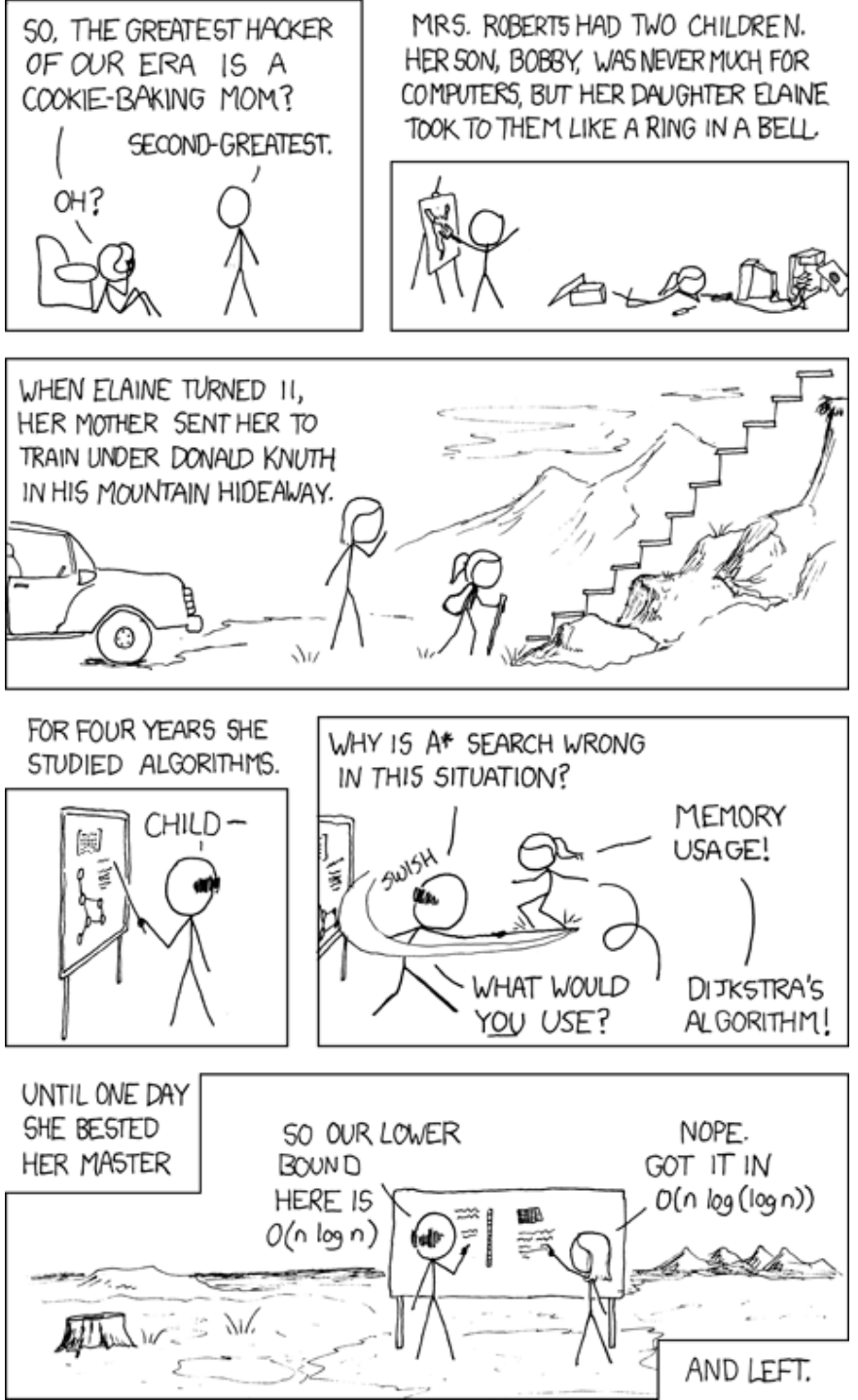
Ordering of functions

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^2$$

$$n^2 \ll n^3 \ll 2^n \ll 3^n \ll n!$$

See `plot_functions.m`

Short break



xkcd

Example: comparing findmin algorithms

```
output = findmin(input, m)  % can be implemented different ways

for each ith output (there are m of these)
    for each jth input (there are n of these)
        if j is not used  and input(j) < output(i)
           output(i) = input(j);
           j_out = j;
       end
    end
    mark that j_out has been used as an output
end
return output
```

```
output = findmin_sort(input, m)
sorted_input = sort(input, ascending);
output = sorted_input(1…m);
```

See `findmin_cost_compare_script.m`

Big-O

$f(n)$ is $O(g(n))$ iff there are $c > 0, k > 0$ such that $0 \leq f(n) \leq cg(n)$ for every $n \geq k$

$\boldsymbol{f(n)}$	$\mathbf{O(}\boldsymbol{g(n)}\mathbf{)}$?
$3 + n^2 + sqrt(n)$	n^2
$2 + 2n^2 + n$	n^2
n^3	$n + n^2$
$1000 + 100n$	n^2
n^5	2^n
nm	$n \log n$

$\Theta(g(n))$

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$

$\boldsymbol{f(n)}$	$\boldsymbol{\Theta(g(n))}$?
$3 + n^2 + sqrt(n)$	n^2
$2 + 2n^2 + n$	n^2
$1000 + 100n$	n^2
$2^n + n$	$n!$
n^5	2^n
nm	$n \log n$

Proving Big-O with induction

Claim: $10n^2$ is $O(2^n)$

Definition: $f(n)$ is $O(g(n))$ iff there are $c > 0, k > 0$ such that

$0 \leq f(n) \leq cg(n)$ for every $n \geq k$.

Choose c and k: $c = 10, k = 4$

Proof: Suppose n is an integer. We need to show $10n^2 \leq 10 \cdot 2^n$ for $n \geq 4$.

Use induction on n.

Base case:

Induction: Suppose $10n^2 \leq 10 \cdot 2^n$ for $n = 4, 5, \ldots, k$. We need to show $10(k+1)^2 \leq 10 \cdot 2^{k+1}$.

Things to remember

- Model algorithm complexity in terms of how much the cost increases as the input/parameter size increases

- In characterizing algorithm computational complexity, we care about
  - Large inputs
  - Dominant terms

- $1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll 3^n \ll n!$

- $f(n)$ is $O(g(n))$ if the dominant terms in $f(n)$ are equivalent (up to some constant factor) or dominated by the dominant terms in $g(n)$

- $f(n)$ is $\Theta(g(n))$ if the dominant terms in $f(n)$ are equivalent to those in $g(n)$

Legend

"An Indian temple in Kashi Vishwanath contains a large room with three time-worn posts in it surrounded by 64 golden disks. 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 world will end."

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

Towers of Hanoi

By André Karwath aka Aka - Own work, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=85401

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:
  - Example: The tower of Hanoi puzzle can be solved for any number of disks. *Base*: it works for one and two disks.

- Graphs:
  - Example: Any fully connected graph with n nodes has $n(n-1)/2$ edges. *Base*: A graph with one node has 0 edges.

- Trees:
  - 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.

Algorithms

- Practice analyzing runtime based on pseudocode

```
01  closestpair(p_1, ..., p_n) : array of 2D points)
02          best1 = p_1
03          best2 = p_2
04          bestdist = dist(p_1,p_2)
05          for i = 1 to n
06                  for j = 1 to n
07                          newdist = dist(p_i,p_j)
08                          if (i ≠ j and newdist < bestdist)
09                                  best1 = p_i
10                                  best2 = p_j
11                                  bestdist = newdist
12          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

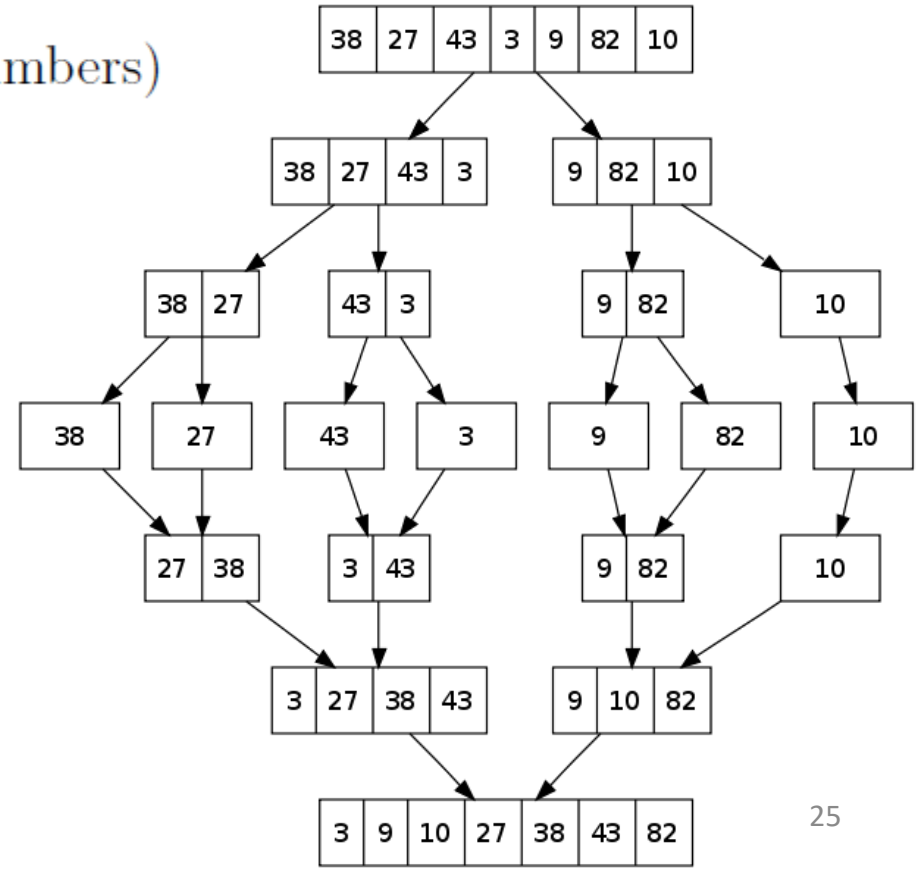
```
01  merge(L1,L2: sorted lists of real numbers)
02        O = emptylist
03        while (L1 is not empty or L2 is not empty)
04              if (L1 is empty)
05                    move head(L2) to the tail of O
06              else if (L2 is empty)
07                    move head(L1) to the tail of O
08              else if (head(L1) <= head(L2))
09                    move head(L1) to the tail of O
10              else move head(L2) to the tail of O
11        return O
```

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

```
01  mergesort(L = a_1, a_2, ..., a_n: list of real numbers)
02        if (n = 1) then return L
03        else
04              m = ⌊n/2⌋
05              L_1 = (a_1, a_2, ..., a_m)
06              L_2 = (a_{m+1}, a_{m+2}, ..., a_n)
07              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

```
01  hanoi(A,B,C: pegs, d_1, d_2 ... d_n: disks)
02          if (n = 1) move d_1 = d_n from A to B.
03          else
04                  hanoi(A,C,B,d_1, d_2, ... d_{n-1})
05                  move d_n from A to B.
06                  hanoi(C,B,A,d_1, d_2, ... d_{n-1})
```

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