# S O T N I R G

Algorithms & Analysis

How many pebbles?

# heuristic

# Heuristics

- Not necessarily *correct* (but gets you a "*good enough*" answer)
- Advantage: *fast* (often way faster than finding an exact solution)
- Famous example: the Traveling Salesman Problem

- Given N cities with a given cost of traveling between each pair, what is the cheapest way to travel to all of them?

Arriving

Departing

| | NYC | SF | CHICAGO |
|---|---|---|---|
| NYC | NA | $250 | $120 |
| SF | $210 | NA | $150 |
| CHICAGO | $100 | $115 | NA |

| | |
|---|---|
| NYC → SF → CHI | $400 |
| NYC → CHI → SF | $235 |
| SF→ NYC → CHI | $330 |
| SF → CHI → NYC | $250 |
| CHI → NYC → SF | $350 |
| CHI → SF → NYC | $325 |

# brute force

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

…etc.

# Algorithms

- Step-by-step instructions
- Complete (gets you an answer)
- Finite (…given enough time)
- May be Exact (provably correct)
  - Downside: some problems are very hard / slow
- May be Approximate (using some heuristic)
  - Upside: often much faster for very hard problems

How can we compare algorithms?

THE
BIG
O

# Algorithm Analysis: Big O Notation

- A *comparative* way to classify different algorithms
- Based on how they *scale* (respond to input size)
- Tells you the *shape* of *growth curve* (*time* vs *input size(s))*
- For *big enough* inputs
- Establishing an *upper bound* on the time
  - Not worse than this. Might be better, but it ain't worse!
- Including just the *highest order* term
  - In $f(n) = n^3 + 5n + 3$, only $n^3$ matters as $n$ gets large
- *Ignores constants* (mostly irrelevant; $0.1 \cdot n^2$ will overtake $10 \cdot n$)

What?

Adapted from The Algorithm Design Manual

# Big O: comparative

- A very coarse, broad tool – big simplification
- Only useful when algorithms have *different* Big O notations
  - O(n) will always beat $O(n^2)$, *for big enough n*
- If two algorithms have the same Big O, we don't know much. One might actually be quite slower than the other.

# Two O(n) Functions

```javascript
function findColors (arr) {
  var colors = {
    red: true,
    orange: true,
    yellow: true,
    green: true,
    blue: true
  };
  arr.forEach(function (val, i) {
    if (colors[val]) console.log(i, val);
  });
}
```

```javascript
function findColorsSlow (arr) {
  arr.forEach(function (val, i) {
    if (val === 'red') console.log(i, val);
  });
  arr.forEach(function (val, i) {
    if (val === 'orange') console.log(i, val);
  });
  arr.forEach(function (val, i) {
    if (val === 'yellow') console.log(i, val);
  });
  arr.forEach(function (val, i) {
    if (val === 'green') console.log(i, val);
  });
  arr.forEach(function (val, i) {
    if (val === 'blue') console.log(i, val);
  });
}
```

O(n)
5n
O(n)
n
BOTH are O(n)
running time
Size of Input (n)
y
x
100
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100

0.5n²
O(n²)
5n
O(n)
n
O(n)
running time
Size of Input (n)
y
x
0
10
20
30
40
50
60
70
80
90
100

n!
2n
n2
n· log(n)
n
running time
Common Big O Notations
log(n)
Size of Input (n)

*Source: Skiena, The Algorithm Design Manual

| input size n | log n | n | n·log n | $n^2$ | $2^n$ | n! |
|---|---|---|---|---|---|---|
| 10 | 0.003 µs | 0.01 µs | 0.03 µs | 0.1 µs | 1 µs | 3.63 ms |
| 20 | 0.004 µs | 0.02 µs | 0.09 µs | 0.4 µs | 1 ms | 77.1 years |
| 30 | 0.005 µs | 0.03 µs | 0.15 µs | 0.9 µs | 1 sec | $8.4 \times 10^{15}$ |
| 40 | 0.005 µs | 0.04 µs | 0.21 µs | 1.6 µs | 18.3 min | |
| 50 | 0.006 µs | 0.05 µs | 0.28 µs | 2.5 µs | 13 days | |
| 100 | 0.007 µs | 0.10 µs | 0.64 µs | 10.0 µs | $4 \times 10^{13}$ yrs | |
| 1 000 | 0.010 µs | 1.00 µs | 9.97 µs | 1 ms | | |
| 10 000 | 0.013 µs | 10.00 µs | ~130.00 µs | 100 ms | | |
| 100 000 | 0.017 µs | 100.00 µs | 1.7 ms | 10 sec | | |
| 1 000 000 | 0.020 µs | 1 ms | 19.9 ms | 16.7 min | | |
| 10 000 000 | 0.023 µs | 10 ms | 230.0 ms | 1.16 days | | |
| 100 000 000 | 0.027 µs | 100 ms | 2.66 sec | 115.7 days | | |
| 1 000 000 000 | 0.030 µs | 1 sec | 29.90 sec | 31.7 years | | |

| Big O | Name | Think | Example |
|---|---|---|---|
| **O(1)** | *Constant* | Doesn't depend on input | get array value by index |
| **O(log n)** | *Logarithmic* | Using a tree | find min element of BST |
| **O(n)** | *Linear* | Checking (up to) all elements | search through linked list |
| **O(n · log n)** | *Loglinear* | A tree for each element | merge sort average & worst case |
| **O($n^2$)** | *Quadratic* | Checking pairs of elements | bubble sort average & worst case |
| **O($2^n$)** | *Exponential* | Generating all subsets | brute-force n-long binary number |
| **O(n!)** | *Factorial* | Generating all permutations | the Traveling Salesman |

| Data Structure | Time Complexity | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |
| Array | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Singly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Doubly-Linked List | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Skip List | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) |
| Hash Table | - | O(1) | O(1) | O(1) | - | O(n) | O(n) | O(n) |
| Binary Search Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) |

# (Some) Classic Sorting Algorithms

- Bubble
- Selection
- Insertion
- Merge: 1945 Jon von Neumann
- Quick: 1959 Tony Hoare
- Heap: 1964 J. W. J. Williams
- Radix: 1887 Hermann Hollerith, for his Tabulating Machine
- Bogo?

# Bubble Sort

6 5 3 1 8 7 2 4

# Bubble Sort

1. Loop over elements
2. Swap anything that's out of order
3. Repeat 1-2 until there are no swaps

# Merge Sort

6 5 3 1 8 7 2 4

# Merge Sort

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

# Merge Sort (iterative)

1. Divide array of n elements into n arrays of 1 element
2. Merge neighboring arrays in sorted order
3. Repeat 2 until there's only one array

# Merge Sort (recursive)

1. If array is one element, good job it's sorted!
2. Otherwise, split the array and merge sort each half
3. Merge combined halves into sorted whole

# Big O

| | Bubble Sort | Merge Sort |
|---|---|---|
| **Time** | $O(n^2)$ | $O(n \cdot \log n)$ |
| **Space** | $O(1)$ | $O(n)$ |

# Why is merge sort faster?

# Merge Sort Speedup

- Combining two lists that are each already sorted into one list that is sorted is a linear time operation
- There are $\log_2(n)$ steps needed to go from $n$ lists of one item each to one list of $n$ items

# Special Note

# Rob Pike's 5 Rules of Programming

Bell Labs
Unix Team
UTF-8
Go Language
...and a lot more

- You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

- Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

- Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy.

- Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

- Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

# WORKSHOP: GO!