

Dropping Glass Balls

- ❖ Tower with N Floors
- ❖ Given 2 glass balls
- ❖ Want to determine the *lowest* floor from which a ball can be dropped and will break
- ❖ How?

- ❖ What is the most efficient algorithm?
- ❖ How many drops will it take for such an algorithm (as a function of N)?

Glass balls revisited (more balls)

- ❖ Assume the number of floors is 100
 - ❖ In the best case how many balls will I have to drop to determine the lowest floor where a ball will break?
 - ❖ In the worst case, how many balls will I have to drop?
- | | |
|--|--|
| <ol style="list-style-type: none"> 1. 1 2. 2 3. 10 4. 16 5. 17 6. 18 7. 20 8. 21 9. 51 10. 100 | <ol style="list-style-type: none"> 1. 1 2. 2 3. 10 4. 16 5. 17 6. 18 7. 20 8. 21 9. 51 10. 100 |
|--|--|

If there are n floors, how many balls will you have to drop? (*roughly*)

What is big-Oh about? (preview)

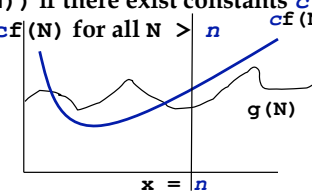
- ❖ Intuition: avoid details when they don't matter, and they don't matter when input size (N) is big enough
 - For polynomials, use only leading term, ignore coefficients

$$\begin{array}{lll}
 y = 3x & y = 6x-2 & y = 15x + 44 \\
 y = x^2 & y = x^2-6x+9 & y = 3x^2+4x
 \end{array}$$

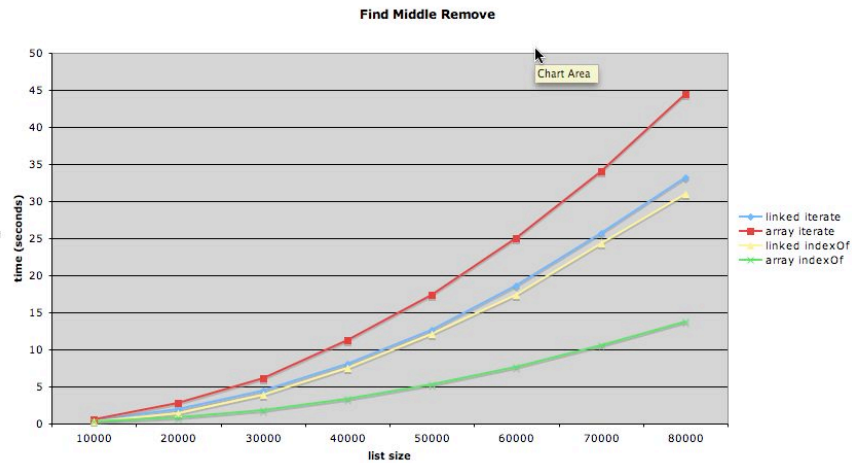
- ❖ The first family is $O(n)$, the second is $O(n^2)$
 - Intuition: family of curves, generally the same shape
 - More formally: $O(f(n))$ is an *upper-bound*, when n is large enough the expression $cf(n)$ is larger
 - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time

More on O-notation, big-Oh

- ❖ Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
 - Allows us to compare algorithms *in the limit*
 - $20N$ hours vs N^2 microseconds: *which is better?*
- ❖ O-notation is an upper-bound, this means that N is $O(N)$, but it is also $O(N^2)$; we try to provide *tight* bounds. Formally:
 - A function $g(N)$ is $O(f(N))$ if there exist constants c and n such that $g(N) < cf(N)$ for all $N > n$



Which graph is “best” performance?



CompSci 100E

4.5

Big-Oh calculations from code

- ❖ Search for element in an array:
 - What is complexity of code (using O-notation)?
 - What if array doubles, what happens to time?

```
for(int k=0; k < a.length; k++) {
    if (a[k].equals(target)) return true;
};
return false;
```

- ❖ Complexity if we call N times on M-element vector?
 - What about best case? Average case? Worst case?

CompSci 100E

4.6

Amortization: Expanding ArrayLists

- ❖ Expand capacity of list when add() called
- ❖ Calling add N times, doubling capacity as needed

Item #	Resizing cost	Cumulative cost	Resizing Cost per item	Capacity After add
1	0	0	0	1
2	2	2	1	2
3-4	4	6	1.5	4
5-8	8	14	1.75	8
$2^{m+1} - 2^{m+1}$	2^{m+1}	$2^{m+2} - 2$	around 2	2^{m+1}

CompSci 100E

4.7

- ❖ What if we grow size by one each time?

Some helpful mathematics

- ❖ $1 + 2 + 3 + 4 + \dots + N$
 - $N(N+1)/2$, exactly = $N^2/2 + N/2$ which is $O(N^2)$ why?
- ❖ $N + N + N + \dots + N$ (total of N times)
 - $N*N = N^2$ which is $O(N^2)$
- ❖ $N + N + N + \dots + N + \dots + N + \dots + N$ (total of 3N times)
 - $3N*N = 3N^2$ which is $O(N^2)$
- ❖ $1 + 2 + 4 + \dots + 2^N$
 - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

- ❖ Impact of last statement on adding 2^{N+1} elements to a vector

CompSci 100E

□ $1 + 2 + \dots + 2^N + 2^{N+1} = 2^{N+2} - 1 = 4 \times 2^N - 1$ which is $O(2^N)$

resizing + copy = total (let $x = 2^N$)

Running times @ 10^6 instructions/sec

N	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

Loop Invariants

- ❖ Want to reason about the correctness of a proposed iterative solution
- ❖ Loop invariants provide a means to effectively reason about the correctness of code

```
while !done do
  // what is true at every step
  // Update/iterate
  // maintain invariant
od
```

Bean Can game

- ❖ Can contains N black beans and M white beans initially
- ❖ Emptied according to the following repeated process
 - Select two beans from the can
 - If the beans are:
 - same color: put a black bean back in the can
 - different colors: put a white bean back in the can
 - Player who chooses the color of the remaining bean wins the game
- ❖ Analyze the link between the initial state and the final state
- ❖ Identify a property that is preserved as beans are removed from the can

- *Invariant* that characterizes the removal process

Bean Can Algorithm

```
while (num-beans-in-can > 1) do
  pick 2 beans randomly
  if bean1-color == bean2-color then
    put-back black bean
  else
    put-back white bean
od
```

Bean Can Analysis

- ❖ What happens each turn?
 - Number of beans in can is decreased by one
 - Number of white beans is either reduced by 2 or 0
 - Number of black beans is either reduced by 1 or 0
- ❖ Examine the final states for 2 bean and 3 bean initial states
- ❖ Any guesses for the correct strategy?

- ❖ What is the process invariant?

The Game of Nim

- ❖ Two Piles of counters with N and M counters in each pile
- ❖ 2 players take turns:
 - Remove some number of counters (≥ 1) from one pile
 - Player who removes last counter wins
- ❖ Properties
 - *Complete information*: could exhaustively search for winning solution
 - *Impartial*: same moves are available for each player

Nim Analysis

- ❖ Denote state by (x,y) : number of counters in each pile
- ❖ What about simple case of $(1,1)$?

- ❖ For whom is $(1,1)$ a “safe” state?

- ❖ How about $(1,2)$ or $(1,3)$?

- ❖ How about $(2,2)$?

- ❖ What is the *invariant* to be preserved by the winning player?

Nim Algorithm

```
// reach a state (x,y) where x=y on
// opponent's
// turn and then follow below algorithm

while !empty(pile1) && !empty(pile2) do
  let opponent remove q counters from a
  pile
  remove q counters from other pile
od
```

Numbers from Ends

- ❖ Game begins with some even number of numbers on a line
10 5 7 9 6 12
- ❖ Players take turns removing numbers from the ends while keeping running sum of numbers collected so far
- ❖ Player with largest sum wins
- ❖ Complete information but how to win without search?

Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

- ❑ Alexander et. al, 1977
- ❑ A text on architecture!

- ❖ What is a programming or design pattern?
- ❖ Why are patterns important?

What is a pattern?

- ❖ "... a three part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, ... , and the rule which tells us how to create that thing, and when we must create it."

Christopher Alexander

- ❑ **name** *factory, aka virtual constructor*
- ❑ **problem** *delegate creation responsibility: expression tree nodes*
- ❑ **solution** *createFoo() method returns aFoo, bFoo,...*
- ❑ **consequences** *potentially lots of subclassing, ...*

- ❖ more a recipe than a plan, micro-architecture, frameworks, language idioms made abstract, less than a principle but more than a heuristic

- ❖ **patterns capture important practice in a form that makes the practice accessible**

Patterns are discovered, not invented

- ❖ You encounter the same "pattern" in developing solutions to programming or design problems
 - ❑ develop the pattern into an appropriate form that makes it accessible to others
 - ❑ fit the pattern into a language of other, related patterns
- ❖ Patterns transcend programming languages, but not (always) programming paradigms
 - ❑ OO folk started the patterns movement
 - ❑ language idioms, programming templates, programming patterns, case studies

Programming Problems

- ❖ Microsoft interview question (1998)

3	3	5	5	7	8	8	8
---	---	---	---	---	---	---	---

- ❖ Dutch National Flag problem (1976)

- ❖ Remove Zeros (AP 1987)

2	1	0	5	0	0	8	4
---	---	---	---	---	---	---	---

- ❖ Quicksort partition (1961, 1986)

4	3	8	9	1	6	0	5
3	1	0	4	8	9	6	5

- ❖ Run-length encoding (SIGCSE 1998)



11 3 5 3 2 6 2 6 5 3 5 3 5 3 10

Removing Duplicates

```
void crunch(ArrayList<String> list)
{
    int lastUniqueIndex = 0;
    String lastUnique = list.get(0);
    for(int k=1; k < list.size(); k++)
    {
        String current = list.get(k);
        if (current != lastUnique)
        {
            list.set(++lastUniqueIndex, current);
            lastUnique = current;
        }
    }
    for (int k=list.size()-1; k > lastUniqueIndex; k--)
        list.remove(k);
}
```

One loop for linear structures

- ❖ Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but data is stored sequentially, e.g., in an array or file. Programming based on control leads to more problems than programming based on structure.

Therefore, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately guarded conditionals to implement the control

Consequences: one loop really means loop according to structure, do not add loops for control: what does the code look like for run-length encoding example?

Coding Pattern

- ❖ **Name:**
 - ❑ one loop for linear structures
- ❖ **Problem:**
 - ❑ Sequential data, e.g., in an array or a file, must be processed to perform some algorithmic task. At first it may seem that multiple (nested) loops are needed, but developing such loops correctly is often hard in practice.
- ❖ **Solution:**
 - ❑ Let the structure of the data guide the coding solution. Use one loop with guarded/if statements when processing one-dimensional, linear/sequential data
- ❖ **Consequences:**
 - ❑ Code is simpler to reason about, facilitates develop of loop invariants, possibly leads to (slightly?) less efficient code