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?
  1. 1
  2. 2
  3. 10
  4. 16
  5. 17
  6. 18
  7. 20
  8. 21
  9. 51
  10. 100

- In the worst case, how many balls will I have to drop?
  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
    
    \[
    \begin{align*}
    y &= 3x \\
    y &= 6x - 2 \\
    y &= 15x + \\
    y &= x^2 \\
    y &= x^2 - 6x + 9 \\
    y &= 3x^2 + 4x
    \end{align*}
    \]

- The first family is O(n), the second is O(n^2)
  - For polynomials, use only leading term, ignore coefficients
  - 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 formally: O(\(f(n)\)) is an upper-bound, when n is large enough the expression cf(n) is larger

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, n such that g(N) < cf(N) for all N > n
**Which graph is “best” performance?**

![Graph showing performance comparison](image)

**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?

```java
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?

**Amortization: Expanding ArrayLists**

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

<table>
<thead>
<tr>
<th>Item #</th>
<th>Resizing cost</th>
<th>Cumulative cost</th>
<th>Resizing Cost per item</th>
<th>Capacity After add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3-4</td>
<td>4</td>
<td>6</td>
<td>1.5</td>
<td>4</td>
</tr>
<tr>
<td>5-8</td>
<td>8</td>
<td>14</td>
<td>1.75</td>
<td>8</td>
</tr>
<tr>
<td>$2^m+1 - 2^{m+1}$</td>
<td>$2^{m+1}$</td>
<td>$2^{m+2} - 2$</td>
<td>around 2</td>
<td>$2^{m+1}$</td>
</tr>
</tbody>
</table>

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

**Some helpful mathematics**

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

- Impact of last statement on adding $2^{N+1}$ elements to a vector
  - $1 + 2 + \ldots + 2^N + 2^{N+1} = 2^{N+2} - 1 = 4 \times 2^N - 1$ which is $O(2^N)$

- Resizing + copy = total (let $x = 2^N$)
Running times @ $10^6$ instructions/sec

<table>
<thead>
<tr>
<th>N</th>
<th>$O(\log N)$</th>
<th>$O(N)$</th>
<th>$O(N \log N)$</th>
<th>$O(N^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.00100</td>
<td>0.01000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.13290</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.66100</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>

Loop Invariants

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

```plaintext
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 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

```plaintext
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

```java
// 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?

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

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
## Programming Problems

- **Microsoft interview question (1998)**

- **Dutch National Flag problem (1976)**

- **Remove Zeros (AP 1987)**

- **Quicksort partition (1961, 1986)**

- **Run-length encoding (SIGCSE 1998)**

## Removing Duplicates

```java
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.