How to be a good computer engineer?

- Good engineers are lazy, otherwise
  - every door in a building
  - every light switch
  - every power outlet
  - every screw
  - ...

- Lazy engineers spent minimum effort to solve a problem
  - never reinvent the wheel
  - never start from scratch
  - always *reuse* existing tools

- Lazy computer engineers write minimum code to solve a problem
- Today’s topic: How to become a lazy computer engineer?
  - Lazy computer engineers use **generics**
  - Lazy computer engineers use **recursion**
  - Lazy computer theoreticians use **asymptotic notation**
1. Generics

2. Recursion

3. Asymptotic Notation
Generic Linked List

- What is a list (of integers)?

- Why do we need a linked list?

- What are the functions that we normally need to manipulate a list?

- Given an object \( x \), how do we check if \( x \) is in the list? (we call this function, “find(\( x \))”)
Now, what do I do if I need a list of strings? Do I need to re-design the whole list?

But I am lazy, so what should I do?

Approach 1:

Approach 2:
class FindMaxDemo
{
  public static void main( String [] args )
  {
    * Test findMax on Shape and String objects.
    Shape [] shape = { new Circle( 2.0 ),
                      new Square( 3.0 ),
                      new Rectangle( 3.0, 4.0 )};
    String [] st1 = { "Joe", "Bob", "Bill", "Zek" };
Find Max

```java
{  
    System.out.println(findMax(arr, new CaseInsensitiveComparator());
    String arr = "ZEBRA", "ILLIGATOR", "CRUCODILE";
}

public static void main(String args[])
{
    class TestProgram
    {
        return this.comparatorWelcomeThis();
        public int compare(String this, String string)
        {
            class CaseInsensitiveComparator implements Comparator<String>
            {
                return arr[maxindex];
                
                if (comparator(arr[i], arr[maxIndex] < 0)
                    for (int i = 1; i < arr.size(); ++i)
                    
                int maxindex = 0;
                AnyType findMax(AnyType[] arr, Comparator<? super AnyType> comp)
                public static <AnyType>
                    // precondition: size(arr) > 0.
                    // Generic findMax, with a function object.
                    
```
Fibonacci numbers $\text{fib}(n)$:

$$\text{fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{if } n > 1 
\end{cases}$$  \hfill (1)

Example: The first 10 Fibonacci numbers are:

\{0, 1, ____, ____, ____, ____, ____, ____, ____, ____, ____\}
Problem: What is \( \text{fib}(200) \)? What about \( \text{fib}(n) \), where \( n \) is any positive integer?

**Algorithm 3.1**: \( \text{fib}(n) \)

```java
if \( n = 0 \)
    then return \((0)\)
if \( n = 1 \)
    then return \((1)\)
return \((\text{fib}(n - 1) + \text{fib}(n - 2))\)
```

Questions that we should ask ourselves.

- Is the algorithm correct?
- What is the running time of our algorithm?
- Can we do better?
Analyze Our First Algorithm

- Is the algorithm correct?
  - Yes, we simply follow the definition of Fibonacci numbers
- How fast is the algorithm?
  - If we let the run time of $\text{fib}(n)$ be $T(n)$, then we can formulate
    \[ T(n) = T(n - 1) + T(n - 2) + 3 \approx 1.6^n \]
  - $T(200) \geq 2^{139}$
  - The world fastest computer, which can run $2^{56}$ instructions per second (93 Peta FLOPS, Peta=$10^{15}$), will take $2^{83}$ seconds to compute. ($2^{83}$ seconds = $3 \times 10^8$ billion years, Sun turns into a red giant star in 4 to 5 billion years, the Universe is about 13.82 billion years old)
  - Can Moose’s law, which predicts that CPU get 1.6 times faster each year, solve our problem?
  - No, because the time needed to compute $\text{fib}(n)$ also have the same “growth” rate
    - if we can compute $\text{fib}(100)$ in exactly a year,
    - then in the next year, we will still spend a year to compute $\text{fib}(101)$
    - if we want to compute $\text{fib}(200)$ within a year, we need to wait for 100 years.
Can we do better?
Yes, because many computations in the previous algorithm are repeated.

Algorithm 3.2: \texttt{fib}(n)


c\texttt{omment:} Initially we create an array $A[0 \cdots n]$

$A[0] \leftarrow 0$, $A[1] \leftarrow 1$

\texttt{for} $i = \{2 \cdots n\}$

\hspace{1em} \texttt{do} $A[i] = A[i - 1] + A[i - 2]$

\texttt{return} ($A[n]$)
Theoretical analysis of time efficiency

- Provide *machine independent* measurements
- Estimate the bottleneck of the algorithm
- The size of the input increases $\rightarrow$ algorithms run longer $\Rightarrow$. Typically we are interested in how efficiency scales w.r.t. input size
- To measure the running time, we could
  1. count all operations executed.
  2. or determine the number of the **basic operation** as a function of input size
- **Basic operation**: the operation that contributes most towards the running time
Some of the commonly seen functions representing the number of the basic operation $C(n) =$

1. $n$
2. $n^2$
3. $n^3$
4. $\log_{10}(n)$
5. $n \log_{10}(n)$
6. $\log^2_{10}(n)$
7. $\sqrt{n}$
8. $2^n$
9. $n!$

Can you order them by their growth rate?
Orders of Growth

- Test functions using some values

<table>
<thead>
<tr>
<th>n</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>1024</td>
<td>$3.6 \times 10^6$</td>
</tr>
<tr>
<td>100</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$1.3 \times 10^{30}$</td>
<td>$9.3 \times 10^{157}$</td>
</tr>
<tr>
<td>1000</td>
<td>$10^6$</td>
<td>$10^9$</td>
<td>$1.1 \times 10^{301}$</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>$10^8$</td>
<td>$10^{12}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>$\log_{10}(n)$</th>
<th>$n \log_{10}(n)$</th>
<th>$\log^2_{10}(n)$</th>
<th>$\sqrt{n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>3.16</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>200</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>3</td>
<td>3000</td>
<td>9</td>
<td>31.6</td>
</tr>
<tr>
<td>10000</td>
<td>4</td>
<td>40000</td>
<td>16</td>
<td>100</td>
</tr>
</tbody>
</table>

(see Weiss pg 203)

- Now, we can order the functions by their growth rate

$log_{10}(n) < log^2_{10}(n) < \sqrt{n} < n < n \log_{10}(n) < n^2 < n^3 < 2^n < n!$
The main goal of algorithm analysis is to estimate **dominate computation steps** $C(n)$ when the **input size** $n$ is large.

Computer scientists classify $C(n)$ into a set of functions to help them concentrate on trend (i.e., order of growth).

Asymptotic notation has been developed to provide a tool for studying order of growth:

- $O(g(n))$: a set of functions with the same or smaller order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in O(n^2)$
  - $2^n + n^{100} - 2 \in O(n!)$
  - $2n + 6 \notin O(\log n)$

- $\Omega(g(n))$: a set of functions with the same or larger order of growth as $g(n)$
  - $2n^2 - 5n + 1 \in \Omega(n^2)$
  - $2^n + n^{100} - 2 \notin \Omega(n!)$
  - $2n + 6 \in \Omega(\log n)$
Example: Maximum contiguous subsequence sum

Don’t play: 0 gain
How Would you find Best Increase?

<table>
<thead>
<tr>
<th>i</th>
<th>price</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>886</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>890</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>880</td>
<td>-10</td>
</tr>
<tr>
<td>4</td>
<td>890</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>899</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>911</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>903</td>
<td>-8</td>
</tr>
<tr>
<td>8</td>
<td>913</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>920</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>924</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>927</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>921</td>
<td>-6</td>
</tr>
<tr>
<td>13</td>
<td>919</td>
<td>-2</td>
</tr>
<tr>
<td>14</td>
<td>887</td>
<td>-32</td>
</tr>
<tr>
<td>15</td>
<td>902</td>
<td>15</td>
</tr>
</tbody>
</table>

Several names for the Problem
- Maximum contiguous subsequence sum (text)
- Maximum Subarray (wikip)
- Find start and end time with largest payoff out of all possible

Find a Solution
- **Input** is the array delta[]
- **Output**: (start, end, payoff) such that payoff is as large as possible
- Can optionally *not invest* for no payoff; return (-1,-1,0)
Algorithm 1: Brute Force

```java
maxSubsequenceCube(int A[]) {
    bestPayoff = 0
    bestStart = -1
    bestEnd = -1
    for start=0 to A.length-1 {
        for end=start to A.length-1 {
            currentPayoff = 0
            for i=start to end {
                currentPayoff += A[i]
            }
            if(currentPayoff > bestPayoff){
                bestPayoff = currentPayoff
                bestStart = start
                bestEnd = end
            }
        }
    }
    return bestPayoff, bestStart, bestEnd
}
```

- `A[]` contains deltas
- Try every possible start and end (outer loops)
- Calculate increase from start to end
- Track the best seen
- **Complexity?**
- Anything better
maxSubsequenceQuad(int A[]){
    B = new array size A.length
    B[0] = A[0]
    for i=1 to B.length-1

    best = 0
    bestStart = -1
    bestEnd = -1
    for start=0 to A.length-1 {
        for end=start to A.length-1 {
            current = B[end] - B[start]
            if(current > best){
                best = current
                bestStart = start
                bestEnd = end
            }
        }
    }
    return best, bestStart, bestEnd
}
Proposition: The shortest maximum subsequence beginning at start and finishing at end contains no point mid between them with a lower value than start.

Proof by Contradiction:
- Suppose shortest max subsequence exists, looks like picture.
- x must be lower than end, o/w could form a shorter maximum subsequence start to x
- But if mid is lower then start, sequence mid to end has a larger increase than start to end.

Consequence: If mid drops below start, reset start to mid
Create a faster algorithm based on this property.
Algorithm 3: Scan

```java
maxSubsequenceLinear(int A[]){
    best = 0
    current = 0
    bestStart = -1
    bestEnd = -1
    start = 0
    for end=0 to A.length-1 {
        current += A[end]
        if(current > best){
            best = current
            bestStart = start
            bestEnd = end
        }
        else if(current < 0){
            start = end+1;
            current = 0;
        }
    }
    return best,bestStart,bestEnd;
}
```

- A[] contains deltas
- When sum current falls below zero, move start to end and reset
- Single pass over entire array
Comparisons

- \texttt{maxSubsequenceCube()}: triply nested loops over entire array, \(O(N^3)\)
- \texttt{maxSubsequenceQuad()}: doubly nested loops over entire array, \(O(N^2)\)
- \texttt{maxSubsequenceLinear()}: single loop over entire array, \(O(N)\)

Intuition: for large arrays, \texttt{maxSubsequenceLinear()} will produce answers faster
Lazy computer engineers do **generics**
Lazy computer engineers do **recursion** (with care!)
Lazy computer theoreticians do **asymptotic notation**

- It is not easy to be lazy; you need to try very hard!
- Read: Chapter 5
- Next week: More Big-O, List, Stacks, Queues