(Abstract) Data Structures?

- What are they?

- Why do you have to learn data structures?

- Where will it be used (e.g. in CS 483)?
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
  - ... would be different

- Lazy engineers spent minimum effort to solve a problem
  - never reinvent the wheel
  - never start from scratch
  - always reuse (but don’t steal) existing tools.

- Lazy computer engineers write minimum code to solve a problem

- However, in CS 310, we start our code from scratch so we can learn

- 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
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 Comparable findMax(Comparable[] arr)
  {
    int maxIndex = 0;
    for (int i = 1; i < arr.length; i++)
      if (arr[i].compareTo(arr[maxIndex]) > 0)
      {
        maxIndex = i;
      }
    return arr[maxIndex];
  }
  public static void main(String[] args)
  {
    Shape[] sh1 = { new Circle(2.0),
                    new Square(3.0),
                    new Rectangle(3.0, 4.0) };
    String[] st1 = { "Joe", "Bob", "Bill", "Zeke" };
    System.out.println(findMax(sh1));
    System.out.println(findMax(st1));
  }
}

***

/** *
* Test findMax on Shape and String objects.
*/

***

/** *
* Return max item in arr.
* Precondition: arr.length > 0
*/
```java
public static String findMax(int[] arr) {
    int maxIndex = 0;
    for (int i = 1; i < arr.length; ++i) {
        if (arr[i] > arr[maxIndex]) {
            maxIndex = i;
        }
    }
    return arr[maxIndex];
}
```
Recursion

- Fibonacci numbers \( \text{fib}(n) \):

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

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

```
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.

1. Is the algorithm correct?
2. What is the running time of our algorithm?
3. 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: fib(n)**

**comment:** Initially we create an array $A[0 \cdots n]$  

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

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


**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
Orders of Growth

- 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. $\log_{10}^2(n)$
  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>$\log_{10}^2(n)$</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

$\log_{10}(n) < \log_{10}^2(n) < \sqrt{n} < n < n \log_{10}(n) < n^2 < n^3 < 2^n < n!$
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 (textbook)
- Maximum Subarray (wikipedia)
- 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)

How is payoff computed for start=5 and end=12?
For start=7 and end=10?
Algorithm 1: Brute Force

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
}
Algorithm 2 Alternative: Convert to global Prices

```java
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
}
```

- Initially convert deltas in A to global prices in B
- First price doesn’t matter as interested in changes
- Try every start and end
- Easy to calculate current Payoff
- Memory overhead?
A Helpful Property

Proposition: The shortest maximum subsequence beginning at \text{start} and finishing at \text{end} contains no point \text{mid} between them with a lower value than \text{start}.

Proof by Contradiction:

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

\textbf{Contradiction □}

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

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
Given that array $A$ has $n$ elements,

- `maxSubsequenceCube()`: triply nested loops over entire array, $O(n^3)$
- `maxSubsequenceQuad()`: doubly nested loops over entire array, $O(n^2)$
- `maxSubsequenceLinear()`: single loop over entire array, $O(n)$

Intuition: for large arrays, `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