Tools: Solve Computational Problems

- Algorithmic techniques
  - Brute-force/exhaustive, greedy algorithms, dynamic programming, divide-and-conquer, ...

- Programming techniques
  - Recursion, memo-izing, compute-once/lookup, tables, ...

- Java techniques
  - java.util.*, Comparator, Priority Queue, Map, Set, ...

- Analysis of algorithms and code
  - Mathematical analysis, empirical analysis

Today

Analysis: Algorithms and Data Structures

- We need a vocabulary to discuss performance and to reason about alternative algorithms and implementations
  - It’s faster! It’s more elegant! It’s safer! It’s cooler!

- We need empirical tests and analytical/mathematical tools
  - Given two methods, which is better? Run them to check.
    - 30 seconds vs. 3 seconds, easy. 5 hours vs. 2 minutes, harder
    - What if it takes two weeks to implement the methods?
  - Use mathematics to analyze the algorithm,
    - The implementation is another matter, cache, compiler optimizations, OS, memory,...

Today

What is a list in Java?

- Collection of elements, operations?
  - Add, remove, traverse, ...
  - What can a list do to itself?
  - What can we do to a list?

- Why are there different kinds of lists? Array and Linked
  - Useful in different applications
  - How do we analyze differences?

Next time

Analyze Data Structures

```java
public double removeFirst(List<String> list) {
    double start = System.currentTimeMillis();
    while (list.size() != 1){
        list.remove(0);
    }
    double end = System.currentTimeMillis();
    return (end-start)/1000.0;
}
List<String> linked = new LinkedList<String>();
List<String> array = new ArrayList<String>();
double ltime = splicer.removeFirst(splicer.create(linked,100000));
double atime = splicer.removeFirst(splicer.create(array,100000));

- How much time does it take to remove the first element?
```
Removing first element

<table>
<thead>
<tr>
<th>Size</th>
<th>Link</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.003</td>
<td>0.045</td>
</tr>
<tr>
<td>20</td>
<td>0.001</td>
<td>0.173</td>
</tr>
<tr>
<td>30</td>
<td>0.001</td>
<td>0.383</td>
</tr>
<tr>
<td>40</td>
<td>0.002</td>
<td>0.680</td>
</tr>
<tr>
<td>50</td>
<td>0.002</td>
<td>1.074</td>
</tr>
<tr>
<td>60</td>
<td>0.002</td>
<td>1.530</td>
</tr>
<tr>
<td>70</td>
<td>0.003</td>
<td>2.071</td>
</tr>
<tr>
<td>80</td>
<td>0.003</td>
<td>2.704</td>
</tr>
<tr>
<td>90</td>
<td>0.004</td>
<td>3.449</td>
</tr>
<tr>
<td>100</td>
<td>0.007</td>
<td>4.220</td>
</tr>
</tbody>
</table>

Middle Index Removal

```java
public double removeMiddleIndex(List<String> list) {
    double start = System.currentTimeMillis();
    while (list.size() != 1) {
        list.remove(list.size() / 2);
    }
    double end = System.currentTimeMillis();
    return (end - start) / 1000.0;
}
```

- What operations could be expensive here?
  - Explicit: size, remove
  - Implicit: find \( n \)th element

Remove middle element

<table>
<thead>
<tr>
<th>Size</th>
<th>Link</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.105</td>
<td>0.023</td>
</tr>
<tr>
<td>20</td>
<td>0.472</td>
<td>0.09</td>
</tr>
<tr>
<td>30</td>
<td>0.984</td>
<td>0.192</td>
</tr>
<tr>
<td>40</td>
<td>1.83</td>
<td>0.343</td>
</tr>
<tr>
<td>50</td>
<td>3.026</td>
<td>0.534</td>
</tr>
<tr>
<td>60</td>
<td>4.288</td>
<td>0.767</td>
</tr>
<tr>
<td>70</td>
<td>6.078</td>
<td>1.039</td>
</tr>
<tr>
<td>80</td>
<td>7.885</td>
<td>1.363</td>
</tr>
<tr>
<td>90</td>
<td>9.896</td>
<td>1.663</td>
</tr>
</tbody>
</table>

Jaron Lanier

[http://www.advanced.org/jaron](http://www.advanced.org/jaron)

Jaron Lanier is a computer scientist, composer, visual artist, and author. He coined the term 'Virtual Reality'... he co-developed the first implementations of virtual reality applications in surgical simulation, vehicle interior prototyping, virtual sets for television production, and assorted other areas

"What's the difference between a bug and a variation or an imperfection? If you think about it, if you make a small change to a program, it can result in an enormous change in what the program does. If nature worked that way, the universe would crash all the time."

Lanier has no academic degrees
How fast does the code run?

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage.

Quantitative Measurements of Code

- Typically measure running time. Also measure memory
- Other things to measure?
- What about wall-clock v CPU time? Java: wall-clock

- Typically change size of input/problem to validate runtime hypotheses
- Not the data itself, but the number of data items
- Size of string vs. number of strings in array?

- Doubling hypothesis: What effect does doubling input size have on running time?
  - Linear: time doubles, quadratic: factor of four, ...

Different measures of complexity

- **Worst case**
  - Gives a good upper-bound on behavior
  - Never get worse than this
  - Drawbacks?

- **Average case**
  - What does average mean?
  - Averaged over all inputs? Assuming uniformly distributed random data?
  - Drawbacks?

- **Best case**
  - Linear search, useful?

Notations for measuring complexity

- O-notation or big-Oh: $O(n^2)$ is used in most algorithmic analysis, e.g., in CompSci 130 here. It’s an upper bound in the limit
  - Correct to say that linear algorithm is $O(n^2)$, but useful?

- Theta-notation or $\Theta(n^2)$ is a tight bound, solid guarantee that algorithmic analysis is exact, both upper and lower bound

- Omega is lower bound: $\Omega(n \log n)$ is a lower bound for comparison based sorts
  - Can’t do better than that, very hard to prove

- Sedgewick/Wayne uses tilde notation $\sim n^2$ means leading term is $n^2$
  - We’ll use this, but abuse big-Oh since we want “best” answer
Multiplying and adding big-Oh

- Suppose we do a linear search then we do another one
  - What is the complexity?
  - If we do 100 linear searches?
  - If we do n searches on a vector of size n?

- What if we do binary search followed by linear search?
  - What are big-Oh complexities? Sum?
  - What about 50 binary searches? What about n searches?

- What is the number of elements in the list (1,2,2,3,3,3)?
  - What about (1,2, ..., n,n,...,n)?
  - How can we reason about this?

Helpful formulae

- We always mean base 2 unless otherwise stated
  - What is log(1024)?
  - $\log(xy) = \log(x) + \log(y)$
  - $y^\log(x) = x^{\log(y)}$
  - $n \log(2) = n$
  - $2^{(\log n)} = n$

- Sums (also, use sigma notation when possible)
  - $1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1 = \sum_{i=0}^{2^k}$
  - $1 + 2 + 3 + \ldots + n = n(n+1)/2 = \sum_{i=1}^{n}$
  - $a + ar + ar^2 + \ldots + ar^{n-1} = a(r^n - 1)/(r-1) = \sum_{i=0}^{n-1}$

Why we study recurrences/complexity?

- Tools to analyze algorithms
- Machine-independent measuring methods
- Familiarity with good data structures/algorithms

- What is CS person: programmer, scientist, engineer?
  - scientists build to learn, engineers learn to build

- Mathematics is a notation that helps in thinking, discussion, programming

Recurrences

- Summing Numbers
  ```
  int sum(int n)
  {
    if (0 == n) return 0;
    else return n + sum(n-1);
  }
  ```

- What is complexity? justification?
- $T(n) =$ time to compute sum for $n$
  - $T(n) = T(n-1) + 1$
  - $T(0) = 1$

- instead of 1, use $O(1)$ for constant time
  - independent of $n$, the measure of problem size
Solving recurrence relations

- plug, simplify, reduce, guess, verify?

\[ T(n) = T(n-1) + 1 \]
\[ T(0) = 1 \]
\[ T(n) = T(n-1) = T(n-1) + 1 \]
\[ T(n) = T(n-2) + 1 = T(n-2) + 2 \]
\[ T(n) = T(n-3) + 1 + 1 = T(n-3) + 3 \]

Now, let \( k = n \), then \( T(n) = T(0) + n = 1 + n \)

- get to base case, solve the recurrence: \( O(n) \)

Complexity Practice

- What is complexity of `Build`? (what does it do?)

```java
ArrayList<Integer> build(int n)
{
    if (0 == n) return new ArrayList<Integer>(); // empty
    ArrayList<Integer> list = build(n-1);
    for(int k=0;k < n; k++){
        list.add(n);
    }
    return list;
}
```

- Write an expression for \( T(n) \) and for \( T(0) \), solve.

Recognizing Recurrences

- Solve once, re-use in new contexts
  - \( T \) must be explicitly identified
  - \( n \) must be some measure of size of input/parameter
    - \( T(n) \) is the time for quicksort to run on an \( n \)-element vector

\[ T(n) = T(n/2) + O(1) \] binary search \( O(\log n) \)
\[ T(n) = T(n-1) + O(1) \] sequential search \( O(n) \)
\[ T(n) = 2T(n/2) + O(1) \] tree traversal \( O(n) \)
\[ T(n) = 2T(n/2) + O(n) \] quicksort \( O(n \log n) \)
\[ T(n) = T(n-1) + O(n) \] selection sort \( O(n^2) \)

- Remember the algorithm, re-derive complexity

Sergey Brin

- Simple ideas sometimes can change the world [wikipedia]
  - Works because of scale
  - Co-created pagerank (Larry Page), which evaluates links to a page and the importance of those links, based on the importance of the page from which the links come which …
    - Recursion?
    - Turtles all the way down!