Announcements:
• Please use bug-submit for code problems.
• Watch the newsgroup and class web site for updates, hints, useful new utilities, etc.

Readings for today:  Data Structures (Into Java), Chapter 1;

Readings for next topics:  Data Structures, Chapter 2–4

What are the questions?
• Cost is a principal concern throughout engineering:
  “An engineer is someone who can do for a dime what any fool can do for a dollar.”
• Cost can mean
  - Operational cost (for programs, time to run, space requirements).
  - Development costs: How much engineering time? When delivered?
  - Costs of failure: How robust? How safe?
• Is this program fast enough? Depends on:
  - For what purpose;
  - What input data.
• How much space (memory, disk space)?
  - Again depends on what input data.
• How will it scale, as input gets big?

Enlightening Example
Problem:  Scan a text corpus (say $10^7$ bytes or so), and find and print the 20 most frequently used words, together with counts of how often they occur.

• Solution 1 (Knuth): Heavy-duty data structures
  - Hash Trie implementation, randomized placement, pointers galore, several pages long.

• Solution 2 (Doug McIlroy): UNIX shell script:
  tr -c -s '[:alpha:]' '[\n*]' < FILE | sort | uniq -c | sort -n -r -k 1,1 | sed 20q
• Which is better?
  - #1 is much faster,
  - but #2 took 5 minutes to write and processes 20MB in 1 minute.
  - I pick #2.
• In most cases, anything will do: Keep It Simple.

Cost Measures (Time)
• Wall-clock or execution time
  - You can do this at home:
    time java FindPrimes 1000
  - Advantages: easy to measure, meaning is obvious.
  - Appropriate where time is critical (real-time systems, e.g.).
  - Disadvantages: applies only to specific data set, compiler, machine, etc.

• Number of times certain statements are executed:
  - Advantages: more general (not sensitive to speed of machine).
  - Disadvantages: doesn’t tell you actual time, still applies only to specific data sets.

• Symbolic execution times:
  - That is, formulas for execution times or statement counts in terms of input size.
  - Advantages: applies to all inputs, makes scaling clear.
  - Disadvantage: practical formula must be approximate, may tell very little about actual time.
Asymptotic Cost

• Symbolic execution time lets us see shape of the cost function.
• Since we are approximating anyway, pointless to be precise about certain things:
  - Behavior on small inputs:
    * Can always pre-calculate some results.
    * Times for small inputs not usually important.
  - Constant factors (as in “off by factor of 2”):
    * Just changing machines causes constant-factor change.
• How to abstract away from (i.e., ignore) these things?

Handy Tool: Order Notation

• Idea: Don’t try to produce specific functions that specify size, but rather families of similar functions.
• Say something like "f is bounded by g if it is in g’s family.”
• For any function $g(x)$, the functions $2g(x)$, $1000g(x)$, or for any $K > 0$, $K \cdot g(x)$, all have the same “shape”. So put all of them into $g$’s family.
• Any function $h(x)$ such that $h(x) = K \cdot g(x)$ for $x > M$ (for some constant $M$) has $g$’s shape “except for small values.” So put all of these in $g$’s family.
• If we want upper limits, throw in all functions that are everywhere $\leq$ some other member of $g$’s family. Call this family $O(g)$ or $O(g(n))$.
• Or, if we want lower limits, throw in all functions that are everywhere $\geq$ some other member of $g$’s family. Call this family $\Omega(g)$.
• Finally, define $\Theta(g) = O(g) \cap \Omega(g)$—the set of functions bracketed by members of $g$’s family.

Big Oh

• Goal: Specify bounding from above.

Here, $f(x) \leq 2g(x)$ as long as $x > 1$.
So $f(x)$ is in $g$’s upper-bound family, written $f(x) \in O(g(x))$.
...even though $f(x) > g(x)$ everywhere.

Big Omega

• Goal: Specify bounding from below:

Here, $f'(x) \geq \frac{1}{2}g(x)$ as long as $x > 1$.
So $f'(x)$ is in $g$’s lower-bound family, written $f'(x) \in \Omega(g(x))$.
...even though $f(x) < g(x)$ everywhere.
In fact, we also have $f'(x) \in O(g(x))$ and $f(x) \in \Omega(g(x))$ and so we can also write $f(x), f'(x) \in \Theta(g(x))$. 
Why It Matters

- Computer scientists often talk as if constant factors didn’t matter at all, only the difference of $\Theta(N)$ vs. $\Theta(N^2)$.
- In reality they do, but we still have a point: at some point, constants get swamped.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$16 \lg n$</th>
<th>$\sqrt{n}$</th>
<th>$n$</th>
<th>$n \lg n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>1.4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>48</td>
<td>2.8</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>64</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65636</td>
</tr>
<tr>
<td>32</td>
<td>80</td>
<td>5.7</td>
<td>32</td>
<td>160</td>
<td>32768</td>
<td>4.2e9</td>
<td>4.2e9</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
<td>8</td>
<td>64</td>
<td>384</td>
<td>262144</td>
<td>1.8e19</td>
<td>1.8e19</td>
</tr>
<tr>
<td>128</td>
<td>112</td>
<td>11</td>
<td>128</td>
<td>896</td>
<td>1.6e10</td>
<td>3.4e18</td>
<td>3.4e18</td>
</tr>
<tr>
<td>1024</td>
<td>160</td>
<td>32</td>
<td>1024</td>
<td>1.0e6</td>
<td>1.1e10</td>
<td>1.8e30</td>
<td>1.8e30</td>
</tr>
<tr>
<td>20</td>
<td>320</td>
<td>1024</td>
<td>2.1e7</td>
<td>1.1e12</td>
<td>1.2e18</td>
<td>6.7e31</td>
<td>6.7e31</td>
</tr>
</tbody>
</table>

How big a problem can you solve in a given time?

- Entries show the size of problem that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.

<table>
<thead>
<tr>
<th>Time (µsec) for problem size $N$</th>
<th>1 second</th>
<th>1 hour</th>
<th>1 month</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg N$</td>
<td>$10^{200000}$</td>
<td>$10^{1000000000}$</td>
<td>$10^8 \cdot 10^{11}$</td>
<td>$10^9 \cdot 10^{14}$</td>
</tr>
<tr>
<td>$N$</td>
<td>$10^6$</td>
<td>$3.6 \cdot 10^9$</td>
<td>$2.7 \cdot 10^{12}$</td>
<td>$3.2 \cdot 10^{15}$</td>
</tr>
<tr>
<td>$N \lg N$</td>
<td>63000</td>
<td>$1.3 \cdot 10^8$</td>
<td>$7.4 \cdot 10^{10}$</td>
<td>$6.9 \cdot 10^{13}$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>10000</td>
<td>60000</td>
<td>$1.6 \cdot 10^6$</td>
<td>$5.6 \cdot 10^7$</td>
</tr>
<tr>
<td>$N^3$</td>
<td>100</td>
<td>1500</td>
<td>14000</td>
<td>150000</td>
</tr>
<tr>
<td>$2^N$</td>
<td>20</td>
<td>32</td>
<td>41</td>
<td>51</td>
</tr>
</tbody>
</table>

Some Intuition on Meaning of Growth

- It’s also true that the worst-case time is $O(N^2)$, since $N \in O(N^2)$ also: Big-Oh bounds are loose.
- The worst-case time is $\Omega(N)$, since $N \in \Omega(N)$, but that does not mean that the loop always takes time $N$, or even $K \cdot N$ for some $K$.
- Instead, we are just saying something about the function that maps $N$ into the largest possible time required to process an array of length $N$.
- To say as much as possible about our worst-case time, we should try to give a $\Theta$ bound: in this case, we can: $\Theta(N)$.
- But again, that still tells us nothing about best-case time, which happens when we find X at the beginning of the loop. Best-case time is $\Theta(1)$.

Using the Notation

- Careful!

<table>
<thead>
<tr>
<th>Time (µsec) for problem size $N$</th>
<th>1 second</th>
<th>1 hour</th>
<th>1 month</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg N$</td>
<td>$10^{200000}$</td>
<td>$10^{1000000000}$</td>
<td>$10^8 \cdot 10^{11}$</td>
<td>$10^9 \cdot 10^{14}$</td>
</tr>
<tr>
<td>$N$</td>
<td>$10^6$</td>
<td>$3.6 \cdot 10^9$</td>
<td>$2.7 \cdot 10^{12}$</td>
<td>$3.2 \cdot 10^{15}$</td>
</tr>
<tr>
<td>$N \lg N$</td>
<td>63000</td>
<td>$1.3 \cdot 10^8$</td>
<td>$7.4 \cdot 10^{10}$</td>
<td>$6.9 \cdot 10^{13}$</td>
</tr>
<tr>
<td>$N^2$</td>
<td>10000</td>
<td>60000</td>
<td>$1.6 \cdot 10^6$</td>
<td>$5.6 \cdot 10^7$</td>
</tr>
<tr>
<td>$N^3$</td>
<td>100</td>
<td>1500</td>
<td>14000</td>
<td>150000</td>
</tr>
<tr>
<td>$2^N$</td>
<td>20</td>
<td>32</td>
<td>41</td>
<td>51</td>
</tr>
</tbody>
</table>
Effect of Nested Loops

- Nested loops often lead to polynomial bounds:
  for (int i = 0; i < A.length; i += 1)
  for (int j = 0; j < A.length; j += 1)
    if (A[i] == A[j]) return true;
  return false;

- Clearly, time is $O(N^2)$, where $N = A$.length. Worst-case time is $\Theta(N^2)$.

- Loop is inefficient though:
  for (int i = 0; i < A.length; i += 1)
    for (int j = i+1; j < A.length; j += 1)
      if (A[i] == A[j]) return true;
  return false;

- Now worst-case time is proportional to
  
  $N - 1 + N - 2 + \ldots + 1 = N(N - 1)/2 \in \Theta(N^2)$

(assume asymptotic time unchanged by the constant factor).

Recursion and Recurrences: Fast Growth

- Silly example of recursion:
  ```java
  /** True iff X is a substring of S */
  boolean occurs (String S, String X) {
    if (S.equals (X)) return true;
    if (S.length () <= X.length ()) return false;
    return
    if (i != j && A[i] == A[j]) return true;
    return false;
  }
  ```

- In the worst case, both recursive calls happen.

- Define $C(N)$ to be the worst-case cost of `occurs(S,X)` for $S$ of length $N$, $X$ of fixed size $N_0$, measured in # of calls to `occurs`.

  - Then $C(N) = \begin{cases} 1, & \text{if } N \leq N_0, \\ 2C(N-1) + 1 & \text{if } N > N_0 \end{cases}$

  - So $C(N)$ grows exponentially:
    
    $C(N) = 2C(N-1) + 1 = 2(2\ldots2 + 1) + 1 = 2\ldots2 \cdot 1 + 1 + 1 = 2^N - N_0 - 1 + 1 = 2^N - 1 \in \Theta(2^N)$

Another Typical Pattern: Merge Sort

- Assuming that size of $L$ is $N = 2^k$, worst-case cost function, $C(N)$, counting just merge time (% # items merged):

  $C(N) = \begin{cases} 1, & \text{if } N < 2; \\ 2C(N/2) + N, & \text{if } N \geq 2, \end{cases}$

  $= 2(2C(N/4) + N/2) + N$

  $= 4C(N/4) + N + N$

  $= 8C(N/8) + N + N + N$

  $= N \cdot 1 + N + N + \ldots + N$

  $= N + N \lg N \in \Theta(N \lg N)$

- In general, $\Theta(N \lg N)$ for arbitrary $N$ (not just $2^k$).
Amortization: Expanding Vectors

- When using array for expanding sequence, best to double size of array to grow it. Here’s why.
- If array is size $s$, doubling its size and moving $s$ elements to the new array takes time $\propto 2s$.
- Cost of inserting $N$ items into array, doubling size as needed, starting with array size 1:

<table>
<thead>
<tr>
<th>To Insert</th>
<th>Resizing Cost</th>
<th>Cumulative Cost</th>
<th>Resizing Cost per Item</th>
<th>Array Size After Insertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item #</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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 to 4</td>
<td>4</td>
<td>6</td>
<td>1.5</td>
<td>4</td>
</tr>
<tr>
<td>5 to 8</td>
<td>8</td>
<td>14</td>
<td>1.75</td>
<td>8</td>
</tr>
<tr>
<td>$2^m + 1$ to $2^{m+1}$</td>
<td>$2^m+1$</td>
<td>$2^m+2 - 2$</td>
<td>$\approx 2$</td>
<td>$2^{m+1}$</td>
</tr>
</tbody>
</table>

- If we spread out (amortize) the cost of resizing, we average about 2 time units on each item: “amortized insertion time is 2 units.”
- So even though worst-case time for adding one element to array of $N$ elements is $2N$, time to add $N$ elements is $\Theta(N)$, not $\Theta(N^2)$.

Demonstrating Amortized Time: Potential Method

- To formalize the argument, associate a potential, $\Phi_i \geq 0$, to the $i^{th}$ operation that keeps track of "saved up" time from cheap operations that we can "spend" on later expensive ones. Start with $\Phi_0 = 0$.
- Now define the amortized cost of the $i^{th}$ operation as $a_i = c_i + \Phi_{i+1} - \Phi_i$, where $c_i$ is the real cost of the operation.
- On cheap operations, we artificially set $a_i > c_i$ and increase $\Phi$ ($\Phi_{i+1} > \Phi_i$).
- On expensive ones, we typically have $a_i \ll c_i$ and greatly decrease $\Phi$ (but don’t let it go negative—may not be “overdrawn”).
- We try to do all this so that $a_i$ remains as we desired (e.g., $O(1)$ for expanding array), without allowing $\Phi_i < 0$.
- Requires that we choose $a_i$ so that $\Phi_i$ always stays ahead of $c_i$.

Application to Expanding Arrays

- When adding to our array, the cost, $c_i$, of adding element #i when the array already has space for it is 1 unit.
- The array does not initially have space when adding items 1, 2, 4, 8, 16, ... in other words at item $2^n$ for all $n \geq 0$. So,
  - $c_i = 1$ if $i \geq 0$ and is not a power of 2; and
  - $c_i = 3i + 1$ (allocate $2i$ items, copy $i$ items, and then add item #i) when $i$ is a power of 2.
- So on each operation #$2^n$ we’re going to need to have saved up at least $3 \cdot 2^{n-1}$ units of potential to cover the expense, and we have the preceding $2^{n-1}$ operations to do it (the ones since the preceding doubling operation).
- To do so, just choose $a_i = 7$ (or could let $a_0 = 1, a_1 = 4$)
- Here’s what happens:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_i$</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$a_i$</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>$\Phi_i$</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>15</td>
<td>9</td>
<td>15</td>
<td>21</td>
<td>27</td>
<td>29</td>
<td>15</td>
<td>21</td>
<td>27</td>
<td>33</td>
<td>39</td>
<td>45</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>