CS61B Lecture #16

Announcements:

• CSUA help session on Subversion today 4-6PM in the Wozniak lounge (north side of Soda on 4th floor).

• 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, Head First Java, Chapter 16.
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:
  ```bash
  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 results 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.

\[ M = 1 \]

\[ f(x) \leq 2g(x) \text{ as long as } x > 1, \]

• 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:

  \[ g(x) = f'(x) \]

  \[ M = 1 \]

  \[ M = 1 \]

- 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

  \[ f(x), f'(x) \in \Theta(g(x)). \]
Using the Notation

- Can use this order notation for any kind of real-valued function.
- We will use them to describe cost functions. Example:

```java
/** Find position of X in list L. Return -1 if not found */
int find (List L, Object X) {
    int c;
    for (c = 0; L != null; L = L.next, c += 1)
        if (X.equals (L.head)) return c;
    return -1;
}
```

- Choose representative operation: number of .equals tests.
- If \( N \) is length of \( L \), then loop does at most \( N \) tests: worst-case time is \( N \) tests.
- In fact, total # of instructions executed is roughly proportional to \( N \) in the worst case, so can also say worst-case time is \( O(N) \), regardless of units used to measure.
- Use \( N > M \) provision (in defn. of \( O(\cdot) \)) to handle empty list.
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>4,096</td>
<td>65,636</td>
</tr>
<tr>
<td>32</td>
<td>80</td>
<td>5.7</td>
<td>32</td>
<td>160</td>
<td>1024</td>
<td>32,768</td>
<td>$4.2 \times 10^9$</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
<td>8</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>$1.8 \times 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>112</td>
<td>11</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>$2.1 \times 10^9$</td>
<td>$3.4 \times 10^{38}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>1,024</td>
<td>160</td>
<td>32</td>
<td>1,024</td>
<td>10,240</td>
<td>$1.0 \times 10^6$</td>
<td>$1.1 \times 10^9$</td>
<td>$1.8 \times 10^{308}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>320</td>
<td>1,024</td>
<td>$1.0 \times 10^6$</td>
<td>$2.1 \times 10^7$</td>
<td>$1.1 \times 10^{12}$</td>
<td>$1.2 \times 10^{18}$</td>
<td>$6.7 \times 10^{315,652}$</td>
</tr>
</tbody>
</table>
Careful!

- 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 $\Theta(1)$. 
Effect of Nested Loops

• Nested loops often lead to polynomial bounds:
  ```java
  for (int i = 0; i < A.length; i += 1)
      for (int j = 0; j < A.length; j += 1)
          if (i != j && 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:
  ```java
  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
  $$N - 1 + N - 2 + \ldots + 1 = N(N - 1)/2 \in \Theta(N^2)$$
  (so 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
        occurs (S.substring (1), X) ||
        occurs (S.substring (0, S.length ()-1), X);
}
```

- In the worst case, both recursive calls happen.

- Consider a fixed size for X, say $N_0$.

- Define $C(N)$ to be the worst-case cost of $\text{occurs}(S,X)$ for $S$ of length $N$, measured in # of calls to $\text{occurs}$. Then

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

- So $C(N)$ grows exponentially:

\[
C(N) = 2C(N - 1) = 2 \cdot 2C(N - 2) = \ldots = 2 \cdot 2 \cdot \ldots \cdot 2 \cdot 1 = 2^{N-N_0} \in \Theta(2^N)
\]
Binary Search: Slow Growth

/** True X iff is an element of S[L .. U]. Assumes
  * S in ascending order, 0 <= L <= U-1 < S.length. */
boolean isIn (String X, String[] S, int L, int U) {
  if (L > U) return false;
  int M = (L+U)/2;
  int direct = X.compareTo (S[M]);
  if (direct < 0) return isIn (X, S, L, M-1);
  else if (direct > 0) return isIn (X, S, M+1, U);
  else return true;
}

• Here, worst-case time, $C(D)$, (as measured by # of string comparisons), depends on size $D = U - L + 1$.

• We eliminate $S[M]$ from consideration each time and look at half the rest. Assume $D = 2^k - 1$ for simplicity, so:

\[
C(D) = \begin{cases} 
0, & \text{if } D \leq 0, \\
1 + C((D - 1)/2), & \text{if } D > 0.
\end{cases}
\]

\[
= 1 + 1 + \ldots + 1 + 0
\]

\[
= k = \lceil \log D \rceil \in \Theta(\log D)
\]
Another Typical Pattern: Merge Sort

List sort (List L) {
  if (L.length () < 2) return L;
  Split L into L0 and L1 of about equal size;
  L0 = sort (L0);  L1 = sort (L1);
  return Merge of L0 and L1
}

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

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

\[
= 2(2C(N/4) + N/2) + N
\]
\[
= 4C(N/4) + N + N
\]
\[
= 8C(N/8) + N + N + N
\]
\[
= N \cdot 1 + \underbrace{N + N + \ldots + N}_{k=\lg N}
\]
\[
= N + N \lg N \in \Theta(N \lg N)
\]

• 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 Item #</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>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 \( \Theta(N^2) \).