Measuring the performance of algorithms

- **How fast is fast enough?**

  ```cpp
  int search(const Vector<string> & names, const string & key)
  // pre:  names contains names.size() entries
  // post: return index of first occurrence of key in names
  //       return -1 if not found
  {
    int k; int len = names.size();
    for(k=0; k < len; k++)
      if (names[k] == key) return k;
    return -1;
  }
  ```

- **In “early years”, time this function on vectors of different sizes, compute a measure for a specific machine/architecture**

- **In current times: how fast on P100? P200? ultrasparc?**
Timing code/theoretical analysis

- We can time code segments with CTimer class from “ctimer.h” (see search.h, search.cc, dotiming.cc)
- We can analyze the function/algorithm using tools from mathematics

- What kind of data should we test with?
- What kind of assumptions can we make to do the mathematical analysis?
From Practice to Theory

- How many string comparisons made in search()?
  - worst case
  - average case

- Is it enough to count string comparisons?
  - arithmetic operations/comparisons
  - other code

- Is there a noticeable pattern in the timing results?
## Searching for all words in several files

<table>
<thead>
<tr>
<th>Platform</th>
<th>Word</th>
<th>Time (s)</th>
<th>Error (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gnu, linux, P100</td>
<td>poe</td>
<td>1.41</td>
<td>0.0006</td>
</tr>
<tr>
<td></td>
<td>melv.</td>
<td>37.86</td>
<td>0.0026</td>
</tr>
<tr>
<td></td>
<td>hamlet</td>
<td>162.45</td>
<td>0.0051</td>
</tr>
<tr>
<td></td>
<td>hawth.</td>
<td>888.23</td>
<td>0.0104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform</th>
<th>Word</th>
<th>Time (s)</th>
<th>Error (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gnu, solarisX86/P200</td>
<td>poe</td>
<td>0.81</td>
<td>0.0004</td>
</tr>
<tr>
<td></td>
<td>melv.</td>
<td>24.02</td>
<td>0.0017</td>
</tr>
<tr>
<td></td>
<td>hamlet</td>
<td>105.31</td>
<td>0.0033</td>
</tr>
<tr>
<td></td>
<td>hawth.</td>
<td>601.7</td>
<td>0.0070</td>
</tr>
</tbody>
</table>

- Load all words, then search for each word once
- Which machine is best?
- Determine times for *Romeo and Juliet*, 25788 words?
- Trends?
- Melville on ultrasparc is 30.09, 0.0021; *hamlet, hawth*?
- Generalize the analysis
P100 Timings

- rough shape of each curve?
  - all word search: quadratic
  - per word search: linear

- Analyze all word search

\[ \text{#comparisons} = 1 + 2 + \ldots + n \]
when searching \( n \) words

\[ \text{total is } n(n+1)/2 \]

practice using \( \sum \) notation

- what about curve? equation?
- use of big-Oh notation
Towards big-Oh

- family of quadratics
  - $y = x^2$
  - $y = 0.5 \times x^2$
  - $y = x^2 + 5$
- disregard constants
- disregard coefficients
- disregard low-order terms

- in the limit this is ok, gives a good yardstick
Determining complexity with big-Oh

- runtime, space \textit{complexity} refers to mathematical notation for algorithm (not really to code, but ok)

- typical examples:

```c
sum = 0;
for(k=0; k < n; k++)
{
    if (a[k] == key) sum++;
}
return sum;
```

```c
for(k=0; k < n; k++)
{
    min = k;
    for(j=k+1; j < n; j++)
    {
        if (a[j] < a[min]) min = j;
        Swap(a[min], a[k]);
    }
}
```

- what are complexities of these?
Recurrences

- **Counting nodes**

  ```c
  int length(Node * list)    
  {    
      if (0 == list) return 0;    
      else return 1 + length(list->next);    
  }
  ```

- **What is complexity? justification?**

- **T(n) = time to compute length for an n-node list**

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

- **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?**

\[
\begin{align*}
T(n) &= T(n-1) + 1 \\
T(0) &= 1 \end{align*}
\]

\[
\begin{align*}
T(n) &= [T(n-2) + 1] + 1 \\
&= [(T(n-3) + 1) + 1] + 1 \\
&= T(n-k) + k \quad \text{find the pattern!}
\end{align*}
\]

- **get to base case, solve the recurrence**
Why must 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
Complexity Practice

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

```cpp
Node * Build(int n) {
    if (0 == list) return 0;
    else {
        Node * first = new Node(n,Build(n-1));
        for(int k = 0; k < n-1; k++)
            first = new Node(n,first->next);
        return first;
    }
}
```

- Write an expression for $T(n)$ and for $T(0)$, solve.
Makefile information

- A Makefile/makefile is a collection of
  - targets, dependencies, rules
  - information on how to build programs
  - information on where files are located

```
clean:
    /bin/rm -f *.o *~

doana: $(OBJ)
    $(CXX) $(CXXFLAGS) -o $@ $(OBS) $(TLIB)
```

- tracking dependencies is important, minimizes recompiles, can be automated
- minimal knowledge needed for wordtrack