Measuring performance of algorithms

- How fast is fast enough?
  ```cpp
  bool search(const tvector<string> & a, const string & key)
  // pre: a contains a.size() entries
  // post: return true if and only if key found in a
  {
    int k; int len = a.size();
    for(k=0; k < len; k++)
      if (a[k] == key) return true;
    return false;
  }
  ```
- 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 P300, P500, G4, ultraspacc?
  - Does compiler make a difference? Other factors?

Empirical and Theoretical results

- We run the program to time it (e.g., using CTimer)
  - What do we use as data? Why does this matter
  - What do we do about the kind of machine being used?
  - What about the load on the machine?
- Use mathematical/theoretical reasoning:
  - The algorithm takes time linear in the size of the vector
  - Double size of vector means double the time
  - What about searching twice, is this still linear?
  - We use big-Oh, a mathematical notation for describing a class of functions

What is big-Oh about?

- A family of functions that are “the same in the limit”
  - For polynomials, use only leading term, ignore coefficients
    
    \[
    y = x^2 \quad y = x^2 - 6x + 9 \quad y = 3x^2 + 4x
    \]
  - What do these look like when graphed? What about quotient of one divided by the other, in the limit?
- Running time for input of size \( N \) is \( 10N \) or \( 2N \) or \( N + 1 \)
  - Running time is \( O(N) \), call this linear
  - Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm

Running times @ \( 10^6 \) instructions/sec

<table>
<thead>
<tr>
<th>( N )</th>
<th>( O(N) )</th>
<th>( O(N) )</th>
<th>( O(N \log N) )</th>
<th>( O(N^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000001</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000015</td>
<td>0.00100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>
Determining complexity with big-Oh

- Runtime, space 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++;
}
```

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

- 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?
  ```c
  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
  ```c
  ➤ independent of n, the measure of problem size
  ```

Solving recurrence relations

- Plug, simplify, reduce, guess, verify?
  ```c
  T(n) = T(n-1) + 1
  T(0) = 1
  T(n) = [T(n-2) + 1] + 1
      = [T(n-3) + 2 + 1] + 1
      = T(n-k) + k  find the pattern!
  ```

- Get to base case, solve the recurrence

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?
  ```c
  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?)

```c
Node * Build(int n)
{
    if (0 == n) 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.

Consider a modification to MultiSet

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right

  > How do we search?
  > How do we insert?

  > How are lists and trees related?

How do we print all values in a tree?

- When is root printed?
  - After left subtree, before right subtree.

```c
void Visit(Node * t)
{
    if (t != 0)
    {
        Visit(t->prev); cout << t->info << endl; Visit(t->next);
    }
}
```

- Inorder traversal

Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?

- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?

- How do we define worst and average cases
  - Compare to multiset README, what is worst case for different implementations?
  - What are complexities of multiset implementations?
Binary Trees

- Linked lists have efficient insertion and deletion, but inefficient search
  - arrays: search is efficient, insertion and deletion are not
- Binary trees are structures that can be used to yield efficient insertion/deletion and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - insertion is as efficient as binary search in array, insertion/deletion as efficient as linked list (once node found)
  - binary trees are inherently recursive, difficult to process trees non-recursively, but possible (recursion never required, but often makes coding/algorithms simpler)

Binary trees (continued)

- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node N to M then N is parent of M
  - M is child of N
  - leaf node has no children
  - internal node has 1 or 2 children
  - path is sequence of nodes, N₀, N₁, ..., Nₖ
  - Nᵢ is parent of Nᵢ₊₁
  - sometimes edge instead of node
  - depth (level) of node: length of root-to-node path
  - level of root is 1
  - height of node: length of longest node-to-leaf path
  - height of tree is height of root

Binary trees (continued)

- Trees can have many shapes: short/bushy, long/stringy
  - if height is h, number of nodes is between 2ʰ⁻¹ and 2ʰ⁻¹
  - single node tree: height = 1, if height = 3

  ![Binary tree diagram]

- C++ implementation, similar to doubly-linked list

```cpp
struct Tree
{
    string info;
    Tree * left;
    Tree * right;
};
```

Tree functions

- Compute height of a tree, what is complexity?
  ```cpp
  int height(Tree * root)
  {
      if (root == 0) return 0;
      else
      {
          return 1 + max(height(root->left),
                          height(root->right));
      }
  }
  ```

- Modify function to compute number of nodes in a tree, does complexity change?
  - What about computing number of leaf nodes?
Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root

Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one

```cpp
bool isBalanced(TreeNode *root) {
    if (root == 0) return true;
    else {
        return isBalanced(root->left) &&
               isBalanced(root->right) &&
               abs(height(root->left) - height(root->right)) <= 1;
    }
}
```

What is complexity?

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?

- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify that pattern is correct

sidebar: solving recurrence

\[
T(n) = 2T(n/2) + O(n) \quad T(1) = 1
\]

What about 2n? 3n?

\[
T(n) = 2[2T(n/4) + n/2] + n
= 4T(n/4) + n + n
= 4[8T(n/8) + n/4] + 2n
= 8T(n/8) + 3n
= \ldots \text{eureka!}
= 2^kT(n/2^k) + kn
\]

Let \(2^k = n\)

\[
k = \log n, \text{ this yields } 2^{\log n}T(n/2^{\log n}) + n(\log n) = nT(1) + n(\log n) = O(n \log n)
\]