Admin

- Today's topics
  - Algorithm analysis, big-O notation, intro to sorting
- Reading
  - Ch 7
- Midterm next Tuesday evening
  - Terman Aud 7-9pm

Algorithm analysis

- Problems can often be solved multiple ways
  - Work forward or backward, iteration or recursion, vector or set, precise answer vs. estimation
  - e.g. sorting, searching, counting, ...
- How to evaluate/compare alternatives?
  - Often interested in execution performance
    - Time spent and memory used
    - Should also consider ease of developing, verifying, maintaining code
  - Be sure complexity is worthwhile!
- Brainstorm: algorithm to count people in this room

Evaluating performance

- Empirically—time with stopwatch
  - + Practical, real life results
  - - Have to write & debug code to test it
  - - Subject to variation (hardware, OS, other activity, etc.)
- Mathematically— analyze algorithm
  - + Can analyze without implementing code
  - + Abstract setting, not tied to specific environment
  - - Empirical results may not exactly match (esp for small size inputs)

Statement counts

```c
double CtoF(double cTemp)
{
    return cTemp*9.0/5.0 + 32;
}
```

- Each statement costs 1¢
- Multiply, divide, add, return
- Total = 4¢
- Does value of input matter?
More statement counts

double Average(Vector<int> &v)
{
    int sum = 0;
    for (int i = 0; i < v.size(); i++)
    {
        sum += v[i];
    }
    return double(sum)/v.size();
}

Count statements
- Outside loop: 4 statements (init sum, init i, divide, return)
- Loop body: 3 statements (test, add, incr i) per vector element
- Total = 3N + 4

Does input matter?
- Double size of vector -- how change time required?

More statement counts

void GetExtremes(Vector<int> &v, int &min, int &max)
{
    min = max = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] > max) max = v[i];
    }
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min) min = v[i];
    }
}

Count statements
- Loop: test i, compare, update, incr i 4 per iteration * N iterations
- 2 loops
- Outside loop: init i, init min/max
- 4 + 8N

Comparing algorithms

Summarize statement counts
- Use only largest term, ignore others, drop all coefficients
- Time = 3n + 5    -> O(n)
- Time = 10n - 2   -> O(n)
- Time = 1/2n^2 - n -> O(n^2)
- Time = 2^n + n^3 -> O(2^n)

Describe growth curve of algorithm in the limit
- Intuition: avoid details when they don’t matter, and they don’t matter
  when input size (N) is big enough

More formally:
- O(f(n)) is an upper-bound on the time required
- Time(n) <= C(f(n) for some constant C and sufficiently large value n

Big-O notation

More statement counts

void GetExtremes(Vector<int> &v, int &min, int &max)
{
    min = max = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] > max) max = v[i];
    }
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min) min = v[i];
    }
}
Using big-O to predict times

◇ For an O(n) algorithm:
  • 5,000 elements takes 3.2 seconds
  • 10,000 elements takes 6.4 seconds
  • 20,000 elements takes ...?

◇ For an O(n^2) algorithm:
  • 5,000 elements takes 2.4 seconds
  • 10,000 elements takes 9.6 seconds
  • 20,000 elements takes ...?

Best-worst-average case

```cpp
bool Search(Vector<string> &names, string key) {
    for (int i=0; i < names.size(); i++)
        if (names[i] == key) return true;
    return false;
}
```

◇ What if key is first? middle? last? What if not found?

◇ Best case
  • Super-fast in some situations, often not that valuable

◇ Worst case
  • Upper bound on how bad it can get

◇ Average case
  • Averaged over all possible inputs, can be harder to compute precisely

Analyzing recursive algorithms

```cpp
int Factorial(int n) {
    if (n == 0) return 1;
    else return n * Factorial(n-1);
}
```

◇ T(n) is time used for input n
  \[ T(n) = \begin{cases} 
    1 & \text{if } n = 0 \\
    1 + T(n-1) & \text{otherwise}
  \end{cases} \]

◇ This is a recurrence relation

Solving recurrences

◇ Repeated substitution expands recurrence
  \[ T(n) = 1 + T(n-1) \]
  \[ = 1 + (1 + T(n-2)) \]
  \[ = 1 + (1 + (1 + T(n-3))) \]
  ...

◇ Generalize pattern
  \[ = i^*1 + T(n-i) \]

◇ Solve for \( i = n \)
  \[ = n + T(0) \]
  \[ = n + 1 \]
  \[ \Rightarrow O(n) \]
Another example

```c
void MoveTower(int n, char src, char dst, char tmp)
{
    if (n > 0) {
        MoveTower(n-1, src, tmp, dst);
        MoveOneDisk(src, dst);
        MoveTower(n-1, tmp, dst, src);
    }
}
```

Set up recurrence

\[ T(n) = \begin{cases} 
    1 & \text{if } n = 0 \\
    1 + 2T(n-1) & \text{otherwise} 
\end{cases} \]

Solving recurrences

- Repeated substitution
  \[ T(n) = 1 + 2T(n-1) \]
  \[ = 1 + (2 + 4T(n-2)) \]
  \[ = 1 + (2 + (4 + 8T(n-3))) \]
  \[ \vdots \]
- Generalize pattern
  \[ = 1 + 2 + 4 + 8 + 16 + \ldots + 2^{i-1} + 2^iT(n-i) \]
- Solve for \( n-i = 0 \) (i = n)
  \[ = 2^n - 1 + 2^nT(0) \]
  \[ = 2^{n+1} - 1 \]
  \[ = 2^{n+1} \Rightarrow O(2^n) \]

Growth patterns

<table>
<thead>
<tr>
<th>N</th>
<th>O(lgN)</th>
<th>O(N)</th>
<th>O(NlgN)</th>
<th>O(N^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</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>0.000010</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>

10^6 instr/sec runtimes

- O(1)
- O(logN)
- O(N)
- O(NlogN)
- O(N^2)
- O(2^N)
- O(N)
- O(logN)
- O(1)

```

Growth patterns

\[ O(1) \]
\[ O(\log N) \]
\[ O(N) \]
\[ O(N \log N) \]
\[ O(N^2) \]
\[ O(2^N) \]
Sorting!

- Very common to need data in order
  - Viewing, printing
  - Faster to search, find min/max, compute median/mode, etc.
- Lots of different sorting algorithms
  - From the simple to very complex
  - Some optimized for certain situations (lots of duplicates, almost sorted, etc.)
  - Typically sort array/vector, but algorithms usually can be adapted for other data structures (e.g. linked list)

Selection sort

- Select smallest and move to front
  - Search to find minimum
  - Place in first slot
  - Could move elements over to make space, but faster to just swap with current first
  - Repeat for second smallest, third, so on

Selection sort code

```cpp
void SelectionSort(Vector<int> &arr)
{
    for (int i = 0; i < arr.size()-1; i++) {
        int minIndex = i;
        for (int j = i+1; j < arr.size(); j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        Swap(arr[i], arr[minIndex]);
    }
}
```