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

```
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];
   }
}
```

- 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

- Count statements
  - CtoF (4)   Average (3N+4)   GetExtremes(8N + 4)
  - Will GetExtremes always take more time than Average or CToF?
- Consider growth patterns
  - If Average takes 2 ms for 1000 elems, estimate for 2000 or 10000?
  - What about GetExtremes?

Big-O notation

- 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) <= Cf(n) for some constant C and sufficiently large value n

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

```
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

```
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) = { 1 if n is 0
  1 + T(n-1) otherwise }
- 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 => O(n)

Another example

```
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

  $T(n) = \{ 1$ if $n = 0$

  $1 + 2T(n-1)$ otherwise }

Solving recurrences

- Repeated substitution

  $T(n) = 1 + 2T(n-1)$

  $= 1 + (2 + 4T(n-2))$

  $= 1 + (2 + (4 + 8T(n-3)))$

  ...

- Generalize pattern

  $= 1+2+4+8+16+\ldots+ 2^{i-1} + 2^{i}T(n - i)$

  $= 2^{i} - 1+ 2^{i}T(n - i)$

- Solve for $n-i = 0$ $(i = n)$

  $= 2^{n} - 1 + 2^{n} T(0)$

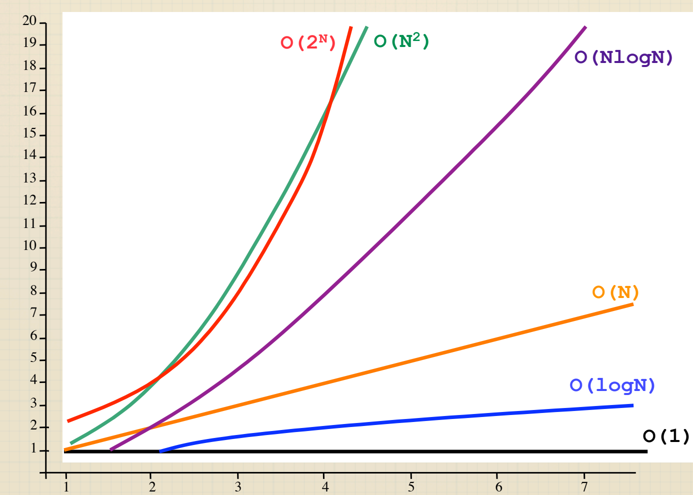
  $= 2^{n+1} - 1$

  $= 2^{n+1} \Rightarrow O(2^{n})$

10^6 instr/sec runtimes

N	O(lgN)	O(N)	O(NlgN)	O(N²)
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

Growth patterns



Sorting!

- Very common to need data in order
  - Viewing, printing
  - Faster to search, find min/max, compute median/mode, etc.
- Lots of different sorting algoritms
  - 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

```
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]);
  }
```