

Analyzing Algorithms

- Consider three solutions to SortByFreqs, also code used in Anagram assignment
 - Sort, then scan looking for changes
 - Insert into Set, then count each unique string
 - Find unique elements without sorting, sort these, then count each unique string
- We want to discuss trade-offs of these solutions
 - Ease to develop, debug, verify
 - Runtime efficiency
 - Vocabulary for discussion

Cost

"An engineer is someone who can do for a dime what any fool can do for a dollar."

- Types of costs:
 - Operational
 - Development
 - Failure
- Is this program fast enough? What's your purpose? What's your input data?
- How will it *scale*?
- Measuring cost
 - Wall-clock or execution time
 - Number of times certain statements are executed
 - Symbolic execution times
 - Formula for execution time in terms of *input size*
 - Advantages and disadvantages?

Data processing example

- Scan a large (~ 10⁷ bytes) file
- Print the 20 most frequently used words together with counts of how often they occur
- Need more specification?
- How do you do it?

Possible solutions

1. Use heavy duty data structures (Knuth)
 - Hash tries implementation
 - Randomized placement
 - Lots o' pointers
 - Several pages
 2. UNIX shell script (Doug McIlroy)

```
tr -cs "[:alpha:]" "\\n*" < FILE | \  
sort | \  
uniq -c | \  
sort -n -r -k 1,1 | \  
head -20
```
- Which is better?
 - K.I.S.S.?



Dropping Glass Balls

- Tower with N Floors
- Given 2 glass balls
- Want to determine the *lowest* floor from which a ball can be dropped and will break
- How?

- What is the most efficient algorithm?
- How many drops will it take for such an algorithm (as a function of N)?

CPS 100

6.5

Glass balls revisited

- Assume the number of floors is 100
- In the best case how many balls will I have to drop to determine the lowest floor where a ball will break?
 1. 1
 2. 2
 3. 10
 4. 16
 5. 17
 6. 18
 7. 20
 8. 21
 9. 51
 10. 100
- In the worst case, how many balls will I have to drop?
 1. 1
 2. 2
 3. 10
 4. 16
 5. 17
 6. 18
 7. 20
 8. 21
 9. 51
 10. 100

If there are n floors, how many balls will you have to drop? (*roughly*)

CPS 100

6.6

What is big-Oh about? (preview)

- Intuition: avoid details when they don't matter, and they don't matter when input size (N) is big enough
 - For polynomials, use only leading term, ignore coefficients

$$\begin{array}{lll} y = 3x & y = 6x-2 & y = 15x + 44 \\ y = x^2 & y = x^2-6x+9 & y = 3x^2+4x \end{array}$$

- The first family is $O(n)$, the second is $O(n^2)$
 - Intuition: family of curves, generally the same shape
 - More formally: $O(f(n))$ is an *upper-bound*, when n is large enough the expression $cf(n)$ is larger
 - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time

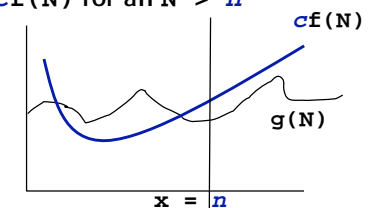
CPS 100

6.7

More on O-notation, big-Oh

- Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
 - Allows us to compare algorithms *in the limit*
 - $20N$ hours vs N^2 microseconds: *which is better?*
- O-notation is an upper-bound, this means that N is $O(N)$, but it is also $O(N^2)$; we try to provide *tight* bounds. Formally:

- A function $g(N)$ is $O(f(N))$ if there exist constants c and n such that $g(N) < cf(N)$ for all $N > n$



CPS 100

6.8

Big-Oh calculations from code

- Search for element in an array:
 - What is complexity of code (using O-notation)?
 - What if array doubles, what happens to time?

```
for(int k=0; k < a.length; k++) {
    if (a[k].equals(target)) return true;
};
return false;
```

- Complexity if we call N times on M-element vector?
 - What about best case? Average case? Worst case?

Big-Oh calculations again

- Alcohol APT: first string to occur 3 times
 - What is complexity of code (using O-notation)?

```
for(int k=0; k < a.length; k++) {
    int count = 0;
    for(int j=0; j <= k; k++) {
        if (a[j].equals(a[k])) count++;
    }
    if (count >= 3) return a[k];
};
return ""; // nothing occurs three times
```

- What happens to time if array doubles in size?
- $1 + 2 + 3 + \dots + n-1$, why and what's O-notation?

Amortization: Expanding ArrayLists

- Expand capacity of list when add() called
- Calling add N times, doubling capacity as needed

| Item # | Resizing cost | Cumulative cost | Resizing Cost per item | Capacity After add |
|---------------------|---------------|-----------------|------------------------|--------------------|
| 1 | 0 | 0 | 0 | 1 |
| 2 | 2 | 2 | 1 | 2 |
| 3-4 | 4 | 6 | 1.5 | 4 |
| 5-8 | 8 | 14 | 1.75 | 8 |
| | | | | |
| $2^{m+1} - 2^m + 1$ | 2^m | $2^{m+2} - 2$ | around 2 | 2^{m+1} |

- What if we grow size by one each time?

Some helpful mathematics

- $1 + 2 + 3 + 4 + \dots + N$
 - $N(N+1)/2$, exactly = $N^2/2 + N/2$ which is $O(N^2)$ why?
- $N + N + N + \dots + N$ (total of N times)
 - $N*N = N^2$ which is $O(N^2)$
- $N + N + N + \dots + N + \dots + N + \dots + N$ (total of $3N$ times)
 - $3N*N = 3N^2$ which is $O(N^2)$
- $1 + 2 + 4 + \dots + 2^N$
 - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$
- Impact of last statement on adding $2^N + 1$ elements to a vector
 - $1 + 2 + \dots + 2^N + 2^{N+1} = 2^{N+2} - 1 = 4 \times 2^N - 1$ which is $O(2^N)$

Running times @ 10^6 instructions/sec

| N | $O(\log N)$ | $O(N)$ | $O(N \log N)$ | $O(N^2)$ |
|---------------|-------------|----------|---------------|---------------|
| 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 |

Recursion Review

- Recursive functions have two key attributes
 - There is a *base case*, sometimes called the *exit case*, which does not make a recursive call
 - All other cases make recursive call(s), the results of these calls are used to return a value when necessary
 - Ensure that every sequence of calls reaches base case
 - Some measure decreases/moves towards base case
 - “Measure” can be tricky, but usually it’s straightforward
- Example: sequential search in an array
 - If first element is search key, done and return
 - Otherwise look in the “rest of the array”
 - How can we recurse on “rest of array”?

Sequential search revisited

- What does the code below do? How would it be called initially?
 - Another overloaded function `search` with 2 parameters?

```
boolean search(String[] a, int index,
               String target)
{
    if (index >= a.length) return false;
    else if (a[index].equals(target))
        return true;
    else return search(a, index+1, target);
}
```

- What is complexity (big-Oh) of this function?

Recursion and Recurrences

```
boolean occurs(String s, String x)
{ // post: returns true iff x is a substring of s
  if (s.equals(x)) return true;
  if (s.length() <= x.length()) return false;
  return occurs(s.substring(1, s.length()), x) ||
         occurs(s.substring(0, s.length()-1), x);
}
```

- In worst case, both calls happen
- Say $C(N)$ is the worst case cost of `occurs(s, x)`, $N == s.length()$
 - $C(N) = 1$ if $N < x.length()$
 - $C(N) = 2C(N-1)$ if $N > x.length()$
- What is $C(N)$?

Binary search revisited

```
bool bsearch(String[] a, int start, int end,
             String target)
{
    // base case
    if (start == end)
        return a[start].equals(target);

    int mid = (start + end)/2;
    int comp = a[mid].compareTo(target);
    if (comp == 0)           // found target
        return true;
    else if (comp < 0)      // target on right
        return bsearch(a, mid + 1, end);
    else                    // target on left
        return bsearch(a, start, mid - 1);
}
```

- What is the big-Oh of bsearch?

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?
scientists build to learn, engineers learn to build

- Mathematics is a notation that helps in thinking, discussion, programming

The Power of Recursion: Brute force

- Consider the TypingJob APT problemn: What is minimum number of minutes needed to type n term papers given page counts and three typists typing one page/minute? (assign papers to typists to minimize minutes to completion)
 - Example: {3, 3, 3 ,5 ,9 ,10 ,10} as page counts
- How can we solve this in general? Suppose we're told that there are no more than 10 papers on a given day.
 - How does the constraint help us?
 - What is complexity of using brute-force?

Recasting the problem

- Instead of writing this function, write another and call it
- ```
// @return min minutes to type papers in pages
int bestTime(int[] pages)
{
 return best(pages,0,0,0,0);
}
```
- What cases do we consider in function below?
- ```
int best(int[] pages, int index,
         int t1, int t2, int t3)
// returns min minutes to type papers in pages
// starting with index-th paper and given
// minutes assigned to typists, t1, t2, t3
{
}
}
```