Today

• Finish discussing queues

• Review math essential to algorithm analysis
  – Proof by induction
  – Powers of 2
  – Exponents and logarithms

• Begin analyzing algorithms
  – Using asymptotic analysis (continue next time)

Mathematical induction

Suppose $P(n)$ is some predicate (mentioning integer $n$)
  – Example: $n \geq n/2 + 1$

To prove $P(n)$ for all integers $n \geq c$, it suffices to prove
1. $P(c)$ – called the “basis” or “base case”
2. If $P(k)$ then $P(k+1)$ – called the “induction step” or “inductive case”

Why we will care:
To show an algorithm is correct or has a certain running time no matter how big a data structure or input value is
(Our “$n$” will be the data structure or input size.)
Example

\[ P(n) = \text{"the sum of the first } n \text{ powers of 2 (starting at 0) is } 2^n-1\"
\]

Theorem: \( P(n) \) holds for all \( n \geq 1 \)
Proof: By induction on \( n \)
\begin{itemize}
  \item Base case: \( n=1 \). Sum of first 1 power of 2 is \( 2^0 \), which equals 1.
    And for \( n=1 \), \( 2^n-1 \) equals 1.
  \item Inductive case:
    \begin{itemize}
      \item Assume the sum of the first \( k \) powers of 2 is \( 2^k-1 \)
      \item Show the sum of the first \( (k+1) \) powers of 2 is \( 2^{k+1}-1 \)
    \end{itemize}
    Using assumption, sum of the first \( (k+1) \) powers of 2 is
    \[ (2^k-1) + 2^{k+1}-1 = (2^k-1) + 2^k = 2^{k+1}-1 \]
\end{itemize}

Powers of 2

\begin{itemize}
  \item A bit is 0 or 1
  \item A sequence of \( n \) bits can represent \( 2^n \) distinct things
    \begin{itemize}
      \item For example, the numbers 0 through \( 2^n-1 \)
    \end{itemize}
  \item \( 2^{10} \) is 1024 ("about a thousand", kilo in CSE speak)
  \item \( 2^{20} \) is "about a million", mega in CSE speak
  \item \( 2^{30} \) is "about a billion", giga in CSE speak
\end{itemize}

Java: an \texttt{int} is 32 bits and signed, so "max int" is "about 2 billion"
\texttt{long} is 64 bits and signed, so "max long" is \( 2^{63}-1 \)

Therefore...

Could give a unique id to...
\begin{itemize}
  \item Every person in the U.S. with 29 bits
  \item Every person in the world with 33 bits
  \item Every person to have ever lived with 38 bits (estimate)
  \item Every atom in the universe with 250-300 bits
\end{itemize}

So if a password is 128 bits long and randomly generated,
do you think you could guess it?

Logarithms and Exponents

\begin{itemize}
  \item Since so much is binary in CS \( \log \) almost always means \( \log_2 \)
  \item Definition: \( \log_2 x = y \) if \( x = 2^y \)
  \item So, \( \log_2 1,000,000 = \) "a little under 20"
  \item Just as exponents grow very quickly, logarithms grow very slowly
\end{itemize}

See Excel file for plot data – play with it!
**Logarithms and Exponents**

- Since so much is binary $\log$ in CS almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2 1,000,000 = "a little under 20"
- Just as exponents grow very quickly, logarithms grow very slowly

**Properties of logarithms**

- $\log(A\times B) = \log A + \log B$
  - So $\log(N^k) = k \log N$
- $\log(A/B) = \log A - \log B$
- $\log(\log x)$ is written $\log \log x$
  - Grows as slowly as $2^y$ grows fast
- $(\log x)(\log x)$ is written $\log^2 x$
  - It is greater than $\log x$ for all $x > 2$
Log base doesn’t matter much!

"Any base $B$ log is equivalent to base 2 log within a constant factor"
- And we are about to stop worrying about constant factors!
- In particular, $\log_2 x = 3.22 \log_{10} x$
- In general, $\log_B x = (\log_A x) / (\log_A B)$

Algorithm Analysis

As the “size” of an algorithm’s input grows (integer, length of array, size of queue, etc.):
- How much longer does the algorithm take (time)
- How much more memory does the algorithm need (space)

Because the curves we saw are so different, we often only care about “which curve we are like”

Separate issue: Algorithm correctness – does it produce the right answer for all inputs
- Usually more important, naturally

Example

- What does this pseudocode return?
  
  ```
  x := 0;
  for i=1 to N do
    for j=1 to i do
      x := x + 3;
  return x;
  ```

- Correctness: For any $N \geq 0$, it returns...

Example

- What does this pseudocode return?
  
  ```
  x := 0;
  for i=1 to N do
    for j=1 to i do
      x := x + 3;
  return x;
  ```

- Correctness: For any $N \geq 0$, it returns $3N(N+1)/2$
- Proof: By induction on $n$
  - $P(n)$ = after outer for-loop executes $n$ times, $x$ holds $3n(n+1)/2$
  - Base: $n=0$, returns 0
  - Inductive: From $P(k)$, $x$ holds $3k(k+1)/2$ after $k$ iterations.
    Next iteration adds $3(k+1)$, for total of $3(k(k+1)/2 + 3(k+1) = (3k(k+1) + 6(k+1))/2 = (k+1)(3k+6)/2 = 3(k+1)(k+2)/2$
Example

- How long does this pseudocode run?
  
  ```
  x := 0;
  for i=1 to N do
    for j=1 to i do
      x := x + 3;
  return x;
  ```

- Running time: For any $N \geq 0$,
  - Assignments, additions, returns take “1 unit time”
  - Loops take the sum of the time for their iterations

- So: $2 + 2^*(number\ of\ times\ inner\ loop\ runs)$
  - And how many times is that…

Example

- How long does this pseudocode run?
  
  ```
  x := 0;
  for i=1 to N do
    for j=1 to i do
      x := x + 3;
  return x;
  ```

- The total number of loop iterations is $N*(N+1)/2$
  - This is a very common loop structure, worth memorizing
  - Proof is by induction on $N$, known for centuries
  - This is proportional to $N^2$, and we say $O(N^2)$, “big-Oh of”
    - For large enough $N$, the $N$ and constant terms are irrelevant, as are the first assignment and return
    - See plot… $N*(N+1)/2$ vs. just $N^2/2$

Lower-order terms don’t matter

$N*(N+1)/2$ vs. just $N^2/2$

Recurrence Equations

- For running time, what the loops did was irrelevant, it was how many times they executed.

- Running time as a function of input size $n$ (here loop bound):
  
  $$T(n) = n + T(n-1)$$
  
  (and $T(0) = 2$ish, but usually implicit that $T(0)$ is some constant)

- Any algorithm with running time described by this formula is $O(n^2)$

- “Big-Oh” notation also ignores the constant factor on the high-order term, so $3N^2$ and $17N^2$ and $(1/1000) N^2$ are all $O(N^2)$
  - As $N$ grows large enough, no smaller term matters
  - Next time: Many more examples + formal definitions
**Big-O: Common Names**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>constant (same as ( O(k) ) for constant ( k ))</td>
</tr>
<tr>
<td>O((\log n))</td>
<td>logarithmic</td>
</tr>
<tr>
<td>O((n))</td>
<td>linear</td>
</tr>
<tr>
<td>O((n \log n))</td>
<td>“(n \log n)”</td>
</tr>
<tr>
<td>O((n^2))</td>
<td>quadratic</td>
</tr>
<tr>
<td>O((n^3))</td>
<td>cubic</td>
</tr>
<tr>
<td>O((n^k))</td>
<td>polynomial (where ( k ) is an constant)</td>
</tr>
<tr>
<td>O((k^n))</td>
<td>exponential (where ( k ) is any constant &gt; 1)</td>
</tr>
</tbody>
</table>

Pet peeve: “exponential” does not mean "grows really fast", it means "grows at rate proportional to \(k^n\) for some \(k>1\)"
- A savings account accrues interest exponentially (\(k=1.01\)?)
- If you don’t know \( k \), you probably don’t know it’s exponential