Announcements

• Quiz 5 is today

• Homework 6 is due on Thursday

• Read Section 2.3 (Functions), 2.4 (Sequences and Summations), and 2.5 (Cardinality of Sets) by Thursday
Algorithms

An algorithm is a clearly defined, step-by-step procedure for solving a problem.

Properties of an algorithm:
• Input and output domains are specified.
• Each step is precisely described.
• Each step is executable in finite time.
• Typically applicable to a range of inputs, not just a specific one.
• Must halt with output in finite time.

Determining if a given algorithm halts on a particular input is an unsolvable problem. (meaning, we cannot write an algorithm to do it.)

aka, a recipe
Algorithms

No one knows if this algorithm halts on all inputs:

Algorithm weird

Input: x integer
Output: y integer

1. y=0
2. if x = 1, output y and halt.
3. elseif x is even
4. x = x/2; y++;
5. elseif x is odd
6. x = 3x + 1; y++;
7. goto 2

It halts on input $x \leq$ billions

5 $\rightarrow$ 16,8,4,2,1
3 $\rightarrow$ 10,5,16,8,4,2,1
6 $\rightarrow$ 3,10,5,16,8,4,2,1
7 $\rightarrow$ 22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1
Algorithms

An *iterative* algorithm is one that repeats the same sequence of steps a number of times.

- for loops
- while loops
- repeat loops
- goto??

The running time of an iterative algorithm depends on the number of times the loop is invoked.
Algorithms

How many times does “twiddle-thumbs” happen?

1. for i = 1 to n
2. for j = 1 to m
3. twiddle-thumbs

Complexity is \( O(mn) \)

The “time complexity” of an algorithm is a measure of its running time.

But different machines run at different speeds!

So we give running times in terms of big-oh, since different machines affect run times by constant factors.
Algorithm MAX

Input: $x_1, x_2, ..., x_n$, an array of numbers
Output: $x_n'$, the maximum of $x_1, x_2, ..., x_n$

1. for $j = 1$ to $n-1$
2. if $x_j > x_{j+1}$ then
3. $temp = x_{j+1}$
4. $x_{j+1} = x_j$
5. $x_j = temp$

Complexity is $O(n)$
Algorithm MAX
Input: $x_1, x_2, ..., x_n$, an array of numbers
Output: $y$, the maximum of $x_1, x_2, ..., x_n$

1. for $j = 1$ to $n-1$
2.  if $x_j > x_{j+1}$ then
3.     temp = $x_{j+1}$
4.     $x_{j+1} = x_j$
5.     $x_j = \text{temp}$

Can we PROVE this algorithm works?

The proof of correctness for an iterative algorithm is typically done via induction.
Algorithms

Algorithm MAX
Input: \(x_1, x_2, \ldots, x_n\), an array of numbers
Output: \(y\), the maximum of \(x_1, x_2, \ldots, x_n\)

1. \(\text{for } j = 1 \text{ to } n-1\)
2. \(\text{if } x_j > x_{j+1} \text{ then}\)
3. \(\text{temp } = x_{j+1}\)
4. \(x_{j+1} = x_j\)
5. \(x_j = \text{temp}\)

Prove that \(x_{j+1} = \max\{x_1, x_2, \ldots, x_{j+1}\}\) after the \(j^{th}\) iteration of the loop.

Note that the last iteration \((j = n-1)\) gives the result we really want.
Algorithms

Algorithm MAX
Input: $x_1, x_2, ..., x_n$, an array of numbers
Output: $x_n$, the maximum of $x_1, x_2, ..., x_n$
1. for $j = 1$ to $n-1$
2. if $x_j > x_{j+1}$ then
3. \hspace{1em} temp = $x_{j+1}$
4. \hspace{1em} $x_{j+1} = x_j$
5. \hspace{1em} $x_j = \text{temp}$

Base case ($j = 0^{\text{th}}$ iteration): $x_1 = \max\{x_1\}$

IH: assume assertion holds for $j = n^{\text{th}}$ iteration.
On $n+1^{\text{st}}$ iteration, $x_{n+1}$ is compared with $x_{n+2}$ and max is swapped into $x_{n+2}$.
But $x_{n+2} = \max\{x_{n+1}, x_{n+2}\} = \max\{x_1, x_2, ..., x_{n+1}, x_{n+2}\}$ by IH.
How long does the “linear search algorithm” take?

Suppose the data is (2, 8, 3, 9, 12, 1, 6, 4, 10, 7) and we’re looking for # 4.

The running time of the algorithm depends on the particular input to the problem. In this case we have two different complexity measures:

- Worst case complexity - running time on worst input
- Average case - average running time among all inputs

Worst case for linear search is time n.

Average case for linear search is time \((1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2\). Both are \(O(n)\).
Algorithm Complexity

How long does the “binary search algorithm” take?

Binary search
Input: a sorted array of numbers $a_1, a_2, ... a_n$ and a number $x$
Output: position $i$ where $a_i = x$, if $x$ is in the list

1. $i = 1$, $j = n$;
2. while $i < j$
3. $m = \lfloor (i + j)/2 \rfloor$; /* midpt of range $(i,j)$*/
4. if $x > a_m$ then $i = m + 1$
5. else $j = m$
6. if $x = a_i$ then output “$x$ is in position $a_i$”
7. else output “$x$ is not in list”

The main point of showing you this code is to remind you that the range is cut in half at every iteration.
Algorithm Complexity

How long does the “binary search algorithm” take?

Binary search
Input: a sorted array of numbers $a_1, a_2, \ldots a_n$ and a number $x$
Output: position $i$ where $a_i = x$, if $x$ is in the list

1. $i = 1$, $j = n$;
2. while $i < j$
3. $m = \lfloor (i + j)/2 \rfloor$; /* midpt of range $(i,j)$*/
4. if $x > a_m$ then $i = m + 1$
5. else $j = m$
6. if $x = a_i$ then output “$x$ is in position $a_i$”
7. else output “$x$ is not in list”

If $n$ is a power of 2, $n = 2^k$, then $k = \log n$ iterations occur.

If $n$ is not a power of 2, let $k$ be the number so that $2^k < n < 2^{k+1}$, and imagine that the array has $2^{k+1}$ elements. Then $k+1 < \log n + 1 = O(\log n)$
Running times

It’s fun to make comparisons about the running times of algorithms of various complexities.

<table>
<thead>
<tr>
<th>Input size</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>.00001s</td>
<td>.00002s</td>
<td>.00003s</td>
<td>.00004s</td>
<td>.00005s</td>
<td>.00006s</td>
</tr>
<tr>
<td>$n^2$</td>
<td>.0001s</td>
<td>.0004s</td>
<td>.0009s</td>
<td>.0016s</td>
<td>.0025s</td>
<td>.0036s</td>
</tr>
<tr>
<td>$n^5$</td>
<td>.1s</td>
<td>3.2s</td>
<td>24.3s</td>
<td>1.7m</td>
<td>5.2m</td>
<td>13m</td>
</tr>
<tr>
<td>$3^n$</td>
<td>.059s</td>
<td>58m</td>
<td>6.5y</td>
<td>3855c</td>
<td>2x10^8c</td>
<td>1.3x10^{13}c</td>
</tr>
</tbody>
</table>

But computers are getting faster! Maybe we can do better.
## Running times

Compare the sizes of problems solvable in 1 hour now, vs the size of problem we could solve if we had a 100 times faster machine.

<table>
<thead>
<tr>
<th>Algorithmic complexity</th>
<th>Input size we can solve w today’s machines (1hr)</th>
<th>Input size we can solve w. 100x faster machines (1hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$N_1$</td>
<td>$100xN_1$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$N_2$</td>
<td>$10xN_2$</td>
</tr>
<tr>
<td>$n^5$</td>
<td>$N_3$</td>
<td>$2.5xN_3$</td>
</tr>
<tr>
<td>$3^n$</td>
<td>$N_4$</td>
<td>$N_4 + 4.19$</td>
</tr>
</tbody>
</table>
Another Example

A dangling complexity question.

Why would we ever use “worst case complexity”?

Advantages:
• Easiest to analyze.
• Bounds the worst possible thing that could happen (conservative).
• Don’t have to decide or assume what typical inputs are (or find a distribution on inputs).

Disadvantage:
• Bizarre inputs could make worst-case running time seem horrible, when most inputs terminate in reasonable time.
Recursive algorithms

Factorial (n)

Input: an integer n > 0.
Output: n!

1. If n = 1 then output 1
2. else
3. output n x Factorial(n-1)

Let T(n) denote the running time of the algorithm on input of size n.

T(n) = C + T(n-1)
T(1) = C

T(n) = C + (C + T(n-2))
   = C + (C + (C + T(n-3))) = ... = nC = O(n)
1) Suppose that the only paper money consists of 3-dollar bills and 10-dollar bills. Show that any dollar amount greater than 17 dollars could be made from a combination of these bills.

2) Suppose \( \{a_n\} \) is defined recursively by:
\[ a_n = a_{n-1}^2 - 1 \text{ and } a_0 = 2 \]

Find the first four outputs \((a_1 \text{ through } a_4)\)
1) Suppose that the only paper money consists of 3-dollar bills and 10-dollar bills. Show that any dollar amount greater than 17 dollars could be made from a combination of these bills.

Base Step
• P (18): Eighteen dollars can be made using six 3-dollar bills.

• Inductive Step
• P (k) → P (k + 1): Suppose that k dollars can be formed, for some \( k \geq 18 \). If at least two 10-dollar bills are used, replace them by seven 3-dollar bills to form \( k + 1 \) dollars. Otherwise (that is, at most one 10-dollar bill is used), at least three 3-dollar bills are being used, and three of them can be replaced by one 10-dollar bill to form \( k + 1 \) dollars.
Practice Problems (Solutions)

2) Suppose \( \{a_n\} \) is defined recursively by:
\[ a_n = a_{n-1}^2 - 1 \text{ and } a_0 = 2 \]

Find \( a_1 \) through \( a_4 \)

\[
\begin{align*}
a_1 &= 2^2 - 1 = 3 \\
a_2 &= 3^2 - 1 = 8 \\
a_3 &= 8^2 - 1 = 63 \\
a_4 &= 63^2 - 1 = 3968
\end{align*}
\]