CS 151
Algorithm Analysis
An algorithm is a clearly specified set of instructions to solve a problem.
It is defined by its input, its desired output, and its (precise) description, usually in pseudocode.

Given any algorithm, you should have two questions:

1. Does my algorithm work?
   (is it provably correct?)
2. Does my algorithm work efficiently?
   a. how much time does it take?
      (what is its time complexity?)
   b. how much space does it need?
      (what is its space complexity?)

Proving algorithms correct is the subject of a future course (280–algorithms). We’ll only touch on it occasionally in 151.

In this course, we’re mostly concerned with time complexity.
Time Complexity

Why should we care about the time complexity of an algorithm?
1. to know how fast it is. Duh.
2. to be able to compare it to other algorithms.

So, we want a standardized way of determining the time complexity of an algorithm.

Idea #1:
• implement each candidate
• run them
• time the results

Q: Is there anything wrong with this approach?
Why should we care about the time complexity of an algorithm?

1. to know how fast it is. Duh.
2. to be able to compare it to other algorithms.

So, we want a standardized way of determining the time complexity of an algorithm.

Idea #2:
• find a function that expresses running time as a function of problem size
• should be able to determine this function without actual implementation
• this function should be independent of hardware and software environments
• this function should take into account all cases (from best to worst)

Vague details:
• just count up the “major” operations in pseudocode, ignoring small details
• find an upper bound on running time, i.e. a worst-case bound
Examples

ex. sum = 0
    for i=1 to n do
        sum = sum + i

ex. for i=1 to n do
    for j=1 to n do
        sum = sum + i*j

ex. for i=1 to n do
    for j=1 to m do
        a[i][j] = i*j

ex. for i=1 to n do
    for j=i to n do
        foo.bar()

ex. while n > 1 do
    n = n/2
ex. algorithm to get max value in an array
input: an array A of integers
goal/output: return the maximum value in A

max = A[0]
for i=1 to |A|-1 do
    if A[i] > max then
        max = A[i]
return max

The running time may change for different input arrays A.

For example: if the max value in A is A[0] then we never enter the if-statement. This makes the running time slightly faster.
Conversely, if the array A is monotonically increasing, then we have to enter the if-statement every iteration. This is the worst-case (slowest).
So, in analyzing time complexity, should we be
a. optimistic?
b. pessimistic?
c. neither?

Computer scientists do all three.

a. optimistic? We call this best-case analysis.
b. pessimistic? We call this worst-case analysis.
c. neither? We can do average-case analysis.

We’ll mostly focus on worst-case analysis, with the occasional average case thrown in for good measure. We’ll save the optimism for other people.
Some of the running time functions we’ve seen are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$T(n) = c$</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$T(n) = c_1 \cdot \log_2(n) + c_2$</td>
</tr>
<tr>
<td>linear</td>
<td>$T(n) = c_1 \cdot n + c_2$</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$T(n) = c_1 \cdot n \log_2(n) + c_2$</td>
</tr>
<tr>
<td>quadratic</td>
<td>$T(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3$</td>
</tr>
<tr>
<td>cubic</td>
<td>$T(n) = c_1 \cdot n^3 + c_2 \cdot n^2 + c_3 \cdot n + c_4$</td>
</tr>
<tr>
<td>exponential</td>
<td>$T(n) = c_1 \cdot 2^n$</td>
</tr>
<tr>
<td>factorial</td>
<td>$T(n) = c_1 \cdot n!$</td>
</tr>
</tbody>
</table>
So, if we really only care about efficiency as input size tends to infinity... then

• we can ignore multiplicative constants, e.g. $2n$, $3n$, $1000n$ all same as $n$
• we can ignore lower-order additive terms, e.g. $n^2+n$ same as $n^2$

Obviously, this is not fool-proof

• under this system, $1000n$ is somehow better than $n^2$
• (which, it is, if $n$ gets large enough!)
• under this system, $1000000n^2 + 100000n$ is equal to $n^2$.
• (well, they’re both better than $n^3$ and worse than $n \log n$)
Def: \( T(n) \) is \( O(f(n)) \) if there are positive constants \( c, n_0 \) such that
\[
T(n) \leq c \cdot f(n) \text{ for all } n \geq n_0
\]
That is, when the input gets large enough (bigger than \( n_0 \)), the running time is at most a constant time \( f(n) \).

ex. \( 2n+2 \) is \( O(n) \)
(i.e. \( f(n) = n \).) What \( c \) and \( n_0 \) fit the definition?

ex. \( 2n^2+3n+5 \) is \( O(n^2) \)

ex. \( n(n+1)/2 \) is \( O(?) \)

Note that although \( f(n) \) is an upper bound, it may not be the tightest bound.
ex. \( n \) is \( O(n^2) \)
You should always try to find the best (i.e. smallest) upper bound.
Big Oh Notation

So really, the bottom line is this:

To determine the big-Oh running time of an algorithm, you should:
- count the number of simple operations (but not incredibly precisely)
- find the dominant term as $n$ (the input size) goes to infinity
- lose the constants
- what you’ve got left is your answer. Voila!
You’ll want to remember this order (from fastest to slowest).

Suppose \( n = 1000 \) and each operation takes 1ms. Then we can compare...

<table>
<thead>
<tr>
<th>name</th>
<th>big-Oh</th>
<th>time-to-process</th>
<th>max-n-per-day</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>( O(c) )</td>
<td>1ms</td>
<td></td>
</tr>
<tr>
<td>logarithmic</td>
<td>( O(\log_2(n)) )</td>
<td>9.9ms</td>
<td></td>
</tr>
<tr>
<td>linear</td>
<td>( O(n) )</td>
<td>1s</td>
<td>86,400,000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( O(n \log_2(n)) )</td>
<td>9.9s</td>
<td>3,943,234</td>
</tr>
<tr>
<td>quadratic</td>
<td>( O(n^2) )</td>
<td>16.67min</td>
<td>9,295</td>
</tr>
<tr>
<td>cubic</td>
<td>( O(n^3) )</td>
<td>11.57days</td>
<td>442</td>
</tr>
<tr>
<td>exponential</td>
<td>( O(2^n) )</td>
<td>3.395x10^{290}</td>
<td>26</td>
</tr>
<tr>
<td>factorial</td>
<td>( O(n!) )</td>
<td>???</td>
<td>11</td>
</tr>
</tbody>
</table>
Recall: $T(n)$ is $O(f(n))$ if there are positive constants $c, n_0$ such that

$$T(n) \leq c \cdot f(n) \text{ for all } n \geq n_0$$

That is, $c \cdot f(n)$ is an upper bound on $T(n)$ for all $n \geq n_0$

Def: $T(n)$ is $\Omega(f(n))$ if there are positive constants $c, n_0$ such that

$$T(n) \geq c \cdot f(n) \text{ for all } n \geq n_0$$

That is, $c \cdot f(n)$ is an lower bound on $T(n)$ for all $n \geq n_0$

So $n$ is $O(n^2)$, but $n$ is not $\Omega(n^2)$, and $n^2$ is $\Omega(n)$, but $n^2$ is not $O(n)$.

Def: $T(n)$ is $\Theta(f(n))$ if and only if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

That is, $c \cdot f(n)$ is a matching upper and lower bound on $T(n)$.

Def: $T(n)$ is $o(f(n))$ if and only if $T(n)$ is $O(f(n))$ but not $\Theta(f(n))$.

Def: $T(n)$ is $\omega(f(n))$ if and only if $T(n)$ is $\Omega(f(n))$ but not $\Theta(f(n))$. 

Thursday, September 13, 12
input: a list \( L \) of \( n \) items, a desired item \( x \)
goal: determine whether \( x \) is in the list \( L \)

linear search:

```plaintext
linearSearch( list L, item x )
for i=1 to \( |L| - 1 \) do
    if A[i] == x then
        return true
    return false
```

best case: item is at the front, and this algorithm takes time \( O(1) \)
worst case: item is at the end, or is not in the list at all, and algorithm is \( O(n) \)
Thus we consider this algorithm to be \( O(n) \)
Binary Search

input: a sorted list L of n items, a desired item x

goal: determine whether x is in the list L

binarySearch( sorted list L, item x )
found = false
low = 0, high = |L| - 1
while( low < high )
    mid = (low + high) / 2
    if x == A[mid] then
        return true
    if x < A[mid] then
        high = mid - 1
    else
        low = mid + 1
return false

best case: item is in the middle, and this algorithm takes time O(1)
worst case: item is not in the list at all, and algorithm is O(log n) because we repeatedly divide our search space into 2 (can only do this log n times).