Objectives:

- overview of policies, procedures, and course material
- deal with simple examples of algorithm analysis
- learn notation for growth of functions ("big oh", "big omega", etc.)
- look at a simple example of divide and conquer: our first algorithm design technique

CSC 505 focus is on...

solving problems efficiently on a computer (resources are time and space)
- Algorithm design techniques
- Algorithm analysis techniques
- Data structures
- Problem complexity (inherent difficulty)

Some useful formulae

You should be familiar with these already. Please review them and be sure you understand how each can be proved.

\[
\log_a n = \frac{\log_b n}{\log_b a}
\]
\[
x^{\log_a y} = y^{\log_a x}
\]
\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}
\]
\[
\sum_{i=1}^{n} i^2 = \frac{n(n + 1)(2n + 1)}{6}
\]
\[
\sum_{i=0}^{t} r^i = \frac{1 - r^{t+1}}{1 - r} \quad \text{if } r \neq 1
\]
\[
\sum_{i=0}^{k} ia^i = \frac{a(1 - a^k)}{(1 - a)^2} - \frac{ka^{k+1}}{1 - a}
\]
\[
\sum_{i=0}^{\infty} ia^i = \frac{a}{(1 - a)^2}
\]

Analysis

As an indicator of execution time, determine the number of basic operations as a function of problem size.
Example 1: Square Matrix Multiplication

\[
\begin{bmatrix}
  c_{11} & c_{12} & \cdots & c_{1n} \\
  \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{bmatrix} =
\begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix} \times
\begin{bmatrix}
  b_{11} & b_{12} & \cdots & b_{1n} \\
  \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{bmatrix}
\]

where

\[c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\]

Problem size is \(n\) — by convention. You could argue for \(2n^2\) since there are \(2n^2\) distinct scalar inputs, but that makes the algebra complicated.

Basic operations are scalar multiplication and addition.

An algorithm for matrix multiplication:

\[
\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
  \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
  \quad \quad c[i,j] \leftarrow 0 \\
  \quad \quad \text{for } k \leftarrow 1 \text{ to } n \text{ do} \\
  \quad \quad \quad c[i,j] \leftarrow c[i,j] + a[i,k] + b[k,j]
\]

Number of scalar multiplications is ___.
Number of scalar additions is ___.

Example 2: Linear Search

Determine whether \(x\) is one of \(A[1], A[2], \ldots, A[n]\) (and retrieve other information about \(x\)).

Problem size is \(n\).

Basic operation: compare \(x\) with an element of array \(A\).

Algorithm:

\[
i \leftarrow 1 \\
\text{while } (i \leq n) \text{ and } (A[i] \neq x) \text{ do} \\
  \quad i \leftarrow i + 1 \\
\text{if } i > n \text{ then } i \leftarrow 0 \quad \triangleright \text{return a 0 if } x \text{ is not found}
\]

Number of comparisons is ___.
Worst case? ___
Best case? ___
Average case? ___

Example 3: Insertion Sort

Arrange \(A[1], \ldots, A[n]\) in ascending order (numerically or lexicographically).

Problem size is \(n\).

Basic operation: compare two elements of \(A\).
Algorithm (Insertion Sort):

1. \( j \leftarrow 2 \)  
   \( \triangleright \text{INVARIANT: } A[1], \ldots, A[j-1] \text{ are in ascending order} \)

2. while \( j \leq n \) do
   \( \triangleright \text{Insert } A[j] \text{ into its proper position among } A[1], \ldots, A[j-1]. \)
   
3.  
   \( x \leftarrow A[j]; i \leftarrow j-1; \)
   \( \triangleright \text{INVARIANT: } A[j] < A[k] \text{ for all } k \text{ satisfying } i+1 \leq k \leq j-1. \)

4. while \((i > 0)\) and \((A[i] > x)\) do
   
5.  
   \( A[i+1] \leftarrow A[i]; i \leftarrow i - 1 \)

6.  
   \( A[i+1] \leftarrow x \)

7.  
   \( j \leftarrow j + 1 \)

Aside about loop invariants:

A \textit{loop invariant} is a logical assertion that is true immediately before the loop's condition is tested, no matter how many iterations of the loop are performed. It can be written as a comment before the \textbf{while} statement.

To be a valid invariant, the assertion must satisfy two conditions:

1. It must be true right before the first iteration of the loop (usually in a trivial way). For example, because \( j = 2 \) after line (1), the outer loop invariant says that \( A[1] \) is in ascending order.

2. Truth of the assertion at the beginning of a loop iteration must imply truth at the end. For example, when \( A[j] \) is inserted into the proper place among \( A[1], \ldots, A[j-1] \), then \( A[1], \ldots, A[j] \) will be in ascending order. After line (7), it will still be the case that \( A[1], \ldots, A[j-1] \) are in ascending order.

To be useful, the assertion must satisfy a third condition:

3. The invariant, along with the negation of the loop condition, should imply the correctness of the algorithm (or some other useful property of the algorithm). For example, the loop condition for the outer loop is \( j \leq n \). Its negation is \( j > n \). If \( j > n \) and \( A[1], \ldots, A[j-1] \) are in ascending order, then the array is in sorted order.

To establish complete correctness of the algorithm, we also have to prove that \( A[1], \ldots, A[n] \) at the end of the algorithm are the same set as at the beginning.

Number of element comparisons.

Worst case? ___  
Best case? ___  
Average case? ___

**Approach A**

Use summation to account for number of basic operations in each loop.  

In the worst case, the inner loop (lines 4-5) requires \( j - 1 \) comparisons (that is what happens if \( i \) reaches 0 or 1 before we exit the loop). The outer loop is executed for values of \( j \) ranging from 2 to \( n \), so the total is

\[
\sum_{j=2}^{n} (j - 1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}
\]

**Approach B**

The number of comparisons in each inner loop is the number of "shifts" (times when \( x < A[i] \) and \( A[i] \) moves one position to the right). [If we exit the loop before \( i = 0 \), there is an extra comparison]
with no corresponding shift, but this does not affect the worst-case analysis."

Each shift occurs because of an inversion, a pair of elements that were in the wrong order initially, i.e. a pair \( i, j \) with \( 1 \leq i < j \leq n \) for which \( A[i] > A[j] \) in the original input array. This means that the number of comparisons in insertion sort is the number of inversions. In the worst case, every pair is an inversion, and the number of inversions is the number of pairs \( \binom{n}{2} = n(n-1)/2 \).

**Asymptotic (growth rate) analysis**

A further abstraction that we use in algorithm analysis is to characterize functions (recall that number of basic operations, a measure of time, is expressed as a function of input size) in terms of growth classes.

Matrix multiplication time grows as \( n^3 \)
Linear search time grows as \( n \)
Insertion sort time grows as \( n^2 \) or \( k \), where \( k = \) number of inversions

**Why is growth rate important?**

Actual execution time assuming 1,000,000 basic operations per second.

<table>
<thead>
<tr>
<th>input size</th>
<th>Execution time, where ( n = ) input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>300 \cdot n</td>
</tr>
<tr>
<td>100</td>
<td>.003 sec</td>
</tr>
<tr>
<td>1000</td>
<td>.03 sec</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>.3 sec</td>
</tr>
<tr>
<td>( 10^5 )</td>
<td>3 sec</td>
</tr>
<tr>
<td>1 sec</td>
<td>30 sec</td>
</tr>
</tbody>
</table>

\[ \ldots \] and now: problem size as a function of available time\(^a\)

<table>
<thead>
<tr>
<th>input size</th>
<th>1 sec ((10 \times))(^b)</th>
<th>1 min ((10 \times))</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,333</td>
<td>33,333</td>
<td>200,000</td>
</tr>
<tr>
<td>2,000</td>
<td>16,000</td>
<td>82,000</td>
</tr>
<tr>
<td>392</td>
<td>1,240</td>
<td>3,038</td>
</tr>
<tr>
<td>180</td>
<td>388</td>
<td>706</td>
</tr>
<tr>
<td>39</td>
<td>43</td>
<td>45</td>
</tr>
</tbody>
</table>

\[ \ldots \] and now: problem size as a function of available time\(^a\)

\(^b\)running on a 10 times faster machine

**Growth “classes” of functions**

\( \text{O}(g(n)) \) big oh: upper bound on the growth rate of a function; that is, a function belongs to class \( \text{O}(g(n)) \) if \( g(n) \) is an upper bound on its growth rate
\( \text{Ω}(g(n)) \) big omega: lower bound on the growth rate of a function
\( \Theta(g(n)) \) big theta: exact bound on the growth rate of a function
\( o(g(n)) \) little oh: used to denote functions that grow more slowly than \( g(n) \); for example, sometimes it’s useful to characterize the number of basic operations as something like \( 3n + o(n) \) to indicate that it’s \( \text{O}(n) \) with a small leading constant
\( \omega(g(n)) \) little omega: denotes functions that grow faster than \( g(n) \); rarely used but included for completeness

For example, a function describing the (worst case) number of basic operations of an algorithm might be \( \text{O}(n^2) \) and \( \text{Ω}(n \lg n) \) since it’s difficult to pin down exactly.
If we find that there’s a set of example inputs for which the growth rate in number of basic operations is \( n^2 \), then we can also say it’s \( \Omega(n^2) \) (the fact that it’s \( \Omega(n \lg n) \) is still true!) and conclude that it’s \( \Theta(n^2) \).

If, on the other hand, we’re able to prove that it never grows faster than \( n \lg n \), we can say that it’s \( O(n \lg n) \) (the fact that it’s \( O(n^2) \) is still true) and conclude that it’s \( \Theta(n \lg n) \).

Precise definitions of big oh and big omega

\[
f(n) \in O(g(n)) \iff \text{there exist } c > 0 \text{ and } n_0 > 0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0.
\]

\[
f(n) \in \Omega(g(n)) \iff \text{there exist } c > 0 \text{ and } n_0 > 0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0.
\]

\[
\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))
\]

Many authors abuse notation and say \( f(n) = O(g(n)) \) when they mean \( f(n) \in O(g(n)) \); same with \( \Omega(g(n)) \) and \( \Theta(g(n)) \). This is not a problem.

However, many technical papers will incorrectly use \( O(g(n)) \) when they mean \( \Omega(g(n)) \) or \( \Theta(g(n)) \). For example, “matrix multiplication requires at least \( O(n^2) \) operations, possibly as many as \( O(n^3) \).”

Examples illustrating big oh and big omega

\[
n^2 + 2n + \lg n \in O(n^3)
\]

Proof. \[
n^2 + 2n + \lg n \leq n^2 + 2n + n \quad \text{as long as } n \geq 1
\]
\[
= n^2 + 3n
\]
\[
\leq n^3 + 3n^3 \quad \text{(if } n \geq 1)\]
\[
= 4n^3
\]

This satisfies the definition of \( O(n^3) \) with \( c = 4 \) and \( n_0 = 1 \).

\[
n^3 - 10n^2 \notin O(n^2)
\]

Proof. Otherwise there must exist \( c > 0 \) and \( n_0 > 0 \) with \( n^3 - 10n^2 \leq cn^2 \) for all \( n \geq n_0 \).

But then \( n^3 \leq (c + 10)n^2 \) (for all \( n \geq n_0 \)) and \( n \leq c + 10 \). The latter is impossible for a given \( c \) and all \( n \geq n_0 \).

\[
5n^3 - 3n^2 + 2n - 6 \in \Theta(n^3)
\]

Proof.

First show that it’s in \( O(n^3) \):

\[
5n^3 - 3n^2 + 2n - 6 \leq 5n^3 + 2n \leq 7n^3 \quad \text{when } n \geq 1
\]

so it’s \( O(n^3) \) with \( c = 7 \) and \( n_0 = 1 \).

Then that it’s in \( \Omega(n^3) \):

\[
5n^3 - 3n^2 + 2n - 6 \geq 5n^3 - 3n^2 - 6 \geq \frac{5}{2}n^3 \quad \text{when } \frac{5}{2}n^3 \geq 3n^2 + 6 \text{ or } n \geq 2
\]

(good enough)

Of course, the above examples are only meant to emphasize the definition and illustrate what must be proved in order to claim time bounds in particular growth class.

The examples seem to suggest that we (i) figure out the exact number of basic operations as a function of \( n \), the input size, and then (ii) prove that what we figured out belongs to a particular growth class.
In fact, we can simplify analysis by skipping step (i). Recall insertion sort. We know the number of comparisons is \( O(n^2) \) — there are at most \( n \) iterations of the outer loop and at most \( n \) of the inner loop.

We can also argue \( \Omega(n^2) \) — for the last \( \lceil n/2 \rceil \) iterations of the outer loop, \( j \) is at least \( n/2 \). In the worst case (when \( A[j] \) is less than all of \( A[1], \ldots, A[j-1] \)), there are \( j-1 \) iterations of the inner loop (and the worst case happens every time when the array is in reverse order). So the number of comparisons is at least \( n/2(n/2-1) = n^2/4 - n/2 \), which is \( \geq n^2/8 \) when \( n \geq 4 \) (do the math!).

Examples involving logarithms and exponents

\[ \ln n \in \Theta(\log n) \]

Proof. Recall that \( \ln n = \log_e n \) and \( \log n = \log_2 n \). Using one of the mathematical identities on the first page, we have

\[ \ln n = \frac{\log n}{\log e} \]

So \( c \log n \leq \ln n \leq c \log n \), where \( c = \frac{1}{\log e} \), for all \( n \geq 1 \), which proves both \( O(\log n) \) and \( \Omega(\log n) \).

\[ e^n \notin O(n^t) \] for any fixed \( t \).

Proof. Otherwise there exist \( c > 0 \) and \( n_0 > 0 \) with \( e^n \leq cn^t \) for all \( n \geq n_0 \). But then (taking natural log’s of both sides) \( n \leq \log c + t \log n \). This translates into (divide each side by \( \log n \))

\[ \frac{n}{\ln n} \leq \frac{\ln c}{\ln n} + t \leq \ln c + t \quad \text{(when } n \geq e) \]

On the other hand,

\[ \lim_{n \to \infty} \frac{n}{\ln n} = \lim_{n \to \infty} \frac{1}{n/\ln n} = \infty \quad \text{(what rule did we use?)} \]

Little oh and little omega

\( f(n) \in o(g(n)) \) iff for all \( c > 0 \) there exists \( n_0 > 0 \) such that \( 0 \leq f(n) < cg(n) \) for all \( n \geq n_0 \).

\( f(n) \in \omega(g(n)) \) iff for all \( c > 0 \) there exists \( n_0 > 0 \) such that \( 0 \leq cg(n) < f(n) \) for all \( n \geq n_0 \).

Observations:

- \( f(n) \in \omega(g(n)) \) iff \( g(n) \in o(f(n)) \)
- \( f(n) \in o(g(n)) \) implies \( f(n) \notin \Omega(g(n)) \)
- \( f(n) \in \omega(g(n)) \) implies \( f(n) \notin O(g(n)) \)

For example, \( \left[ 2^n \in o(3^n) \right] \).

Proof. \( \lim_{n \to \infty} (2/3)^n = 0 \) and, by definition of limit, for any \( c > 0 \), there is an \( n_0 > 0 \) with \( (2/3)^n < c \) for all \( n \geq n_0 \). This means that \( 2^n < c3^n \) for all \( n \geq n_0 \), as desired.

Limits and asymptotic notation

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \quad \text{implies} \quad f(n) \in o(g(n)), \quad \text{that is,} \quad f(n) \notin \Omega(g(n)) \]
So limits can be helpful in determining the growth rate of functions.

Warning: The converses of the above are not necessarily true. Limits may not exist in some cases where growth classes are well-defined.

You might be tempted to conjecture that \( o(g(n)) = O(g(n)) - \Theta(g(n)) \). Not so. Consider the following functions (as suggested by the drawing):

Here (assuming constants have been appropriately scaled), \( f(n) \in O(g(n)) \) but \( f(n) \notin \Omega(g(n)) \) (it keeps “dipping down” to 0), so \( f(n) \in O(g(n)) - \Theta(g(n)) \). But \( f(n) \notin o(g(n)) \) (it keeps “rising back up” to the value of \( g(n) \)).

DIVIDE AND CONQUER

To solve (an instance of) a problem \( P \):

IF (the instance of) \( P \) is “large enough” THEN

- Divide (the instance of) \( P \) into smaller instances of the same problem
- Recurse to solve the smaller instances
- Combine solutions of the smaller instances to create a solution for the original instance

ELSE

- Solve (the instance of) \( P \) directly

ENDIF
Example: Binary search

Search for key \(x\) in a sorted array \(A[1], \ldots, A[n]\).

IF the array has more than one element THEN
  • Divide the array in half (conceptually)
  • Compare \(x\) with the last element of the lower half.
  • if \(x\) is larger, then recurse to search for \(x\) in the upper half; else recurse to search for \(x\) in the lower half.
  • return the result of the recursive search; there is no real combine step — this is tail recursion and can easily be implemented as a loop
ELSE
  • Compare the single element to \(x\) and return index if they’re equal
ENDIF

Details:

```c
function BinarySearch(A, x, lower, upper) is
  ▷ returns index of \(x\) if \(x\) is one of \(A[lower], \ldots, A[upper]\)
  ▷ or 0 if it isn’t.
  ▷ Initially called with lower = 1 and upper = n.

  if lower < upper then
    mid ← ⌊(lower + upper)/2⌋
    if \(x \leq A[mid]\) then
      return BinarySearch(A, x, lower, mid)
    else return BinarySearch(A, x, mid + 1, upper)
  else ▷ lower = upper
    if \(x = A[lower]\) then return lower
    else return 0 ▷ Not found
  end BinarySearch
```

Recursive insertion sort

(as a way of introducing mergesort)

IF the array has more than one element THEN
  • Divide the array into two parts, one that includes only the last element, the other the remaining elements.
  • Recurse to sort all but the last element.
  • Combine by inserting the last element into the recursively sorted array.
ELSE
  • Do nothing (a one-element array is already sorted)
ENDIF

Mergesort

IF the array has more than one element THEN
  • Divide the array in half (conceptually).
  • Recurse to sort both the lower half and the upper half.
  • Combine the two sorted halves by merging them.
ELSE
  • Do nothing.
ENDIF