CS 220: Discrete Structures and their Applications

Measuring algorithm running time using big $O$ analysis
We have two algorithms: \texttt{alg1} and \texttt{alg2} that solve the same problem, and you want fast running time.

How do we choose between the algorithms?
Measuring the running time of algorithms

Possible solution:
Implement the two algorithms and compare their running times

Issues with this approach:
- How are the algorithms coded? We want to compare the algorithms, not the implementations.
- What computer should we use? Results may be sensitive to this choice.
- What data should we use?
Measuring the running time of algorithms

Objective: analyze algorithms independently of specific implementations, hardware, or data

Observation: An algorithm’s execution time is related to the number of operations it requires

Solution: count the number of steps, i.e. constant time, operations the algorithm will perform for an input of given size

Example: copying an array with $n$ elements requires $\ldots$ operations.
def linear_search(array, value):
    for i in range(len(array)):
        if array[i] == value:
            return i
    return -1

What is the maximum number of steps linear search takes for an array of size n?
example: binary search

def binary_search(array, value, lo, hi):
    # precondition: array is sorted
    # postcondition: if value in array[lo...hi] return its position
    # else return -1
    if (lo>hi):
        r = -1
    else:
        mid = (lo+hi)/2
        if (array[mid]==value):
            r = mid
        elif array[mid]>value:
            r = binary_search(array, value, lo, mid-1)
        else:
            r = binary_search(array, value, mid+1, hi)
    return r
The time complexity of an algorithm is defined by a function $f: \mathbb{N} \rightarrow \mathbb{N}$ such that $f(n)$ is the maximum number of atomic operations performed by the algorithm on any input of size $n$. 
growth rates

Algorithm A requires $\frac{n^2}{2}$ operations to solve a problem of size $n$

Algorithm B requires $5n+10$ operations to solve a problem of size $n$

Which one would you choose?
When we increase the size of input $n$, how does the execution time grow for these algorithms?

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2 / 2$</td>
<td>.5</td>
<td>2</td>
<td>4.5</td>
<td>8</td>
<td>12.5</td>
<td>18</td>
<td>24.5</td>
<td>32</td>
</tr>
<tr>
<td>$5n+10$</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>50</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2 / 2$</td>
<td>1250</td>
<td>5,000</td>
<td>500,000</td>
<td>50,000,000</td>
<td>5,000,000,000</td>
</tr>
<tr>
<td>$5n+10$</td>
<td>260</td>
<td>510</td>
<td>5,010</td>
<td>50,010</td>
<td>500,010</td>
</tr>
</tbody>
</table>
growth rates

Algorithm A
Algorithm B
growth rates

Algorithm A requires $n^2/2$ operations to solve a problem of size $n$
Algorithm B requires $5n + 10$ operations to solve a problem of size $n$

For large enough problem size algorithm B is more efficient

We focus on the growth rate:

- Algorithm A requires time proportional to $n^2$
- Algorithm B requires time proportional to $n$
Order of magnitude analysis

**Big O:** A function $f(n)$ is $O(g(n))$ if there are two positive constants, $c$ and $n_0$, such that

$$f(n) \leq c \cdot g(n) \quad \forall n > n_0$$
Order of magnitude analysis

**Big O**: A function $f(n)$ is $O(g(n))$ if there are two positive constants, $c$ and $n_0$, such that

$$f(n) \leq c \cdot g(n) \quad \forall n > n_0$$

Focus is on the shape of the function
- Ignore the multiplicative constant

Focus is on large $x$
- $n_0$ allows us to ignore behavior for small $x$
Order of magnitude analysis

**Big O**: A function $f(n)$ is $O(g(n))$ if there are two positive constants, $c$ and $n_0$, such that

$$f(n) \leq c \cdot g(n) \quad \forall n > n_0$$

Focus is on the shape of the function

- Ignore the multiplicative constant

Focus is on large $x$

- $n_0$ allows us to ignore behavior for small $x$

$c$ and $n_0$ are **witnesses** to the relationship that $f(x)$ is $O(g(x))$
$f(x)$ is $O(g(x))$
\( f(x) \) is \( \Omega(g(x)) \)
Let $f$ and $g$ be functions. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants $c$ and $n_0$ s.t.,

$$f(x) \geq c \ g(x)$$

whenever $x > n_0$.
$f(x)$ is $\Theta(g(x))$
Let $f$ and $g$ be functions. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

\[ c_1 g(x) \]

\[ c_2 g(x) \]
Question

$f(n) = n^2 + 3n$

Is $f(n) O(n^2)$

why?
Question

\[ f(n) = n + \log n \]

Is \( f(n) \) \( O(n) \)?

why?
Question

\[ f(n) = n \log n + 2n \]

Is \( f(n) \) \( O(n) \)?

why?
Question

\[ f(n) = n \log n + 2n \]

Is \( f(n) \) \( O(n \log n) \)?

why?
worst/average case analysis

Worst case
- just how bad can it get: the maximum number of steps
- our focus in this course

Average case
- number of steps expected “usually”
- In this course we will hand wave when it comes to average case

Best case
- The smallest number of steps

Example: searching for an item in an unsorted array
Careful, this graph is misleading! Why? Small values of n. Make a table for $n^3$ and $2^n$ ($n=2,4,8,16,32$)
common shapes: constant

$O(1)$

Examples:
- Any integer/double arithmetic/logic operation
- Accessing a variable or an element in an array
Questions

Which is an example of constant time operations?

A. An integer/double arithmetic operation
B. Accessing an element in an array
C. Determining if a number is even or odd
D. Sorting an array
E. Finding a value in a sorted array
Common Shapes: Linear

\( O(n) \)

\[ f(n) = a*n + b \]

- \( a \) is the slope
- \( b \) is the Y intersection

Are all linear functions the same \( O \) ?
Which are examples of linear time operations?

A. Summing n numbers
B. adding an element in a linked list
C. binary search
D. Accessing A[i] in list A.
Other Shapes: Sublinear
log₂₈ = 3
2⁴ = 16 \quad \log₂_{16} = 4

logₐₙ: (\# \text{ of digits to represent } n \text{ in base } b) - 1

We usually work with base 2

log₂ₙ: \text{ number of times you can divide } n \text{ by } 2 \text{ until you get to } 1

log₂ₙ \text{ algorithms often break a problem in 2 halves and then solve 1 half}

The logarithm is a very slow-growing function
Properties of logarithms

- \( \log(x \cdot y) = \log x + \log y \)
- \( \log(x^a) = a \log x \)
- \( \log_a n = \log_b n / \log_b a \)

notice that \( \log_b a \) is a constant so

\[
\log_a n = O(\log_b n) \text{ for any } a \text{ and } b
\]

logarithm is a very slow-growing function
I have a number between 0 and 63

How many (Y/N) questions do you need to find it?

is it >= 32   N
is it >= 16   Y
is it >= 24   N
is it >= 20   N
is it >= 18   Y
is it >= 19   Y

What’s the number?

33
Guessing game

I have a number between 0 and 63
How many questions do you need to find it?

- is it >= 32  N  0
- is it >= 16  Y  1
- is it >= 24  N  0
- is it >= 20  N  0
- is it >= 18  Y  1
- is it >= 19  Y  1

What’s the number?  19  (010011 in binary)
$O(\log n)$ in algorithms

$O(\log n)$ occurs in divide and conquer algorithms, when the problem size gets chopped in half (third, quarter,...) every step

(About) how many times do you need to divide

1,000 by 2 to get to 1 ?
1,000,000 ?
1,000,000,000 ?
Question

Which is an example of a log time operation?

A. Determining max value in an unsorted array
B. Pushing an element onto a stack
C. Binary search in a sorted array
D. Sorting an array
Other Shapes: Superlinear

Polynomial \(x^a\), exponential \(a^x\)
quadratic

\[ O(n^2) : \]

\[
\begin{aligned}
\text{for } i \text{ in range}(n) & : \\
& \text{for } j \text{ in range}(n) : \\
& \quad \ldots
\end{aligned}
\]
Give a Big O bound for the following function.

\[ f(n) = (3n^2 + 8)(n + 1) \]

(a) \( O(n) \)
(b) \( O(n^3) \)
(c) \( O(n^2) \)
(d) \( O(1) \)

Is \( f(n) = O(n^4) \)?

What is the BEST (smallest) big O bound for \( f(n) \)?
Big-O for Polynomials

Theorem: Let
\[ f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]
where \( a_n, a_{n-1}, \ldots, a_1, a_0 \) are real numbers.

Then \( f(x) \) is \( O(x^n) \)

Example: \( x^2 + 5x \) is \( O(x^2) \)

Are all quadratic functions the same \( O \)? All cubic?
combinations of functions

Additive Theorem:

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1 + f_2)(x) \) is \( O(\max(|g_1(x)|, |g_2(x)|)) \).

Multiplicative Theorem:

Suppose that \( f_1(x) \) is \( O(g_1(x)) \) and \( f_2(x) \) is \( O(g_2(x)) \). Then \( (f_1f_2)(x) \) is \( O(g_1(x)g_2(x)) \).
practical analysis

Sequential
- Big-O bound: *steepest growth dominates*
- Example: copying of array, followed by binary search
  - $n + \log(n)$  \( O(?) \)

Embedded code
- Big-O bound *multiplicative*
- Example: a for loop with \( n \) iterations and a body taking \( O(\log n) \)  \( O(?) \)
dependent loops

....
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) {
        ...
    }
}
...

i = 0: inner-loop iters = 0
i = 1: inner-loop iters = 1
   ...
   :
   :
i = n-1: inner-loop iters = n-1

Total = 0 + 1 + 2 + ... + (n-1)
f(n) = n*(n-1)/2

O(n²)
public int f7(int n){
    int s = n;
    int c = 0;
    while(s>1){
        s/=2;
        for(int i=0;i<n;i++)
            for(int j=0;j<=i;j++)
                c++;
    }
    return c;
}
public int f7(int n){
    int s = n;
    int c = 0;
    while(s>1){
        s/=2;
        for(int i=0;i<s;i++)
            c++;
    }
    return c;
}
recursion

Number of operations depends on:
- number of calls
- work done in each call

Examples:
- factorial: how many recursive calls?
- binary search?
- merge sort?
- Fibonacci? (hint: draw the call tree)

```python
def h(n):
    if n==1: return 1
    else: return h(n-1) + h(n-1)

def f(n):
    if n<2: return 1
    else: return f(n-1) + f(n-2)
```
Practical Analysis - Recursion

Number of operations depends on:
- number of calls
- work done in each call

Examples:
- factorial: how many recursive calls?
- binary search?

We will devote more time to analyzing recursive algorithms later in the course.
Example Recursive Code

```java
public int divCo(int n) {
    if (n <= 1)
        return 1;
    else
        return 1 + divCo(n-1) + divCo(n-1);
}
```

How many recursive calls?
hint: draw the call tree

Big O complexity?

How much work per call?
What is the role of “return 1” and return 1+...”?
final comments

✓ Order-of-magnitude analysis focuses on large problems
✓ If the problem size is always small, you can probably ignore an algorithm’s efficiency
✓ Weigh the trade-offs between an algorithm’s time requirements and its memory requirements, expense of programming/maintenance...