Lecture 3: Algorithm Analysis
Foundational Data Structures
(Review of Some 280 Material)

**Asymptotic Algorithm Analysis**

An algorithm with complexity \( f(n) \) is said to be not slower than another algorithm with complexity \( g(n) \) if \( f(n) \) is bounded by \( g(n) \) for large \( n \).

Commonly written as \( f(n) = O(g(n)) \)
(read: \( f(n) \) is big-Oh \( g(n) \)),
a.k.a. the asymptotic (or big-Oh) notation

**Big-Oh – Definition**

\[ f(n) = O(g(n)) \] if and only if there are constants \( c > 0 \) and \( n_0 \geq 0 \) such that \( f(n) \leq c \cdot g(n) \) whenever \( n \geq n_0 \).

**Is \( n = O(n^2) \)?**

Let \( f(n) = 8n + 128 \) and \( g(n) = n^2 \).

\( f(n) \) is \( O(g(n)) \) if \( 8n + 128 \leq c \cdot n^2 \) for some \( c > 0 \) and \( n_0 \geq 0 \).

Let \( c = 1 \), clearly, for \( n = 8, f(n) > g(n) \).

At what value of \( n_0 \) is \( g(n) > f(n), \forall n \geq n_0 \)?

How about for \( c = 2 \) and \( c = 4 \)?
Big-Oh – Definition

As long as there is a \( c > 0 \), and \( n_0 \geq 0 \) such that
\( c \cdot g(n) \geq f(n) \) for all \( n \geq n_0 \), we say that \( f(n) = O(g(n)) \)

In this example, \( 8n + 128 = O(n^2) \)

Mathematically:
\[
f(n) = O(g(n)) \iff \exists c > 0, n_0 \geq 0 \mid \forall n, n \geq n_0, f(n) \leq c \cdot g(n)
\]
\[
O(g(n)) = \{ f(n) : \exists c > 0, n_0 \geq 0 \mid \forall n, n \geq n_0, 0 \leq f(n) \leq c \cdot g(n) \}
\]

So more accurately, \( f(n) \in O(g(n)) \)
but conveniently people write \( f(n) = O(g(n)) \),
though \textbf{NOT} \( f(n) \leq O(g(n)) \)

Big-Oh: Sufficient (but not necessary) Condition

If \[
\lim_{n \to \infty} \left( \frac{f(n)}{g(n)} \right) = c < \infty
\]
then \( f(n) \) is \( O(g(n)) \)

\[
\begin{align*}
\log_2 n &= O(2n) \\
f(n) &= \log_2 n \\
g(n) &= 2n \\
\lim_{n \to \infty} \frac{\log n}{2n} &= \lim_{n \to \infty} \frac{1}{2n} \\
&= 0 = c < \infty \\
\Rightarrow \log_2 n &= O(2n)
\end{align*}
\]

\[
\begin{align*}
\sin \left( \frac{n}{100} \right) &= O(100) \\
f(n) &= \sin \left( \frac{n}{100} \right) \\
g(n) &= 100 \\
\lim_{n \to \infty} \frac{\sin \left( \frac{n}{100} \right)}{100} &= \frac{\sin \left( \frac{n}{100} \right)}{\frac{n}{100}} \\
&= \frac{1}{1} = c = 1
\end{align*}
\]

Condition does not hold but nevertheless it is true that \( f(n) = O(g(n)) \)

Big-Oh – Definition

In other words, we only care about LARGE \( n \), it doesn’t matter what \( c \) is

\* obviously, \( c \) cannot be \( 10^{100} \) (one googol, the conjectured upper bound on the number of atoms in the observable universe!)

Also, asymptotically, \( n^2 + k = O(n^2) \), \( k \) constant (Why?)

L’Hôpital’s Rule

If \[
\lim_{x \to c} f(x) = \lim_{x \to c} g(x) = 0 \text{ or } \pm \infty
\]
and \( \lim_{x \to c} \frac{f'(x)}{g'(x)} \) exists then

\[
\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}
\]

Also useful, derivative of log:

\[
\frac{d}{dx} \log_b(x) = \frac{1}{x \ln(b)}
\]

\[
\frac{d}{dx} \ln(f(x)) = \frac{f'(x)}{f(x)}
\]
Log Identities

<table>
<thead>
<tr>
<th>Identity</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\log_{a}(xy) = \log_{a}x + \log_{a}y)</td>
<td>(\log_{a}(12) = )</td>
</tr>
<tr>
<td>(\log_{a}(x/y) = \log_{a}x - \log_{a}y)</td>
<td>(\log_{a}(4/3) = )</td>
</tr>
<tr>
<td>(\log_{a}(x^r) = r \log_{a}x)</td>
<td>(\log_{a}8 = )</td>
</tr>
<tr>
<td>(\log_{a}(1/x) = -\log_{a}x)</td>
<td>(\log_{a}(1/4) = )</td>
</tr>
<tr>
<td>(\log_{a}x = \frac{\ln x}{\ln a})</td>
<td>(\log_{a}9 = )</td>
</tr>
<tr>
<td>(k = \log_{a}n \iff 2^k = n)</td>
<td>(\log_{a}a = ?)</td>
</tr>
<tr>
<td>(\log_{a}1 = ?)</td>
<td>(\log_{a}1 = ?)</td>
</tr>
</tbody>
</table>

Power Identities

<table>
<thead>
<tr>
<th>Identity</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a^{(m+n)} = a^m a^n)</td>
<td>(2^5 = )</td>
</tr>
<tr>
<td>(a^{(m-n)} = a^m / a^n)</td>
<td>(2^{3-2} = )</td>
</tr>
<tr>
<td>((a^m)^n = a^{mn})</td>
<td>((2^3)^3 = )</td>
</tr>
<tr>
<td>(a^{-1} = \frac{1}{a})</td>
<td>(2^{-4} = )</td>
</tr>
<tr>
<td>(a^0 = ?)</td>
<td>(a^0 = ?)</td>
</tr>
<tr>
<td>(a^1 = ?)</td>
<td>(a^1 = ?)</td>
</tr>
</tbody>
</table>

Big-Oh: We Can Drop Constants

\(3n^2 + 7n + 42 = O(n^2)\)?

\[ f(n) = 3n^2 + 7n + 42 \]
\[ g(n) = n^2 \]

Definition

<table>
<thead>
<tr>
<th>(c &gt; 0, n_0 \geq 0) such that</th>
<th>Sufficient Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(n) \leq c \cdot g(n), \forall n \geq n_0)</td>
<td>(\lim_{n \to \infty} \frac{f(n)}{g(n)} = c &lt; \infty)</td>
</tr>
<tr>
<td>(f(n) = O(g(n)))</td>
<td>(= \lim_{n \to \infty} \frac{3n^2 + 7n + 42}{n^2})</td>
</tr>
<tr>
<td>(g(n) = n^2)</td>
<td>(= \lim_{n \to \infty} \frac{6n + 7}{2n})</td>
</tr>
<tr>
<td>(c = 5)</td>
<td>(= \lim_{n \to \infty} \frac{6}{2})</td>
</tr>
</tbody>
</table>

How Fast is Your Code?

Is \(f(n) = O(2^n)\)?
Is \(f(n) = O(n^2)\)?
Is \(f(n) = O(n \log n)\)?
Is \(f(n) = O(n)\)?
Is \(f(n) = O(\log n)\)?

Let \(f(n)\) be the complexity of your code, how fast would you advertise it as?

While \(f(n) = O(g(n)) \Rightarrow f(n) = g(n)\), you want to pick a \(g(n)\) that is as close to \(f(n)\) as possible (a “tight” bound)
Big-Oh – More Common Mistakes

**Mistake #3:** Let $f(n) = g_1(n) \times g_2(n)$
If $f(n) \leq cg_1(n)$ where $c = g_2(n)$,
then $f(n) = O(g_1(n))$ (NOT)

**Mistake #4:** Let $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, and $g_1(n) < g_2(n)$
$\Rightarrow f_1(n) < f_2(n)$ (NOT)

Counter-example:
$f_1(n) = ?$
$g_1(n) = ?$
$f_2(n) = ?$
$g_2(n) = ?$

Big-Oh – Rules

**Rule 1:** For $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

Example: $f_1(n) = n^3 \in O(n^3)$, $f_2(n) = n^2 \in O(n^2)$
$f_1(n) + f_2(n) = O(?)$

**Rule 2:** For $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
$f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$

• If your code calls a function within a loop, the complexity of your code is the complexity of the function you call times the loop's complexity

**Rule 3:** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$
then $f(n) = O(h(n))$

Relatives of Big-Oh

Big-Omega ($\Omega()$): asymptotic lower bound
For $f(n) > 0$, $\forall n \geq 0$, $f(n) = \Omega(g(n))$ if
$\exists c > 0, n_0 > 0$ \suchthat $\forall n \geq n_0, f(n) \geq c \times g(n)$
$h_1(n) = O(h_2(n)) \iff h_2(n) = \Omega(h_1(n))$

Big-Theta ($\Theta()$):

$f(n) = \Theta(g(n))$ iff
$f(n) = O(g(n))$ and
$f(n) = \Omega(g(n))$

$f(n)$ grows as fast as $g(n)$
Big-Theta

Does \( f(n) = \Theta(g(n)) \Rightarrow g(n) = \Theta(f(n)) \)?

Does \( f(n) = \Theta(g(n)) \Rightarrow f(n) = g(n) \)?

Does \( f(n) = \Theta(g(n)) \Rightarrow f(n) \) is the same order as \( g(n) \)?

Relatives of Big-Oh

little-oh (\( o() \)):

\[
 f(n) = o(g(n)) \text{ if } f(n) = O(g(n)) \text{ but } f(n) \neq \Theta(g(n))
\]

In contrast to \( O() \), \( o() \) is for all \( c > 0 \), whereas \( O() \) only requires there exists \( c > 0 \); so \( O() \) is sloppier than \( o() \), which is why we use it more often!

Example: \( 2n^2 = O(n^2) \) is asymptotically tight, but \( 2n = O(n^2) \) is not

little-omega (\( \omega() \)):

\[
 f(n) = \omega(g(n)) \text{ iff } g(n) = o(f(n))
\]

In the Limit

\( O() : f(n) = O(g(n)) \iff f(n) \leq c_1 g(n) \) and \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c_1 \)

\( \Omega() : f(n) = \Omega(g(n)) \iff f(n) \geq c_2 g(n) \) and \( \lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c_2 \)

\( \Theta() : f(n) = \Theta(g(n)) \iff \text{both } \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c_1 \text{ and } \lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c_2 \)

\( o() : f(n) = o(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)

\( \omega() : f(n) = \omega(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \)

The Common Case: Empirical Performance Evaluation

If \( n_0 > \) the common case \( n \), the asymptotic analysis result is not very useful . . . .

To determine the common case performance, given known workload, run empirical performance measurement/evaluation

Note that common case performance is not necessarily the average case performance (Why not?)

Empirical evaluation is also useful for evaluating complex algorithm or large software systems
Experiment Setup

Factors that affect the accuracy of your empirical performance evaluation:
- system speed
- system load
- compiler optimization

What you need:
- workload generator: must generate realistic common cases
- reduce system variability:
  - use the same compiler
  - use the same machine
  - minimize concurrent/background tasks
  - for shared systems, run experiment around the same time of day

Measuring Time

```c
#include <iostream>
#include <sys/resource.h>
#include <sys/time.h>

void main()
{
    struct rusage startu;
    struct rusage endu;
    getrusage(RUSAGE_SELF, &startu);
    //---- Do computations here
    getrusage(RUSAGE_SELF, &endu);
    double start_sec = startu.ru_utime.tv_sec + startu.ru_utime.tv_usec/1000000.0;
    double end_sec = endu.ru_utime.tv_sec + endu.ru_utime.tv_usec/1000000.0;
    double duration = end_sec - start_sec;
}
```

Struct `rusage` contains other useful information, e.g., memory usage

Empirical Results

Repeat experiment several times with the same input and take the average or minimum

Plot algorithm runtimes for varying input sizes

Include a large range to accurately display trend

Runtime looks linear ...

Analysis vs. Evaluation

When experimental results differ from analysis ...
- check for correctness in complexity analysis
- check for error in coding
- extra loop
- algorithm implemented is different from the one analyzed!
- if no error, experiment may simply have not covered worst case scenario
- external factors, e.g., hardware/software system (performance) bug?
Self-Study Questions

1. Which of these are true? Why?
   10^{100} = O(1)
   3n^2 + 45n^3 = O(n^3)
   3^x = O(2^n)
   2^x = O(3^n)
   45log(n) + 45n = O(log(n))
   log(n^2) = O(log(n))
   [log(n)]^2 = O(log(n))

2. Let \( f(n) = (n - 1)n/2, c = n/2 \), is \( f(n) = O(n) \)?
   If so, why? If not, what’s the big-Oh of \( f(n) \)?

3. Is \( \log n = O(n) \)?
   Is \( \log n = O(n^2) \)?
   Which is a tighter bound?

4. Given 4 consecutive statements: \( S_1; S_2; S_3; S_4 \); Let \( S_1 = O(\log n) \),
   \( S_2 = O(\log n^2) \), \( S_3 = O(n) \), \( S_4 = O(3n) \). What is the big-Oh time
   complexity of the four consecutive statements together? Prove it mathematically using
   the definition of big-Oh.

5. You ran two programs to completion. Both have running time of 10 ms.
   Can you say that the two programs have the same big-Oh time complexity? Why or why not?

6. Find \( f(n) \) and \( g(n) \), such that \( f(n) \) is not \( O(g(n)) \) and \( g(n) \) is not \( O(f(n)) \)

Foundational Data Structures

Data structures from which we build abstract data types (ADTs):
- arrays
- linked lists

Example ADTs?

Since they are so foundational to all the more complicated data structures, it is of
upmost importance that you thoroughly understand how to work with them

Arrays Review

What is an array?

```c
char ar[] = {'m', 'e', '2', '8', '1'};
char c = ar[2];
// now we have c==2
```

```c
cast ptr = ar;
// now ptr points to “ee281”
ptr = &ar[1];
// now ptr points to “e281”
// same as ptr = ar+1;
```

Copying with Pointers

How can we copy data from \( src_ar \) to \( dest_ar \)?

```c
double size = 4;
double src_ar[] = {3, 5, 6, 1};
double dest_ar[size];
```

Without pointer

```c
for (int i = 0; i < size; i++){
    dest_ar[i] = src_ar[i];
}
```

With pointer?

```c
In which cases would you want to use pointers?
```
Arrays: Common Bugs

Two most common bugs (in various guises):

1. out-of-bound access
   • index variable not initialized
   • null-termination error
   • off-by-one errors
   • bounds not checked

2. dangling pointers into/out of array elements
   • pointers in array not de-allocated → memory leak
   • when moved (or realloc-ed), pointers to array elements not moved

Index Variable Not Initialized

What’s the bug?

```c
int i;
printf( "\%c\n", y[i]);
```

Correct programs always
run correctly on correct input

Buggy programs sometimes
run correctly on correct input
• sometimes they crash
even when input doesn’t change!

Off-by-One Errors

```c
const int size = 5;
int x[size];

// set values to 0-4
for(int j=0; j<size; j++){
    x[j] = j;
}
// copy values from above
for(int k=0; k<=(size-1); k++) {
    x[k] = x[k+1];
}
// set values to 1-5
for(int m=1; m<size; m++){
    x[m-1] = m;
}
```

NULL-termination Errors

```c
int i;
char x[10];
strcpy(x, "0123456789");

// allocate memory
char* y = (char*)malloc(strlen(x));

for(i = 1; i < 11; i++) {
    y[i] = x[i];
}
y[i] = '\0';
printf("\%s\n", y);
```

Look up/confirm the behavior of various libraries by reading the manual pages
(under Linux or Mac OS X) or http://www.cplusplus.com/reference/clibrary/
Bounds Not Checked

```c
int main(int argc, const char* argv[]) {
    char name[20];
    strcpy(name, argv[1]);
}
```

What errors may occur when running the code?

How can the code be made safer?

Container Classes

Wrapper for objects
- allows for control/protection over editing of objects
- e.g., adding bounds checking to arrays

Container class operations:
- Constructor
- Destructor
- addElement()
- removeElement()
- getElement()
- getSize()
- copy()
- assign()

Example of a Container Class:
Adding Bounds Checking to Arrays

```c
class Array{
    int* data; // array data
    unsigned int length; // array size
    // Why aren’t data and length public?

public:
    // Constructor:
    Array(unsigned len=0):length(len) {
        data = (len ? new char[len] : NULL);
    }
    // other methods to follow in next slides...
};
```

Array Class: Inserting an Element

```c
bool insert(int index, double val){
    if (index >= size || index < 0)
        return false;
    for(int i=size-1; i > index; i--){
        data[i] = data[i-1];
    }
    data[index] = val;
    return true;
}
```

```c
ar = 1.6 3.1 4.2 5.9
ar.insert(1, 3.4);
```

Are arrays desirable when many insertions are needed?
Array Class: Complexity of Insertion

```cpp
bool insert(int index, double val) {
    if (index >= size || index < 0)
        return false;
    for(int i=size-1; i > index; i--){
        data[i] = data[i-1];
    }
    data[index] = val;
    return true;
}
```

- Best case: $O(1)$
- Worst case: $O(n)$
- Average case: $O(n)$

Memory Leak

If `ar` is deleted/freed using either:
- `free(ar);` or `delete ar;`

objects it points to become inaccessible, causing memory leak

How to delete `ar` correctly?

Memory Ownership

Array Class: Append Example

Original `ar` = `array A` = [ ]

How can we append one more element `●`?

Create a new `temp_ar` = [ ]

Copy existing elements into new array
and add new element:

`New ar` = [ ]

Delete old array so that memory can be reused
(but be careful of dangling pointers!)

Why do we have to make a new array?
How big shall we make the new array?

Array Class: Append Example

Original `ar` = [ ]

How can we append one more element `●`?

Create a new `temp_ar` = [ ]

Copy existing elements into new array
and add new element:

`New ar` = [ ]

Delete old array so that memory can be reused
(but be careful of dangling pointers!)

Why do we have to make a new array?
How big shall we make the new array?
Dangling Pointers

Say we have a binary search tree (BST) pointing to elements in an unsorted array `ar` (the BST acts as an “index” to speed up search).

Now if we need a larger array, we’d need to reallocate a larger chunk of memory and copy each element of the old array to the new array.

Leak if the BST is not updated and continues to point to the old space.

Amortized Complexity

A type of worst-case complexity analysis spread out over a given input size.

Considers the average cost over a sequence of operations

- in contrast: best/worst/average-case only considers a single operation

Justifies the cost of expandable arrays.

Array Class: Complexity of Append

Appending $n$ additional elements to an already full array of size $n$.

On first append

- double array size from $n$ to $2n$ (1 step)
- copy $n$ items from original array to new array ($n$ steps)

On remaining $n-1$ appends

- place element in appropriate location ($n-1$ times 1 step)

Total: $1 + n + (n-1) = 2n$ steps

Amortized complexity of appending additional $n$ elements: $2n/n = 2$ steps per append = $O(1)$

Pros and Cons of Arrays

Name 2 advantages of using an array:

Name 3 disadvantages of using an array:
2D Arrays in C/C++

```c
int arr[3][3];
int val = 0;

// For each row
for (int r=0; r < 3; r++){
    // For each column
    for (int c=0; c < 3; c++){
        arr[r][c] = val++;
    }
}
```

Limitations:

- indexing cumbersome, makes code hard to read
- prefer to address elements as `array[][]`

2D Arrays with Row Pointers

```c
int data = (int *) malloc(9*sizeof(int));
int *arr[3]; // array of row pointers

// assign row pointers
for (int r=0, nc=3; r < 3; r++) {
    arr[r] = &array[r*nc];
}
int val=0;
for(int r=0; r < 3; r++){ //rows
    for(int c=0; c < 3; c++){ // columns
        arr[r][c] = val++;
    }
}
```

Use this method in your programming assignments

Self-Study Questions

1. Who owns the memory in a container class?
2. What are the disadvantages of arrays?
3. Why do you need a `const` and a non-`const` version of some operators? What should a non-`const` `op[]` return?
4. How many destructor calls (min, max) can be invoked by:
   - `operator delete`
   - `operator delete[]`
5. Why would you use a pointer-based copying algorithm?
6. Are C++ strings null-terminated?
7. Give two examples of off-by-one bugs.
8. How do I set up a 2D array class?
10. Discuss the pros and cons of pointers and references when implementing container classes.