CSC 6013 Algorithms and Discrete Structures I **Michael Bradley**

Week 4: Algorithms and Asymptotic Analysis

Class presentation (= reduced version of full class notes)

Big idea #1: Algorithms

What is an algorithm?

Informally, a strategy for solving a problem.

Historically, a Latinized version of the name of Persian mathematician

Abū Ja'far Muhammad ibn Mūsā al-Khwārizmī (800-867)

who wrote the oldest known math book about using symbols to solve equations titled

al-Kitīb al-mukhtasar fī hisāb al-jabr wa'l muqābala

(The compendious book on calculation by completion and balancing).

al-jabr = completion = "add the same quantity to both sides of the equation"

al-muqābala = balancing = "subtract the same quantity form both sides of the equation"

A more useful definition of algorithm in computer science is:

A well-defined computational procedure

that takes some value, or set of values, as input

and

produces a value, or set of values, as output.

More concisely,

A sequence of computational steps (instructions)

that convert an input to an output.

We will often write a computational problem as an input and output relation.

Three examples of algorithms from the class notes:

Example 1: algorithm to count the number of positive values in an array of real numbers

```
1       # Input: An array A [0..n-1] of real numbers
2       # Output: Return the number of positive values in the array
3       def countPositiveElements(A):
4               count = 0
5               for x in A:
6                       if x>0:
7                               count += 1
8               return count
```

Example 5: algorithm to determine if all the elements in an array are unique

```
1       # Input: An array A if integers.
2       # Output: True is elements are unique; otherwise, False.
3       def uniqueElements(A):
4               for i in range(0, len(A)-1):
5                       for j in range(i+1, len(A)):
6                               if A[i] == A[j]:
7                                       return False
8               return True
```

Example 6: algorithm to count the number of digits in the binary representation of a number

```
1       # Input: n a non-negative integer.
2       # Output: Return the number of bits in the
3       #       binary expansion of n.
4       def binaryDigitCount(n):
5               count = 1
6               while n>1:
7                       count = count+1
8                       n = n//2
9               return count
```

Big idea #2: Asymptotic analysis of algorithms

Given an algorithm,

for an input of size n,

find a function of n, T(n), that describes the amount of work done by the algorithm

usually, the number of times some fundamental step is executed

usually, in the worst case (the maximum over all valid inputs of size n)

usually, provide an upper bound (rather than an exact count)

the Big-Oh class of the algorithm

From page 3 of class notes

Example 1: algorithm to count the number of positive values in an array of real numbers

```
1       # Input: An array A [0..n-1] of real numbers
2       # Output: Return the number of positive values in the array
3       def countPositiveElements(A):
4               count = 0
5               for x in A:
6                       if x>0:
7                               count += 1
8               return count
```

Using the formal strategy from the book, what is our input size?
Answer: n.

What's the worst-case running time?

To formally compute the worst-case running time, we would determine the cost for each line and the number of times each line executes. The results are in the table:

Line	Cost	Count
4	c_1	1
5	c_2	n
6	c_3	n
7	c_4	n (in the worst case)
8	c_5	1

We then total the cost multiplied by the count for each line.

$T(n) = c_1(1) + c_2(n) + c_3(n) + c_4(n) + c_5(1)$	group like terms together
$= (c_2 + c_3 + c_4)n + (c_1 + c_5)1$	collapse constants into new constants
$= c_6(n) + c_7$	for $n \geq 1$, $1 \leq n$
$\leq c_6(n) + c_7(n)$	group like terms together
$= (c_6 + c_7)n$	collapse constants into a new constant
$= c_8(n)$	

Since $T(n) \leq$ some constant multiple of n, we say
$T(n)$ is in the asymptotic class $O(n)$
and we write $T(n) = O(n)$.

Informally, we can identify the basic operation to be the comparison in line 6. Since this is performed once on each of the n passes through the for loop in line 6, the work is O(n). Done.

From page 27 of class notes

Example 5: algorithm to determine if all the elements in an array are unique

```
1       # Input: An array A if integers.
2       # Output: True if elements are unique; otherwise, False.
3       def uniqueElements(A):
4               for i in range(0, len(A)-1):
5                       for j in range(i+1, len(A)):
6                               if A[i] == A[j]:
7                                       return False
8               return True
```

Using the formal strategy from the book, what is our input size?
Answer: n.

What's the worst-case running time?

To formally compute the worst-case running time, we would determine the cost for each line and the number of times each line executes. The results are in the table:

Line	Cost	Count
4	c_1	n - 1
5	c_2	(n-1) + (n-2) + (n-3) + … + 1 = $\frac{(n-1)n}{2}$
6	c_3	(n-1) + (n-2) + (n-3) + … + 1 = $\frac{(n-1)n}{2}$
7	c_4	1 (at most)
8	c_5	1 (at most)

we then total the cost multiplied by the count for each line

$T(n) = c_1 (n-1) + c_2 \frac{(n-1)n}{2} + c_3 \frac{(n-1)n}{2} + \max(c_4, c_5)$

use the distributive law to remove all the parentheses

$= c_1 n – c1 + ½ c_2 n^2 - ½ c_2 n + ½ c_3 n^2 - ½ c_3 n + \max(c_4, c_5)$

group like terms together

$= (½ c_2 + ½ c_3) n^2 + (c_1 - ½c_2 - ½c_3) n + (-c_1 + \max(c_4, c_5))$

replace each parenthesized group of constants by a new named constant

$= c_6 n^2 + c_7 n + c_8$

when n > 1, $n < n^2$ and $1 < n^2$, so this expression is smaller than the new expression with these higher powered terms substituted

$< c_6 n^2 + c_7 n^2 + c_8 n^2$

factoring out n^2 from each term and replacing the group of constants by a new named constant

$= c_9 n^2$

since T(n) < some constant multiple of n^2, T(n) is in the Big-Oh class

$= O(n^2)$

To informally determine the Big-Oh class for the worst-case running time, we can identify the basic operation to be the comparison in line 6.
Because we have nested loops, that causes the inner loop to run one less time on each initiation.
Therefore, this operation will be performed (n-1) + (n-2) + (n-3) + …. + 3 +2 +1 = n(n-1)/2 times.
This simplifies to be = $\frac{1}{2}n^2 - \frac{1}{2}n$.
Focusing only on this highest-powered term and ignoring its coefficient, we get $O(n^2)$. Done.

From page 29 of class notes

Example 6: algorithm to count the number of digits in the binary representation of a number

```
1       # Input: n a non-negative integer.
2       # Output: Return the number of bits in the
3       #       binary expansion of n.
4       def binaryDigitCount(n):
5               count = 1
6               while n>1:
7                       count = count+1
8                       n = n//2
9               return count
```

Using the formal strategy from the book, what is our input size?
Answer: n.

What's the worst-case running time?

To formally compute the worst-case running time, we would determine the cost for each line and the number of times each line executes.

It is easy to see that lines 5 and 9 are each executed once. What about lines 6, 7, and 8?
On the first pass through the loop we have the original value of n.
Inside the loop, n is divided by 2 and rounded down so it is replaced by n//2.
So on the second pass through the loop we have (the original value of n)//2.
Inside the loop, the current value of n is divided by 2 and rounded down.
So on the third pass through the loop we have (the original value of n)//4.
On the next pass through the loop we have (the original value of n)//8.
Then (the original value of n)//16.
Then (the original value of n)//32.
On the kth pass through the loop, we have (the original value of n)//(2^{k-1}).
The number of times you can divide a positive integer by 2 and round down, before the result becomes ==1, is called the base-2 logarithm of the number or, log2(n) from python's math library.
So, lines 6, 7, and 8 are each executed log2(n) times.

In the table, we have:

Line	Cost	Count
5	c_1	1
6	c_2	log2(n)
7	c_3	log2(n)
8	c_4	log2(n)
9	c_5	1

Now, we can total the cost multiplied by the count for each line

$T(n) = c_1 (1) + c_2 (\log2(n)) + c_3 (\log2(n)) + c_4 (\log2(n)) + c_5 (1)$

group like terms together

$= (c_2 + c_3 + c_4) \log2(n) + (c_1 + c_5)$

replace each parenthesized group of constants by a new named constant

$= (c_6) \log2(n) + (c_7)$

when n > 2, we have 1 < log2(n), so this expression is smaller than the new expression with this larger term substituted

$< (c_6) \log2(n) + (c_7) \log2(n)$

group like terms together

$= (c_6 + c_7) \log2(n)$

replace the parenthesized group of constants by a new named constant

$= c_8 \log2(n)$

Since T(n) < some constant multiple of log2(n), T(n) is in the Big-Oh class

$= O(\log2(n))$

To informally determine the Big-Oh class for the worst-case running time, we can identify the basic operation to be the integer division operation in line 8 (n = n//2).
Since this is executed log2(n) times, we get O(log2(n)). Done.

In addition to the Big-Oh class of an algorithm, we have the Big-Omega and Big-Theta classes.

Also, we need to learn about the mathematics of logarithms and summations.

There are also other details to learn.

So, read the class notes and read the textbook.

Big idea #3: Big-Oh class, Big-Omega, and Big-Theta

(see class notes pages 19-21 for more details)

For the uniqueElements algorithm we saw that in the worst-case T(n) = O(n²). We got this result by showing that T(n) ≤ cn² for some constant c (does not depend on n) or large vales of n.

O(n²) is the class of all functions that are bounded above by a multiple of n² for sufficiently large values of n. In formal mathematical notation,

$$O(n^2) = \{f(n) | \exists c\ , n_0 > 0\ such\ that\ 0\ \leq\ f(n) \leq c \cdot n^2\ , \forall\ n \geq n_0\}$$

Since T(n) is in this class of functions, we should write T(n) ∈ O(n²) to indicate that T(n) is an element in this set, but the common notation is to write that T(n) = O(n²). We say that we have an n² algorithm in the worst case, or an algorithm whose worst case is n².

If we were able to show that T(n) ≥ c n² for some constant c, we have established a lower bound. This means that the algorithm does at least c n² work in the worst case. We would then say that T(n) is in the class Big-Omega(n²). In formal mathematical notation,

$$\Omega(n^2) = \{f(n) | \exists c\ , n_0 > 0\ such\ that\ 0 < c \cdot n^2\ \leq f(n)\ , \forall\ n \geq n_0\}$$

Note, that Ω(n²) is the set of all functions bounded below by some constant multiple of n².

If we were able to show that T(n) ≥ c_1 n² for some constant c_1 and T(n) ≤ c_2n² for some constant c_2, then we have a multiple of n² as a lower bound and another multiple of n² as an upper bound so

$$c_1\, n^2 \leq T(n) \leq c_2\, n^2.$$

If we can find lower and upper bounds like this, we would then say that T(n) is in the class Big-Theta(n²). In formal mathematical notation,

$$\theta(n^2) = \{f(n) | \exists c_1\ , c_2, n_0 > 0\ such\ that\ 0 < c_1 \cdot n^2 \leq\ f(n) \leq c_2 n^2\ , \forall\ n \geq n_0\}$$

There was nothing special about n² in this discussion. We can provide similar definitions for O(n) or Ω(log2(n)) or θ(n³), etc. The corresponding definitions of these classes work for any function g(n).

Read fuller details in the class notes (pages 19-21).

Read the whole story in the textbook.

Big idea #4: Summations (see class notes pages 14-17 for more details)

The uppercase Greek letter sigma (Σ) is used to denote the sum of a sequence of terms.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n$$

Think of a "for loop" summing the terms from an array.

Often, the general term a_i is a function of the subscript/index *i.*

$$\sum_{i=1}^{4} i(i+1) = 1(2) + 2(3) + 3(4) + 4(5) = 2 + 6 + 12 + 20 = 40$$

A different letter (such as j or k) can be used for the index/subscript.

$$\sum_{j=1}^{5} j^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$$

The initial value of the index/subscript can be another number instead of 1.

$$\sum_{i=0}^{4} 2^i = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31$$

Certain types of summations occur often enough that we should know the following formulas:

Sums of sums: If the general term of a summation is a sum of terms, we can create separate summations for each term.

$$\sum (a_i + b_i) = \sum a_i + \sum b_i$$

ex.

$$\sum_{j=1}^{n} (j^3 + 5j) = \sum_{j=1}^{n} j^3 + \sum_{j=1}^{n} 5j$$

Sums with constant coefficients: If every term of a summation includes the same constant, we can factor it out in front of the summation.

$$\sum c \cdot a_i = c \sum a_i$$

ex.

$$\sum_{j=1}^{n} 5j^2 = 5\sum_{j=1}^{n} j^2$$

Sums of constants: If the general term is a constant (k in the following formula), we are simply adding n copies of the constant.

$$\sum_{i=1}^{n} k = k + k + k + \cdots + k = k \cdot n$$

ex.

$$\sum_{i=1}^{n} 3 = 3 + 3 + 3 + \cdots + 3 = 3n$$

Sums of consecutive positive integers:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

ex.

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + \cdots + 10 = \frac{10(11)}{2} = 55$$

Sums of squares of consecutive positive integers:

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

ex.

$$\sum_{i=1}^{10} i^2 = 1^2 + 2^2 + 3^2 + \cdots + 10^2 = \frac{10(11)(21)}{6} = 385$$

Sums of positive integer powers of 2: (Note that the summation starts with i = 0.)

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + 2^2 \ldots + 2^n = 2^{n+1} - 1$$

ex.

$$\sum_{i=0}^{7} 2^i = 2^0 + 2^1 + 2^2 \ldots + 2^7 = 2^8 - 1 = 255$$

You should become comfortable reading and writing this summation notation. This is not a complete list; we will see additional formulas.

Big idea #5: Logarithms base 2 (see class notes page 14 for more details)

Fact: Every positive real number can be written as a power of 2.

For example, 8 can be written as 2^3 .

32 = 2^5.

1,024 = 2^{10}.

½ = 2^{-1}.

1 = 2^0.

In general, if n = 2^k, then *k* = $log_2(n)$, which is also written as *lg(n)* or *lg n*. In Python's math library, this is log2(n) or log(n, 2).

In other words, the base-2 logarithm of a positive integer *n* is the power of 2 that "produces" *n*.

A more helpful way to think about base-2 logarithms is, for any positive number n, lg(n) means:

if you start with n, how many times can you divide by 2 before you get to a fraction between 0 and 1?

8/2 = 4, 4/2 = 2 , 2/2 = 1 since we "divided by 2" 3 times, lg(8) = 3.

32 → 16 → 8 → 4 → 2 → 1, since we "divided by 2", 5 times, lg(32) = 5.

Some properties of logarithms that we will use frequently are:

1. $2^{\lg(n)} = n$ ← This is the equation version of the previous sentence "In other words,…"

2. $lg(a^b) = b \cdot lg(a)$ ← This is the logarithm version of the exponent formula $(a^m)^n = a^{m \cdot n}$ since

$$2^{b \cdot \lg(a)} = \left(2^{\lg(a)}\right)^b = a^b.$$

3. $lg(2) = 1$ ← This follows from the fact that $2^1 = 2$

4. $lg(2^n) = n$ ← This follows from formulas 2 and 3.

5. $lg(n^k) = k \cdot \lg(n)$ ← This follows from formula 2.

We will need other properties of logarithms later, but we will pick them up as we go rather than learn them all now.