Week 4

- Introduce the **Analysis of Complexity**, also called **Algorithmic Analysis**, or “where Big O Notation comes from”.
- The techniques of algorithmic analysis will be applied to the various data structures, searching and sorting techniques developed in the rest of the course.
- Used to provide understanding as to why one algorithm is better than another in terms of its expected execution time.

Analysis of Complexity

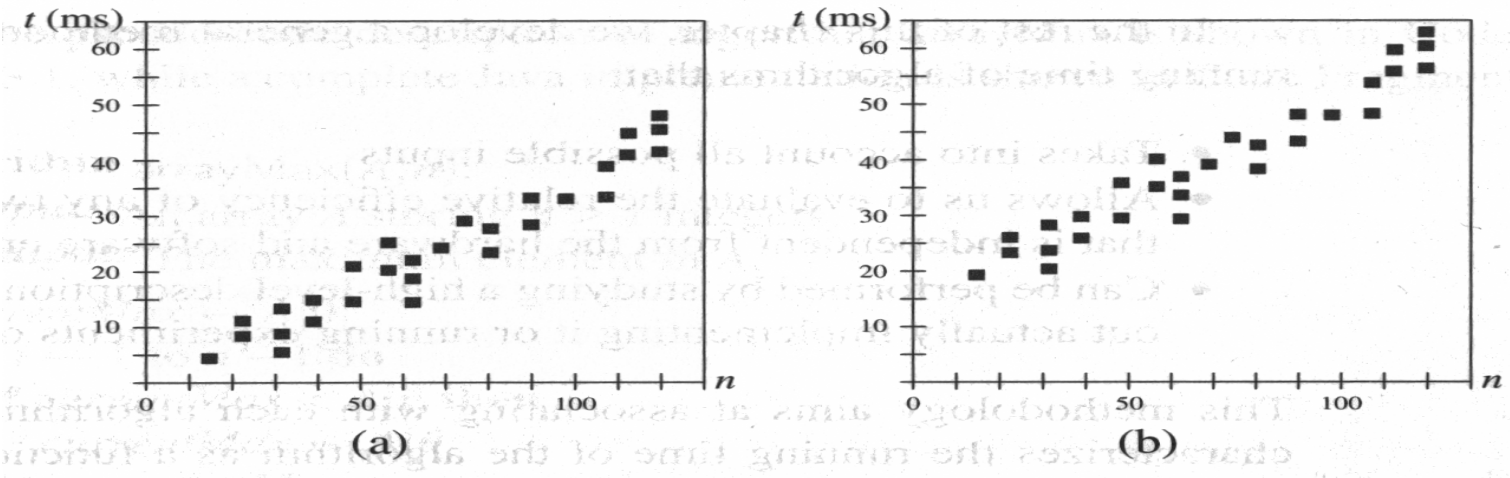
- Once **correct** data structures have been built, **efficient** code is required to manipulate that data - sorting, searching, entering, displaying, saving, etc.
- “Correct” implies “error free” and “robust”.
- “Efficient” mainly means “**minimum running time**”.
- These days, hardware is not a restriction - memory and magnetic storage is cheap. But users must be impatient, otherwise why do processors keep getting faster?

Analysis of Complexity - Cont.

- As datasets get larger and larger, execution time becomes a big issue.
- "**Complexity**" is how a program actually executes - all the steps actually taken that sum up to the measured running time of the program - this is very complex…
- "**Analysis of complexity**" is a means of simplifying this complexity to the point where algorithms can be compared on a simple basis.

Analysis of Complexity - Cont.

• Measure execution time t, in milliseconds, against sample size, n:



• (a) is a faster computer, (b) is a slower one.

Analysis of Complexity - Cont.

- As expected, the actual running time of a program depends not only on the efficiency of the algorithm, but on many other variables:
  - Processor speed & type.
  - Bus speed.
  - Amount of memory.
  - Operating system.
  - Extent of multi-tasking.
  - …

Analysis of Complexity - Cont.

- In order to compare algorithm speeds experimentally, all other variables must be kept constant.
- But experimentation is a lot of work - It would be better to have some theoretical means of predicting algorithm speed, or at least relative speed.
- We want to be able to say that "Algorithm A will be faster than Algorithm B, if both are tested under the same conditions".

Analysis of Complexity - Cont.

- The advantage would be that:
  - It would be independent of the type of input.
  - It would be independent of the hardware and software environment.
  - The algorithm does not need to be coded (and debugged!).

- An example follows.
- But first, define what a "**primitive operation**" is.

Analysis of Complexity - Cont.

- A **primitive operation** takes a unit of time. The actual length of time will depend on external factors such as the hardware and software environment.
- Select operations that would all take about the same length of time. For example:
  - Assigning a value to a variable.
  - Calling a method.
  - Performing an arithmetic operation.
  - Comparing two numbers.
  - Indexing an array element.
  - Following an object reference.
  - Returning from a method.

Analysis of Complexity - Cont.

- Each of these kinds of operations would take the same amount of time on a given hardware and software platform.
- Count primitive operations for a simple method.
- Example - find the highest integer in an array:

Analysis of Complexity - Cont.

```
public static int arrayMax(int[] A) {

   int currentMax = A[0]; // 3 operations
   int n = A.length; // 3 operations
   for (int i = 0; i < n; i++)// 2+3n
      if (currentMax < A[i]) // 3n
        currentMax = A[i]; // 0 to 2n
   return currentMax; // 1
} // end arrayMax
```

- Number of primitive operations:
  - Minimum = 3+3+2+3n+3n+1= 9 + 6n
  - Maximum = 9 + 8n

Analysis of Complexity - Cont.

- Or, **Best case** is 9 + 6n, **Worst case** is 9 + 8n.
- What is the **Average case**?
- If the maximum element had exactly the same probability of being in each of the array locations then the average case would be = 9 + 7n.
- But if data is not randomly distributed, then getting the average case is very difficult.
- It is much easier to just assume the worst case analysis, 9 + 8n.

Analysis of Complexity - Cont.

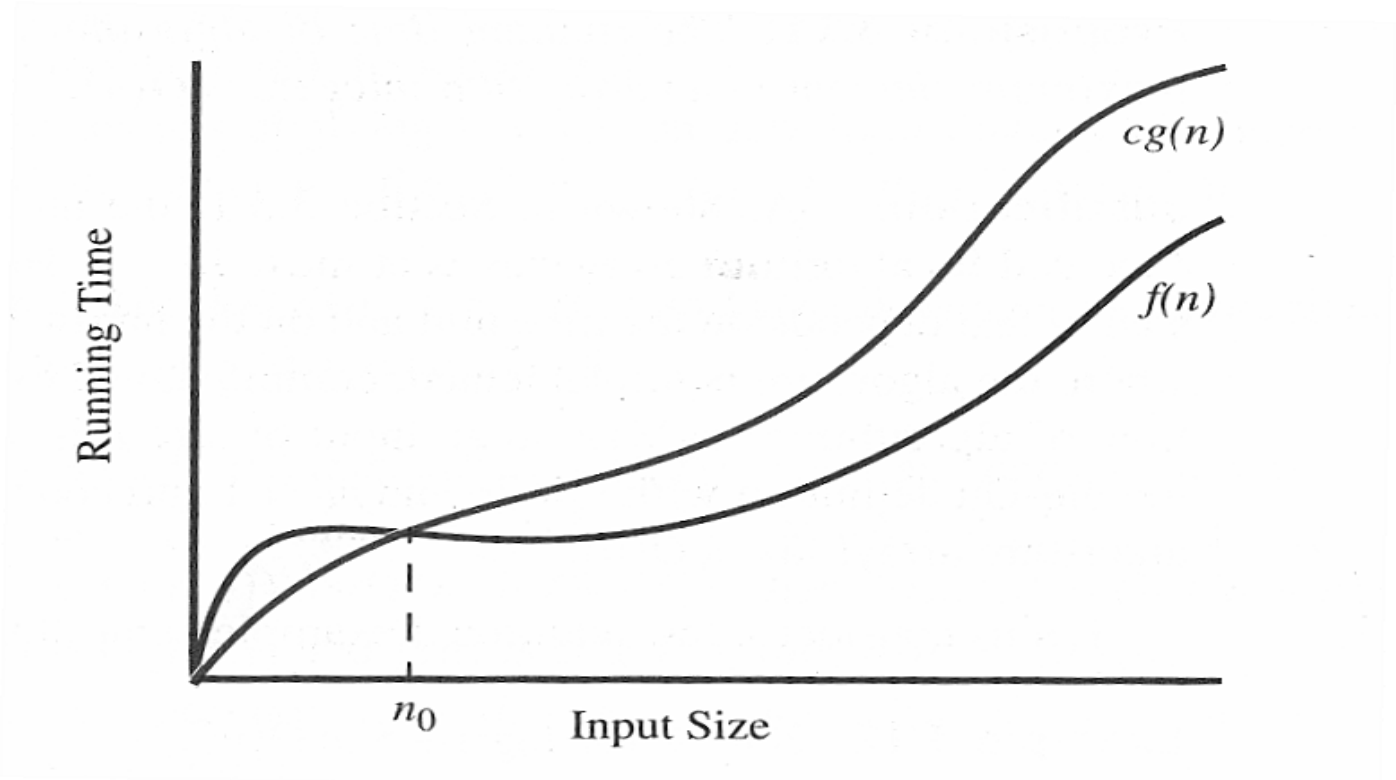
- Consider the case when n gets very large (**Asymptotic Analysis**).
- Then 8n is much greater than 9, so just using 8n is a good approximation.
- Or, it is even easier to say that the running time grows proportionally to n.
- Or the order is to the first power of n, or just first order, or **O(n)**.
- As expected, it makes sense to just ignore the constant values in the equation.

"Big O" Notation

- Mathematical definition of **Big O Notation**:

Let $f(n)$ and $g(n)$ be functions that map nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant, c, where $c > 0$ and an integer constant n_0, where $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$.

"Big O" Notation - Cont.



"Big O" Notation - Cont.

- $f(n)$ is the function that describes the actual time of the program as a function of the size of the dataset, n.
- The idea here is that $g(n)$ is a much simpler function that what $f(n)$ is.
- Given all the assumptions and approximations made to get any kind of a function, it does not make sense to use anything more than the simplest representation.

"Big O" Notation - Cont.

- For example, consider the function:

$$f(n) = n^2 + 100n + \log_{10}n + 1000$$

- For small values of n the last term, 1000, dominates. When n is around 10, the terms 100n + 1000 dominate. When n is around 100, the terms n^2 and 100n dominate. When n gets much larger than 100, the n^2 dominates all others. The $\log_{10}n$ term never gets to play at all!
- So it would be safe to say that this function is $O(n^2)$ for values of $n > 100$.

"Big O" Notation - Cont.

- Similarly a function like:

$$20n^3 + 10n(\log(n)) + 5$$

is $O(n^3)$. (It can be proved that the function is $\leq 35n^3$ for $n \geq 1$. So c is 35 and n_0 is 1.)

"Big O" Notation - Cont.

$3\log(n) + \log(\log(n))$

- is O(log(n)) since the function is $\leq 4\log(n)$ for $n \geq 2$. ($c = 4$, $n_0 = 2$).
- Note that the Big O notation actually provides an infinite series of equations, since c and n_0 can have any number of values.
- The constants c and n_0 are not usually presented when an algorithm is said to be "Big O" of some function - but there are times when it is important to know their magnitude.

"Big O" Notation - Cont.

- Common big O notations:
  - Constant O(1)
  - Logarithmic O(log(n))
  - Linear O(n)
  - N log n O(n(log(n)))
  - Quadratic $O(n^2)$
  - Cubic $O(n^3)$
  - Polynomial $O(n^x)$
  - Exponential $O(2^n)$

"Big O" Notation - Cont.

- How these functions vary with n. Assume a rate of 1 instruction per μsec (micro-second):

"Big O" Notation - Cont.

Notation	**n** 10	10^2	10^3	10^4	10^5	10^6
O(1)	1 µsec	1 µsec	1 µsec	1 µsec	1 µsec	1 µsec
O(log(n))	3 µsec	7 µsec	10 µsec	13 µsec	17 µsec	20 µsec
O(n)	10 µsec	100 µsec	1 msec	10 msec	100 msec	1 sec
O(nlog(n))	33 µsec	664 µsec	10 msec	13.3 msec	1.6 sec	20 sec
$O(n^2)$	100 µsec	10 msec	1 sec	1.7 min	16.7 min	11.6 days
$O(n^3)$	1 msec	1 sec	16.7 min	11.6 days	31.7 years	31709 years
$O(2^n)$	10 msec	3e17 years				

"Big O" Notation - Cont.

- So algorithms of $O(n^3)$ or $O(2^n)$ are not practical for any more than a few iterations.
- Quadratic, $O(n^2)$ (common for nested "for" loops!) gets pretty ugly for n > 1000.

- To figure out the big O notation for more complicated algorithms, it is necessary to understand some mathematical concepts.

Some Math…

- Logarithms and exponents - a quick review:
- By definition:

$$\log_b(a) = c, \text{ when } a = b^c$$

More Math…

- Some rules when using logarithms and exponents - for a, b, c positive real numbers:
  - $\log_b(ac) = \log_b(a) + \log_b(c)$
  - $\log_b(a/c) = \log_b(a) - \log_b(c)$
  - $\log_b(a^c) = c(\log_b(a))$
  - $\log_b(a) = (\log_c(a))/\log_c(b)$
  - $b^{\log_c(a)} = a^{\log_c(b)}$
  - $(b^a)^c = b^{ac}$
  - $b^a b^c = b^{a+c}$
  - $b^a/b^c = b^{a-c}$

More Math…

- Summations:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \ldots + f(b)$$

- If a does not depend on i:

$$\sum_{i=1}^{n} af(i) = a\sum_{i=1}^{n} f(i)$$

More Math…

- Also:

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

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

Still More Math…

- The **Geometric Sum**, for any real number, a > 0 and a ≠ 1.

$$\sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}$$

- Also for 0 < a < 1:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}$$

Still More Math…

- Finally, for a > 0, a ≠ 1, and n ≥ 2:

$$\sum_{i=1}^{n} i a^{i} = \frac{a - (n+1)a^{(n+1)} + n a^{(n+2)}}{(1-a)^{2}}$$

Properties of Big O Notation

1. If d(n) is O(f(n)), then ad(n) is still O(f(n)), for any constant, a > 0.
2. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n) + e(n) is O(f(n) + g(n)).
3. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)e(n) is O(f(n)g(n)).
4. If d(n) is O(f(n)) and f(n) is O(g(n)), then d(n) is O(g(n)).

Properties of Big O Notation – Cont.

5. If f(n) is a polynomial of degree d (ie. $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$, then f(n) is $O(n^d)$.
6. $\log_a(n)$ is $O(\log_b(n))$ for any a, b > 0. That is to say $\log_a(n)$ is $O(\log_2(n))$ for any a > 0 (and $a \neq 1$).
7. $(\log(n))^x$ is $O(n^y)$ for x, y > 0.

"Big O" Notation - Cont.

- While correct, it is considered "bad form" to include constants and lower order terms when presenting a Big O notation.
- For most comparisons, the simplest representation of the notation is sufficient.

"Big O" Notation - Cont.

- For example, show that $2n^3 + 4n^2\log(n)$ is really just $O(n^3)$.
- This could be shown just by looking at the relative numeric significance of the terms as we did for an earlier example, but it is easier to use the properties of Big O notation:
  - log(n) can be considered O(n) in this case (Rule 7)
  - $4n^2$ is $O(n^2)$ (Rule 1)
  - $O(n^2)O(n)$ is $O(n^3)$ (Rule 3)
  - $2n^3$ is $O(n^3)$ (Rule 1)
  - $O(n^3) + O(n^3)$ is $O(n^3)$ (Rule 2)

"Big O" Notation – Cont.

- Basically, just look for the highest order term in a multi-term expression, in order to simplify the notation.

Analysis of Complexity - Cont.

- The analysis of complexity is mathematical in nature.
- So, it is often necessary to offer logical proofs of a conjecture or a hypothesis.
- I will offer only a rapid overview of some mathematical techniques for justification - a complete treatment is beyond the scope of this course.

Simple Justification Techniques

- **Proof by Example**:
  - Sometimes it is sufficient to produce only one example to prove a statement true (but not often).
  - The more common use of "Proof by Example" is to Disprove a statement, by producing a **counterexample**.
  - For example, propose that the equation 2^i-1 only yields prime numbers. To disprove this proposition, it is sufficient to provide the example of i=4 that evaluates to 15, which is not prime.

Simple Justification Techniques - Cont.

- Proof by using the **Contrapositive**:
  - For example to justify the statement “if a is true then b is true”, sometimes it is easier to prove that if “a is false then b is false”.

Simple Justification Techniques - Cont.

- Proof by using **Contradiction**:
  - To prove that statement “q is true”, propose that q is false and then show that this statement leads to a logical contradiction (such as $2 \neq 2$, or $1 > 3$).

Simple Justification Techniques - Cont.

- Proof by **Induction**:
  - Very common since algorithms often use loops.
  - Used to show that q(n) will be true for

    all $n \geq 1$.
  - Start by showing that q(n) is true for n=1, and possibly for n=2,3,…
  - (Induction step) Assume that q is true for some value k, where k < n.
  - Show that if q(k) is true then q(n) is true.

Simple Justification Techniques - Cont.

- Example of proof by Induction:
- Prove that:

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

- Base case, n=1, is true since 1 = 2/2.
- n=2 is true since 3 = 6/2.
- (Induction step) Assume true for some k < n. So if k=n-1:

$$\sum_{i=1}^{k} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$$

Simple Justification Techniques - Cont.

- And:

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

- Then:

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

Application of Big O to Loops

- For example, a single "**`for`**" loop:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
```

- 2 operations before the loop, 6 operations inside the loop. Total is 2 + 6n. This gives the loop **O(n)**.

Application of Big O to Loops - Cont.

- Nested "**`for`**" loops, where both loop until n:

```
for (i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j < n; j++)
        sum[i] += a[i, j];
}
```

- Outer loop: 1+5n+n(inner loop). Inner loop = 1+7n. Total = $1+6n+7n^2$, so algorithm is **$O(n^2)$**.

Application of Big O to Loops - Cont.

- Nested "**`for`**" loops where the inner loop goes less than n times (used in simple sorts):

```
for (i = 0; i < n; i++) {
    sum[i] = 0;
    for (j = 0; j < i; j++)
        sum[i] += a[i, j];
}
```

- Outer loop: 1 + 6n + 7i times, for i = 1, 2, 3, …, n-1.

Application of Big O to Loops - Cont.

- Or:

$$1 + 6n + \sum_{i=1}^{n-1} 7i = 1 + 6n + 7\sum_{i=1}^{n-1} i = 1 + 6n + \frac{7n(n-1)}{2}$$

- Which is O(n) + O(n^2), which is finally just **O(n^2)**.

Application of Big O to Loops - Cont.

- Nested **`for`** loops are usually $O(n^2)$, but not always!

```
for (i = 3; i < n; i++) {
    sum[i] = 0;
    for (j = i-3; j <= i; j++)
        sum[i] += a[i, j];
}
```

- Inner loop runs 4 times for every value of i. So, it is actually **O(n)**, linear, not quadratic.

Another Example - Binary Search

- Binary search is a very efficient technique, but it must be used on a sorted searchset (an array, in this case):

```
// returns the location of "key" in A
// if "key" is not found, -1 is returned
public int binSearch (int[] A, int key) {

  int lo = 0; //lo,hi, and mid are positions
  int hi = A.length - 1;
  int mid = (lo + hi) / 2;

  while (lo <= hi) {
      if (key < A[mid])
            hi = mid - 1;
      else if (A[mid] < key)
            lo = mid + 1;
      else return mid;
      mid = (lo + hi) / 2;
  } // end while
  return -1;
} // end binSearch
```

Another Example - Binary Search - Cont.

- For the best case, the element matches right at the middle of the array, and the loop only executes once.
- For the worst case, **`key`** will not be found and the maximum number of loops will occur.
- Note that the loop will execute until there is only one element left that does not match.
- Each time through the loop the number of elements left is halved, giving the progression below for the number of elements:

Another Example - Binary Search - Cont.

- Number of elements to be searched progression:

$$n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots, \frac{n}{2^m}$$

- The last comparison is for $n/2^m$, when the number of elements is one.
- So, $n/2^m = 1$, or $n = 2^m$.
- Or, $m = \log(n)$.
- So, algorithm is **O(log(n))**.

Another Example - Binary Search - Cont.

- S**equential search** searches by comparing each element in order, starting at the beginning of the array.
- It can use un-sorted arrays.
- Best case is when first element matches.
- Worst case is when last element matches.
- So, sequential search algorithm is **O(n)**.
- Binary search at O(log(n)) is better than sequential at O(n).
- Major reason to sort datasets!

A Caution on the Use of "Big O"

- Big O notation usually ignores the constant c.

- For example, algorithm A has the number of operations = $10^8 n$, and algorithm B has the form $10n^2$.
- A is then O(n), and B is $O(n^2)$.
- Based on this alone, A would be chosen over B.
- But, if c is considered, B would be faster for n up to 10^7 - which is a very large dataset!
- So, in this case, B would be preferred over A, in spite of the Big O notation.

A Caution on the Use of "Big O" – Cont.

- For "mission critical" algorithm comparison, nothing beats experiments where actual times are measured.
- When measuring times, keep everything else constant, just change the algorithm (same hardware, OS, type of data, etc.).

"Relatives" of Big O Notation

- Big Ω, or "Big Omega":

Let $f(n)$ and $g(n)$ be functions that map nonnegative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ if there is a real constant, c, where $c > 0$ and an integer constant n_0, where $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for every integer $n \geq n_0$.

- So, Big Ω can represent a lower bound for f(n), when Big O represents an upper bound.
- Otherwise, Big Ω behaves in much the same way.

"Relatives" of Big O Notation - Cont.

- Big Θ, or "Big Theta":

> Let $f(n)$ and $g(n)$ be functions that map nonnegative integers to real numbers. We say that $f(n)$ is $\Theta(g(n))$ if there are real constants, c_1 *and* c_2, where $c_1, c_2 > 0$ and an integer constant n_0, where $n_0 \geq 1$ such that, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for every integer $n \geq n_0$.

- Big Θ is of interest in asymptotic complexity, since you can see that, for a large enough n, if f(n) is O(g(n)) and f(n) is Ω(g(n)), then f(n) is also Θ(g(n)).