O Notation (Big Oh)

We want to give an upper bound on the amount of time it takes to solve a problem.

defn: $v(n) = O(f(n)) \iff \exists$ constants $c$ and $n_0$ such that $|v(n)| \leq c|f(n)|$ whenever $n > n_0$

Termed complexity: has nothing to do with difficulty of coding or understanding, just time to execute.

Important tool for analyzing and describing the behavior of algorithms

Is an $n^2$ algorithm always better than a $n^3$ algorithm? No, it depends on the specific constants, but if $n$ is large enough, an $O(n^3)$ is always slower.

Complexity Class: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$

For small $n$, $c$ may dominate.

Intractable: all known algorithms are of exponential complexity

Measuring Complexity

1. Additive Property: If two statements follow one another, the complexity of the two of them is the larger complexity.
   
   $O(m) + O(n) = O(\max(n,m))$

2. If/Else: For the fragment
   
   ```
   if cond then S1
   else S2
   ```

   The complexity is the running time of the $cond$ plus the larger of the complexities of $S1$ and $S2$.

3. Multiplicative Property:
   
   For Loops: the complexity of a for loop is at most the complexity of the statements inside the for loop times the number of iterations. However, if each iteration is different in length, it is more accurate to sum the individual lengths rather than just multiply.

Nested For Loops

Analyze nested for loops inside out. The total complexity of a statement inside a group of nested for loops is the complexity of the statement multiplied by the product of the size of each for loop.

Example 1

```for i = 1,m
    for j = 1,n
        x++;
```
O(mn)

Consider a pictorial view of the work. Let each row represent the work done in an iteration. The area of the figure will then represent the total work.

```
  n
  |
  v
m
```

**Example 2**

```c
for (beg = 1; beg < m; beg++)
    for (j = beg; j < m; j++)
        x++;
```

In this example, our complexity picture is triangular.

```
  m
  |
  v
m
```

**Recursive Algorithms**

The complexity for recursive algorithms requires additional techniques. There are three general techniques for finding complexity. The pictorial approach, the formulaic approach, and the mathematics approach. You should be able to use any two of them.

**Example 3**

```c
procedure B(N)
    for i = 1, N
        x = x + i
    Call B(N/2)
    Call B(N/2)
end procedure
```

If we let \( T(n) \) represent the time to solve this problem, the time is represented recursively as \( T(n) = 2T(n/2) + n \). This is called a recurrence relation.
In our pictorial view, we let each line represent a layer of recursions (The first call is the first row, the two calls at the second level comprise the second row, the four third level calls represent the third row.) The picture represents the call itself (ignoring costs incurred by the recursive calls). In other words, to determine the size of the first row, measure how much work is done in that call not counting the recursive calls it makes.

Example 4

```
procedure B(N)
    for i = 1, N
        x = x + i
    Call B(N/2)
end procedure
```

If we let \( T(n) \) represent the time to solve this problem, the time is represented recursively as \( T(n) = 1 + T(n/2) + n \).
Example 5

procedure B(N)
  x = x + i
  Call B(N/2)
end procedure

If we let $T(n)$ represent the time to solve this problem, the time is represented recursively as $T(n) = T(n/2) + 1$.

\[
\begin{array}{c}
\log n \\
O(\log n)
\end{array}
\]

Example 6

procedure B(N)
  x = x + i
  Call B(N/2)
  Call B(N/2)
end procedure

If we let $T(n)$ represent the time to solve this problem, the time is represented recursively as $T(n) = 2T(n/2) + 1$.

\[
\begin{array}{c}
1 \\
2n \\
O(n)
\end{array}
\]

A Formula Approach

Mathematicians have developed a formula approach to determining complexity.

Theorem:

$T(n) = a \ T(n/b) + O(n^k)$

if $a > b^k$ the complexity is $O(n \log_{b^k} a)$

if $a = b^k$ the complexity is $O(n^k \log n)$
if $a < b^k$ the complexity is $O(n^k)$

In this case:

- $a$ represents the number of recursive calls you make
- $b$ represents the number of pieces you divide the problem into
- $k$ represents the power on $n$ which represents the work you do in breaking the problem into two pieces or putting them back together again. In other words, $n^k$ represents the work you do independent of the recursive calls you make. Note this corresponds to one row in our pictorial representation.

Let's use the theorem to revisit the same problems we solved pictorially.

**Example 3**

```plaintext
procedure B(N)
    for i = 1, N
        x = x + i
        Call B(N/2)
        Call B(N/2)
    end procedure
```

$a=2$ $b=2$ $k=1$ (as work is $n$)

Since $a=b^k$, we are in the ``equals'' case.

if $a= b^k$ the complexity is $O(n^k \log n) = O(n \log g)$ which is exactly what our pictures told us.

**Example 4**

```plaintext
procedure B(N)
    for i = 1, N
        x = x + i
        Call B(N/2)
    end procedure
```

$a=1$ $b=2$ $k=1$

Since $a < b^k$ we are in the ``less than'' case.

if $a < b^k$ the complexity is $O(n^k) = O(n)$ which is exactly what our pictures told us.

**Example 5**

```plaintext
procedure B(N)
    x = x + i
    Call B(N/2)
end procedure
```
a=1 b=2 k=0 as the work is 1 or n^0

a = b^k as 1 = 2^0 we are in the `equals`` case.

if a = b^k the complexity is O(n^k log n) = O(log n) which is exactly what our pictures told us.

**Example 6**

```plaintext
procedure B(N)
    x = x + i
    Call B(N/2)
    Call B(N/2)
end procedure
```

a = 2 b = 2 k = 0

Since a > b^k we are in the `greater than`` case.

if a > b^k the complexity is O(n log_b a) = O(n log_2 2) = O(n^1) which is exactly what our pictures told us.

**Recurrence Relations**

In analyzing algorithms, we often encounter progressions. Most calculus books list the formulas below, but you should be able to derive them.

**Arithmetic Progression**

Example: \( S = 3 + 5 + 7 + 9 + \ldots + (2n - 1) + (2n + 1) \)

Writing the same sum backwards:
\( S = (2n + 1) + (2n - 1) + (2n - 3) + (2n - 5) + \ldots + 5 + 3 \)

If we add the two S's together,

\[
S = \begin{align*}
3 & + 5 & + 7 & + 9 & + \ldots & + (2n-1) & + (2n+1) \\
S & = (2n+1) & + (2n-1) & + (2n-3) & + (2n-5) & + \ldots & + 5 & + 3 \\
2S & = (2n+4) & + (2n+4) & + \ldots & + (2n+4) & + (2n+4) \\
2S & = n(2n+4) & & & & & & \\
S & = n(n+2)
\end{align*}
\]

**Geometric Progression**

Example: \( S = 2 + 4 + 8 + 16 + 32 + \ldots + 2^n \)

Multiplying both sides by the base:
\( 2S = 4 + 8 + 16 + 32 + \ldots + 2^n + 2 + 2^n \)

Subtracting S from 2S we get
Mathematically Solving Recurrence Relations

While the math types may like this approach, it usually confuses the non-math types more than helps.

Example 7

Suppose our recurrence relation is:
\[ T_n = 2T_{n/2} + 2 \]
and
\[ T_2 = 1 \]

We use the telescoping technique (plug and chug technique). (You may have learned something you like better in your discrete math classes.) We set up a series of equations. Since we would never stop for infinite \( n \), let \( n = 32 \) to see the pattern, and generalize for any \( n \).

\[
egin{align*}
T_n &= 2T_{n/2} + 2 \\
2T_{n/2} &= 2(2T_{n/4} + 2) = 2^2T_{n/4} + 2^2 \\
2^2T_{n/4} &= 2^2(2T_{n/8} + 2) = 2^3T_{n/8} + 2^3 \\
2^3T_{n/8} &= 2^3(2T_{n/16} + 2) = 2^4T_{n/16} + 2^4 \\
2^4T_{n/16} &= 2^4T_2 = 2^4 + 1 = 2^4 
\end{align*}
\]

Now by substituting each equation into the previous equation we have

\[
T_{32} = 2 + 2^2 + 2^3 + 2^4 + 2^4 = \sum_{i=1}^{4} 2^i + n/2
\]

Notice the upper limit on the summation is related to the size of \( n \). Since \( 2^4 = n/2 \) in this case, we conjecture the upper bound (call it \( b \)) will always be determined by \( 2^b = n/2 \), i.e. \( b = \log_2 n - 1 \). We would actually have to consider a few more cases to convince ourselves this is the actual relationship. Notice, however, that we expect to perform the recursion a logarithmic number of times as we halved the problem size each time.

\[
T_{32} = 2 + 2^2 + 2^3 + 2^4 + 2^b = \sum_{i=1}^{b} 2^i + n/2
\]
using our geometric sum technique

\[ \sum_{i=1}^{b} 2^i = 2^{b+1} - 1 \]

Thus,

\[ T_n = 2^{b+1} - 2 + n/2 = 2 + 2^b - 2 + n/2 = 2(n/2) - 2 + n/2 = 3n/2 - 2 \]

It would be nice to be able to check our work: For the case of \( n = 32 \), we got

\[ T_{32} = 2 + 2^2 + 2^3 + 2^4 = 46 \]

Since \( 3(32)/2 - 2 = 46 \), we are reasonably certain we did the computations correctly.

**Telescoping, again**: If that was confusing, you could also have done the same thing as follows:

\[ T_n = 2T_{n/2} + 2 \]

\[ T_n = 2(2T_{n/4} + 2) + 2 \]

\[ T_n = 4T_{n/4} + 4 + 2 \]

\[ T_n = 4(2T_{n/8} + 2) + 4 + 2 \]

\[ T_n = 8T_{n/8} + 8 + 4 + 2 \]

\[ T_n = 8(2T_{n/16} + 2) + 8 + 4 + 2 \]

\[ T_n = 16T_{n/16} + 16 + 8 + 4 + 2 \]

But since \( n=32 \)

\[ T_n = 16T_2 + 16 + 8 + 4 + 2 \]

\[ T_n = 16 + 16 + 8 + 4 + 2 \]

\[ T_{32} = 2 + 2^1 + 2^5 + 2^4 + 2^4 = \sum_{i=1}^{b} 2^i + n/2 \]

Thus,

\[ T_n = 2^{b+1} - 2 + n/2 = 2 + 2^b - 2 + n/2 = 2(n/2) - 2 + n/2 = 3n/2 - 2 \]

**Example 8**

Suppose the relation was

\[ T_n = 2T_{n/2} + n \]

and

\[ T_1 = 1 \]

This is a common form. You break a problem of size \( n \) into two equal sized pieces (\( n/2 \)) and have to solve each piece (as long as \( n/2 \) is bigger than our base case). The breaking apart of the problem and the accumulating the results takes time proportional to \( n \).
This technique won't always work, but for the type of recurrence relations we see in algorithm analysis, it is often useful.

Divide each side of the recurrence relation by \( n \). The choice will become clearer as we proceed.

\[
\frac{T_n}{n} = \frac{2T_{n/2}}{n/2} + 1
\]

This equation is valid for any power of two, so:

\[
\frac{T_{n/2}}{n/2} = \frac{T_{n/4}}{n/4} + 1
\]

and

\[
\frac{T_{n/4}}{n/4} = \frac{T_{n/8}}{n/8} + 1
\]

\[
\frac{T_{n/8}}{n/8} = \frac{T_{n/16}}{n/16} + 1
\]

\[
\frac{T_2}{2} = \frac{T_1}{1} + 1
\]

Now add up all of the equations. This means we add up all the terms on the left hand side and set the result to the sum of all the terms on the right hand size. Observe, virtually all terms exist on both sides of the equation and may be cancelled. After everything is cancelled, we get

\[
\frac{T_n}{n} = \frac{T_1}{1} + \log n
\]

\[
T_n = n + n \log n
\]

**Example 9**

Consider a sorting algorithm in which you repeatedly search the unsorted portion of an array for the smallest element. The smallest element is swapped with the first element in the unsorted portion, and the lower limit of the unsorted portion is incremented. If we have \( n \) elements, it takes \( n-1 \) compares to find the smallest, 1 swap, and then sort the remaining \( n-1 \) elements.

\[
T_n = n + T_{n-1}
\]

If there are only two elements, there is a compare and at most 1 swap.

\[
T_2 = 2
\]

To see the pattern, let's assume \( n=6 \)

\[
T_n = n + T_{n-1}
\]

\[
T_{n-1} = n - 1 + T_{n-2}
\]
\[ T_{n-2} = n - 2 + T_{n-3} \]

\[ T_{n-3} = n - 3 + T_{n-4} \]

Backwards substitution yields: \[ T_5 = \frac{2}{2} + 3 + 4 + 5 + \cdots \quad \text{in general,} \]
\[ T_n = 2 + 3 + 4 + 5 + \cdots + n = \frac{(n+2)(n-1)}{2} \]