Happy Thursday, April 2.

*Analysis of algorithms* is concerned with determining how much time (or memory) an algorithm uses, as a function of the size of the input.

We will be spending just a little time looking at analysis of algorithms. The ideas are simple. The key is to pay attention to definitions and facts. Read the material and learn the facts.

**Functions**

You should be familiar polynomials, such as \( n^2 + 5n \). For our purposes, the only characteristic of a polynomial that matters is the degree of the polynomial. \( n^2 + 5n \) is a quadratic polynomial (degree 2).

An important function is \( \log_2(n) \), the logarithm to base 2 of \( n \). The logarithm grows very slowly as \( n \) grows. You can get an estimate of \( \log_2(n) \) by starting with \( n \) and doing a sequence of halvings, stopping at 1, rounding down to the nearest integer at each step. Suppose that \( n = 10 \).

\[
\begin{align*}
10 \\
5 \\
2 \\
1 
\end{align*}
\]

There are 4 numbers, but only three steps of halving and rounding down to the nearest integer. That tells you that \( \log_2(10) \approx 3 \), because it took 3 steps of halving. In fact, the estimate is off by no more than 1; \( 3 \leq \log_2(10) \leq 4 \). Let’s do the same thing starting at 35.

\[
\begin{align*}
35 \\
17 \\
8 \\
4 \\
2 \\
1 
\end{align*}
\]

It takes 5 steps of halving to reach 1. So \( 5 \leq \log_2(35) \leq 6 \).
Exercises

Read page 35A in the notes. Do the exercises at the bottom of page 35A.

Big-O notation

We would like to get a rough estimate of how large a function is. Suppose that $f(n)$ and $g(n)$ are two functions of $n$.

We say that $f(n)$ is $O(g(n))$ (f(n) is “big Oh” of $g(n)$) provided there is a constant $c$ so that $f(n) \leq cg(n)$.

That definition is not complicated. Here are some examples.

Example. $n^2$ is $O(3n^2 + 1)$. Choose $c = 1$.

Example. $3n^2 + 1$ is $O(n^2)$. Choose $c = 4$. Notice that

$$3n^2 + 1 \leq 3n^2 + n^2 = 4n^2$$

All you need to remember about polynomials is this.

1. Suppose that $f(n)$ and $g(n)$ are both polynomials of degree $d$. Then $f(n)$ is $O(g(n))$.

2. Suppose that $f(n)$ is a polynomial of degree $d_f$ and $g(n)$ is a polynomial of degree $d_g$. Then $f(n)$ is $O(g(n))$ exactly when $d_f \leq d_g$.

Example. $n^4$ is $O(5n^5)$ because $n^4$ has degree 4, $5n^5$ has degree 5 and $4 \leq 5$.

Example. $n^5$ is not $O(n^4)$ because $5n^5$ has degree 5, $n^4$ has degree 4, and $5 \nleq 4$.

Big-Theta notation

There is another notation that is more precise than big-O notation. Θ is an upper case Greek letter theta.

Suppose that $f(n)$ and $g(n)$ are two functions of $n$. We say that $f(n)$ is $\Theta(g(n))$ (f(n) is big-Theta of $g(n)$) provided
1. \( f(n) \) is \( O(g(n)) \).

2. \( g(n) \) is \( O(f(n)) \).

For polynomials, all you need to know is: Suppose \( f(n) \) is a polynomial of degree \( d_f \) and \( g(n) \) is a polynomial of degree \( d_g \). Then \( f(n) \) is \( \Theta(g(n)) \) exactly when \( d_f = d_g \).

For example:

**Example.** \( 3n^3 \) is \( \Theta(20n^3 + n^2) \).

**Example.** \( 10n^2 + 2 \) is \( \Theta(n^2) \).

**Example.** \( n^2 \) is not \( \Theta(n^3) \).

**Example.** \( n^3 \) is not \( \Theta(n^2) \).

### Big-O and big-Theta notation and algorithms

When we want to know how efficient an algorithm is, we ideally find a function \( f(n) \) so that the algorithm takes time that is \( \Theta(f(n)) \) on inputs of size \( n \).

**Example.** Suppose that \( s \) is a null-terminated string of length \( n \). Function \( \text{strlen}(s) \) takes time that is \( \Theta(n) \) to find the length of \( s \). Why? Because it looks at each character in \( s \) once.

**Example.** Suppose that \( L \) is a linked list whose length is \( n \). It takes time that is \( \Theta(n) \) to find the length of \( L \).

**Example.** Suppose that \( s \) is a null-terminated string of length \( n \). How much time does it take to compute \( \text{strlen}(s) \) \( n \) times? If you buy 20 things and they cost $5 each, then you pay $100. You multiply. If you do \( n \) steps and each step takes time about \( n \), then the total time is about \( n^2 \). You multiply. So it takes time \( \Theta(n^2) \) to compute \( \text{strlen}(s) \) \( n \) times.

**Example.** Suppose that \( x \) and \( y \) are two values of type \text{int}. It takes only one machine-language instruction to compute \( x + y \). Obviously, that is a fixed amount of time. If \( f(n) = 1 \), then \( f(n) \) is a polynomial of degree 0, and \( f(n) \) is \( \Theta(1) \).

### Big-O and big-Theta notation and logarithms

Logarithms grow very slowly, much slower than any polynomial. We will encounter algorithms whose time function is \( \Theta(n \log_2(n)) \). That is
only slightly worse than $\Theta(n)$. Here are a few approximate values of $n$, $n \log_2(n)$ and $n^2$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log_2(n)$</th>
<th>$n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>700</td>
<td>10,000</td>
</tr>
<tr>
<td>1000</td>
<td>10,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>10,000</td>
<td>300,000</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>

**Exercises**

Read page 36A in the notes. Do the exercises at the bottom of page 36A.