Graph Proof (cont)

There are two cases:
- $S(H) \cup \{v\}$ is independent.
  Then $S(G) = S(H) \cup \{v\}$.
- $S(H) \cup \{v\}$ is not independent.
  Let $w \in S(H)$ such that $(w, v) \in E$.
  Every vertex in $N(v)$ can be reached by $w$ with path of length $\leq 2$.
  So, $S(G) = S(H)$.

By Strong Induction, the theorem holds for all $G$.

Fibonacci Numbers

Define Fibonacci numbers inductively as:

\[
F(1) = F(2) = 1 \\
F(n) = F(n-1) + F(n-2), n > 2.
\]

Theorem: $\forall n \geq 1, F(n)^2 + F(n+1)^2 = F(2n+1)$.

Induction Hypothesis:

$F(n-1)^2 + F(n)^2 = F(2n-1)$.

Fibonacci Numbers (3)

With a stronger theorem comes a stronger IH!

Theorem:

\[
F(n)^2 + F(n+1)^2 = F(2n+1) \text{ and} \\
F(n)^2 + 2F(n)F(n-1) = F(2n).
\]

Induction Hypothesis:

$F(n-1)^2 + F(n)^2 = F(2n-1)$ and

$F(n-1)^2 + 2F(n-1)F(n-2) = F(2n-2)$.

Another Example

Theorem: All horses are the same color.

Proof: $P(n)$: If $S$ is a set of $n$ horses, then all horses in $S$ have the same color.

Base case: $n = 1$ is easy.

Induction Hypothesis: Assume $P(i), i < n$.

Induction Step:
- Let $S$ be a set of horses, $|S| = n$.
- Let $S' = S - \{h\}$ for some horse $h$.
- By IH, all horses in $S'$ have the same color.
- Let $h'$ be some horse in $S'$.
- IH implies $\{h, h'\}$ have all the same color.

Therefore, $P(n)$ holds.
Algorithm Analysis

- We want to “measure” algorithms.
- What do we measure?
- What factors affect measurement?
- Objective: Measures that are independent of all factors except input.

Time Complexity

- Time and space are the most important computer resources.
- Function of input: \( T(\text{input}) \)
- Growth of time with size of input:
  - Establish an (integer) \( n \) for inputs
  - \( n \) numbers in a list
  - \( n \) edges in a graph
- Consider time for all inputs of size \( n \):
  - Time varies widely with specific input
  - Best case
  - Average case
  - Worst case
- Time complexity \( T(n) \) counts steps in an algorithm.

Asymptotic Analysis

- It is undesirable/impossible to count the exact number of steps in most algorithms.
  - Instead, concentrate on main characteristics.
- Solution: Asymptotic analysis
  - Ignore small cases:
    - Consider behavior approaching infinity
  - Ignore constant factors, low order terms:
    - \( 2n^2 \) looks the same as \( 5n^2 + n \) to us.

O Notation

O notation is a measure for “upper bound” of a growth rate.
- pronounced “Big-oh”

Definition: For \( T(n) \) a non-negatively valued function, \( T(n) \) is in the set \( O(f(n)) \) if there exist two positive constants \( c \) and \( n_0 \) such that \( T(n) \leq cf(n) \) for all \( n > n_0 \).

Examples:
- \( 5n + 8 \in O(n) \)
- \( 2n^2 + n \log n \in O(n^2) \in O(n^3 + 5n^2) \)
- \( 2n^2 + n \log n \in O(n^2) \in O(n^3 + n^2) \)
O Notation (cont)

We seek the “simplest” and “strongest” \( f \).

Big-O is somewhat like “\( \leq \)";
\[ n^2 \in O(n^2) \text{ and } n^2 \log n \in O(n^2), \]
- \( n^2 \neq n^2 \log n \)
- \( n^2 \in O(n^2) \) while \( n^2 \log n \notin O(n^2) \)

A common misunderstanding:
- “The best case for my algorithm is \( n = 1 \) because that is the fastest.” WRONG!
- Big-oh refers to a growth rate as \( n \) grows to \( \infty \).
- Best case is defined for the input of size \( n \) that is cheapest among all inputs of size \( n \).

Growth Rate Graph

2\( ^n \) is an exponential algorithm. 10\( n \) and 20\( n \) differ only by a constant.

Speedups

What happens when we buy a computer 10 times faster?

\[
\begin{array}{|c|c|c|c|c|}
\hline
T(n) & n & n' & \text{Change} & n'/n \\
\hline
10n & 1,000 & 10,000 & n' = 10n & 10 \\
20n & 500 & 5,000 & n' = 10n & 10 \\
5n \log n & 250 & 1,842 & \sqrt{10n} < n' \leq 10n & 7.37 \\
2^n & 70 & 223 & n' = \sqrt{10n} & 3.16 \\
2^n & 13 & 16 & n' = n + 3 & \ldots \\
\hline
\end{array}
\]

- \( n \): Size of input that can be processed in one hour (10,000 steps).
- \( n' \): Size of input that can be processed in one hour on the new machine (100,000 steps).

Some Rules for Use

**Definition:** \( f \) is monotonically growing if \( n_1 \geq n_2 \) implies \( f(n_1) \geq f(n_2) \).

We typically assume our time complexity function is monotonically growing.

**Theorem 3.1:** Suppose \( f \) is monotonically growing,
\[ \forall c > 0 \text{ and } \forall a > 1, (f(n))^c \in O(a^{f(n)}) \]

In other words, an exponential function grows faster than a polynomial function.

**Lemma 3.2:** If \( f(n) \in O(s(n)) \) and \( g(n) \in O(r(n)) \) then
- \( f(n) + g(n) \in O(s(n) + r(n)) \equiv O(\max(s(n), r(n))) \)
- \( f(n)g(n) \in O(s(n)r(n)) \).
- If \( s(n) \in O(h(n)) \) then \( f(n) \in O(h(n)) \)
- For any constant \( k \), \( f(n) \in O(k\cdot s(n)) \)

Assume monotonically growing because larger problems should take longer to solve. However, many real problems have “cyclically growing” behavior.

Is \( O(2^{(n)}(n)) \in O(3^{(n)}(n)) \)? Yes, but not vice versa.

\( 3^n = 1.5^n \times 2^n \) so no constant could ever make 2\( ^n \) bigger than 3\( ^n \) for all \( n \).

functional composition
**Other Asymptotic Notation**

\( \Omega(f(n)) \) – lower bound (\( \geq \))

**Definition:** For \( T(n) \) a non-negatively valued function, \( T(n) \) is in the set \( \Omega(g(n)) \) if there exist two positive constants \( c \) and \( n_0 \) such that \( T(n) \geq cg(n) \) for all \( n > n_0 \).

Ex: \( n^2 \log n \in \Omega(n^2) \).

\( \Theta(f(n)) \) – Exact bound (\( = \))

**Definition:** \( g(n) = \Theta(f(n)) \) if \( g(n) \in O(f(n)) \) and \( g(n) \in \Omega(f(n)) \).

**Important:** It is \( \Theta \) if it is both in big-Oh and in \( \Omega \).

Ex: \( 5n^2 + 4n^2 + 9n + 7 = \Theta(n^2) \)

**Other Asymptotic Notation (cont)**

\( o(f(n)) \) – little o (\( < \))

**Definition:** \( g(n) \in o(f(n)) \) if \( \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \)

Ex: \( n^2 \in o(n^3) \)

\( \omega(f(n)) \) – little omega (\( > \))

**Definition:** \( g(n) \in \omega(f(n)) \) if \( f(n) \in o(g(n)) \).

Ex: \( n^2 \in \omega(n^2) \)

\( \infty(f(n)) \)

**Definition:** \( T(n) = \infty(f(n)) \) if \( T(n) = O(f(n)) \) but the constant in the \( O \) is so large that the algorithm is impractical.

**Aim of Algorithm Analysis**

Typically want to find “simple” \( f(n) \) such that \( T(n) = \Theta(f(n)) \).

- Sometimes we settle for \( O(f(n)) \).

Usually we measure \( T \) as “worst case” time complexity.

Sometimes we measure “average case” time complexity.

**Approach:** Estimate number of “steps”

- Appropriate step depends on the problem.

- Ex: measure key comparisons for sorting

**Summation:** Since we typically count steps in different parts of an algorithm and sum the counts, techniques for computing sums are important (loops).

**Recurrence Relations:** Used for counting steps in recursion.

**Analyzing Problems**

To an algorithm designer, what would it mean to solve a problem?

- Upper bound: The upper bound for the best algorithm that we know.

- Lower bound: The best (biggest) lower bound possible for any algorithm to solve the problem.

Lower bounds are hard!

We know that we understand our problem when the bounds match.

Example: Sorting

Example: Find the minimum value in an unsorted list.

\( \Omega(n^2) \) most useful to discuss cost of problems, not algorithms.

Once you have an equation, the bounds have met. So this is more interesting when discussing your level of uncertainty about the difference between the upper and lower bound.

You have \( \Omega \) when you have the upper and the lower bounds meeting. So \( \Omega \) means that you know a lot more than just Big-oh, and so is preferred when possible.

A common misunderstanding:

- Confusing worst case with upper bound.

- Upper bound refers to a growth rate.

- Worst case refers to the worst input from among the choices for possible inputs of a given size.