Graph Proof (cont)

There are two cases:

1. $S(H) \cup \{v\}$ is independent.
   Then $S(G) = S(H) \cup \{v\}$.
2. $S(H) \cup \{v\}$ 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 ≤ 2.
   So, set $S(G) = S(H)$.

By Strong Induction, the theorem holds for all G.

Graph Proof (cont)

There are two cases:
1. $S(H) \cup \{v\}$ is independent.
   Then $S(G) = S(H) \cup \{v\}$.
2. $S(H) \cup \{v\}$ 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 ≤ 2.
   So, set $S(G) = S(H)$.

By Strong Induction, the theorem holds for all G.

"$S(H) \cup \{v\}$ is not independent" means that there is an edge from something in $S(H)$ to v.
IMPORTANT: There cannot be an edge from v to $S(H)$ because whatever we can reach from v is in $N(v)$ and would have been removed in H.
We need strong induction for this proof because we don't know how many vertices are in $N(v)$.
We must remove $N(v)$ instead of just v because of this case: We remove just v to yield H. S(H) turns out to have something that can be reached from v. So, when we add v back to reform G, v cannot become part of S(G) (because that would violate the definition of independent set). But if v is 3 steps away from anything in S(H), we must add it to satisfy the theorem. So are stuck.

Fibonacci Numbers

Define Fibonacci numbers inductively as:

$$\begin{aligned} F(1) &= F(2) = 1 \\ F(n) &= F(n-1) + F(n-2), n > 2. \end{aligned}$$

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 (2)

Expand both sides of the theorem, then cancel like terms:
$F(2n+1) = F(2n) + F(2n-1)$ and,

$$\begin{aligned} F(n)^2 + F(n+1)^2 &= F(n)^2 + (F(n) + F(n-1))^2 \\ &= F(n)^2 + F(n)^2 + 2F(n)F(n-1) + F(n-1)^2 \\ &= F(n)^2 + F(n-1)^2 + F(n)^2 + 2F(n)F(n-1) \\ &= F(2n-1) + F(n)^2 + 2F(n)F(n-1). \end{aligned}$$

Want: $F(n)^2 + F(n+1)^2 = F(2n+1) = F(2n) + F(2n-1)$
Steps above left us with needing to prove:
$F(2n) + F(2n-1) = F(2n-1) + F(n)^2 + 2F(n)F(n-1)$
So we need to show that: $F(2n) = F(n)^2 + 2F(n)F(n-1)$
To prove the original theorem, we must prove this. Since we must do it anyway, we should take advantage of this in our IH!

Fibonacci Numbers (3)

With a stronger theorem comes a stronger IH!

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

Induction Hypothesis:
$F(n-1)^2 + F(n)^2 = F(2n-1)$ and
$F(n-1)^2 + 2F(n-1)F(n-2) = F(2n-2)$.

Fibonacci Numbers (4)

$$\begin{aligned} & F(n)^2 + 2F(n)F(n-1) \\ &= F(n)^2 + 2(F(n-1) + F(n-2))F(n-1) \\ &= F(n)^2 + F(n-1)^2 + 2F(n-1)F(n-2) + F(n-1)^2 \\ &= F(2n-1) + F(2n-2) \\ &= F(2n). \end{aligned}$$

$$\begin{aligned} F(n)^2 + F(n+1)^2 &= F(n)^2 + [F(n) + F(n-1)]^2 \\ &= F(n)^2 + F(n)^2 + 2F(n)F(n-1) + F(n-1)^2 \\ &= F(n)^2 + F(2n) + F(n-1)^2 \\ &= F(2n-1) + F(2n) \\ &= F(2n+1). \end{aligned}$$

... which proves the theorem. The original result could not have been proved without the stronger induction hypothesis.

Another Example

Theorem: All horses are the same color.

Proof: $\mathbf{P}(n)$: 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 $\mathbf{P}(i), i < n$.
Induction Step:

- Let S be a set of horses, $|S| = n$.
- Let S' be $S - \{h\}$ 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, $\mathbf{P}(n)$ holds.

Another Example

Theorem: All horses are the same color.
Proof: $\mathbf{P}(n)$: 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 $\mathbf{P}(i), i < n$.
Induction Step:
- Let S be a set of horses, $|S| = n$.
- Let S' be $S - \{h\}$ 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.

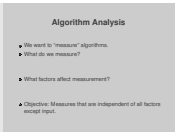
Therefore, $\mathbf{P}(n)$ holds.

The problem is that the base case does not give enough strength to give the **particular** instance of $n = 2$ used in the last step.

If it were true for 2, then the whole proof woudl work. But we cannot get from the base case to an arbitrary 2.

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.



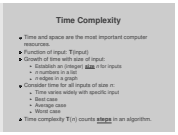
What do we measure?
Time and space to run; ease of implementation (this changes with language and tools); code size

What affects measurement?
Computer speed and architecture; Programming language and compiler; System load; Programmer skill; Specifics of input (size, arrangement)

If you compare two programs running on the same computer under the same conditions, all the other factors (should) cancel out.
Want to measure the relative efficiency of two algorithms without needing to implement them on a real computer.

Time Complexity

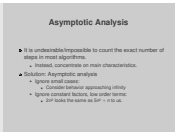
- Time and space are the most important computer resources.
- Function of input: **T**(input)
- Growth of time with size of input:
  - Establish an (integer) **size** n for inputs
  - n numbers in a list
  - n edges in a graph
- Consider time for all inputs of size n:
  - Time varies widely with specific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



Sometimes analyze in terms of more than one variable.
Best case usually not of interest.
Average case is usually what we want, but can be hard to measure.
Worst case appropriate for "real-time" applications, often best we can do in terms of measurement.
Examples of "steps:" comparisons, assignments, arithmetic/logical operations. What we choose for "step" depends on the algorithm. Step cost must be "constant" – not dependent on n.

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.



Undesirable to count number of machine instructions or steps because issues like processor speed muddy the waters.

O Notation

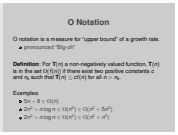
O notation is a measure for "upper bound" of a growth rate.

- pronounced "Big-oh"

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

Examples:

- $5n + 8 \in \mathrm{O}(n)$
- $2n^2 + n\log n \in \mathrm{O}(n^2) \in \mathrm{O}(n^3 + 5n^2)$
- $2n^2 + n\log n \in \mathrm{O}(n^2) \in \mathrm{O}(n^3 + n^2)$



Remember: The time equation is for some particular set of inputs – best, worst, or average case.

O Notation (cont)

We seek the "simplest" and "strongest" f.

Big-O is somewhat like "$\leq$":
$n^2 \in \mathrm{O}(n^3)$ and $n^2 \log n \in O(n^3)$, but

- $n^2 \neq n^2 \log n$
- $n^2 \in \mathrm{O}(n^2)$ while $n^2 \log n \notin \mathrm{O}(n^2)$

O Notation (cont)

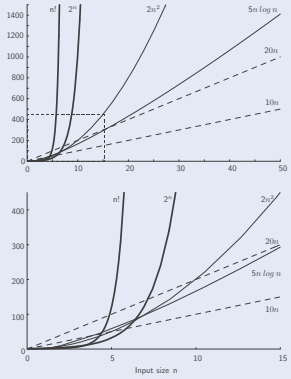
We seek the "simplest" and "strongest" f.

Big-O is somewhat like "≤":
$n^2 \in O(n^3)$ and $n^2 \log n \in O(n^3)$, but
- $n^2 \neq n^2 \log n$
- $n^2 \in O(n^2)$ while $n^2 \log n \notin O(n^2)$

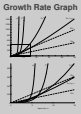
A common misunderstanding:

- "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 ∞.
- Best case is defined for the input of size n that is cheapest among all inputs of size n.

Growth Rate Graph



Growth Rate Graph



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

Speedups

What happens when we buy a computer 10 times faster?

$\mathbf{T}(n)$	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

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).

Speedups

What happens when we buy a computer 10 times faster?

T(n)	n	n'	Change	n'/n
10n	1,000	10,000	n' = 10n	10
20n	500	5,000	n' = 10n	10
5nlog n	250	1,842	√10n < n' < 10n	7.37
2n²	70	223	n' = √10n	3.16
2^n	13	16	n' = n + 3	--

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).

How much speedup? 10 times. More important: How much increase in problem size for same time? Depends on growth rate.
For n^2, if $n = 1000$, then n' would be 1003.
Compare $\mathbf{T}(n) = n^2$ to $\mathbf{T}(n) = n \log n$. For $n > 58$, it is faster to have the $\Theta(n \log n)$ algorithm than to have a computer that is 10 times faster.

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.
$\forall c > 0$ 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, $f(n) \in O(ks(n))$

Some Rules for Use

Definition: f is monotonically growing if n₁ ≥ n₂ implies f(n₁) ≥ f(n₂).
We typically assume our time complexity function is monotonically growing.

Theorem 3.1: Suppose f is monotonically growing. ∀c > 0 and ∀a > 1, (f(n))^c ∈ O(a^f(n))
In other words, an exponential function grows faster than a polynomial function.

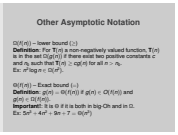
Lemma 3.2: If f(n) ∈ O(s(n)) and g(n) ∈ O(r(n)) then
- f(n) + g(n) ∈ O(s(n) + r(n)) ≡ O(max(s(n), r(n)))
- f(n)g(n) ∈ O(s(n)r(n)).
- If s(n) ∈ O(h(n)) then f(n) ∈ O(h(n))
- For any constant k, f(n) ∈ O(ks(n))

Assume monitonic growth because larger problems should take longer to solve. However, many real problems have "cyclically growing" behavior.
Is $O(2^{f(n)}) \in O(3^{f(n)})$? 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 $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ 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 Θ if it is both in big-Oh and in Ω.
Ex: $5n^3 + 4n^2 + 9n + 7 = \Theta(n^3)$



Ω is most userful 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 Θ when you have the upper and the lower bounds meeting. So Θ means that you know a lot more than just Big-oh, and so is perferred when possible.

A common misunderstanding:

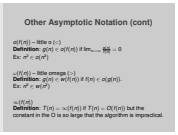
- Confusing worst case with upper bound.
- Upper bound refers to a growth rate.
- Worst case refers to the worst input from among the choices for possible inputs of a given size.

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: $g(n) \in w(f(n))$ if $f(n) \in o(g(n))$.
Ex: $n^5 \in w(n^2)$

$\infty(f(n))$
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.



We won't use these too much.

Aim of Algorithm Analysis

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

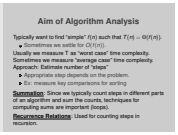
- Sometimes we settle for $O(f(n))$.

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.



We prefer Θ over Big-oh because Θ means that we understand our bounds and they met. But if we just can't find that the bottom meets the top, then we are stuck with just Big-oh. Lower bounds can be hard. For **problems** we are often interested in Ω

– but this is often hard for non-trivial situations!

Often prefer average case (except for real-time programming), but worst case is simpler to compute than average case since we need not be concerned with distribution of input.

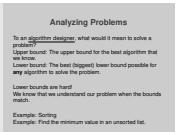
For the sorting example, key comparisons must be constant-time to be used as a cost measure.

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.



Sorting: If you only know simple sorts, your upper bound is $O(n^2)$.
Then you learn better sorts and your upper bound is $O(n \log n)$
A naive lower bound is $\Omega(n)$. Later we learn the proof that no (general) sorting algorithm can have a worst case better than $\Omega(n \log n)$.
At that point, we know that sorting is $\Theta(n)$. Minimum Finding:

The upper bound is $O(n)$ because we know an algorithm to solve it in that time.
The lower bound is $\Omega(n)$ because we have to look at every value to be sure we have the answer