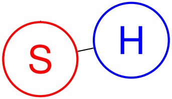
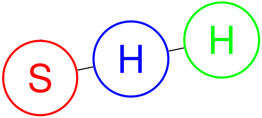
CS170: Algorithms

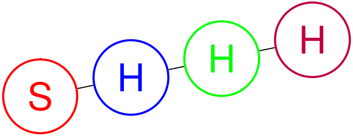
CS170: Algorithms


S
H

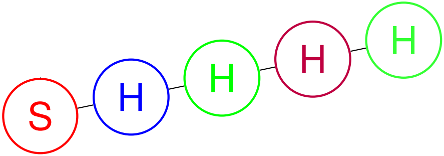
CS170: Algorithms



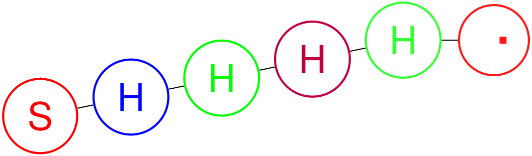
CS170: Algorithms



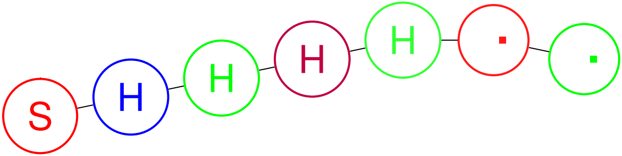
CS170: Algorithms



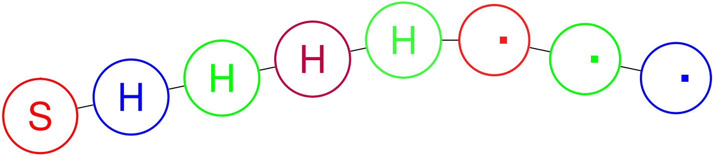
CS170: Algorithms



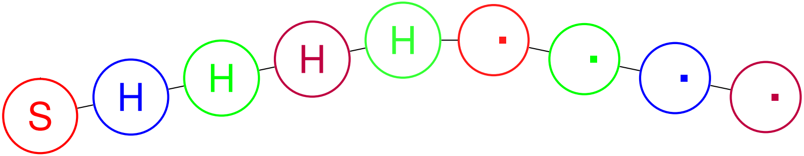
CS170: Algorithms



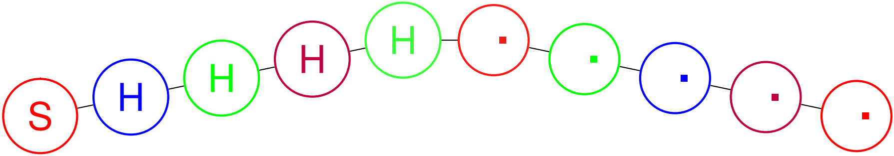
CS170: Algorithms



CS170: Algorithms



CS170: Algorithms



Lecture in a minute.

Horn Formula:

Implications of positive literals with ANDs on one side.

Plus ORs of negatives.

Negative clauses problem only with true literals.

Greedy: only set true if have to.

Lecture in a minute.

Horn Formula:

Implications of positive literals with ANDs on one side.

Plus ORs of negatives.

Negative clauses problem only with true literals.

Greedy: only set true if have to.

Lecture in a minute.

Horn Formula:

Implications of positive literals with ANDs on one side.

Plus ORs of negatives.

Negative clauses problem only with true literals.

Greedy: only set true if have to.

Set Cover:

Given subsets of some elements.

Find: min number of sets that contains every element.

Greedy: choose largest set w.r.t. remaining elements.

$O(\log n)$ approximate solution.

Proof Idea:optimal of size $k \implies$ Cover $1/k$ of the remaining elts.

Lecture in a minute.

Horn Formula:
  Implications of positive literals with ANDs on one side.
  Plus ORs of negatives.
 Negative clauses problem only with true literals.
  Greedy: only set true if have to.

Set Cover:
  Given subsets of some elements.
  Find: min number of sets that contains every element.
  Greedy: choose largest set w.r.t. remaining elements.
  $O(\log n)$ approximate solution.
  Proof Idea:optimal of size $k \implies$ Cover $1/k$ of the remaining elts.

Path Compression:
  $O(m \log^* n)$ time for m finds.
  Some finds expensive but cheap on average.
  Idea: group ranks into $log^* n$ sets.
    Small number of pointers across sets in any find.
    Total movement inside sets $O(n)$.
     Idea: from not more than 2^k nodes of rank k.

Taint Analysis

Taint Analysis

$$a = http.read\_response();$$

Taint Analysis

$a = http.read\_response();$

$\vdots$

Taint Analysis

$$a = http.read\_response();$$

$$\vdots$$

$$b = a + c;$$

$$\vdots$$

Taint Analysis

$$a = http.read\_response();$$
$$\vdots$$
$$b = a + c;$$
$$\vdots$$
$$d = sql\_command(b);$$

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."

Taint Analysis

$$a = http.read\_response();$$

$$\vdots$$

$$b = a + c;$$

$$\vdots$$

$$d = sql\_command(b);$$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:

Taint Analysis

$$a = \mathit{http.read\_response}();$$
$$\vdots$$
$$b = a + c;$$
$$\vdots$$
$$d = \mathit{sql\_command}(b);$$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"

Taint Analysis

$$a = http.read\_response();$$

$$\vdots$$

$$b = a + c;$$

$$\vdots$$

$$d = sql\_command(b);$$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

$\implies A$

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

$\implies A\ ,\ A \implies B$

Taint Analysis

$$a = http.read\_response();$$
$$\vdots$$
$$b = a + c;$$
$$\vdots$$
$$d = sql\_command(b);$$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

$$\implies A\,,\, A \implies B\,,\, \overline{B}.$$

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

$$\implies A \; , \; A \implies B \; , \; \overline{B}.$$

Satisfiable?

Taint Analysis

$a = http.read\_response();$

$\vdots$

$b = a + c;$

$\vdots$

$d = sql\_command(b);$

a is input from web.
"*a* is tainted."
"if *a* is tainted *b* is tainted."
"*b* should not be tainted."
Logic representation:
A - "a is tainted"
B - "b is tainted"

$\implies A\ ,\ A \implies B\ ,\ \overline{B}.$

Satisfiable?

Not in this case.

Horn SAT

Implications:

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$x \wedge y \to z$

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \to z$$

Negative clauses:

Or of negative literals.

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A,\ A \implies B,\ \overline{B}.$$

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A, A \implies B, \overline{B}.$$

Is this satisfiable?

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A, A \implies B, \overline{B}.$$

Is this satisfiable?

True is the problem.

Horn SAT

Implications:
And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:
Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A, A \implies B, \overline{B}.$$

Is this satisfiable?

True is the problem.
If every literal is False:

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A,\ A \implies B,\ \overline{B}.$$

Is this satisfiable?

True is the problem.

If every literal is False:

All $\wedge$ implication statements are good.

Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A, A \implies B, \overline{B}.$$

Is this satisfiable?

True is the problem.

If every literal is False:

All $\wedge$ implication statements are good.

All $\vee$ statements are true.

Horn SAT

Implications:
And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:
Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A,\ A \implies B,\ \overline{B}.$$

Is this satisfiable?

True is the problem.
If every literal is False:
All $\wedge$ implication statements are good.
All $\vee$ statements are true.
except for implication: $\implies A$.

Horn SAT

Implications:
And of positive literals imply **one** positive literal.

$x \wedge y \rightarrow z$

Negative clauses:
Or of negative literals.

$\overline{u} \vee \overline{v}$

Taint Example:

$\implies A$, $A \implies B$, $\overline{B}$.

Is this satisfiable?

True is the problem.
If every literal is False:
All $\wedge$ implication statements are good.
All $\vee$ statements are true.
except for implication: $\implies A$.

This forces a true literal.

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Problem: Find consistent assignment with fewest “True” literals.

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm:

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True

Horn Sat: another view.

$$\begin{aligned} x_1 \wedge x_2 &\implies x_4 \\ x_3 &\implies x_2 \\ x_1 &\implies x_3 \\ x_5 \wedge x_1 &\implies x_3 \\ x_2 \wedge x_6 &\implies x_5 \\ &\implies x_1 \end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\Longrightarrow x_4 \\
x_3 &\Longrightarrow x_2 \\
x_1 &\Longrightarrow x_3 \\
x_5 \wedge x_1 &\Longrightarrow x_3 \\
x_2 \wedge x_6 &\Longrightarrow x_5 \\
&\Longrightarrow x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True

Horn Sat: another view.

$$\begin{aligned} x_1 \wedge x_2 &\implies x_4 \\ x_3 &\implies x_2 \\ x_1 &\implies x_3 \\ x_5 \wedge x_1 &\implies x_3 \\ x_2 \wedge x_6 &\implies x_5 \\ &\implies x_1 \end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True

Horn Sat: another view.

$$\begin{aligned} x_1 \wedge x_2 &\implies x_4 \\ x_3 &\implies x_2 \\ x_1 &\implies x_3 \\ x_5 \wedge x_1 &\implies x_3 \\ x_2 \wedge x_6 &\implies x_5 \\ &\implies x_1 \end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True
Solution:

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True
Solution: $\{x_1, x_2, x_3, x_4\}$ are True

Horn Sat: another view.

$$\begin{aligned} x_1 \wedge x_2 &\implies x_4 \\ x_3 &\implies x_2 \\ x_1 &\implies x_3 \\ x_5 \wedge x_1 &\implies x_3 \\ x_2 \wedge x_6 &\implies x_5 \\ &\implies x_1 \end{aligned}$$

Problem: Find consistent assignment with fewest “True” literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True
Solution: $\{x_1, x_2, x_3, x_4\}$ are True

Could also set x_5 to true, or both x_5 and x_6 to true...

Horn Sat: another view.

$$\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True
Solution: $\{x_1, x_2, x_3, x_4\}$ are True

Could also set x_5 to true, or both x_5 and x_6 to true...but don't!

Horn Sat: another view.

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

Problem: Find consistent assignment with fewest "True" literals.

Greedy algorithm: Only set literals to True if you have to.

Example:
x_1 must be True so x_3 must be True
so x_2 must be True so x_4 must be True
Solution: $\{x_1, x_2, x_3, x_4\}$ are True

Could also set x_5 to true, or both x_5 and x_6 to true...but don't!

Same as horn sat!

Why same as HornSAT?

Horn SAT had negative clauses.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true...

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
is set to true to satisfy a clause

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
    is set to true to satisfy a clause
        so it must be true.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
is set to true to satisfy a clause
so it must be true.

Horn has negative clauses.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
    is set to true to satisfy a clause
        so it must be true.

Horn has negative clauses.

Negative clauses only problem for true variables.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
  is set to true to satisfy a clause
    so it must be true.

Horn has negative clauses.

Negative clauses only problem for true variables.

Any variable that is true must be true.

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
    is set to true to satisfy a clause
        so it must be true.

Horn has negative clauses.

Negative clauses only problem for true variables.

Any variable that is true must be true.

So if a negative clause is false, it must be.

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

...no polynomial time algorithm unless NP = P ...

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

...no polynomial time algorithm unless NP = P ...

"P = NP"?

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

...no polynomial time algorithm unless NP = P ...

"P = NP"?

"There is an efficient algorithm to **find** a solution for every problem whose solution can be efficiently **checked.**"

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

...no polynomial time algorithm unless NP = P ...

"P = NP"?

"There is an efficient algorithm to **find** a solution for every problem whose solution can be efficiently **checked.**"

More later...in the course.

Set Cover.

Input:

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Set Cover.

Input:
Items: $B = \{1,\ldots,n\}$
Sets: $S_1,\ldots,S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.

Set Cover.

Input:
Items: $B = \{1,\ldots,n\}$
Sets: $S_1,\ldots,S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).
Sets: People with job capabilities.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).
Sets: People with job capabilities.

Items: factory needs (touch screens, chips,

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).
Sets: People with job capabilities.

Items: factory needs (touch screens, chips, cheap labor).

Set Cover.

Input:
Items: $B = \{1, \dots, n\}$
Sets: $S_1, \dots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).
Sets: People with job capabilities.

Items: factory needs (touch screens, chips, cheap labor).
Sets: suppliers.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby,perl,python, web,unix,...).
Sets: People with job capabilities.

Items: factory needs (touch screens, chips, cheap labor).
Sets: suppliers.
(Thousands of supliers for GM!!)

Greedy Algorithm

Choose set S_i that has largest number of elts.

Greedy Algorithm

Choose set S_i that has largest number of elts.
    Remove elements in S_i from all sets

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Greedy Algorithm

Choose set S_i that has largest number of elts.

Remove elements in S_i from all sets (Rinse).

Repeat.

Number of sets is number of iterations.

Greedy Algorithm

Choose set S_i that has largest number of elts.
    Remove elements in S_i from all sets (Rinse).
    Repeat.

Number of sets is number of iterations.
How many iterations?

Greedy Algorithm

Choose set S_i that has largest number of elts.
Remove elements in S_i from all sets (Rinse).
Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

Greedy Algorithm

Choose set S_i that has largest number of elts.
    Remove elements in S_i from all sets (Rinse).
    Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \leq$$

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \leq n_t - \frac{1}{k}n_t$$

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \le n_t - \frac{1}{k}n_t = (1 - \frac{1}{k})n_t.$$

Greedy Algorithm

Choose set S_i that has largest number of elts.
Remove elements in S_i from all sets (Rinse).
Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \leq n_t - \frac{1}{k}n_t = (1 - \frac{1}{k})n_t.$$

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

Greedy Algorithm

Choose set S_i that has largest number of elts.
  Remove elements in S_i from all sets (Rinse).
  Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ a set contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \leq n_t - \frac{1}{k}n_t = (1 - \frac{1}{k})n_t.$$

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

When do we stop?

Bound iterations.

When do we stop?

Bound iterations.

When do we stop?

When $n_t < 1$?

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must this be true?

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must this be true?

(A) $t = \log n$

(B) $t = k$

(C) $t = k \ln n$.

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must this be true?

(A) $t = \log n$

(B) $t = k$

(C) $t = k \ln n$.

(C).

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must this be true?

(A) $t = \log n$

(B) $t = k$

(C) $t = k \ln n$.

(C).

Plug in $t = k \ln n$ and $n_t < 1$.

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must this be true?

(A) $t = \log n$

(B) $t = k$

(C) $t = k \ln n$.

(C).

Plug in $t = k \ln n$ and $n_t < 1$.

In more detail...

Bound iterations (really)

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Bound iterations (really)

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

When must $n_t < 1$?

Of course you remember:

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$

Bound iterations (really)

$$n_t \le (1 - \frac{1}{k})^t n_0$$

When must $n_t < 1$?

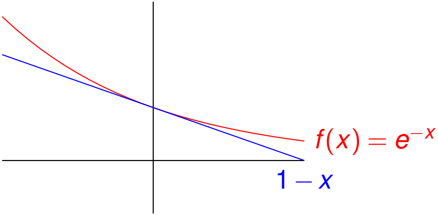
Of course you remember: $(1 - x) \le e^{-x}$ Smile!

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!

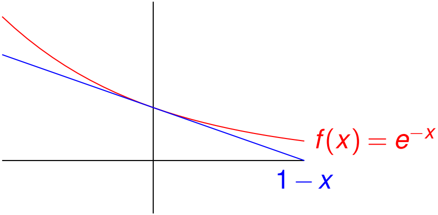


Bound iterations (really)

$n_t \le (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \le e^{-x}$ Smile!

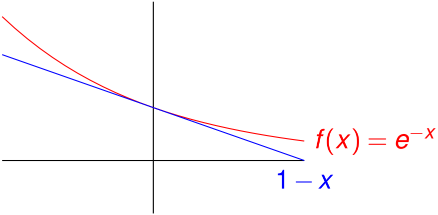


So, $n_t \le (1 - \frac{1}{k})^t n$

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!

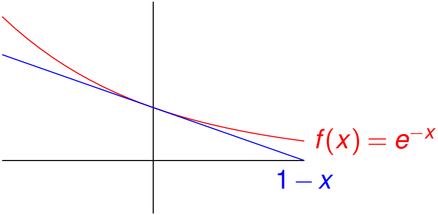


So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n$

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



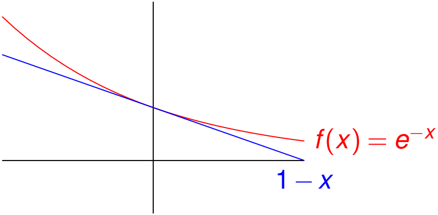
So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

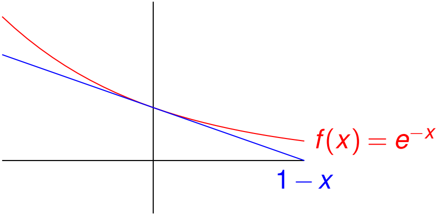
For $t = k \ln n$,

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

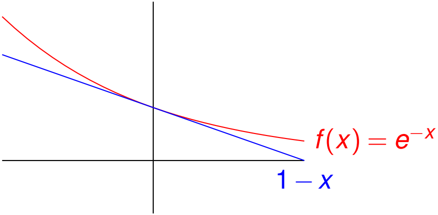
For $t = k \ln n$, $n_t < (e^{-\ln n})n$

Bound iterations (really)

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



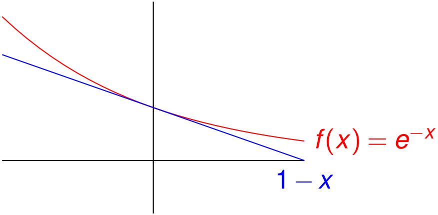
So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}}) n$.

For $t = k \ln n$, $n_t < (e^{-\ln n}) n = (\frac{1}{n}) n$

Bound iterations (really)

$n_t \le (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \le e^{-x}$ Smile!



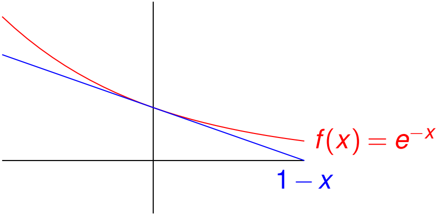
So, $n_t \le (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \le (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



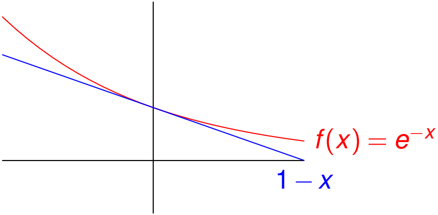
So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.
No elements are uncovered at this time!

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.
No elements are uncovered at this time!

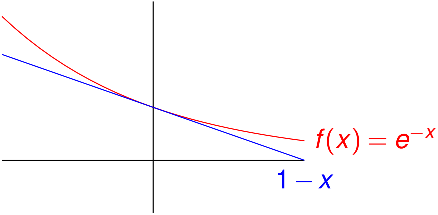
So $t \leq k \ln n$. Number of sets for greedy is at most $k \ln n$!

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.
No elements are uncovered at this time!

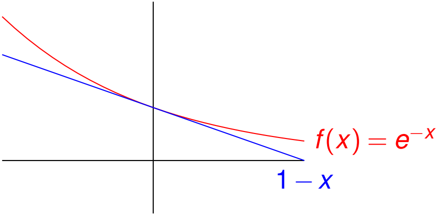
So $t \leq k \ln n$. Number of sets for greedy is at most $k \ln n$!

Within $\ln n$ of k,

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$
When must $n_t < 1$?

Of course you remember: $(1 - x) \leq e^{-x}$ Smile!



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.
No elements are uncovered at this time!

So $t \leq k \ln n$. Number of sets for greedy is at most $k \ln n$!

Within $\ln n$ of k, which is the best possible!

Hmmm...

We did not find optimal solution!

Hmmm...

We did not find optimal solution!

Is there a better analysis?

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No.

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

 Idea: Two sets cover all the elements.

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Idea: Two sets cover all the elements.
One set covers slightly more than half the remaining elements.

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

- Idea: Two sets cover all the elements.
  - One set covers slightly more than half the remaining elements.
  - Give $\Omega(\log n)$ lower bound.

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Idea: Two sets cover all the elements.
One set covers slightly more than half the remaining elements.
Give $\Omega(\log n)$ lower bound.

Is there a better algorithm?

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

- Idea: Two sets cover all the elements.
  - One set covers slightly more than half the remaining elements.
  - Give $\Omega(\log n)$ lower bound.

Is there a better algorithm?

"Probably" not!

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Idea: Two sets cover all the elements.
One set covers slightly more than half the remaining elements.
Give $\Omega(\log n)$ lower bound.

Is there a better algorithm?

"Probably" not!

Again, only if P=NP.

Hmmm...

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Idea: Two sets cover all the elements.
One set covers slightly more than half the remaining elements.
Give $\Omega(\log n)$ lower bound.

Is there a better algorithm?

"Probably" not!

Again, only if P=NP.

More later in the course.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Properties:

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Properties:
(1) Parent has a strictly higher rank.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Properties:
(1) Parent has a strictly higher rank.
(2) Rank doesn't change for internal nodes.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Properties:
(1) Parent has a strictly higher rank.
(2) Rank doesn't change for internal nodes.
(3) $rank(x) = rank(y) = k$

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

find(x)
    if $\pi(x) == x$
        return x
    else
        find($\pi(x)$)

union(x,y)
    r_x = find(x)
    r_y = find(y)
    if rank(r_x) $<$ rank(r_y):
        $\pi(r_x) = r_y$
    else:
        $\pi(r_y) = r_x$
        if rank(r_x) == rank(r_y):
            rank(r_x) $+= 1$

Properties:
(1) Parent has a strictly higher rank.
(2) Rank doesn't change for internal nodes.
(3) $rank(x) = rank(y) = k$
  (i) Each have $\geq 2^k$ vertices in sets

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x. Initially: rank(x) $= 0$.

makeset(x) $\pi(x) = x$.

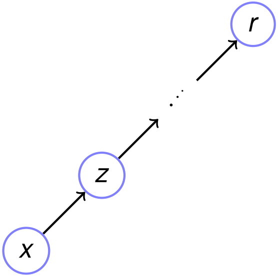
find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

union(x,y)
  r_x = find(x)
  r_y = find(y)
  if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
  else:
    $\pi(r_y) = r_x$
    if rank(r_x) == rank(r_y):
      rank(r_x) $+= 1$

Properties:
(1) Parent has a strictly higher rank.
(2) Rank doesn't change for internal nodes.
(3) $rank(x) = rank(y) = k$
  (i) Each have $\geq 2^k$ vertices in sets
  (ii) and the sets are disjoint.

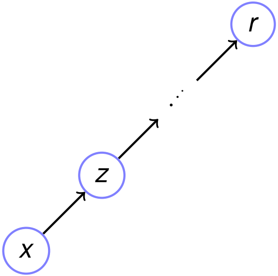
Path Compression

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)



Path Compression

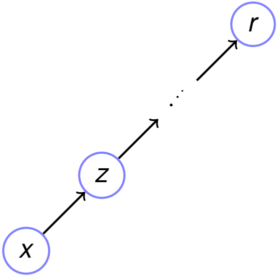
find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)



What happens if we find(x) again?

Path Compression

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

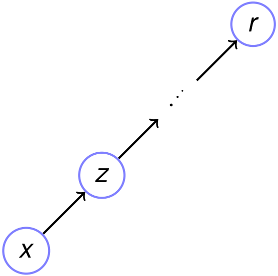


What happens if we find(x) again?

Chase again!

Path Compression

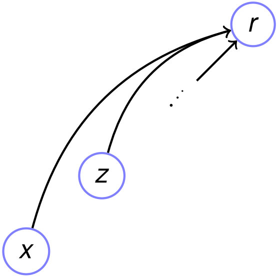
find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)



What happens if we find(x) again?

Chase again!

find(x)
  if $\pi(x) == x$
    return x
  else
    $\pi(x) = \text{find}(\pi(x))$
    return $\pi(x)$



Is this better..

..asymptotically?

Is this better..

..asymptotically?

Take a deep breath.

Is this better..

..asymptotically?

Take a deep breath.

Fancy stuff..next!

Is this better..

..asymptotically?

Take a deep breath.

Fancy stuff..next!

Don’t worry.

Is this better..

..asymptotically?

Take a deep breath.

Fancy stuff..next!

Don’t worry.

...do try..

Is this better..

..asymptotically?

Take a deep breath.

Fancy stuff..next!

Don’t worry.

...do try..you’ll get smarter!

Path Compression Analysis

Union is same.

Path Compression Analysis

Union is same. Only affects root nodes.

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
    rank to parent is higher

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

Every find is asymptotically faster?

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
    rank to parent is higher and $\geq 2^k$ node in rank k

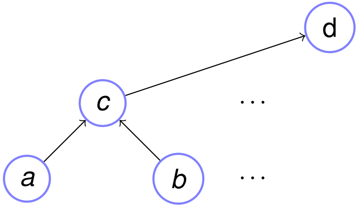
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

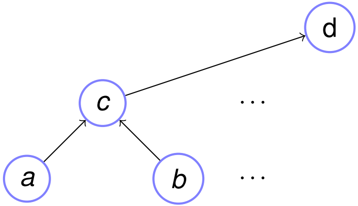
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c)

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
    rank to parent is higher and $\geq 2^k$ node in rank k

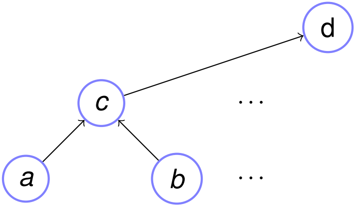
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.

rank to parent is higher and $\geq 2^k$ node in rank k

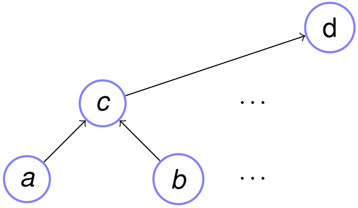
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)

. . .

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
rank to parent is higher and $\geq 2^k$ node in rank k

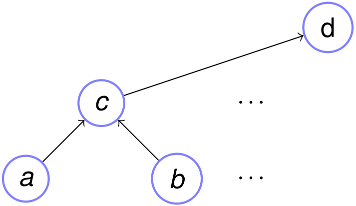
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)
$\cdots$ union(c,d)

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.

rank to parent is higher and $\geq 2^k$ node in rank k

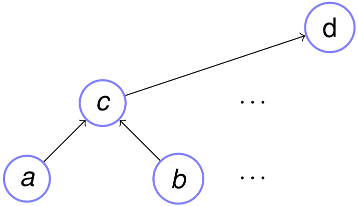
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)

$\cdots$ union(c,d)

union subtree roots to build tree

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
  rank to parent is higher and $\geq 2^k$ node in rank k

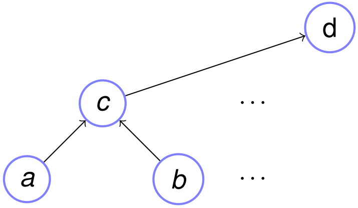
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)
  $\cdots$ union(c,d)
union subtree roots to build tree
find(a)

Path Compression Analysis

Union is same. Only affects root nodes.

Rank properties still hold.
    rank to parent is higher and $\geq 2^k$ node in rank k

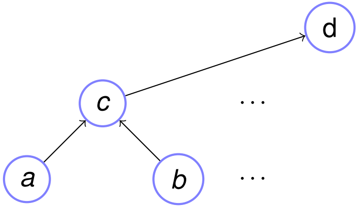
Every find is asymptotically faster?

No. Can make a find take $\Theta(\log n)$ time.

Do you see how?

(A) Yes

(B) No



union(a,c) union(b,c)
    $\cdots$ union(c,d)
union subtree roots to build tree
find(a)
$\Theta(\log n)$ time for this find.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$,

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

$\log 1,000,000$ versus $\log^* 1,000,000$?

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

$\log 1,000,000$ versus $\log^* 1,000,000$?

20 versus 5.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

$\log 1,000,000$ versus $\log^* 1,000,000$?

20 versus 5.

$\log 1,000,000^{1,000,000}$ versus $\log^* 1,000,000^{1,000,000}$?

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

$\log 1,000,000$ versus $\log^* 1,000,000$?

20 versus 5.

$\log 1,000,000^{1,000,000}$ versus $\log^* 1,000,000^{1,000,000}$?

$20,000,000$ versus 6.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time.

$O(\log^* n)$ time on average!

$\log^* n$ is number of times one takes $\log$ to get to 1.

$\log^*(16)$?

(A) 4

(B) 2

(C) 3

C. $\log 16 = 4$, $\log 4 = 2$, $\log 2 = 1$. 3 times.

Also $2^{2^2} = 16$. height of powers of two!

$\log 1,000,000$ versus $\log^* 1,000,000$?

20 versus 5.

$\log 1,000,000^{1,000,000}$ versus $\log^* 1,000,000^{1,000,000}$?

$20,000,000$ versus 6.

Grows very slowly.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Who else amortizes?

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Who else amortizes?

Bankers!

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Who else amortizes?

Bankers!

Hand out some money

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Who else amortizes?

Bankers!

Hand out some money
..... use it to pay for each pointer change.

Amortized Analysis.

Show that m finds take $O(m \log^* n)$ time in total.

$O(\log^* n)$ time on average!

Amortize cost = average over many operations.

Who else amortizes?

Bankers!

Hand out some money
..... use it to pay for each pointer change.

Only hand out $O(m \log^* n)$ dollars.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1,\ldots 2^k\} \cdots$

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\},\{2,3,4\},\{5,\ldots,16\}\cdots\{k+1,\ldots 2^k\}\cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1,\ldots 2^k\} \cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1, \ldots 2^k\} \cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B. Each group grows by powering two!

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1, \ldots 2^k\} \cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B. Each group grows by powering two!

How many internal nodes ever get rank r?

(A) $O(n/2^r)$

(B) $\Theta(n)$

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1,\ldots 2^k\} \cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B. Each group grows by powering two!

How many internal nodes ever get rank r?

(A) $O(n/2^r)$

(B) $\Theta(n)$

A.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\}, \{2,3,4\}, \{5,\ldots,16\} \cdots \{k+1,\ldots 2^k\} \cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B. Each group grows by powering two!

How many internal nodes ever get rank r?

(A) $O(n/2^r)$

(B) $\Theta(n)$

A. Each contained $\geq 2^r$ nodes when root.

Handing out dollars.

Will hand out money to internal nodes
.....to pay for them changing pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Divide non-zero ranks into levels.

$\{1\},\{2,3,4\},\{5,\ldots,16\}\cdots\{k+1,\ldots 2^k\}\cdots$

How many groups of ranks?

(A) $\Theta(\log n)$

(B) $\Theta(\log^* n)$

B. Each group grows by powering two!

How many internal nodes ever get rank r?

(A) $O(n/2^r)$

(B) $\Theta(n)$

A. Each contained $\geq 2^r$ nodes when root. Separate nodes.

Handing out money!

Will hand out money to internal nodes

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

If in set of ranks $\{k+1, \ldots, 2^k\}$ give node 2^k dollars.

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

If in set of ranks $\{k+1, \ldots, 2^k\}$ give node 2^k dollars.

$O(n/2^r)$ internal nodes of rank r.

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
    rank will no longer change!

If in set of ranks $\{k+1,\ldots,2^k\}$ give node 2^k dollars.

$O(n/2^r)$ internal nodes of rank r.

Total Doled out:

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

If in set of ranks $\{k+1,\ldots,2^k\}$ give node 2^k dollars.

$O(n/2^r)$ internal nodes of rank r.

Total Doled out:
In a group:

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

If in set of ranks $\{k+1, \ldots, 2^k\}$ give node 2^k dollars.

$O(n/2^r)$ internal nodes of rank r.

Total Doled out:
In a group: $2^k(n/2^{k+1} + n/2^{k+2} \cdots + n/2^{2^k}) = O(n)$.

Handing out money!

Will hand out money to internal nodes
.....since they change pointers in find.

Notice: When a node becomes an internal node.
rank will no longer change!

If in set of ranks $\{k+1,\ldots,2^k\}$ give node 2^k dollars.

$O(n/2^r)$ internal nodes of rank r.

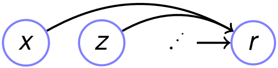
Total Doled out:
In a group: $2^k(n/2^{k+1}+n/2^{k+2}\cdots+n/2^{2^k})=O(n)$.
$O(\log^* n)$ groups. Total money: $O(n\log^* n)$.

Bounding find cost.

Bound cost of find operation.

Bounding find cost.

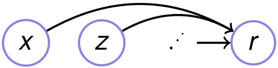
Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

Bounding find cost.

Bound cost of find operation.

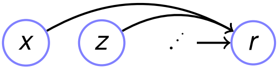


rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

Bounding find cost.

Bound cost of find operation.

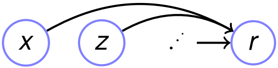


$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

Bounding find cost.

Bound cost of find operation.



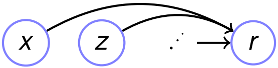
$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

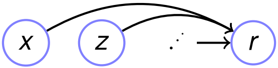
$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

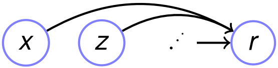
$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

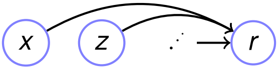
cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

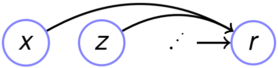
Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

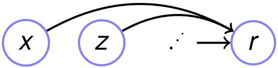
Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

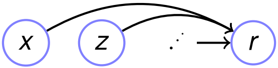
$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

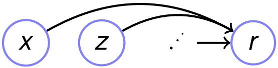
Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

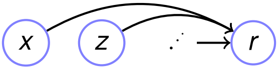
Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

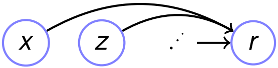
Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

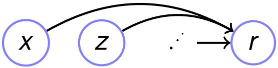
Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

only 2^k ranks in group.

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

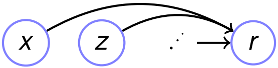
Enough money?

only 2^k ranks in group.

each node in group has 2^k dollars.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

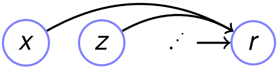
Enough money?

only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

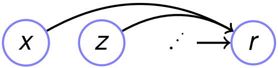
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$.

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

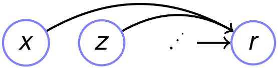
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost:

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

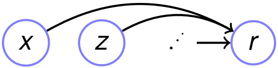
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n)\log^* n)$!

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

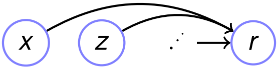
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n) \log^* n)$! !

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

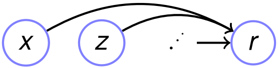
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n)\log^* n)$! ! !

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

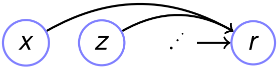
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\Longrightarrow$ Total find cost: $O((m+n)\log^* n)$! ! ! !

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

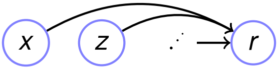
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n)\log^* n)$! ! ! ! !

Bounding find cost.

Bound cost of find operation.



$rank(x)$ and $rank(r)$ in different groups.

$rank(z)$ and $rank(r)$ in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1, \ldots, 2^{k+1}\}$

Enough money?

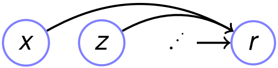
only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n)\log^* n)$! ! ! ! !
!

Bounding find cost.

Bound cost of find operation.



rank(x) and *rank*(r) in different groups.

rank(z) and *rank*(r) in same group.

$O(1)$ plus

cost of changing pointers to point to higher ranked nodes

Per Operation part.

$O(\log^* n)$ pointers that point to node to a higher group.

Total per operation cost over m finds: $O(m \log^* n)$.

Amortized Part.

Node uses its dollars to pay for changing a pointer within group.

Recall group: $\{k+1,\ldots,2^{k+1}\}$

Enough money?

only 2^k ranks in group.

each node in group has 2^k dollars. Enough money!

Total money: $O(n \log^* n)$. $\implies$ Total find cost: $O((m+n)\log^* n)$! ! ! ! !
! !

Instant Replay

Intuition:

Instant Replay

Intuition:

Some operations may be expensive.

Instant Replay

Intuition:

Some operations may be expensive.
...but modify data structure so they won't be in future.

Instant Replay

Intuition:

Some operations may be expensive.
...but modify data structure so they won't be in future.
(Path Compression!)

Instant Replay

Intuition:

Some operations may be expensive.
...but modify data structure so they won't be in future.
(Path Compression!)

Place credits in data structure to pay for these modifications.

Instant Replay

Intuition:

Some operations may be expensive.
...but modify data structure so they won't be in future.
(Path Compression!)

Place credits in data structure to pay for these modifications.

Still..

Instant Replay

Intuition:

Some operations may be expensive.
...but modify data structure so they won't be in future.
(Path Compression!)

Place credits in data structure to pay for these modifications.

Still..

fancy business.

Ackerman’s function.

Can we do better?

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Grows fast.

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Grows fast.

$f^{-1}(n)$ grows slowly!

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Grows fast.

$f^{-1}(n)$ grows slowly! For example $f^{-1}((10^6)^{10^6}) = 5$.

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Grows fast.

$f^{-1}(n)$ grows slowly! For example $f^{-1}((10^6)^{10^6}) = 5$.

Ackermans function grows even faster: computable but grows faster than any primitive recursive function.

Ackerman's function.

Can we do better?

$f(k)$ is $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ of height k.

Grows fast.

$f^{-1}(n)$ grows slowly! For example $f^{-1}((10^6)^{10^6}) = 5$.

Ackermans function grows even faster: computable but grows faster than any primitive recursive function.

There is MST algorithm that runs in $O(m\alpha(m,n))$ where $\alpha(m,n)$ is inverse Ackerman's function.

Lecture in a minute.

Horn Formula:

- Implications of positive literals with ANDs on one side.
- Plus ORs of negatives.

Negative clauses problem only with true literals.

- Greedy: only set true if have to.

Lecture in a minute.

Horn Formula:
  Implications of positive literals with ANDs on one side.
  Plus ORs of negatives.
 Negative clauses problem only with true literals.
  Greedy: only set true if have to.

Lecture in a minute.

Horn Formula:

Implications of positive literals with ANDs on one side.

Plus ORs of negatives.

Negative clauses problem only with true literals.

Greedy: only set true if have to.

Set Cover:

Given subsets of some elements.

Find: min number of sets that contains every element.

Greedy: choose largest set w.r.t. remaining elements.

$O(\log n)$ approximate solution.

Proof Idea:optimal of size $k \implies$ Cover $1/k$ of the remaining elts.

Lecture in a minute.

Horn Formula:
Implications of positive literals with ANDs on one side.
Plus ORs of negatives.
Negative clauses problem only with true literals.
Greedy: only set true if have to.

Set Cover:
Given subsets of some elements.
Find: min number of sets that contains every element.
Greedy: choose largest set w.r.t. remaining elements.
$O(\log n)$ approximate solution.
Proof Idea:optimal of size $k \implies$ Cover $1/k$ of the remaining elts.

Path Compression:
$O(m \log^* n)$ time for m finds.
Some finds expensive but cheap on average.
Idea: group ranks into $log^* n$ sets.
Small number of pointers across sets in any find.
Total movement inside sets $O(n)$.
Idea: from not more than 2^k nodes of rank k.