15-251
Great Theoretical Ideas in Computer Science
Counting

The Power of One
Algebra

The Power of X
Asymptotics

The Power of O
How to add 2 n-bit numbers
How to add 2 n-bit numbers
How to add 2 n-bit numbers
How to add 2 n-bit numbers
How to add 2 n-bit numbers
How to add 2 n-bit numbers

“Grade school addition”
Time complexity of grade school addition

\[ T(n) = \text{amount of time grade school addition uses to add two } n\text{-bit numbers} \]
Time complexity of grade school addition

What do we mean by “time”?
Our Goal

We want to define “time” in a way that transcends implementation details and allows us to make assertions about grade school addition in a very general yet useful way.
Roadblock ???

A given algorithm will take different amounts of time on the same inputs depending on such factors as:

- Processor speed
- Instruction set
- Disk speed
- Brand of compiler
On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.
On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.

Pick any particular computer \( M \) and define \( c \) to be the time it takes to perform on that computer.
On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.

Pick any particular computer $M$ and define $c$ to be the time it takes to perform $+$ on that computer.

Total time to add two $n$-bit numbers using grade school addition:
On any reasonable computer, adding 3 bits and writing down the two bit answer can be done in constant time.

Pick any particular computer $M$ and define $c$ to be the time it takes to perform on that computer.

Total time to add two $n$-bit numbers using grade school addition:

$$cn \quad [i.e., \: c \: \text{time for each of } n \: \text{columns}]$$
On another computer $M'$, the time to perform $\square$ may be $c'$. 
On another computer $M'$, the time to perform $M$ may be $c'$.

Total time to add two $n$-bit numbers using grade school addition:
On another computer $M'$, the time to perform may be $c'$. 

Total time to add two $n$-bit numbers using grade school addition:

$c'n$ [c’ time for each of $n$ columns]
# of bits in the numbers

Machine M: cn

Machine M': c’n
The fact that we get a line is invariant under changes of implementations. Different machines result in different slopes, but the time taken grows linearly as input size increases.
Thus we arrive at an implementation-independent insight:

Grade School Addition is a linear time algorithm
Thus we arrive at an implementation-independent insight:

**Grade School Addition is a linear time algorithm**

This process of abstracting away details and determining the rate of resource usage in terms of the problem size $n$ is one of the fundamental ideas in computer science.
Time vs Input Size

For any algorithm, define

**Input Size** = # of bits to specify its inputs.
For any algorithm, define

\textbf{Input Size} = \# of bits to specify its inputs.

Define

\textbf{TIME}_n = \text{the worst-case amount of time used by the algorithm on inputs of size } n
For any algorithm, define

**Input Size** = # of bits to specify its inputs.

Define

**TIME**\(_n\) = the worst-case amount of time used by the algorithm on inputs of size \(n\)

We often ask: **What is the growth rate of **Time\(_n\)** **?
How to multiply 2 n-bit numbers.
How to multiply 2 n-bit numbers.

\[
\begin{array}{c}
\times \\
\hline
\end{array}
\]

\[
\begin{array}{ccccccccccc}
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
\end{array}
\]

\[
\begin{array}{c}
\{ \\
n^2 \\
\}
\end{array}
\]
How to multiply 2 n-bit numbers.

The total time is bounded by $cn^2$ (abstracting away the implementation details).
Grade School Addition: Linear time
Grade School Multiplication: Quadratic time
Grade School Addition: Linear time
Grade School Multiplication: Quadratic time

No matter how dramatic the difference in the constants, the **quadratic curve** will eventually dominate the **linear curve**
How much time does it take to square the number $n$ using grade school multiplication?
Grade School Multiplication: Quadratic time

c($\log n)^2$ time to square the number n
Grade School Multiplication: Quadratic time

Input size is measured in bits, unless we say otherwise.

c(\log n)^2 \text{ time to square the number } n

Input size is measured in bits, unless we say otherwise.
How much time does it take?

Nursery School Addition
Input: Two $n$-bit numbers, $a$ and $b$
Output: $a + b$
How much time does it take?

Nursery School Addition
Input: Two \( n \)-bit numbers, \( a \) and \( b \)
Output: \( a + b \)

Start at \( a \) and increment (by 1) \( b \) times
How much time does it take?

Nursery School Addition
Input: Two \( n \)-bit numbers, \( a \) and \( b \)
Output: \( a + b \)

Start at \( a \) and increment (by 1) \( b \) times

\( T(n) = ? \)
How much time does it take?

Nursery School Addition
Input: Two \( n \)-bit numbers, \( a \) and \( b \)
Output: \( a + b \)

Start at \( a \) and increment (by 1) \( b \) times

\( T(n) = ? \)

If \( b = 000\ldots0000 \), then NSA takes almost no time
How much time does it take?

Nursery School Addition

Input: Two $n$-bit numbers, $a$ and $b$
Output: $a + b$

Start at $a$ and increment (by 1) $b$ times

$T(n) = ?$

If $b = 000\ldots0000$, then NSA takes almost no time

If $b = 1111\ldots11111$, then NSA takes $cn2^n$ time
Worst Case Time

**Worst Case Time** $T(n)$ for algorithm A:

$T(n) = \max_{\text{all permissible inputs } X \text{ of size } n} \text{(Running time of algorithm A on input } X).$
What is $T(n)$?

Kindergarten Multiplication

Input: Two $n$-bit numbers, $a$ and $b$

Output: $a \times b$
What is $T(n)$?

Kindergarten Multiplication
Input: Two $n$-bit numbers, $a$ and $b$
Output: $a \times b$

Start with $a$ and add $a$, $b-1$ times
What is $T(n)$?

Kindergarten Multiplication
Input: Two $n$-bit numbers, $a$ and $b$
Output: $a \times b$

Start with $a$ and add $a$, $b-1$ times

Remember, we always pick the WORST CASE input for the input size $n$.

Thus, $T(n) = cn2^n$
Thus, Nursery School adding and Kindergarten multiplication are exponential time.

They scale HORRIBLY as input size grows.

Grade school methods scale polynomially: just linear and quadratic. Thus, we can add and multiply fairly large numbers.
If $T(n)$ is not polynomial, the algorithm is not efficient: the run time scales too poorly with the input size.
If $T(n)$ is not polynomial, the algorithm is not efficient: the run time scales too poorly with the input size.

This will be the yardstick with which we will measure “efficiency”.
Multiplication is efficient, what about “reverse multiplication”? 
Multiplication is efficient, what about “reverse multiplication”?

Let’s define FACTORING(N) to be any method to produce a non-trivial factor of N, or to assert that N is prime.
Factoring The Number N
By Trial Division
Factoring The Number N By Trial Division

Trial division up to $\sqrt{N}$
Factoring The Number N By Trial Division

Trial division up to $\sqrt{N}$

for $k = 2$ to $\sqrt{N}$ do
  if $k \mid N$ then
    return "N has a non-trivial factor k"
  return "N is prime"
Factoring The Number N By Trial Division

Trial division up to $\sqrt{N}$

for $k = 2$ to $\sqrt{N}$ do
    if $k | N$ then
        return “N has a non-trivial factor k”
    return “N is prime”

$c \sqrt{N} (\log N)^2$ time if division is $c (\log N)^2$ time
Factoring The Number N
By Trial Division

Trial division up to $\sqrt{N}$

for $k = 2$ to $\sqrt{N}$ do
    if $k \mid N$ then
        return “N has a non-trivial factor k”
    return “N is prime”

$c \sqrt{N} (\log N)^2$ time if division is $c (\log N)^2$ time

Is this efficient?
Factoring The Number N By Trial Division

Trial division up to $\sqrt{N}$

for $k = 2$ to $\sqrt{N}$ do
    if $k | N$ then
        return “N has a non-trivial factor k”
    return “N is prime”

c $\sqrt{N} (\log N)^2$ time if division is $c (\log N)^2$ time

Is this efficient?

No! The input length $n = \log N$. Hence we’re using $c 2^{n/2} n^2$ time.
Can we do better?
Can we do better?

We know of methods for FACTORING that are sub-exponential (about $2^{n^{1/3}}$ time) but nothing efficient.
Notation to Discuss Growth Rates

For any monotonic function \( f \) from the positive integers to the positive integers, we say

“\( f = O(n) \)” or “\( f \) is \( O(n) \)”
For any monotonic function $f$ from the positive integers to the positive integers, we say

\[ f = O(n) \] or \[ f \text{ is } O(n) \]

If some constant times $n$ eventually dominates $f$
Notation to Discuss Growth Rates

For any monotonic function $f$ from the positive integers to the positive integers, we say

"$f = O(n)$" or "$f$ is $O(n)$"

If some constant times $n$ eventually dominates $f$

[Formally: there exists a constant $c$ such that for all sufficiently large $n$: $f(n) \leq cn$]
$f = O(n)$ means that there is a line that can be drawn that stays above $f$ from some point on.
Other Useful Notation: $\Omega$

For any monotonic function $f$ from the positive integers to the positive integers, we say

“$f = \Omega(n)$” or “$f$ is $\Omega(n)$”
For any monotonic function \( f \) from the positive integers to the positive integers, we say

“\( f = \Omega(n) \)” or “\( f \) is \( \Omega(n) \)”

If \( f \) eventually dominates some constant times \( n \)
For any monotonic function $f$ from the positive integers to the positive integers, we say

“$f = \Omega(n)$” or “$f$ is $\Omega(n)$”

If $f$ eventually dominates some constant times $n$

[Formally: there exists a constant $c$ such that for all sufficiently large $n$: $f(n) \geq cn$]
\[ f = \Omega(n) \] means that there is a line that can be drawn that stays below \( f \) from some point on.
Yet More Useful Notation: $\Theta$

For any monotonic function $f$ from the positive integers to the positive integers, we say

“$f = \Theta(n)$” or “$f$ is $\Theta(n)$”
Yet More Useful Notation: $\Theta$

For any monotonic function $f$ from the positive integers to the positive integers, we say

“$f = \Theta(n)$” or “$f$ is $\Theta(n)$”

if: $f = O(n)$ and $f = \Omega(n)$
\( f = \Theta(n) \) means that \( f \) can be sandwiched between two lines from some point on.
Notation to Discuss Growth Rates

For any two monotonic functions \( f \) and \( g \) from the positive integers to the positive integers, we say

“\( f = O(g) \)” or “\( f \) is \( O(g) \)”
Notation to Discuss Growth Rates

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = O(g)$” or “$f$ is $O(g)$”

If some constant times $g$ eventually dominates $f$
Notation to Discuss Growth Rates

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = \mathcal{O}(g)$” or “$f$ is $\mathcal{O}(g)$”

If some constant times $g$ eventually dominates $f$

[Formally: there exists a constant $c$ such that for all sufficiently large $n$: $f(n) \leq c \, g(n)$]
f = O(g) means that there is some constant c such that \( c \, g(n) \) stays above \( f(n) \) from some point on.
Other Useful Notation: $\Omega$

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = \Omega(g)$” or “$f$ is $\Omega(g)$”
For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = \Omega(g)$” or “$f$ is $\Omega(g)$”

If $f$ eventually dominates some constant times $g$
Other Useful Notation: $\Omega$

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = \Omega(g)$” or “$f$ is $\Omega(g)$”

If $f$ eventually dominates some constant times $g$

[Formally: there exists a constant $c$ such that for all sufficiently large $n$: $f(n) \geq c \cdot g(n)$]
Yet More Useful Notation: $\Theta$

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

“$f = \Theta(g)$” or “$f$ is $\Theta(g)$”
Yet More Useful Notation: $\Theta$

For any two monotonic functions $f$ and $g$ from the positive integers to the positive integers, we say

"$f = \Theta(g)$" or "$f$ is $\Theta(g)$"

If: $f = O(g)$ and $f = \Omega(g)$
• $n = O(n^2)$ ?
• $n = O(n^2)$? Yes!
• $n = O(n^2)$? Yes!

Take $c = 1$

For all $n \geq 1$, it holds that $n \leq cn^2$
• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$?
• $n = O(n^2)$ ? Yes!

• $n = O(\sqrt{n})$ ? No
• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$? No

Suppose it were true that $n \leq c \sqrt{n}$ for some constant $c$ and large enough $n$. 


• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$? No

Suppose it were true that $n \leq c \sqrt{n}$ for some constant $c$ and large enough $n$. Cancelling, we would get $\sqrt{n} \leq c$. Which is false for $n > c^2$. 
• $n = O(n^2)$ ? Yes!
• $n = O(\sqrt{n})$ ? No
• $3n^2 + 4n + 4 = O(n^2)$ ?
• $3n^2 + 4n + 4 = \Omega(n^2)$ ?
• $n^2 = \Omega(n \log n)$ ?
• $n^2 \log n = \Theta(n^2)$ ?
• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$? No

• $3n^2 + 4n + 4 = O(n^2)$? Yes!

• $3n^2 + 4n + 4 = \Omega(n^2)$?

• $n^2 = \Omega(n \log n)$?

• $n^2 \log n = \Theta(n^2)$?
• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$? No

• $3n^2 + 4n + 4 = O(n^2)$? Yes!

• $3n^2 + 4n + 4 = \Omega(n^2)$? Yes!

• $n^2 = \Omega(n \log n)$?

• $n^2 \log n = \Theta(n^2)$?
• $n = O(n^2)$? Yes!

• $n = O(\sqrt{n})$? No

• $3n^2 + 4n + 4 = O(n^2)$? Yes!

• $3n^2 + 4n + 4 = \Omega(n^2)$? Yes!

• $n^2 = \Omega(n \log n)$? Yes!

• $n^2 \log n = \Theta(n^2)$?
• $n = O(n^2)$ ? Yes!
• $n = O(\sqrt{n})$ ? No
• $3n^2 + 4n + 4 = O(n^2)$ ? Yes!
• $3n^2 + 4n + 4 = \Omega(n^2)$ ? Yes!
• $n^2 = \Omega(n \log n)$ ? Yes!
• $n^2 \log n = \Theta(n^2)$ ? No
• $f = O(g)$ and $g = O(h)$
  then $f = O(h)$ ?

• $f = O(g)$
  then $g = \Omega(f)$
• \( f = O(g) \) and \( g = O(h) \)
then \( f = O(h) \)?  

  Yes!

• \( f = O(g) \)
then \( g = \Omega(f) \)
• $f = O(g)$ and $g = O(h)$
  then $f = O(h)$ ?
  Yes!

  \[
  f(n) \leq c \, g(n) \text{ for all } n \geq n_0. \\
  \text{and } g(n) \leq c' \, h(n) \text{ for all } n \geq n_0'. \\
  \]

  So $f(n) \leq (cc') \, h(n) \text{ for all } n \geq \max(n_0, n_0')$

  

• $f = O(g)$
  then $g = \Omega(f)$
• $f = O(g)$ and $g = O(h)$
  then $f = O(h)$? Yes!

\[
f(n) \leq c \ g(n) \text{ for all } n \geq n_0.
\]

\[
\text{and } g(n) \leq c' \ h(n) \text{ for all } n \geq n_0'.
\]

So $f(n) \leq (cc') \ h(n)$ for all $n \geq \max(n_0, n_0')$

• $f = O(g)$
  then $g = \Omega(f)$ Yes!
Names For Some Growth Rates

**Linear Time:** \( T(n) = O(n) \)

**Quadratic Time:** \( T(n) = O(n^2) \)

**Cubic Time:** \( T(n) = O(n^3) \)
Names For Some Growth Rates

Linear Time: \( T(n) = O(n) \)
Quadratic Time: \( T(n) = O(n^2) \)
Cubic Time: \( T(n) = O(n^3) \)

Polynomial Time:
Names For Some Growth Rates

**Linear Time:** $T(n) = O(n)$

**Quadratic Time:** $T(n) = O(n^2)$

**Cubic Time:** $T(n) = O(n^3)$

**Polynomial Time:**

For some constant $k$, $T(n) = O(n^k)$.

Example: $T(n) = 13n^5$
Large Growth Rates

Exponential Time:
Large Growth Rates

**Exponential Time:**
for some constant $k$, $T(n) = O(k^n)$

Example: $T(n) = n2^n = O(3^n)$
Small Growth Rates

**Logarithmic Time:** $T(n) = O(\log n)$

Example: $T(n) = 15 \log_2(n)$
Small Growth Rates

**Logarithmic Time:** $T(n) = O(\log n)$

Example: $T(n) = 15\log_2(n)$

**Polylogarithmic Time:**
for some constant $k$, $T(n) = O(\log^k(n))$
Small Growth Rates

**Logarithmic Time:** $T(n) = O(\log n)$
Example: $T(n) = 15\log_2(n)$

**Polylogarithmic Time:**
for some constant $k$, $T(n) = O(\log^k(n))$

**Note:** These kind of algorithms can’t possibly read all of their inputs.
Small Growth Rates

**Logarithmic Time:** $T(n) = O(\log n)$
Example: $T(n) = 15\log_2(n)$

**Polylogarithmic Time:**
for some constant $k$, $T(n) = O(\log^k(n))$

**Note:** These kind of algorithms can’t possibly read all of their inputs.

A very common example of logarithmic time is looking up a word in a sorted dictionary (binary search)
Some Big Ones

Doubly Exponential Time means that for some constant $k$
Some Big Ones

**Doubly Exponential Time** means that for some constant $k$

$$T(n) = 2^{2^k n}$$
Some Big Ones

Doubly Exponential Time means that for some constant $k$

$$T(n) = 2^{2^k n}$$

Triply Exponential
Some Big Ones

Doubly Exponential Time means that for some constant $k$

$$T(n) = 2^{2^k n}$$

Triply Exponential

$$T(n) = 2^{2^{2^k n}}$$
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}
2STACK(1) = 2
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}

2STACK(1) = 2
2STACK(2) = 4
Faster and Faster: 2STACK

\[ 2\text{STACK}(0) = 1 \]
\[ 2\text{STACK}(n) = 2^{2\text{STACK}(n-1)} \]

\[ 2\text{STACK}(1) = 2 \]
\[ 2\text{STACK}(2) = 4 \]
\[ 2\text{STACK}(3) = 16 \]
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536
Faster and Faster: 2STACK

\[ 2\text{STACK}(0) = 1 \]
\[ 2\text{STACK}(n) = 2^{2\text{STACK}(n-1)} \]

\[ 2\text{STACK}(1) = 2 \]
\[ 2\text{STACK}(2) = 4 \]
\[ 2\text{STACK}(3) = 16 \]
\[ 2\text{STACK}(4) = 65536 \]
\[ 2\text{STACK}(5) \geq 10^{80} \]
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536
2STACK(5) \geq 10^{80}
= \text{atoms in universe}
Faster and Faster: 2STACK

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536
2STACK(5) \geq 10^{80}
   = atoms in universe

2STACK(n) = \underbrace{2 \cdots 2}_{n \text{ times}}
"tower of n 2's"
And the inverse of $2^{STACK}$: $\log^*$

$2^{STACK}(0) = 1$

$2^{STACK}(n) = 2^{2^{STACK}(n-1)}$

$2^{STACK}(1) = 2$

$2^{STACK}(2) = 4$

$2^{STACK}(3) = 16$

$2^{STACK}(4) = 65536$

$2^{STACK}(5) \geq 10^{80}$

$= \text{atoms in universe}$
And the inverse of 2STACK: \( \log^* \)

\[
\begin{align*}
2\text{STACK}(0) &= 1 \\
2\text{STACK}(n) &= 2^{2\text{STACK}(n-1)} \\
2\text{STACK}(1) &= 2 \\
2\text{STACK}(2) &= 4 \\
2\text{STACK}(3) &= 16 \\
2\text{STACK}(4) &= 65536 \\
2\text{STACK}(5) &\geq 10^{80} \\
&= \text{atoms in universe}
\end{align*}
\]

\( \log^*(n) = \# \text{ of times you have to apply the log function to } n \text{ to make it } \leq 1 \)
And the inverse of 2STACK: \( \log^* \)

\[
\begin{align*}
2\text{STACK}(0) &= 1 \\
2\text{STACK}(n) &= 2^{2\text{STACK}(n-1)} \\
2\text{STACK}(1) &= 2 \\
2\text{STACK}(2) &= 4 \\
2\text{STACK}(3) &= 16 \\
2\text{STACK}(4) &= 65536 \\
2\text{STACK}(5) &\geq 10^{80} \\
&= \text{atoms in universe}
\end{align*}
\]

\( \log^*(2) = 1 \)

\( \log^*(n) = \# \text{ of times you have to apply the log function to } n \text{ to make it } \leq 1 \)
And the inverse of 2STACK: log*

\[2^{\text{STACK}(n)} = 2^{2^{\text{STACK}(n-1)}}\]

\[
\begin{align*}
\text{STACK}(0) &= 1 \\
\text{STACK}(1) &= 2 \\
\text{STACK}(2) &= 4 \\
\text{STACK}(3) &= 16 \\
\text{STACK}(4) &= 65536 \\
\text{STACK}(5) &\geq 10^{80}
\end{align*}
\]

\[\log^*(n) = \# \text{ of times you have to apply the log function to } n \text{ to make it } \leq 1\]

\[\log^*(2) = 1, \quad \log^*(4) = 2\]

= atoms in universe
And the inverse of 2STACK: \( \log^* \)

\[
\begin{align*}
2\text{STACK}(0) &= 1 \\
2\text{STACK}(n) &= 2^{2\text{STACK}(n-1)} \\
2\text{STACK}(1) &= 2 \\
2\text{STACK}(2) &= 4 \\
2\text{STACK}(3) &= 16 \\
2\text{STACK}(4) &= 65536 \\
2\text{STACK}(5) &\geq 10^{80} \\
\end{align*}
\]

\( \log^*(2) = 1 \)

\( \log^*(4) = 2 \)

\( \log^*(16) = 3 \)

\( \log^*(n) = \# \text{ of times you have to apply the log function to } n \text{ to make it } \leq 1 \)

\( = \text{ atoms in universe} \)
And the inverse of $2\text{STACK}$: $\log^*$

$2\text{STACK}(0) = 1$

$2\text{STACK}(n) = 2^{2\text{STACK}(n-1)}$

$2\text{STACK}(1) = 2$

$2\text{STACK}(2) = 4$

$2\text{STACK}(3) = 16$

$2\text{STACK}(4) = 65536$

$2\text{STACK}(5) \geq 10^{80}$

$\geq$ atoms in universe

$\log^*(n) = \#$ of times you have to apply the log function to $n$ to make it $\leq 1$
And the inverse of 2STACK: log*

2STACK(0) = 1
2STACK(n) = 2^{2STACK(n-1)}

2STACK(1) = 2
2STACK(2) = 4
2STACK(3) = 16
2STACK(4) = 65536
2STACK(5) ≥ 10^{80}

log*(2) = 1
log*(4) = 2
log*(16) = 3
log*(65536) = 4
log*(atoms) = 5

= atoms in universe

log*(n) = # of times you have to apply the log function to n to make it ≤ 1
So an algorithm that can be shown to run in $O(n \log^* n)$ Time is **Linear Time** for all practical purposes!!
Ackermann\'s Function
Ackermann’s Function

$$A(0, n) = n + 1 \text{ for } n \geq 0$$
Ackermann’s Function

$A(0, n) = n + 1$ for $n \geq 0$

$A(m, 0) = A(m - 1, 1)$ for $m \geq 1$
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]

<table>
<thead>
<tr>
<th></th>
<th>n=0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>m=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ackermann’s Function
Ackermann’s Function

A(0, n) = n + 1 for n \geq 0
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]

\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]

\[ A(4, 2) > \# \text{ of particles in universe} \]
Ackermann’s Function

\[ A(0, n) = n + 1 \text{ for } n \geq 0 \]
\[ A(m, 0) = A(m - 1, 1) \text{ for } m \geq 1 \]
\[ A(m, n) = A(m - 1, A(m, n - 1)) \text{ for } m, n \geq 1 \]

\[ A(4,2) > \# \text{ of particles in universe} \]
\[ A(5,2) \text{ can’t be written out as decimal in this universe} \]
Ackermann’s Function
Ackermann’s Function

Define: $A'(k) = A(k,k)$
Ackermann’s Function

Define: $A'(k) = A(k,k)$

Inverse Ackerman $\alpha(n)$ is the inverse of $A'$
Ackermann’s Function

Define: $A'(k) = A(k,k)$

Inverse Ackerman $\alpha(n)$ is the inverse of $A'$

Practically speaking: $n \times \alpha(n) \leq 4n$
Ackermann’s Function

$A(0, n) = n + 1$ for $n \geq 0$

Define: $A'(k) = A(k, k)$

Inverse Ackerman $\alpha(n)$ is the inverse of $A'$

Practically speaking: $n \times \alpha(n) \leq 4n$
Ackermann’s Function

A(0, n) = n + 1 for n ≥ 0
A(m, 0) = A(m - 1, 1) for m ≥ 1

Define: A’(k) = A(k, k)

Inverse Ackerman α(n) is the inverse of A’

Practically speaking: n × α(n) ≤ 4n
Ackermann’s Function

A(0, n) = n + 1 for n ≥ 0
A(m, 0) = A(m - 1, 1) for m ≥ 1
A(m, n) = A(m - 1, A(m, n - 1)) for m, n ≥ 1

Define: A’(k) = A(k,k)

Inverse Ackerman α(n) is the inverse of A’

Practically speaking: n × α(n) ≤ 4n
The inverse Ackermann function – in fact, $\Theta(n \alpha(n))$ arises in the seminal paper of:

Here’s What You Need to Know…

• How is “time” measured
• Definitions of:
  • $O$, $\Omega$, $\Theta$
  • linear, quadratic time, etc
  • $\log^*(n)$
  • Ackerman Function