15-251
Great Theoretical Ideas in Computer Science

This is The Big Oh!
Lecture 21 (March 31, 2009)

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”

How to add 2 n-bit numbers

```
+ * * * * * * * * *
```

Time complexity of grade school addition

```
+ * * * * * * * * *
```

T(n) = amount of time grade school addition uses to add two n-bit numbers

What do we mean by “time”?

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

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.

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

to be the time it takes to perform on that computer.

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

\[
\sum_{\text{i.e., c time for each of n columns}}^n
\]
On another computer $M'$, the time to perform $\sum$ may be $c'$. The total time to add two $n$-bit numbers using grade school addition:

$c' n$  

[c’ time for each of $n$ columns]

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.

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.

Thus we arrive at an implementation-independent insight:

For any algorithm, define

Input Size = # of bits to specify its inputs.

Define

$TIME_n = \text{the worst-case amount of time used by the algorithm on inputs of size } n$

We often ask: What is the growth rate of $Time_n$?

How to multiply 2 $n$-bit numbers.

$X$

$\sum$

$n^2$

The total time is bounded by $cn^2$ (abstracting away the implementation details).
No matter how dramatic the difference in the constants, the quadratic curve will eventually dominate the linear curve.

Grade School Multiplication: Quadratic time

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

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

Worst Case Time

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

$T(n) = \max_{\text{all permissible inputs } X \text{ of size } n} \text{Runtime}(A,X)$

Runtime$(A,X) =$

Running time of algorithm A on input X.

What is $T(n)$?

Kindergarten Multiplication

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

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 addition 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.

This will be the yardstick with which we will measure “efficiency”.

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

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 $\sqrt{N}$ time if division is $c \log N$ time

Is this efficient?

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

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 $O(n)$” if some constant times $n$ eventually dominates $f$.

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

![Graph showing \( f = \mathcal{O}(n) \)]

**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 \( \Omega(n) \)”

If \( f \) eventually dominates some constant times \( n \)

Formally: there exists a constant \( c \) such that for all sufficiently large \( n \): \( f(n) \geq cn \)

---

\( f = \Theta(n) \) means that there is a line that can be drawn that stays below \( f \) from some point on.

![Graph showing \( f = \Theta(n) \)]

**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: \( f = \mathcal{O}(n) \) and \( f = \Omega(n) \)

---

\( f = \Theta(n) \) means that \( f \) can be sandwiched between two lines from some point on.

![Graph showing \( f = \Theta(n) \)]

**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 \( n \): \( f(n) \leq cg(n) \)
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)”

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 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)”

If: f = O(g) and f = \( \Omega \)(g)

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

\( \cdot \) n = O(n^2) ? Yes!
\( \cdot \) 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

\( \cdot \) 3n^2 + 4n + 4 = O(n^2) ? Yes! 3n^2 + 4n + 4 \( \leq \) 4n^2 for n \( \geq \) 5
\( \cdot \) 3n^2 + 4n + 4 = \( \Omega \)(n^2) ? Yes! 3n^2 + 4n + 4 \( \geq \) 3 n^2 for n \( \geq \) 0
\( \cdot \) n^2/10 = O(n \log n) ? Yes! n^2/10 \( \geq \) (n \log n)/10 for n \( \geq \) 1
\( \cdot \) n^2 \log n = \( \Theta \)(n^2) ? No
\[ n^2 \log n = \Theta(n^2) \? \quad \text{No} \]

Yes, \( n^2 \log n = \Omega(n^2) \)

But, \( n^2 \log n \neq O(n^2) \)

If it were, then \( p^x \log n \leq c \rho^x \) for some \( c \) and large enough \( n \)

But this is false for \( n > 2^c \)

\( (\text{assuming } \log n = \log_2 n) \)

\[ f(n) \leq cn \quad \text{for all } n \geq n_0 \]

\[ g(n) \leq c' h(n) \quad \text{for all } n \geq n'_0 \]

So \( f(n) \leq (cc') h(n) \quad \text{for all } n \geq \max(n_0, n'_0) \)

\[ f = O(g) \quad \text{then } g = \Omega(f) \quad \text{Yes!} \]

\[ f = O(g) \quad \text{and } g = O(h) \]

\[ \text{then } f = O(h) \quad \text{Yes!} \]

---

**Names For Some Growth Rates**

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

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

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

Polynomial Time:
for some constant \( k \), \( T(n) = O(n^k) \).
Example: \( T(n) = 13n^5 \)

---

**Large Growth Rates**

Exponential Time:
for some constant \( k \), \( T(n) = O(k^n) \)
Example: \( T(n) = n2^n = O(3^n) \)

\[ \begin{align*}
Q: \quad n 2^n &= O(2^n) \quad ? n_0 \\
&= O(1) \quad ? n_0' \\
&= \Omega(n) \quad ? n_0 \end{align*} \]

---

**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 \( k \)
\[ T(n) = 2^{2^k n} \]

Triply Exponential
\[ T(n) = 2^{2^{2^k n}} \]
**Faster and Faster: 2STACK**

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

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

---

**And the inverse of 2STACK: log***

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

2STACK(1) = 2 \quad \log^*(2) = 1
2STACK(2) = 4 \quad \log^*(4) = 2
2STACK(3) = 16 \quad \log^*(16) = 3
2STACK(4) = 65536 \quad \log^*(65536) = 4
2STACK(5) \geq 10^{80} \quad \log^*(\text{atoms}) = 5

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

---

**Ackermann’s Function**

<table>
<thead>
<tr>
<th>m \ n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

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**

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

\(A(4,2) > \# \text{ of particles in universe}\)

\(A(5,2)\) can’t be written out as decimal in this universe

---

**Ackermann’s Function**

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

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

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

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


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

Here’s What You Need to Know...