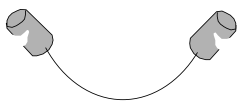
Great Theoretical Ideas In Computer Science

S.Rudich CS 15-251 Spring 2006

V.Adamchik

Lecture 20 Mar 30, 2006 Carnegie Mellon University

1. A Parallel Perspective
2. Compression



How to add 2 n-bit numbers.

```
                                          *
    *  *  *  *  *  *  *  *  *  *  *
+   *  *  *  *  *  *  *  *  *  *  *
---------------------------------
                                  *
```

How to add 2 n-bit numbers.

```
                         *  *  *
    *  *  *  *  *  *  *  *  *  *  *
+   *  *  *  *  *  *  *  *  *  *  *
---------------------------------
                            *  *  *
```

How to add 2 n-bit numbers.

```
 *  *  *  *  *  *  *  *  *  *  *
    *  *  *  *  *  *  *  *  *  *  *
+   *  *  *  *  *  *  *  *  *  *  *
---------------------------------
 *  *  *  *  *  *  *  *  *  *  *  *
```

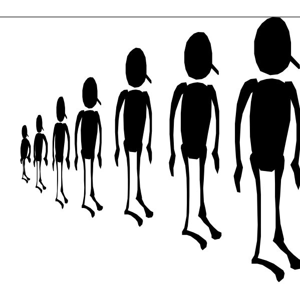
How to add 2 n-bit numbers.

```
 *  *  *  *  *  *  *  *  *  *  *
    *  *  *  *  *  *  *  *  *  *  *
+   *  *  *  *  *  *  *  *  *  *  *
---------------------------------
 *  *  *  *  *  *  *  *  *  *  *  *
```

Let c be the maximum time that it takes you to do

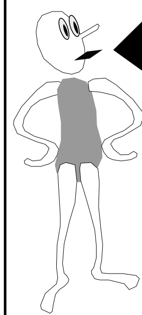
$T(n) \leq c*n$ is proportional to n

If n people agree to add two n bit numbers, can they finish faster than you alone?.



Is it possible to add two n bit numbers in less than linear parallel-time?

Darn those carries !



To find fast parallel ways of adding, let's re-examine grade school addition from the view of a computer circuit.

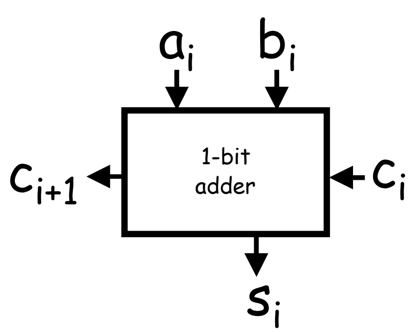
Grade School Addition

$$\begin{array}{rccccc} & c_5 & c_4 & c_3 & c_2 & c_1 \\ & & a_4 & a_3 & a_2 & a_1 & a_0 \\ + & & b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline \end{array}$$

Grade School Addition

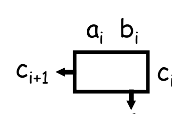
$$\begin{array}{rcccc|c|c} & c_5 & c_4 & c_3 & c_2 & c_1 & \\ & & a_4 & a_3 & a_2 & a_1 & a_0 \\ + & & b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline & & & & & s_1 & \end{array}$$

Adder



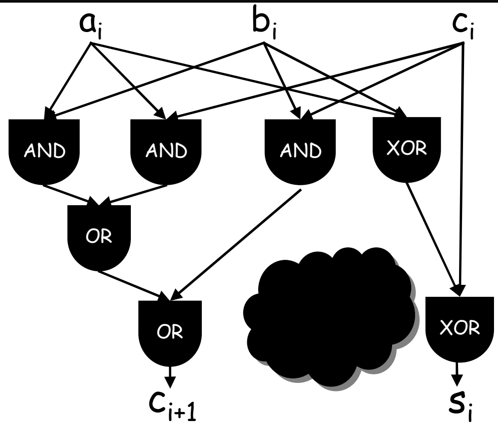
Logical representation of binary:
0 = false, 1 = true

$$\begin{array}{rc|c|c} & c_5c_4c_3c_2 & c_1 & \\ & a_4a_3a_2 & a_1 & a_0 \\ + & b_4b_3b_2 & b_1 & b_0 \\ \hline & & s_1 & \end{array}$$

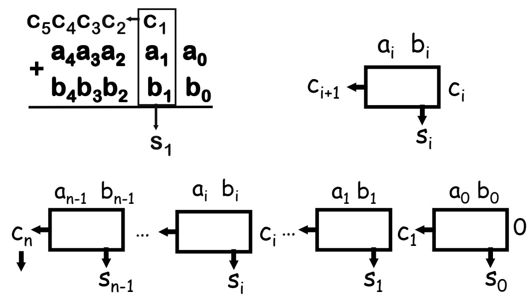


$s_1 = (a_1 \text{ XOR } b_1) \text{ XOR } c_1$

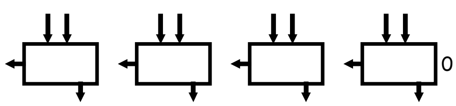
$c_2 = (a_1 \text{ AND } b_1)$
$\text{OR } (a_1 \text{ AND } c_1)$
$\text{OR } (b_1 \text{ AND } c_1)$



Ripple-carry adder

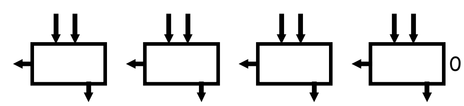


Ripple-carry adder



How long to add two n bit numbers?

Ripple-carry adder



How long to add two n bit numbers?

Propagation time through the ripple-carry adder will be $\Theta(n)$

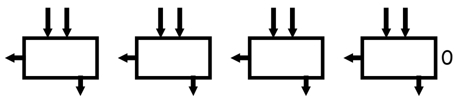
Circuits compute things in parallel.

We can think of the propagation delay as PARALLEL TIME.

Is it possible to add two n bit numbers in less than linear parallel-time?

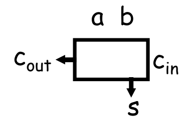
Darn those carries (again)!

If we knew the carries it would be very easy to do fast parallel addition



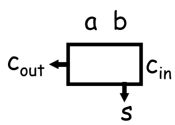
So how do we figure out the carries fast?

What do we know about the carry-out before we know the carry-in?

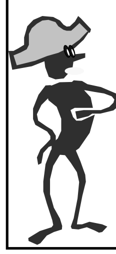


a	b	C_{out}
0	0	0
0	1	C_{in}
1	0	C_{in}
1	1	1

What do we know about the carry-out before we know the carry-in?



a	b	C_{out}
0	0	0
0	1	←
1	0	←
1	1	1



This is just a function of a and b.
We can do this in parallel.

a	b	C_{out}
0	0	0
0	1	←
1	0	←
1	1	1

Idea: do this calculation first.

10← ← ←←←1←←0
1011111[redacted]
\+ 1000000[redacted]

a	b	C_{out}
0	0	0
0	1	←
1	0	←
1	1	1

Note that this just took one step!

Now if we could only replace the ← by 0/1 values…

Idea: do this calculation first.

10← ← ←←←1←←0
1011111101
\+ 1000000110

Idea: do this calculation first.

```
  10← ← ←←←1000
   1011111101
+  1000000110
  ______________
```

Idea: do this calculation first.

```
  10← ←←111000
   1011111101
+  1000000110
  ______________
```

Idea: do this calculation first.

```
  10111111000
   1011111101
+  1000000110
  ______________
```

Idea: do this calculation first.

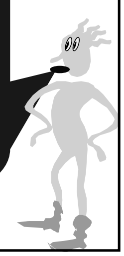
```
  10111111000
   1011111101
+  1000000110
  ______________
  10100000011
```

Once we have the carries, it takes only one more step: $s_i = (a_i \text{ XOR } b_i) \text{ XOR } c_i$

10← ← ←←←1←←0

So, everything boils down to: can we find a fast parallel way to convert each position to its final 0/1 value?

Called the "parallel prefix problem"



Prefix Sum Problem

Input: $X_{n-1}, X_{n-2},\dots,X_1, X_0$

Output: $Y_{n-1}, Y_{n-2},\dots,Y_1, Y_0$

where

$Y_0 = X_0$

$Y_1 = X_0 + X_1$

$Y_2 = X_0 + X_1 + X_2$

$Y_3 = X_0 + X_1 + X_2 + X_3$

$\vdots$

$Y_{n-1} = X_0 + X_1 + X_2 + X_3 + \dots + X_{n-1}$

+ is Addition

Idea to get 10111111000

Can think of 10← ← ← ← ←1← ← 0 as

(1 M (0 M (← M (← M (← M (← M (← M (1 M (← M 0)))))))))

for the operator **M** :

← **M** x = x

1 **M** x = 1

0 **M** x = 0

M	0	1	←
0	0	0	0
1	1	1	1
←	0	1	←

Associative Binary Operator

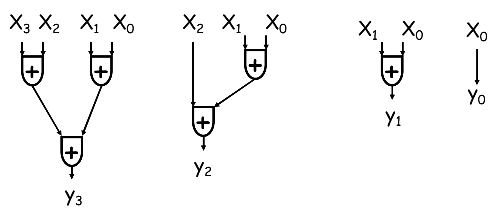
Binary Operator: an operation that takes two objects and returns a third.

- A♠B = C

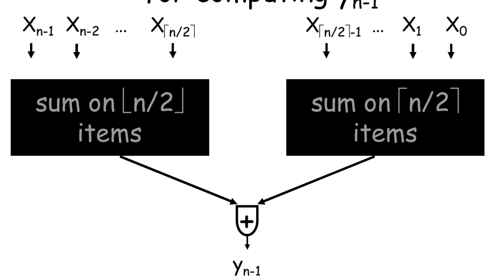
Associative:

- (A ♠ B) ♠ C = A ♠ (B ♠ C)

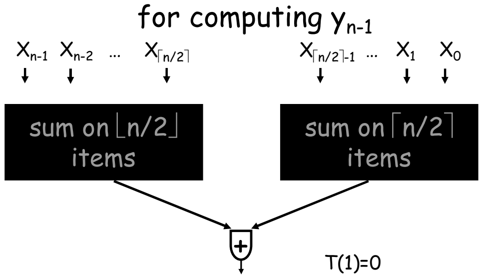
Example circuitry (n = 4)



Divide, conquer, and glue for computing y_{n-1}



Divide, conquer, and glue for computing y_{n-1}



y_{n-1}

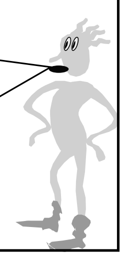
$T(1)=0$

$T(n) = T(\lceil n/2 \rceil) + 1$

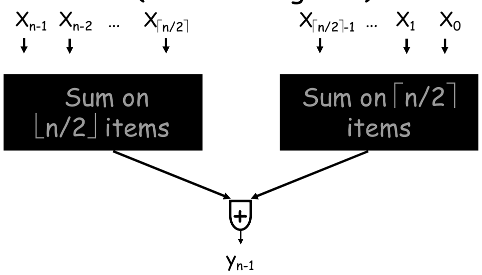
$T(n) = \lceil \log_2 n \rceil$

The parallel time taken is $T(n) = \lceil \log_2 n \rceil$!

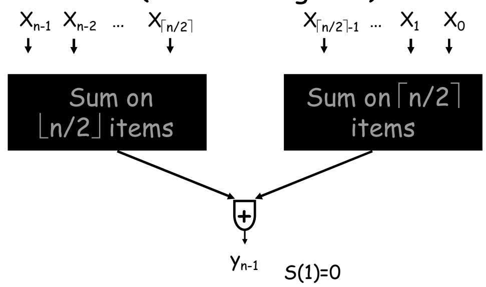
But how many components does this use? What is the size of the circuit?



Size of Circuit (number of gates)



Size of Circuit (number of gates)

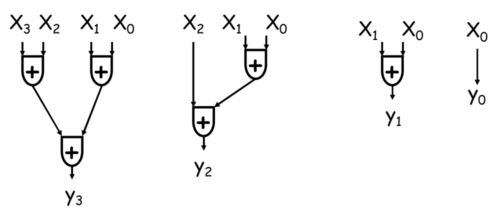


$S(1)=0$

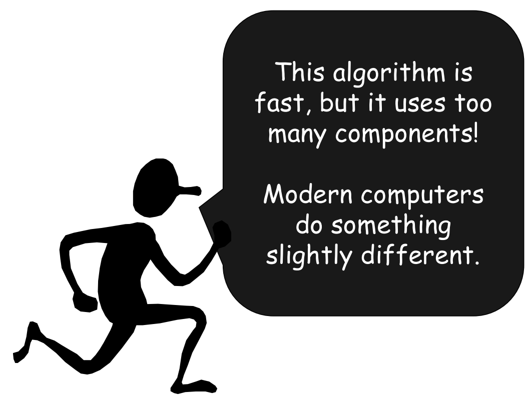
$S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 1$

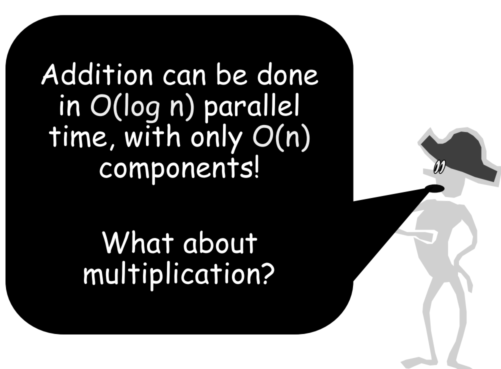
$S(n) = n-1$

Sum of Sizes

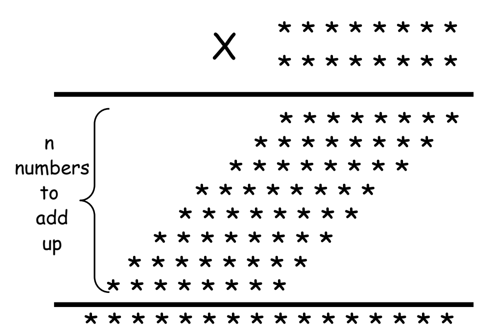


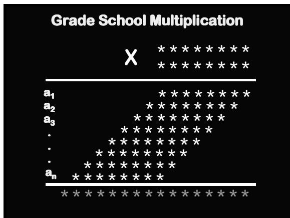
$S(n) = 0 + 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$





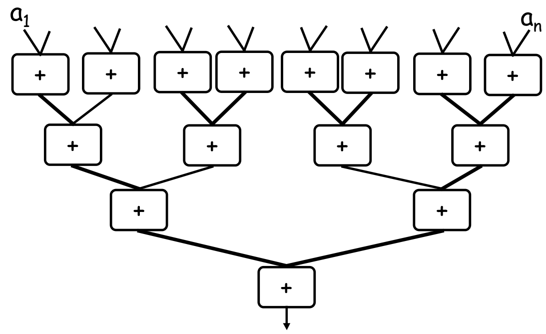
How about multiplication?





We need to add n 2n-bit numbers:
$a_1, a_2, a_3, \ldots, a_n$

Adding these numbers in parallel

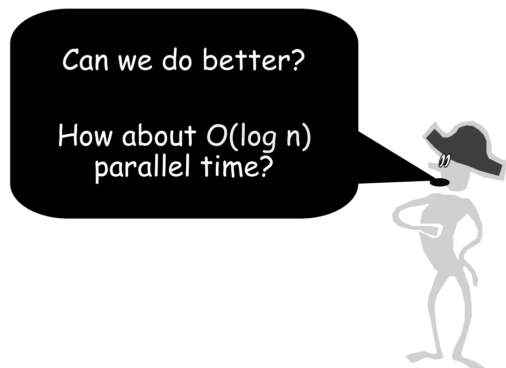


What is the depth of the circuit?

Each addition takes O(log n) parallel time

Depth of tree = $\log_2 n$

Total $O(\log n)^2$ parallel time



How about multiplication?

Here's a really neat trick:

Let's think about how to add 3 numbers to make 2 numbers.

"Carry-Save Addition"

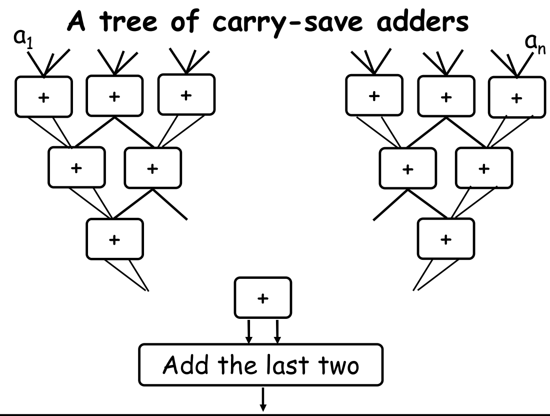
The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

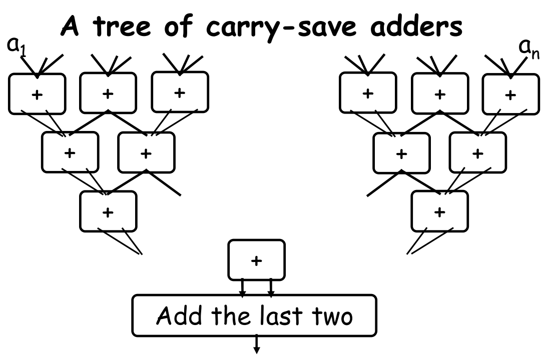
$$\begin{array}{r} 1100111011 \\ +\ 1011111101 \\ +\ 1000000110 \\ \hline \end{array}$$

“Carry-Save Addition”

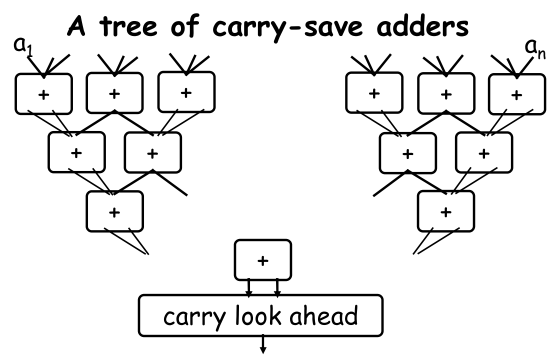
The sum of three numbers can be converted into the sum of 2 numbers in constant parallel time!

```
  1100111011
+ 1011111101
+ 1000000110
------------
  1111000000   ← XOR
+ 10001111110  ← Carries
```



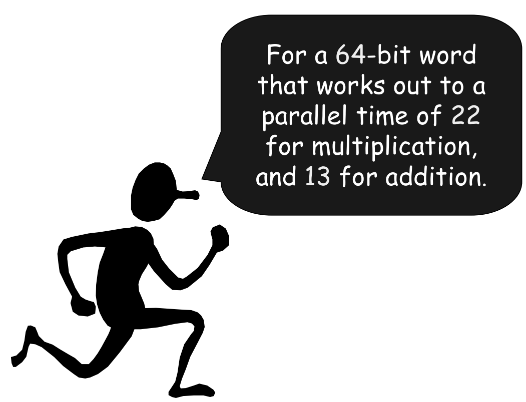


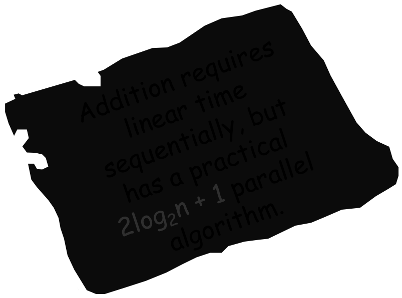
$T(n) \approx \log_{3/2}(n) + [\text{last step}]$



$T(n) \approx \log_{3/2}(n) + 2\log_2 2n + 1$

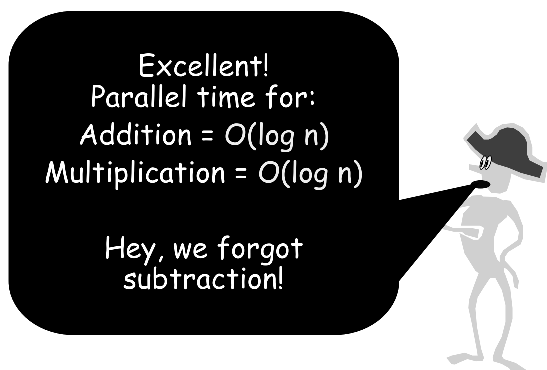
We can multiply in O(log n) parallel time too!

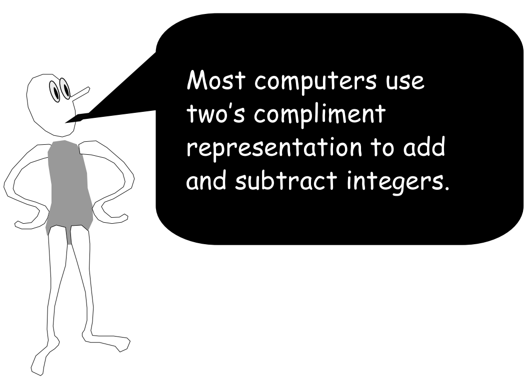




And this is how addition works on commercial chips…..

Processor	n	$2\log_2 n + 1$
80186	16	9
Pentium	32	11
Alpha	64	13

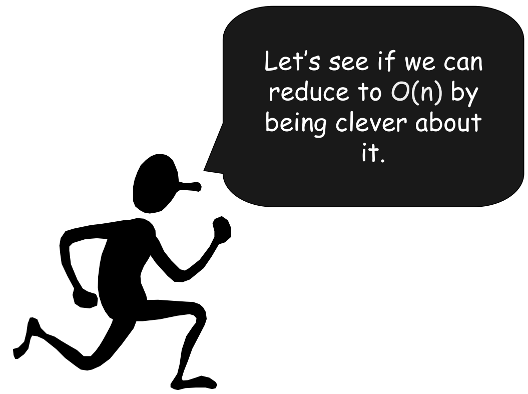


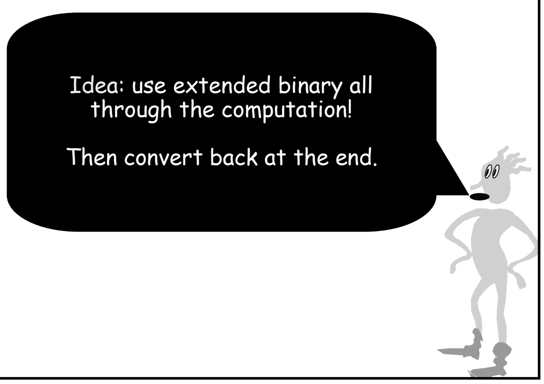


Grade School Division

```
                    * * * * * * * * * *
* * * * * * *  | * * * * * * * *
                 * * * * * * * *
                   * * * * * * * *
                   * * * * * * * *
                     * * * * * * * *
                     * * * * * * * *
                       * * * * * * * *
                       * * * * * * * *
                         * * * * * * * *
```

Suppose we have n bits of precision.
Naïve method: n subtractions costing $2\log_2 n + 1$ each = $\Theta(n \log n)$ parallel time





SRT division algorithm

```
           1  1 -1  1  0   r -1 -1  1
1 0 1 1 | 1 1 1 0 1 1 0 1
          -1 0 -1 -1
          ----------
             1  0 -1  1
            -1  0 -1 -1
            -----------
                  -2  0
            = -1  0  0  1
               1  0  1  1
              ----------
                     1  2
               = 1  0  0  0
                -1  0 -1 -1
                -----------
                   0 -1 -1  1
```

```
    21 r 6        22 r -5
11 | 237      11 | 237
```

Rule: Each bit of quotient is determined by comparing first bit of divisor with first bit of dividend. Easy!

Time for n bits of precision in result:

$$\approx \underbrace{3n}_{\text{1 addition per bit}} + \underbrace{2\log_2(n) + 1}_{\text{Convert to standard representation by subtracting negative bits from positive.}}$$

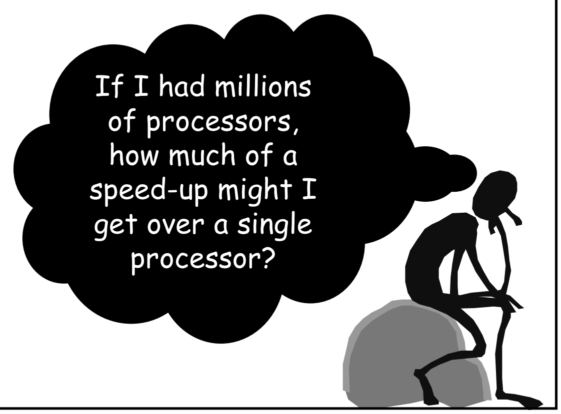
Intel Pentium division error

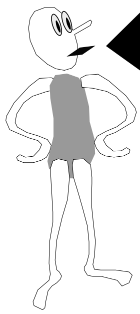
The Pentium uses essentially the same algorithm, but computes more than one bit of the result in each step.

Several leading bits of the divisor and quotient are examined at each step, and the difference is looked up in a table.

The table had several bad entries.

Ultimately Intel offered to replace any defective chip, estimating their loss at $475 million.





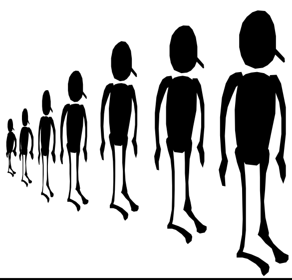
Brent's Law

At best, p processors will give you a factor of p speedup over the time it takes on a single processor.

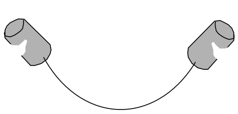
The traditional GCD algorithm will take linear time to operate on two n bit numbers.

Can it be done faster in parallel?

If n^2 people agree to help you compute the GCD of two n bit numbers, it is not obvious that they can finish faster than if you had done it yourself.



Decision Trees and Information: A Question of Bits



20 Questions

S = set of all English nouns

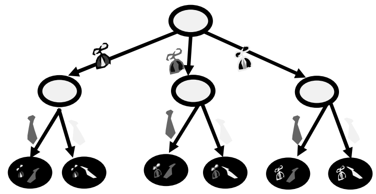
Game:

I am thinking of an element of S.

You may ask up to 20 YES/NO questions.

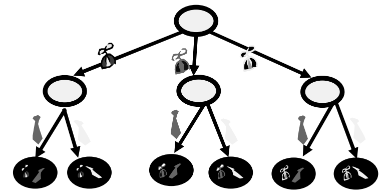
What is a question strategy for this game?

Choice Tree



A choice tree is a rooted, directed tree with an object called a "choice" associated with each edge and a label on each leaf.

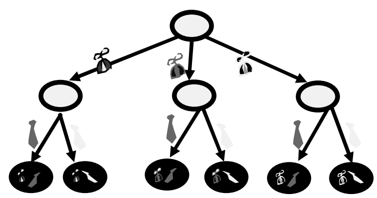
Choice Tree Representation of S



We satisfy these two conditions:

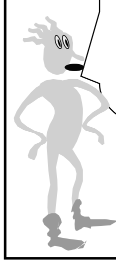
- Each leaf label is in S
- Each element from S on exactly one leaf.

Question Tree Representation of S



I am thinking of an outfit.
Ask me questions until you know which one.

What color is the beanie?
What color is the tie?



When a question tree has at most 2 choices at each node, we will call it a decision tree, or a decision strategy.

Note: Nodes with one choices represent stupid questions, but we do allow stupid questions.

20 Questions

Suppose S = $\{a_0, a_1, a_2, \ldots, a_k\}$

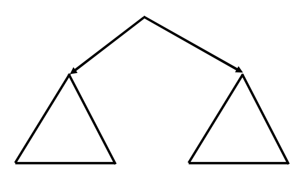
Binary search on S.

First question will be:

"Is the word in $\{a_0, a_1, a_2, \ldots, a_{(k-1)/2}\}$?"

20 Questions Decision Tree Representation

A decision tree with depth at most 20, which has the elements of S on the leaves.



Decision tree for $\{a_0, a_1, a_2, \ldots, a_{(k-1)/2}\}$

Decision tree for $\{a_{(k+1)/2}, \ldots, a_{k-1}, a_k\}$

Decision Tree Representation

Theorem:
The binary-search decision tree for S with k+1 elements { $a_0, a_1, a_2, \ldots, a_k$ } has depth

$$\lceil \log (k+1) \rceil$$
$$= \lfloor \log k \rfloor + 1$$
$$= |k|$$

"the length of k when written in binary"

A lower bound

Theorem: No decision tree for S (with k+1 elements) can have depth $d < \lfloor \log k \rfloor + 1$.

A lower bound

Theorem: No decision tree for S (with k+1 elements) can have depth $d < \lfloor \log k \rfloor + 1$.

Proof:

A depth d binary tree can have at most 2^d leaves.

But $d < \lfloor \log k \rfloor + 1 \Rightarrow$ number of leaves $2^d < (k+1)$

Hence some element of S is not a leaf.

Tight bounds!

The optimal-depth decision tree for any set S with (k+1) elements has depth

$$\lfloor \log k \rfloor + 1 = |k|$$

Recall…

The minimum number of bits used to represent unordered 5 card poker hands

Recall…

The minimum number of bits used to represent unordered 5 card poker hands =

$$\left\lceil \log_2 \binom{52}{5} \right\rceil$$

= 22 bits

= The decision tree depth for 5 card poker hands.

Prefix-free Set

Let T be a subset of $\{0,1\}^*$.

Definition:
T is prefix-free if for any distinct $x,y \in T$,

if $|x| < |y|$, then x is not a prefix of y

Example:
{[redacted]01, [redacted]1} is prefix-free
{0, 01, 10, 11, 101} is not.

Prefix-free Code for S

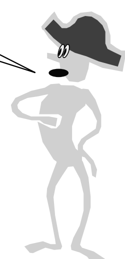
Let S be any set.

Definition: A prefix-free code for S is
a prefix-free set T and
a 1-1 "encoding" function f: S -> T.

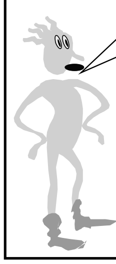
The inverse function f^{-1} is called the "decoding function".

Example: S = {buy, sell, hold}.
T = {0, 110, 1111}.
f(buy) = 0, f(sell) = 1111, f(hold) = 110.

What is so cool about prefix-free codes?



Sending sequences of elements of S over a communications channel



Let T be prefix-free and f be an encoding function. Wish to send $<x_1, x_2, x_3, \ldots>$

Sender: sends $f(x_1)\ f(x_2)\ f(x_3)\ldots$

Receiver: breaks bit stream into elements
of T and decodes using f^{-1}

Sending info on a channel

Example: S = {buy, sell, hold}.
T = {0, 110, 1111}.
f(buy) = 0, f(sell) = 1111, f(hold) = 110.

If we see
00011011111100…
we know it must be
0 0 0 110 1111 110 0 …
and hence
buy buy buy hold sell hold buy …

Morse Code is not Prefix-free!

SOS encodes as …---…

A .-	F ..-.	K -.-	P .--.	U ..-	Z --..
B -...	G --.	L .-..	Q --.-	V ...-	
C -.-.	H	M --	R .-.	W .--	
D -..	I ..	N -.	S ...	X -..-	
E .	J .---	O ---	T -	Y -.--	

Morse Code is not Prefix-free!

SOS encodes as …---…

Could decode as: ..|.-|--|..|. = IAMIE

A .-	F ..-.	K -.-	P .--.	U ..-	Z --..
B -...	G --.	L .-..	Q --.-	V ...-	
C -.-.	H	M --	R .-.	W .--	
D -..	I ..	N -.	S ...	X -..-	
E .	J .---	O ---	T -	Y -.--	

Unless you use pauses

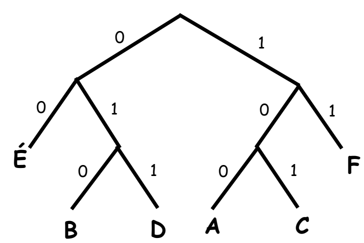
SOS encodes as … --- …

A .-	F ..-.	K -.-	P .--.	U ..-	Z --..
B -...	G --.	L .-..	Q --.-	V ...-	
C -.-.	H	M --	R .-.	W .--	
D -..	I ..	N -.	S ...	X -..-	
E .	J .---	O ---	T -	Y -.--	

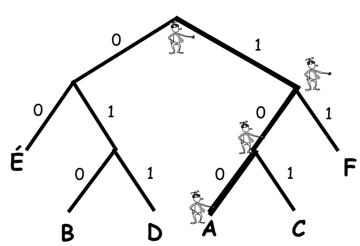
Prefix-free codes are also called "self-delimiting" codes.

Representing prefix-free codes

A = 100
B = 010
C = 101
D = 011
É = 00
F = 11

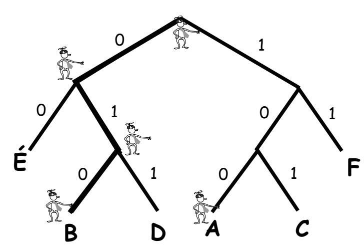


"CAFÉ" would encode as 1011001100
How do we decode 1011001100 (fast)?



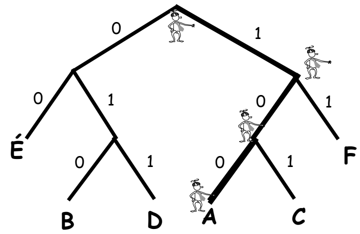
If you see: 1000101000111011001100

can decode as:



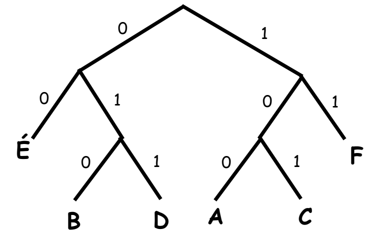
If you see[redacted]000101000111011001100

can decode as: A



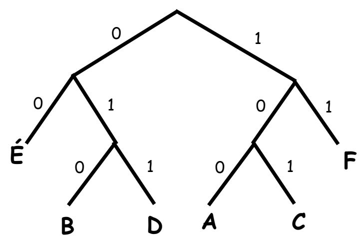
If you see: 1000101000111011001100

can decode as: AB



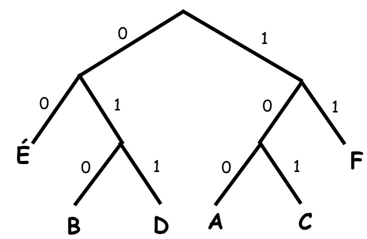
If you see: 1000101000111011001100

can decode as: ABA



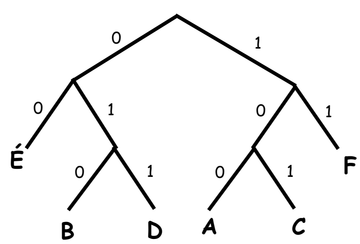
If you see: 1000101000111011001100

can decode as: ABAD



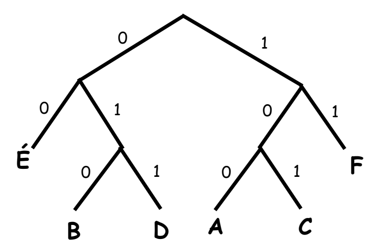
If you see: 1000101000111011001100

can decode as: ABADC



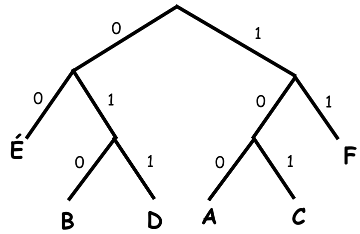
If you see: 1000101000111011001100

can decode as: ABADCA



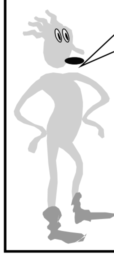
If you see: 1000101000111011001100

can decode as: ABADCAF



If you see: 10001010001110110[redacted]1100

can decode as: ABADCAFÉ



Prefix-free codes are yet another representation of a decision tree.

Theorem:

S has a decision tree of depth d

if and only if

S has a prefix-free code with all codewords bounded by length d

Extends to infinite sets

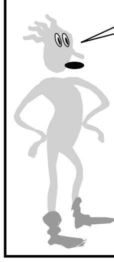
Let S is a subset of Σ^*

Theorem:

S has a decision tree where all length n elements of S have depth $\leq$ D(n)

if and only if

S has a prefix-free code where all length n strings in S have encodings of length $\leq$ D(n)



I am thinking of some natural number k. ask me YES/NO questions in order to determine k.

Let d(k) be the number of questions that you ask when I am thinking of k.

Let D(n) = max { d(k) over n-bit numbers k }.

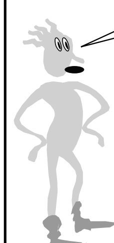
I am thinking of some natural number k - ask me YES/NO questions in order to determine k.

Naïve strategy: Is it 0? 1? 2? 3? …

d(k) = k+1

D(n) = 2^{n+1} since 2^{n+1} -1 uses only n bits.

Effort is exponential in length of k



I am thinking of some natural number k - ask me YES/NO questions in order to determine k.

What is an efficient question strategy?

I am thinking of some natural number k…

Does k have length 1? NO
Does k have length 2? NO
Does k have length 3? NO
…
Does k have length n? YES
Do binary search on strings of length n.

$d(k) = |k| + |k|$
$\quad = 2 (\lfloor \log k \rfloor + 1)$

$D(n) = 2n$

<u>Size First/ Binary Search</u>

Does k have length 1? NO
Does k have length 2? NO
Does k have length 3? NO
…
Does k have length n? YES
Do binary search on strings of length n.

What prefix-free code corresponds to the Size First / Binary Search decision strategy?

$f(k)$ = ($|k|$ - 1) zeros, followed by 1, and then by the binary representation of k

$|f(k)| = 2\ |k|$

Another way to look at f

k = 27 = 11011, and hence $|k| = 5$

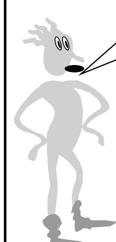
f(k) = 00001 11011

"Fat Binary" ⇔ Size First/Binary Search strategy

Is it possible to beat 2n questions to find a number of length n?

Look at the prefix-free code…

Any obvious improvement suggest itself here?



length of k in unary ⇒ $|k|$ bits
k in binary ⇒ $|k|$ bits

In fat-binary, $D(n) \leq 2n$

Now $D(n) \leq n + 2 (\lfloor \log n \rfloor + 1)$

Can you do better?

better-than-Fat-Binary-code(k) concatenates

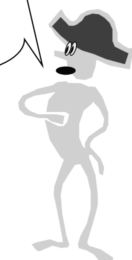
length of k in fat binary $\Rightarrow$ 2||k|| bits

k in binary $\Rightarrow$ |k| bits

Hey, wait!

In a better prefix-free code

RecursiveCode(k) concatenates RecursiveCode(|k|) & k in binary



better-than-Fat-Binary code

better-t-FB ||k|| + 2|||k|||

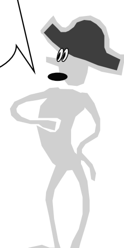
|k| in fat binary $\Rightarrow$ 2||k|| bits

k in binary $\Rightarrow$ |k| bits

Oh, I need to remember how many levels of recursion r(k)

In the final code

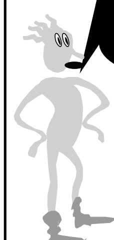
F(k) = F(r(k)) . RecursiveCode(k)



r(k) = log* k

Hence, length of F(k)

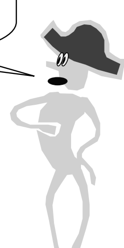
= |k| + ||k|| + |||k||| + … + 1

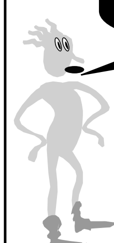


Good, Bonzo! I had thought you had fallen asleep.

Maybe I can do better…

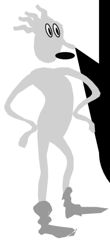
Can I get a prefix code for k with length ≈ log k ?





No!

Let me tell you why length ≈ log k is not possible



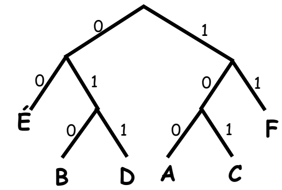
Decision trees have a natural probabilistic interpretation.

Let T be a decision tree for S.

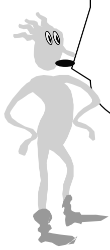
Start at the root, flip a fair coin at each decision, and stop when you get to a leaf.

Each sequence **w** in S will be hit with probability $\mathbf{1/2^{|w|}}$

Random walk down the tree



Pr(F) = ¼, Pr(A) = 1/8, Pr(C) = 1/8, …

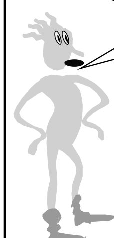


Let T be a decision tree for S (possibly countably infinite set)

The probability that some element in S is hit by a random walk down from the root is

$\sum_{w\in S} 1/2^{|w|} \leq 1$

Kraft Inequality



Let S be any prefix-free code.

Kraft Inequality:
$\sum_{w\in S} 1/2^{|w|} \leq 1$

Fat Binary:
f(k) has 2|k| ≈ 2 log k bits

$\sum_{k\in\mathbb{N}} \frac{1}{2}^{|f(k)|} \leq 1$

$\approx \sum_{k\in\mathbb{N}} 1/k^2$

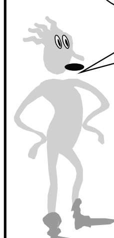
Let S be any prefix-free code.

Kraft Inequality:
$\sum_{w\in S} 1/2^{|w|} \leq 1$

Better-than-FatB Code:
f(k) has |k| + 2||k|| bits

$\sum_{k\in\mathbb{N}} \frac{1}{2}^{|f(k)|} \leq 1$

$\approx \sum_{k\in\mathbb{N}} 1/(k (\log k)^2)$



Let S be any prefix-free code.

Kraft Inequality:
$\sum_{w\in S} 1/2^{|w|} \leq 1$

Ladder Code: k is represented by
|k| + ||k|| + |||k||| + … bits

$\sum_{k\in\mathbb{N}} \frac{1}{2}^{|f(k)|} \leq 1$

$\approx \sum_{k\in\mathbb{N}} 1/(k \log k \log\log k \ldots)$

Let S be any prefix-free code.

Kraft Inequality:
$\sum_{w \in S} 1/2^{|w|} \leq 1$

Can a code that represents k by $|k| = \log k$ bits exist?

No, since $\sum_{k \in \mathbb{N}} 1/k$ diverges !!

So you can't get log n, Bonzo…

Back to compressing words

The optimal-depth decision tree
for any set S with (k+1) elements has
depth
$\lfloor \log k \rfloor + 1$
$\Downarrow$

The optimal prefix-free code
for A-Z + "space" has length
$\lfloor \log 26 \rfloor + 1 = 5$

English Letter Frequencies

But in English, different letters occur with different *frequencies*.

A 8.1%	F 2.3%	K .79%	P 1.6%	U 2.8%	Z .04%
B 1.4%	G 2.1%	L 3.7%	Q .11%	V .86%	
C 2.3%	H 6.6%	M 2.6%	R 6.2%	W 2.4%	
D 4.7%	I 6.8%	N 7.1%	S 6.3%	X .11%	
E 12%	J .11%	O 7.7%	T 9.0%	Y 2.0%	

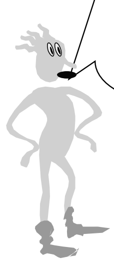
short encodings!

Why should we try to minimize
the maximum length of a codeword?

If encoding A-Z, we will be happy if
the "<u>average codeword</u>" is short.

Given frequencies for A-Z, what is the optimal prefix-free encoding of the alphabet?

I.e., one that minimizes the <u>average</u> code length



Huffman Codes: Optimal Prefix-free Codes Relative to a Given Distribution

Here is a Huffman code based on the English letter frequencies given earlier:

A 1011	F 101001	K 10101000	P 111000	U 00100
B 111001	G 101000	L 11101	Q 1010100100	V 1010101
C 01010	H 1100	M 00101	R 0011	W 01011
D 0100	I 1111	N 1000	S 1101	X 1010100101
E 000	J 1010100110	O 1001	T 011	Y 101011
				Z 1010100111

References

The Mathematical Theory of Communication, by C. Shannon and W. Weaver

Elements of Information Theory, by T. Cover and J. Thomas