[Zec72, FK96]

# Welcome to SIGma

SIGma

Section 1

# Officers in No Particular Order

# Anakin

- Math Major
- Did Computational Group Theory at an REU
- Graph Theory / Optimization Research during the year
- SIGPwny Crypto[1] Gang + Admin team
- Coffee Club
- CA for CS 173 + CS 374

[1] Not that one, the other one

# Aditya

- ECE/Math double major.
- Interned at a satellite internet startup over the summer.
- CA for ECE 411, ECE 391 + SIGARCH co-lead.
- Other interests: FP, EE, Crypto(graphy).

## Sam

- CS PhD
- Doing Computational Geometry with Sariel Har-Peled
- SIGPwny

# Hassam

- Intern at IMC Trading over the summer
- CS Major (takes math classes for fun ???)
- SIGPwny Crypto Gang + Admin team + Infra lead
- CA for CS 341, CS 173
- Compiler research

# We Need Officers!

- This list is smaller than last year
- Reach out to me if you are interested in improving SIGma and making meetings!

Section 2

# Fibonacci Codes

## But Why?

- Almost everything you do online involves sending and receiving messages
- How can we make these messages “robust” to errors?

# Starting From The End

- Suppose we want to uniquely assign the natural numbers a *codeword*
- We want this *code* to have a couple properties
  - Quick to compute
  - Variable length
  - Robust to errors
- It turns out *Fibonacci Numbers* do all this for us

# Our Favorite Sequence

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ | $F_{10}$ | $F_{11}$ | $F_{12}$ | $F_{13}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 |

# Zeckendorf's Theorem

**Theorem ( [Zec72])**

Every natural number $n \geq 1$ can be represented as a unique sum of non-consecutive Fibonacci numbers $F_i$ where $i \geq 2$. *If we allow* $F_0$ *and* $F_1$ *we lose uniqueness*

We call this sum a *Zeckendorf sum*.

- $4 = F_4 + F_2 = 3 + 1$
- $64 = F_{10} + F_6 + F_2 = 55 + 8 + 1$

# Recursion is Induction is Recursion is Induction is . . .

We prove existence by *induction* on our natural number $n$. Suppose that for all natural numbers strictly smaller than $n$, such a Zeckendorf sum exists. There are two cases.

If $n \leq 4$:

$$1 = F_2 \qquad 2 = F_3$$
$$3 = F_4 \qquad 4 = F_4 + F_2$$

If $n > 4$ itself is a Fibonacci number, we are done.

# Recursion is Induction is Recursion is Induction is ...

If $n > 4$ is not a Fibonacci number:

- Since $n > 4$, it is strictly between two consecutive Fibonacci numbers $F_i < n < F_{i+1}$ for some $i \geq 3$
- $n - F_i < n$ so, by *induction*, $n - F_i$ has some Zeckendorf sum
- Note that

$$\begin{aligned} & n - F_i + F_i = n < F_{i+1} = F_{i-1} + F_i \\ \Longrightarrow\ & n - F_i < F_{i-1} \end{aligned}$$

  and thus the Zeckendorf sum of $n - F_i$ does not contain $F_{i-1}$
- Combine the Zeckendorf sum of $n - F_i$ with $F_i$ to obtain a Zeckendorf sum for $n$ $\square$

# Deadly Sins $\implies$ Fast Algorithms

The statement and proof of the theorem helps design a *greedy* algorithm

- The inductive proof implies we should find the largest $F_i \leq n$
- The statement implies that if we picked $F_i$, we should skip $F_{i-1}$
- Our goal is to encode text, so we can precompute an array of Fibonacci numbers ahead of time up to some maximum

```
maximum ← 1114111    ⟪ largest Unicode value U+10FFFF ⟫
F ← [0, 1]
i ← 2
while F[i − 1] ≤ maximum:
    F.append(F[i − 2] + F[i − 1])
    i += 1
```

# Deadly Sins $\Longrightarrow$ Fast Algorithms

```
ZECKENDORF(x):
    i ← max i such that F[i] ≤ x
    rep ← " "
    rem ← x
    while i ≥ 2 :
        if F[i] ≤ rem :
            rem −= F[i]
            rep += 1
            if rem > 0 :
                rep += 1
                i −= 1
        else:
            rep += 0
        i −= 1
    return rep
```

# Deadly Sins $\implies$ Fast Algorithms

**ZECKENDORF**$(x)$:

```
i ← max i such that F[i] ≤ i
rep ← “ ”
rem ← x
while i ≥ 2 :
    if F[i] ≤ rem :
        rem −= F[i]
        rep += 1
        if rem > 0 :
            rep += 1
            i −= 1
    else:
        rep += 0
    i −= 1
return rep
```

$\#$ of $i$ such that $F_i \leq x$

$$= \left\lfloor \log_\phi \left( x\sqrt{5} \right) \right\rfloor = O(\log x)$$

- The while loop does $i = O(\log x)$ iterations
- The work inside the while loop takes $O(1)$ time
- So ZECKENDORF takes $O(\log x)$ time
- Each iteration we add at most 2 characters $\implies$ $|\text{ZECKENDORF}(x)| = O(\log x)$

# Deadly Sins $\implies$ Fast Algorithms

```
ZECKENDORF(x):
    i ← max i such that F[i] ≤ i
    rep ← " "
    rem ← x
    while i ≥ 2 :
        if F[i] ≤ rem :
            rem −= F[i]
            rep += 1
            if rem > 0 :
                rep += 1
                i −= 1
        else:
            rep += 0
        i −= 1
    return rep
```

$\#$ of $i$ such that $F_i \le x$

$$= \left\lfloor \log_\phi \left( x\sqrt{5} \right) \right\rfloor = O(\log x)$$

- The while loop does $i = O(\log x)$ iterations
- The work inside the while loop takes $O(1)$ time
- So ZECKENDORF takes $O(\log x)$ time
- Each iteration we add at most 2 characters $\implies |\text{ZECKENDORF}(x)| = O(\log x)$

# Deadly Sins $\implies$ Fast Algorithms

**Zeckendorf**($x$):

$i \leftarrow \max i$ such that $\mathrm{F}[i] \leq i$
$rep \leftarrow$ “ ”
$rem \leftarrow x$
while $i \geq 2$ :
    if $\mathrm{F}[i] \leq rem$ :
        $rem \mathrel{-}= \mathrm{F}[i]$
        $rep \mathrel{+}= \mathtt{1}$
        if $rem > 0$ :
            $rep \mathrel{+}= \mathtt{1}$
            $i \mathrel{-}= 1$
    else:
        $rep \mathrel{+}= \mathtt{0}$
    $i \mathrel{-}= 1$
return $rep$

$\#$ of $i$ such that $F_i \leq x$

$$= \left\lfloor \log_\phi \left( x\sqrt{5} \right) \right\rfloor = O(\log x)$$

- The while loop does $i = O(\log x)$ iterations
- The work inside the while loop takes $O(1)$ time
- So Zeckendorf takes $O(\log x)$ time
- Each iteration we add at most 2 characters $\implies |\text{Zeckendorf}(x)| = O(\log x)$

# Deadly Sins $\implies$ Fast Algorithms

**ZECKENDORF**$(x)$:

$i \leftarrow \max i$ such that $\mathrm{F}[i] \leq i$
$\quad$ *rep* $\leftarrow$ " "
$\quad$ *rem* $\leftarrow x$
$\quad$ while $i \geq 2$ :
$\quad\quad$ if $\mathrm{F}[i] \leq$ *rem* :
$\quad\quad\quad$ *rem* $-= \mathrm{F}[i]$
$\quad\quad\quad$ *rep* $+=$ 1
$\quad\quad\quad$ if *rem* $> 0$ :
$\quad\quad\quad\quad$ *rep* $+=$ 1
$\quad\quad\quad\quad$ $i \mathrel{-}= 1$
$\quad\quad$ else:
$\quad\quad\quad$ *rep* $+=$ 0
$\quad\quad$ $i \mathrel{-}= 1$
$\quad$ return *rep*

\# of $i$ such that $F_i \leq x$

$$= \left\lfloor \log_\phi \left( x\sqrt{5} \right) \right\rfloor = O(\log x)$$

- The while loop does $i = O(\log x)$ iterations
- The work inside the while loop takes $O(1)$ time
- So **ZECKENDORF** takes $O(\log x)$ time
- Each iteration we add at most 2 characters $\implies$ $|\textbf{ZECKENDORF}(x)| = O(\log x)$

# Undo

**FRODNEKCEZ**($rep[0..n]$):

$res \leftarrow 0$<br>
for $i \leftarrow 0..n$:<br>
    if $rep[i] = 1$:<br>
        $res \mathrel{+}= \mathrm{F}[i+2]$ ⟪ *Remember we don't use $F_0$ or $F_1$* ⟫<br>
return $res$

Runtime analysis:

- Work is constant for each iteration of the for loop $\implies O(n)$
- If $rep[0..n] = \textbf{Zeckendorf}(x)$ then $O(n) = O(\log x)$

# A Fibonacci Code

We now show how to assign natural numbers a code word using the Zeckendorf Decomposition [FK96]

- The length of the Zeckendorf Representation for numbers can vary
  - $\textsc{Zeckendorf}(2) = \texttt{01}$, $\textsc{Zeckendorf}(7) = \texttt{0101}$
- We want to be able to send this bit strings and tell when a character begins and ends
  - Does `0101` correspond to $[2, 2]$ or $[7]$?
- Solution: Add a *"comma"* using an extra `1`
  - $\mathbf{ENC}([2, 2]) = \texttt{011011}$, $\mathbf{ENC}([7]) = \texttt{01011}$

# Heavy Lifting Has Already Been Done

$\underline{\textsc{Enc}(x)}$:
1: return $\textsc{Zeckendorf}(x) + \texttt{1}$ ⟪ *add comma* ⟫

$\underline{\textsc{Dec}(rep[0..n])}$:
1: return $\textsc{Frodnekcez}(rep[0..n-1])$ ⟪ *remove comma* ⟫

Runtime analysis:

- Same as **Zeckendorf** and **Frodnekcez**

# Heavy Lifting Has Already Been Done

```
ENCODE(m[0..n]):
  code ← “ ”
  for i ← 0..n:
      val ← ORD(m[i])
      code += ENC(val)
  return code
```

Runtime analysis:

- To simplify our life, since ORD($m[i]$) is some Unicode value which has a set maximum, **ENC** runs in constant time
  - More precise analysis would require knowledge of the distribution of characters in whatever language being used. Ask your nearest linguist.
- Thus, **ENCODE**($m[0..n]$) runs in $O(n)$ time

# Heavy Lifting Has Already Been Done

$\underline{\textbf{DECODE}(code[0..n]):}$
$m \leftarrow$ " "
$i \leftarrow 0$
while $i \leq n$:
  $j \leftarrow$ smallest $j > i$ such that
  $code[j] = code[j+1] = 1$
  $rep = code[i..j+1]$
  $n \leftarrow \textbf{DEC}(rep)$
  $m \mathrel{+}= \textsc{chr}(n)$
  $i \leftarrow j+1$
return $m$

Runtime analysis:

- By similar logic, $\textbf{DECODE}(code[0..n])$ runs in $O(n)$ time

# An Example

| S | I | G | m | a |
|---|---|---|---|---|
| 83 | 73 | 71 | 109 | 97 |
| 0101001011 | 0001010011 | 0010010011 | 01010100011 | 00001000011 |

# Containment of Errors

*Claim:* When a single error occurs, at most 3 codewords are lost

- We know that $\textbf{ENC}(x)$ ends with `011` for all $x > 1$
- For such $x$, if an error occurs outside of these last 3 bits, only one codeword is lost:
  - If a `01` gets turned into a `11`, if a `0` is deleted, or if a `1` is inserted in some specific spot, then one codeword may turn into two
  - Consider `0101011` $\rightsquigarrow$ `0111011` / `011011` / `01101011`
  - Otherwise, we just misconvert that single codeword

Questions?

*Abstract is a word people use when they haven't gotten used to something*

— EUGENE LERMAN (8/28/2023)

# Question: Containment of Errors

*Claim:* When a single error occurs, at most 3 codewords are lost

- We know that $\textbf{ENC}(x)$ ends with `011` for all $x > 1$
- For such $x$, if an error occurs outside of these last 3 bits, only one codeword is lost:
  - If a `01` gets turned into a `11`, if a `0` is deleted, or if a `1` is inserted in some specific spot, then one codeword may turn into two
  - Consider `0101011` $\rightsquigarrow$ `0111011` / `011011` / `01101011`
  - Otherwise, we just misconvert that single codeword

*Exercise:* Consider what may happen in the cases of insertion, deletion, and bitflipping for each of the last three bits of $\textbf{ENC}(x)$ for $x > 1$

# Bibliography


Aviezri S. Fraenkel and Shmuel T. Kleinb.

Robust universal complete codes for transmission and compression.

*Discrete Applied Mathematics*, 64(1):31–55, January 1996.

E. Zeckendorf.

Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas.

*Bull. Soc. R. Sci. Liège*, 41:179–182, 1972.