Grape Codes

Kunal Marwaha

March 2020

1 Why?

My friend Madison and I were looking through a booklet from Julia Robinson Mathematics Festival, a wonderful math education project. The booklet describes Grape Codes, a version of binary where you can have any number at each digit. Madison asked me: "How many grape codes are there?" This made me very curious. Part of this analysis was completed with another friend, Nick Sherman.

2 Problem

In Grape Code, a number n can be represented by a sequence $\{a_0, a_1, ...\}$ written in reverse (like $...a_2a_1a_0$), where $a_i \in \mathbb{N}$ and $n = \sum_i a_i 2^i$. How many unique sequences exist for a number n?

2.1 The first few cases

Let's define $H(n)$ as the number of unique Grape Codes for a given natural number n. I list a few examples below.

n	Grape Codes	$H(n)$
0	0	1
1	1	1
2	2;10	2
3	3;11	2
4	4;12,20;100	4
5	5;13,21;101	4
6	6;14,22,30;102,110	6
7	7;15,23,31;103,111	6
8	8;16,24,32,40;104,112,120;200;1000	10
9	9;17,25,33,41;105,113,121;201;1001	10
10	(10);18,26,34,42,50;106,114,122,130;202,210;1002,1010	14
11	(11);19,27,35,43,51;107,115,123,131;203,211;1003,1011	14

2.2 How to find Grape Codes

One way to find Grape Codes is to start with the initial number $n = a_0$, with 0 for the rest of the sequence. Then, if you have at least 2 in any digit a_k, subtract 2 from that digit and add 1 to a_{k+1}. Repeat this process until you cannot. At this time, there are only 0s and 1s for all a_i, so it is the unique binary representation of n. All Grape Codes can be found using this procedure (although I am not sure how to rigorously prove this). See 4.1 for a software implementation.

3 Describing $H(n)$

3.1 Monotonicity

Take a number n. Any of its Grape Codes can represent $n+1$ by adding 1 to a_0. So, $H(n)$ is monotonic:

$$H(k+1) \geq H(k) \quad \forall k \in \mathbb{N}$$

3.2 Even and odd patterns

For any odd number n, $a_0 \geq 1$ for any Grape Code (see the procedure in 2.2). So, any Grape Codes can represent $n-1$ by subtracting 1 from a_0. Together with 3.1, any odd number has the same number of Grape Codes as the number before it:

$$H(2k+1) = H(2k) \quad \forall k \in \mathbb{N}$$

3.3 Recursive solution

Define $H_d(n)$ as the number of d-digit Grape Codes, i.e. with $a_k = 0$ for all $k > (d-1)$. Then:

$$H(n) = \lim_{d \to \infty} H_d(n) = H_n(n) = H_{d > \log_2(n)}(n)$$

For even n, consider the number of two-digit Grape Codes. There are $\frac{n}{2} + 1$ of these Grape Codes, each with a unique value of $a_0 \in \{0, 2, 4, ..., n\}$. Together with 3.2, we can find the number of two-digit Grape Codes:

$$\text{(two-digit) } H(n) = H_2(n) = \sum_{u=0}^{\lfloor n/2 \rfloor} 1 = \lfloor n/2 \rfloor + 1$$

Now consider the number of Grape Codes with $a_2 = v$ and all $a_i = 0$ for $i \geq 3$. Given that $n \geq 4v$, we can count the number of two-digit Grape Codes $H_2(n-4v)$. We can then find all three-digit Grape Codes:

$$\text{(three-digit) } H(n) = H_3(n) = \sum_{v=0}^{\lfloor n/4 \rfloor} H_2(n-4v) = \sum_{v=0}^{\lfloor n/4 \rfloor} \lfloor n/2 \rfloor - 2v + 1 = (\lfloor n/4 \rfloor + 1)(\lfloor n/2 \rfloor - \lfloor n/4 \rfloor + 1)$$

In general, we have a recursive formula for $H_d(n)$ (see 4.2 for a software implementation):

$$\text{(d-digit) } H(n) = H_d(n) = \sum_{w=0}^{\lfloor n/2^{d-1} \rfloor} H_{d-1}(n - 2^{d-1}w)$$

3.4 Growth rate

For powers of 2 (i.e. $n = 2^r$ for $r \in \mathbb{N}$), we can simplify the recursive formula for $H_d(n)$:

$$H_d(n) = \sum_{x=0}^{n/2^{d-1}} H_{d-1}(2^{d-1}x)$$

This can be approximated as an integral:

$$H_d(n) \approx \frac{1}{2^{d-1}} \int_{x=0}^{n} dx H_{d-1}(x)$$

Using $H_1(n) = 1$, we can approximate $H(n)$:

$$H(n) = H_{r+1}(n) \approx \frac{1}{2^{r(r+1)/2}} \frac{n^r}{r!} \approx \frac{n^{r/2}}{r!} = \frac{n^{\log_2(n)/2}}{\log_2(n)!}$$

To get the form $H(n) \propto n^k$, we can use Stirling's formula on $\log_2(H(n))$ (where e is Euler's constant):

$$\log_2(H(n)) = \log_2(n)^2/2 - \log_2(\log_2(n)!) \approx \log_2(n)^2/2 - \log_2(n)\big(\log_2(\log_2(n)) - \log_2(e)\big)$$

$$H(n) \approx n^{log_2(n)/2 - log_2(log_2(n)) + \log_2(e)}$$

Figure 1 shows that the approximations made here align well with $H(n)$.

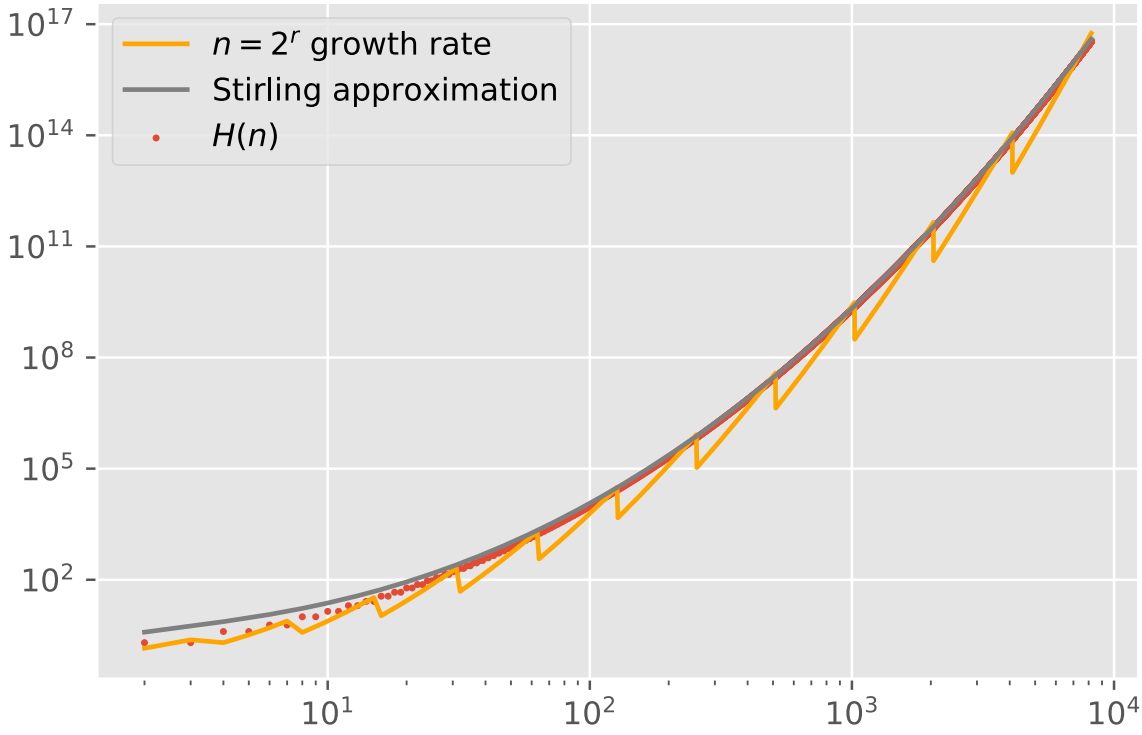

Figure 1: Visualizing $H(n)$ for $n < 2^{13}$ along with approximations of $H(n)$. On a log-log plot, $y = x^{\log(x)}$ looks like a quadratic function. The 2^r growth rate is jagged from implementing $\log_2(n)!$ as $\lfloor \log_2(n) \rfloor !$, which could be removed by using the gamma function. The Stirling approximation smooths this out. On qualitative inspection, the approximations match $H(n)$ quite closely.

3.5 Intuition for the growth rate

Suppose you can choose any number for each digit a_i, as long as $a_i 2^i \leq n$ (i.e. $a_i \leq \frac{n}{2^i}$). The number of total choices is approximately $(\frac{n}{1})(\frac{n}{2})(\frac{n}{4})(\frac{n}{8})...(8)(4)(2)(1)$. There are about $\log_2(n)$ terms in this expression. Notice that terms from each side of the expression can be multiplied together to form n, simplifying the expression:

$$(\frac{n}{1})(\frac{n}{2})(\frac{n}{4})(\frac{n}{8})...(8)(4)(2)(1) \approx n^{\log_2(n)/2} = (\sqrt{n})^{\log_2(n)}$$

This overcounts $H(n)$, as most choices of $\{a_i\}$ will not sum to n. In fact, each choice of digit d (i.e. a_{d-1}) only permits certain collections of $\{a_i\}_{i<(d-1)}$. These choices evenly sample the possibilities of H_{d-1} from 0 to n. Choosing a_1 evenly samples choices of a_0 from 0 to n, so $H_2(k) \propto k$. By induction, $H_d(n) \propto \frac{1}{(d-1)!}$, so $H(n) \propto \frac{1}{\log_2(n)!}$.

4 Simulation

4.1 Procedure to find Grape Codes

Here is Python code that executes the procedure in 2.2 to calculate $H(n)$.

```
def procedure(lst, hashes):
    hashes.add(str(lst))
    for i in range(len(lst)):
```

```
        if lst[i] > 1:
            cp = lst[:]
            cp[i] -= 2
            if i == len(lst) - 1:
                cp.append(0)
            cp[i+1] += 1
            if str(cp) not in hashes:
                hashes |= procedure(cp, hashes)
    return hashes

def H(n):
    return len(procedure([n], set()))
```

4.2 Recursive solution

This Python code calculates $H_d(n)$ recursively. Even without the cache, this runs much more quickly than 4.1.

```
from math import floor
from functools import lru_cache

@lru_cache(maxsize=2**24)
def Hd(d, n):
    if d == 1:
        return 1
    offset = 2**(d-1)
    if n < offset:
        return Hd(d-1, n)
    return sum([Hd(d-1, n - i*offset) for i in range(floor(n/offset) + 1)])
```

4.3 Growth rate approximation

This Python code implements the approximations to $H(n)$ made in 3.4.

```
from math import floor, log, e, factorial
def growth(n):
    return n**(log(n, 2)/2)/factorial(floor(log(n, 2)))

def stirling(n):
    return n**(log(n, 2)/2 + log(e, 2) - log(log(n, 2), 2))
```

Figure 1 compares the approximations with direct calculation of $H(n)$. This code creates that figure:

```
import matplotlib.pyplot as plt
ins = range(2, 2**13)
plt.style.use('ggplot')
plt.scatter(ins, [Hd(13, i) for i in ins], s=3, label="$H(n)$")
plt.plot(ins, list(map(growth, ins)), color='orange', label="$n=2^r$ growth rate")
plt.plot(ins, list(map(stirling, ins)), color='gray', label="Stirling approximation")
plt.yscale('log')
plt.xscale('log')
plt.legend()
```

5 Onwards

Mathematics can be so fun and so beautiful. Reach me at marwahaha@berkeley.edu if you have other curious ideas.