CSE 280
Assignment 1
Andrey Baikadze
A53239410
List 1.

① Signed up

② Let us calculate (c) first

n – ploidy
m – #alleles.

The problem can be reformulated:

$$\begin{cases} i_1 + \dots + i_m = n \\ i_j \geq 0 \quad (i_j \in \mathbb{Z}) \; \forall j \in 1{:}m \end{cases}$$

? $i_1 \dots i_m$ How many tuples?

Standard combinatorial task: stars-and-bars counting

$m-1$ bars and n stars

The total # of places is $n+m-1$ (to put either a bar or a star)

$\Rightarrow$ Put n stars at any order:

$$\binom{n+m-1}{n}$$

# of distinct genotypes.

(a) $n = 6$, $m = 4$ $\Rightarrow$ $\binom{n+m-1}{n} = \frac{6!}{4!2!} = \frac{8!}{6!} = \frac{9!}{6!\cdot 3!} = \frac{7\cdot 8\cdot 9}{3} = 84$

$\Rightarrow$ The number of distinct possible genotypes of hexaploid "bread wheat" is **84**.

(b) ab^3c^2. The probability to generate such a genotype is

$$\binom{6}{3}\binom{3}{2} P_a P_b^3 P_c^2 = 20 \cdot 3 \cdot 0.5 \cdot 0.25^3 \cdot 0.15^2 = 0.0105$$

$\Rightarrow$ The expected # in 1000 simulations is $\approx$ **105**

CSE 280
Assignment 1
Andrey Bakudze
A532 39410
List 2

(3) For a diploid population with m alleles we would like to infer probabilities from the set of eq-s $\{ 2\hat{p}_i\hat{p}_j = \frac{n_{ij}}{n} \}$ // hat means - the estimate of real p // $\forall 1 \le i < j \le m$.
We do not have real hope to solve this set of eq-s explicitly, so instead we formulate a least squares minimization (optimization) problem:

$$\sum_{1 \le i \le j \le m} \left(2\hat{p}_i\hat{p}_j - \frac{n_{ij}}{n} \right)^2 \longrightarrow \min_{\substack{p=(p_1 \dots p_m)^T \\ p_i \ge 0 \\ \sum_{i=1}^{m} p_i = 1}}$$

Maybe the exact solution exists, but one could just run numerical optimization algorithm with constrains (L-BFGS, for example).

In the case of 4 alleles and matrix from the question the minimization gives the following probabilities:

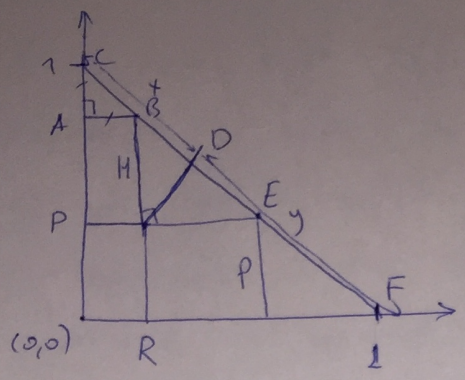
$$p_1 \approx 0.5146 \quad p_2 \approx 0.1705, \quad p_3 \approx 0.2023, \quad p_4 \approx 0.1126.$$

The code is on the next page.

```
from scipy.optimize import minimize
import numpy as np

def MostLikelyFrequences(freqs, n, k):
    def opt_fun(p):
        res = 0
        for i in range(len(freqs)):
            for j in range(len(freqs[i])):
                res += (2 * p[i] * p[i+j+1] - freqs[i][j] / n)**2
        return res

    p_0 = np.ones(k) / k
    return minimize(opt_fun, p_0, bounds=[(0, 1)]*k,
                    constraints={'type':'eq', 'fun': lambda p:
sum(p) - 1})


freqs = [[18, 21, 12], [7, 3], [5]]
n, k = 100, 4
res = MostLikelyFrequences(freqs, n, k)
print(res.x)
print(sum(res.x))
print(MostLikelyFrequences(freqs, n, k))
```

CSE 280
Assignment 1
Audrey Bikadze
A532 39410
List 4

(4)



(a) (?) $P+H+R=1$.

Is easily obtained from ΔABC isosceles triangular with $AB = AC = R$

$\Rightarrow 1-P = H+R \Rightarrow P+H+R=1$

$P = Pr(AA) = Pr^2(A)$
$H = Pr(Aa) = 2\,Pr(A)\,Pr(a)$
$R = Pr(aa) = Pr^2(a)$

(6) (?) $\frac{y}{x} = \frac{Pr(A)}{Pr(a)}$.

$CB = \sqrt{2}R$, $BD = \sqrt{2}H \Rightarrow x = \sqrt{2}(R+H)$

$\Rightarrow y = \sqrt{2}(P+H)$ $\Big| \Rightarrow \frac{y}{x} = \frac{R+\frac{1}{2}H}{P+\frac{1}{2}H}$.

$EF = \sqrt{2}P$

If we assume $HW \Rightarrow \frac{y}{x} = \frac{Pr^2(A) + Pr(A)\,Pr(a)}{Pr(a) + Pr(A)Pr(a)} = \frac{Pr(A)}{Pr(a)}$

(c) (?) $P^2+R^2-2PR-2P-2R+1=0$

Proof: $P+H+R=1 \Rightarrow P^2 + \underbrace{H^2}_{4PR} + R^2 + \underbrace{2PH}_{2P(1-P-R)} + \underbrace{2HR}_{2R(1-P-R)} + 2PR = 1$

$\Rightarrow -P^2-R^2+2R+2R+2P=1 \Rightarrow \boxed{P^2+R^2-2PR-2R-2P=1}$

CSE 280
Assignment 1
Andrey Bzikadze
A53233410
List 5

(5)

(a) I've written a Python script. (next page). List6

dataset 1-3 — perfect ph. exists

dataset 4-6 — does not exist.

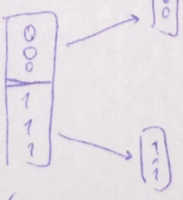
(6) In my implementation $f(n, m) = O(nm)$.

Preliminary step is sorting columns. It can be done in $O(nm)$ time with Radix sort. The constant could be significantly improved with using larger „buckets".

a little offtopic { I once wrote C++ code for Radix sort and ran 200 tests like this: in each test I sorted positive ints from 1 to 10^9 (totally ≤ 50000 ints) the time for sorting 200 arrays was 428 ms. // Need for speed.

The trick for the main part is in using the following recursion that goes from the left to the right column.

For each column we divide the set of rows in two parts:



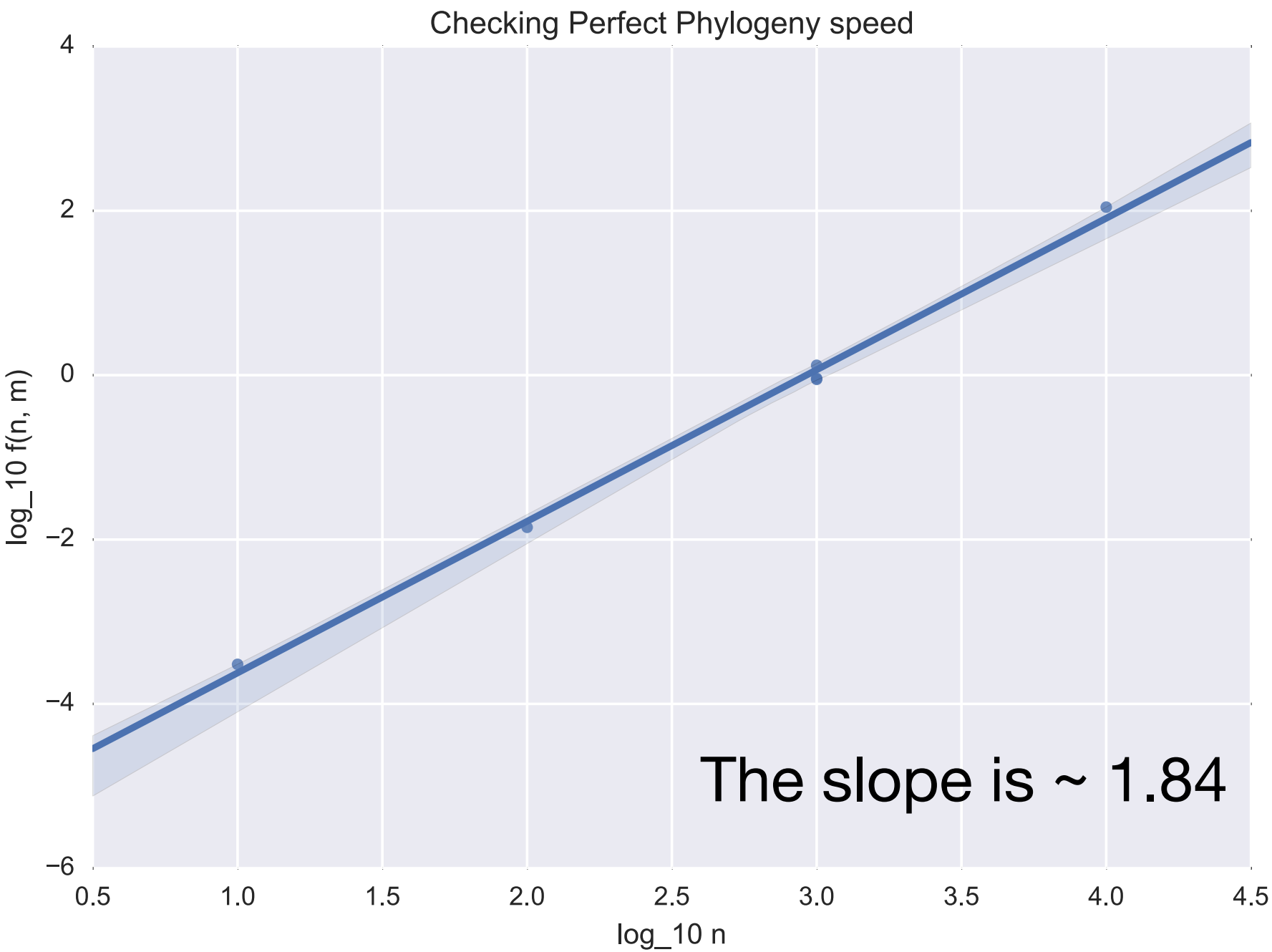
and call the same function recursively from each subtree with the next column.

The total # of recursive calls is $\leq nm$. Moreover, each cell of the matrix is only considered once (in all recursive calls) as we move from the left to the right.

We also have an array with seen mutations. The flag is turned after a specific mutation was considered. If perf. ph. exists each mutation occurs no more than once (after the sorting). So if we try to sign the bit that is already signed → p.p. does not exist.

The plot is on page 7. The slope is 1.8439 in my case

```
import sys
import time

import numpy as np
import seaborn as sns
from scipy import stats
import matplotlib.pylab as plt


sys.setrecursionlimit(100000)


class PerfTree(object):
    def __init__(self, m):
        self.m = np.array(m)
        bin_ns = []
        for i in range(self.m.shape[0]):
            bin_ns.append(''.join(self.m[:, i]))
        for i in range(self.m.shape[1]):
            bin_ns_0, bin_ns_1 = [], []
            for bin_n in bin_ns:
                if bin_n[i] == '1':
                    bin_ns_0.append(bin_n)
                else:
                    bin_ns_1.append(bin_n)
            bin_ns = bin_ns_0 + bin_ns_1
        self.m = np.array([list(x) for x in bin_ns]).T.astype(int)
        self.mut_ind = [0] * self.m.shape[1]

    def Size(self):
        return self.m.shape

    def CheckPerf(self, r_i=None, c_i=0):
        if r_i is None:
            r_i = range(self.m.shape[0])

        if c_i == self.m.shape[1]:
            return True

        r_i_0, r_i_1 = [], []
        for r in r_i:
            if self.m[r, c_i] == 0:
                r_i_0.append(r)
            else:
                r_i_1.append(r)

        if len(r_i_1) and self.mut_ind[c_i]:
            return False
        if len(r_i_1):
            self.mut_ind[c_i] = 1

        res1 = self.CheckPerf(r_i_1, c_i + 1) if len(r_i_1) else
True
        if res1 is False:
            return False
        return self.CheckPerf(r_i_0, c_i + 1) if len(r_i_0) else
True


pattern = 'a1data$.txt'
files = []
for i in range(1, 7):
    files.append(pattern.replace('$', str(i)))

sizes, times = [], []
for f in files:
    print(f)
    with open(f, 'r') as f:
        m = [list(x.strip()) for x in f.readlines()]

    starttime = time.process_time()
    tree = PerfTree(m)
    print(tree.CheckPerf())
    endtime = time.process_time()
    sizes.append(tree.Size()[0])
    times.append(endtime - starttime)
    print(sizes[-1], times[-1])

sizes, times = np.array(sizes), np.array(times)
sizes, times = np.log10(sizes), np.log10(times)
ax = sns.regplot(sizes, times)
ax.set(xlabel='log_10 n', ylabel='log_10 f(n, m)',
       title='Checking Perfect Phylogeny speed')
plt.savefig('3_speed.pdf', format='pdf')
slope, intercept, r_value, p_value, std_err =
stats.linregress(sizes, times)
print(slope, intercept, r_value, p_value, std_err)
```


Checking Perfect Phylogeny speed
log_10 f(n, m)
log_10 n
4
2
0
−2
−4
−6
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
The slope is ~ 1.84

CSE 280
Assignment 1
Andrey Batkadze
A53239410
List 8

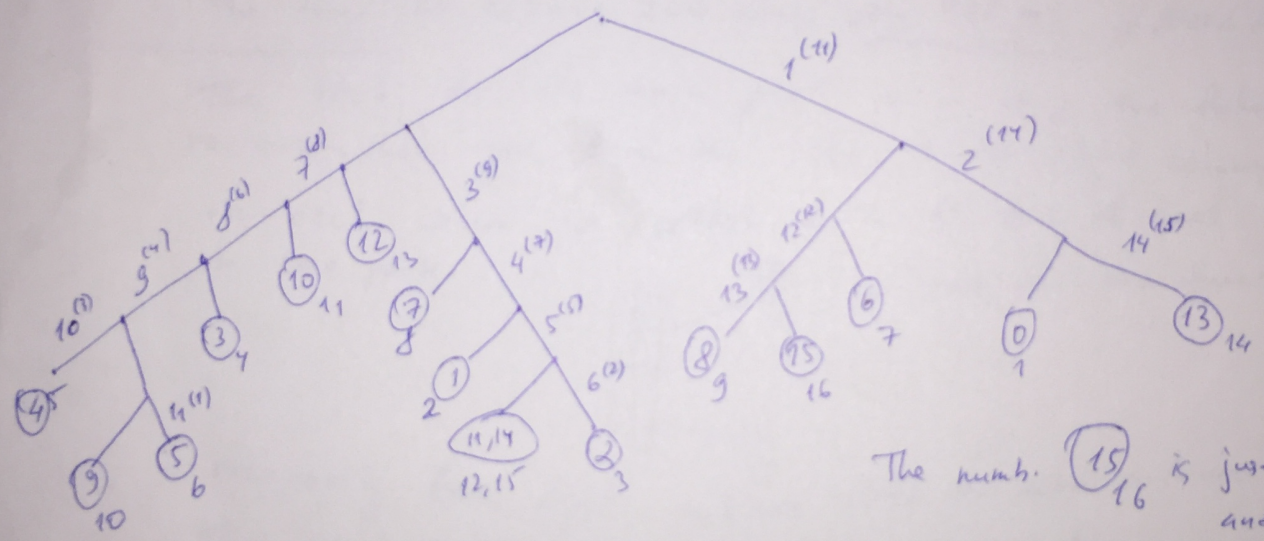
⑥ I'm not sure if we needed to construct a rooted or an unrooted tree. As we don't know that all zeros is an ancestor in this task I've decided to construct an unrooted version.

I sorted columns (after inverting 11-th and 17th):

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	14	9	7	5	2	8	6	4	3	1	12	13	15	10

↑ nobody has

And build the following tree: (rooted variant + original numbering)



The numb. 15/16 is just zero-based and one-based numeration.

My best guess to the migration routes is on page 9.

The 12 and 15 are impossible to distinguish from the SNP matrix given