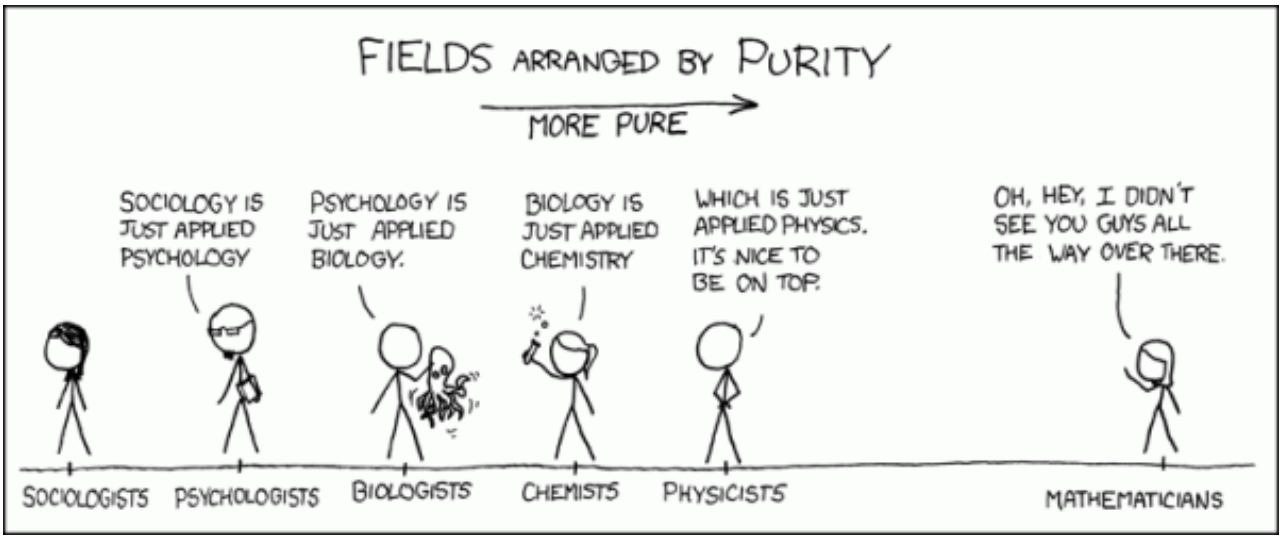
MATH SCRAPBOOK

Uwe Hoffmann

UWE HOFFMANN

MATH SCRAPBOOK

NOTES AND SOLVED PROBLEMS

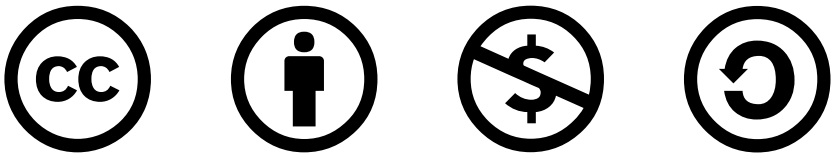


http://xkcd.com/435/

Formatted with LaTeX using the https://tufte-latex.github.io/tufte-latex/ template.

xkcd comics http://xkcd.com, used under CC license.

UWE@CODEMANIC.COM



Version August 2022

Dedicated to my family, in appreciation of their love and support

Preface

Collection of math/cs notes and problems written up over the years for my own amusement. Topics are at the undergraduate college level. Normally notes like these would be written with a pencil in a notebook. LaTeX makes it very easy to produce publishing quality typesetting of mathematical texts, so blame LaTeX for this book. Hopefully others will find some of these notes useful.

I'm an amateur but that hasn't prevented me from enjoying writing these notes just as not being Michael Jordan has never prevented me from enjoying pickup basketball. Describing and explaining has helped me understand the topics involved. Errors and misunderstandings are solely my fault. There is no original content in these notes and I tried to be complete about attributions and citations but if I missed something, I apologize.

Code snippets are mostly in Haskell and Mathematica. I'm not an expert in either but they should work.

Contents

1
Airplane Seating

Problem

A line of n airline passengers is waiting to board a plane. They each hold a ticket to one of the n seats on that flight. (For convenience, let's say that the ith passenger in line has a ticket for the seat number i.) Unfortunately, the first person in line is crazy, and will ignore the seat number on their ticket, picking a random seat to occupy. All of the other passengers are quite normal, and will go to their proper seat unless it is already occupied. If it is occupied they will then find a free seat to sit in, at random. What is the probability that the last (nth) person to board the plane will sit in their proper seat (#n)?

Any seat arrangement under the rules of the problem is a permutation π from the set S_n of permutations of size n. Let's define $A_n \subseteq S_n$ the subset of permutations of size n that are valid seat arrangements.

Let $B_n := \{\pi \in A_n : \pi(n) = n\}$ be the subset of A_n where the last person gets their proper seat. A strategy to solve the problem would be to count $|A_n|$ and $|B_n|$ and then divide them up to get the probability.

We will use the permutation cycle notation $(i_1, i_2, \ldots, i_k)$ for a cycle of length k that maps $i_1 \mapsto i_2 \mapsto \ldots i_k \mapsto i_1$. Also let ι_n be the identity permutation in S_n and let $A_n^* = A_n \setminus \{\iota_n\}$ and $B_n^* = B_n \setminus \{\iota_n\}$.

Let's characterize permutations in A_n^*.

Lemma 1.1. *A permutation $\pi \in A_n^*$ is a cycle of the form*

$$\pi = (1, i_1, i_2, \ldots, i_k) \text{ with } 2 \leq i_1 < i_2 < \ldots < i_k \leq n$$

Proof. Consider $\pi \in A_n^*$. Suppose $\pi(1) = 1$ then under the rules of the problem all other passengers can occupy their seat and $\pi = \iota_n$ which is a contradiction because A_n^* doesn't have the identity permutation.

So there exists a $i_1 \in \{2, \ldots, n\}$ with $\pi(1) = i_1$. i_1 cannot map to any $j < i_1$ because under the rules of the problem every $j < i_1$ maps to itself (every $j < i_1$ finds their seat unoccupied so they take it). So there exists a $i_2 \in \{2, \ldots, n\}$ with $i_2 > i_1$ and $i_1 \mapsto i_2$. And so on. This means that π has at least the cycle $(1, i_1, i_2, \ldots, i_k)$ with $2 \leq i_1 < i_2 < \ldots < i_k \leq n$. It cannot have any other cycles that don't have 1 in them because under the rules of the problem only passenger 1 can start a seat rearrangement and all passengers not affected by that rearrangement will occupy their seat.

□

Definition 1.2. Let $2^{\{2,\ldots,n\}}$ be the set of all subsets of $\{2, \ldots, n\}$. The function $\varphi : 2^{\{2,\ldots,n\}} \to S_n$ is defined as:

$$\begin{aligned} \varphi(\oslash) &= \iota_n \\ \varphi(\{i_1, i_2, \ldots, i_k\}) &= (1, i_1, i_2, \ldots, i_k) \\ &\text{assuming } 2 \leq i_1 < i_2 < \ldots < i_k \leq n \end{aligned}$$

φ is a valid function because for each subset there is only one cycle possible with the monotonically increasing ordering. From lemma 1.1 it then follows that $\varphi(2^{\{2,\ldots,n\}}) = A_n$, so $|A_n| = 2^{n-1}$.

For B_n we apply the same arguments, except we take out the n-th passenger. A permutation $\pi' \in B_n^*$ is a cycle of the form

$$\pi' = (1, i_1, i_2, \ldots, i_k) \text{ with } 2 \leq i_1 < i_2 < \ldots < i_k \leq n-1$$

and there is a function φ' defined as

$$\begin{aligned} \varphi'(\oslash) &= \iota_n \\ \varphi'(\{i_1, i_2, \ldots, i_k\}) &= (1, i_1, i_2, \ldots, i_k) \\ &\text{assuming } 2 \leq i_1 < i_2 < \ldots < i_k \leq n-1 \end{aligned}$$

that defines a bijection from $2^{\{2,\ldots,n-1\}}$ to B_n. It means that $|B_n| = 2^{n-2}$ for $n \geq 2$.

So the probability that the last (nth) person to board the plane will sit in their proper seat is $\frac{|B_n|}{|A_n|} = 0.5$ for $n \geq 2$.

2
Schröder-Bernstein Theorem

BIJECTIONS from one-to-one functions are the topic[1] in this note. The problem statement is known as the Schröder-Bernstein Theorem.

[1] Exercise 1.5.11 on page 32 from Stephen Abbott. *Understanding Analysis*. Springer, 2 edition, 2015. ISBN 978-1-4939-2711-1.

Problem

Let $f : X \to Y$ and $g : Y \to X$ be one-to-one functions. Then there exists a bijection $h : X \to Y$.

The given functions are one-to-one, so for subsets $f(X)$ and $g(Y)$ they are already bijections. This leads to the idea of partitioning X and Y such that we can compose a bijection h piece-wise from f and g^{-1} using the partitions. In particular given a subset $A \subseteq X$, we consider the sets A, $X \setminus A$, $f(A)$, $Y \setminus f(A)$ and $g(Y \setminus f(A))$. We want subsets $A \subseteq X$, such that $A \cap g(Y \setminus f(A)) = \emptyset$, as shown in figure 2.2. Let's define this as property P:

$$\forall A \subseteq X : P(A) \Leftrightarrow A \cap g(Y \setminus f(A)) = \emptyset$$

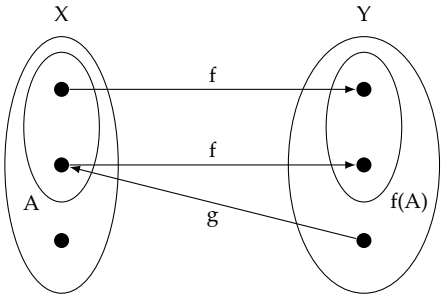


Figure 2.1: A violates $P(A)$

If we have a subset $A \subseteq X$ that satisfies $P(A)$, then we can define the bijection h:

$$h(x) = \begin{cases} f(x) & : x \in A \\ g^{-1}(x) & : x \in g(Y \setminus f(A)) \end{cases}$$

The domain of h is $A \cup g(Y \setminus f(A))$, which is not necessarily equal to X, so we are not done yet. Our goal therefore is to find a subset $A \subseteq X$ that satisfies $P(A)$ and for which $A \cup g(Y \setminus f(A)) = X$.

Let

$$\Lambda = \{A \subseteq X : P(A)\}$$

be the set of all subsets of X that satisfy property P and let $\bar{A}$ be the

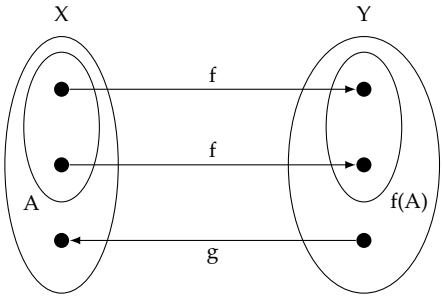


Figure 2.2: A satisfies $P(A)$

union of all such subsets

$$\bar{A} = \bigcup_{A \in \Lambda} A$$

Lemma 2.1. *$\bar{A}$ is the biggest subset of X that satisfies P.*

Proof. First we show that $\bar{A}$ satisfies P. Assume

$$\exists y \in Y \setminus f(\bar{A}) \text{ with } g(y) \in \bar{A}$$

Then there exists a set $A \in \Lambda$ with $g(y) \in A$ [2]. $A \subseteq \bar{A}$, so $f(A) \subseteq f(\bar{A})$. Therefore $Y \setminus f(\bar{A}) \subseteq Y \setminus f(A)$, so $y \in Y \setminus f(A)$. But this contradicts A satisfying property P, so no such y exists. It follows that $\bar{A}$ satisfies P too.

Assume there is a set A' that satisfies P and that is bigger than $\bar{A}$, so $\bar{A} \subseteq A'$. But $A' \in \Lambda$ and $\bar{A} = \bigcup_{A \in \Lambda}$, so $A' \subseteq \bar{A}$. That means $A' = \bar{A}$. □

[2] Because $\bar{A} = \bigcup_{A \in \Lambda}$.

With $\bar{A}$ we can define the partitions $X = \bar{A} \oplus (X \setminus \bar{A})$ and $Y = f(\bar{A}) \oplus (Y \setminus f(\bar{A}))$.

Lemma 2.2.

$$g(Y \setminus f(\bar{A})) = X \setminus \bar{A}$$

Proof. Because $\bar{A}$ satisfies P, we already know that

$$g(Y \setminus f(\bar{A})) \subseteq X \setminus \bar{A}$$

Now assume

$$\exists x \in X \setminus \bar{A} \text{ such that } \forall y \in Y \setminus f(\bar{A}) : g(y) \neq x$$

But then $\bar{A} \cup \{x\}$ satisfies P [3] and is bigger than $\bar{A}$. This contradicts lemma 2.1. So no such x exists and the lemma is proven. □

[3] We have

$$Y \setminus f(\bar{A} \cup \{x\}) \subseteq Y \setminus f(\bar{A})$$

so

$$\forall y \in Y \setminus f(\bar{A} \cup \{x\}) : g(y) \notin \bar{A} \cup \{x\}$$

We can now define the bjection $h : X \to Y$ with

$$h(x) = \begin{cases} f(x) & : x \in \bar{A} \\ g^{-1}(x) & : x \in X \setminus \bar{A} \end{cases}$$

which solves the problem in this section. [4]

[4] The solution uses a nifty proof strategy: maximize a mathematical structure so that its "complement" has no choice but to satisfy a certain property, ie not satisfying the property would contradict the maximality.

3
Bridge Crossings

Problem

Four people begin on the same side of a bridge. You must send them across to the other side in the fastest time possible. It is night. There is one flashlight. A maximum of two people can cross at a time. Any party who crosses, either one or two people, must have the flashlight to see. The flashlight must be walked back and forth, it cannot be thrown, etc. Each person walks at a different speed. A pair must walk together at the rate of the slower person's pace, based on this information: Person 1 takes $t_1 = 1$ minutes to cross, and the other persons take $t_2 = 2$ minutes, $t_3 = 5$ minutes, and $t_4 = 10$ minutes to cross, respectively.

Günter Rote[1] gives a very elegant solution to this puzzle.

[1] Günter Rote. *Crossing the Bridge at Night*. World Wide Web, http://page.mi.fu-berlin.de/~rote/Papers/pdf/Crossing+the+bridge+at+night.pdf, 2002

How many ways are there to let n people cross the bridge under the rules of the original puzzle ?

There are $\binom{n}{2}$ ways to send the first pair over to the other side, there are 2 ways to send the flashlight back with somebody from that side. Now there are $\binom{n-1}{2}$ ways to send the next pair over to the other side from the remaining $n-1$ people on this side and then there are 3 ways to send the flashlight back with somebody from that side etc.

Using the basic product counting principle from combinatorics we get the number of ways P to let n people cross the bridge

$$\begin{aligned} P &= \binom{n}{2} 2 \binom{n-1}{2} 3 \binom{n-2}{2} 4 \dots (n-1) \binom{2}{2} \\ &= (n-1)! \prod_{k=0}^{n-2} \binom{n-k}{2} \end{aligned} \tag{3.1}$$

Taking the product from (3.1) and using the definition of a binomial coefficient we get:

$$\prod_{k=0}^{n-2} \binom{n-k}{2} = \prod_{k=0}^{n-2} \frac{(n-k)!}{2!(n-k-2)!} \tag{3.2}$$

With:

$$\begin{aligned} P_k &= \frac{(n-k)!}{2!(n-k-2)!} \quad \text{and} \\ p_k &= (n-k)! \end{aligned} \tag{3.3}$$

we get:

$$P_k = \frac{p_k}{2!p_{k+2}} \tag{3.4}$$

The product of these P_k can now be simplified to:

$$\begin{aligned} \prod_{k=0}^{n-2} P_k &= \prod_{k=0}^{n-2} \frac{(n-k)!}{2!(n-k-2)!} \\ &= \frac{1}{(2!)^{n-1}} \prod_{k=0}^{n-2} \frac{p_k}{p_{k+2}} \\ &= \frac{1}{2^{n-1}} \frac{p_0}{p_2} \frac{p_1}{p_3} \frac{p_2}{p_4} \cdots \frac{p_{n-3}}{p_{n-1}} \frac{p_{n-2}}{p_n} \\ &= \frac{1}{2^{n-1}} \frac{p_0 p_1}{p_{n-1} p_n} \\ &= \frac{1}{2^{n-1}} n!(n-1)! \end{aligned} \tag{3.5}$$

Using (3.5) we get the solution

$$P = \frac{n!((n-1)!)^2}{2^{n-1}} \tag{3.6}$$

For four people this comes to an astonishing 108 ways to cross the bridge under the rules of the puzzle.

Generating the ways

This section shows a small Haskell program that generates all the possible ways to cross the bridge. It has a helper function *pairs* that generates a list of all possible pairs from a set. It then defines two mutually recursive functions *bridgecrossleft* and *bridgecrossright* for crossing the bridge from the left side as pairs and for a flashlight carrier coming back from the right. The functions pass along the states on the left bank *lbs* and the right bank *rbs*. They generate all possible crossings in their respective direction given the current state. For pairs crossing from the left tuples have the respective pair and for people coming back from the right tuples have the same person in both positions of the tuple. The functions collect the resulting combinations in a list of lists of tuples *rs* (Fig. 3.1). *bridgecross* is the main function taking a list

and calling *bridgecrossleft* because we start on the left with all possible ways of crossing of the first pair.

Calling *bridgecross [1, 2, 3]* we get this result:

```
[
  [(1,2),(1,1),(1,3)],
  [(1,2),(2,2),(2,3)],
  [(1,3),(1,1),(1,2)],
  [(1,3),(3,3),(3,2)],
  [(2,3),(2,2),(2,1)],
  [(2,3),(3,3),(3,1)]
]
```

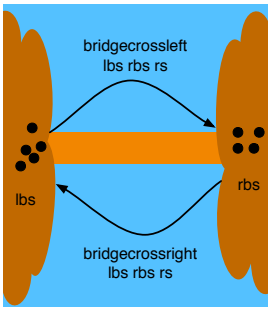


Figure 3.1: Two mutually recursive functions *bridgecrossleft* and *bridgecrossright*.

Listing 3.1: Haskell code

```
pairs :: [a] -> [(a, a)]

pairs xs = let
              rmap :: (a -> [a] -> [b]) -> [a] -> [b]

              rmap f (x:xs) = (f x xs) ++ (rmap f xs)

              rmap f [] = []

              mpairs :: (a -> [a] -> [(a, a)])

              mpairs x xs = map (\y -> (x, y)) xs

           in rmap mpairs xs

bridgecrossleft :: [Int] -> [Int] -> [(Int,Int)]
                                  -> [[(Int,Int)]]

bridgecrossleft lbs rbs rs
        = if (length lbs) >= 2 then
                let
                   ps = pairs lbs
                   f = (\(x,y) ->
                       (bridgecrossright
                          (filter (\z -> (z /= x)
                                      && (z /= y)) lbs)
                            (x:y:rbs) (rs ++ [(x, y)])))
                in foldl (++) [] (map f ps)
          else [rs]

bridgecrossright :: [Int] -> [Int] -> [(Int,Int)]
                                   -> [[(Int,Int)]]

bridgecrossright lbs rbs rs
         = if (length lbs) > 0 then
                let
                   f = (\x ->
                        (bridgecrossleft (x:lbs)
                           (filter (\z -> (z /= x)) rbs)
                               (rs ++ [(x, x)])))
                in foldl (++) [] (map f rbs)
            else [rs]

bridgecross :: [Int] -> [[(Int, Int)]]

bridgecross xs = bridgecrossleft xs [] []
```

4
Cat vs Dog

Bipartite graphs, network flows, matchings and vertex covers are the topics of the problem [1] in this note.

[1] Spotify. Cat vs dog. 2012. URL https://labs.spotify.com/puzzles/

Problem

The latest reality show has hit the TV: "Cat vs. Dog". In this show, a bunch of cats and dogs compete for the very prestigious Best Pet Ever title. In each episode, the cats and dogs get to show themselves off, after which the viewers vote on which pets should stay and which should be forced to leave the show.

Each viewer gets to cast a vote on two things: one pet which should be kept on the show, and one pet which should be thrown out. Also, based on the universal fact that everyone is either a cat lover (i.e. a dog hater) or a dog lover (i.e. a cat hater), it has been decided that each vote must name exactly one cat and exactly one dog.

Ingenious as they are, the producers have decided to use an advancement procedure which guarantees that as many viewers as possible will continue watching the show: the pets that get to stay will be chosen so as to maximize the number of viewers who get both their opinions satisfied. Calculate this maximum number of satisfied viewers.

At first glance this looks similar to a SAT problem [2], something like $(c_1 \wedge \neg d_3)$, $(c_3 \wedge \neg d_1)$, $(d_2 \wedge \neg c_2)$, ... where c_i are the cats and d_j are the dogs. The goal would be to pick the biggest subset of boolean expressions (votes) that are satisfied.

[2] Boolean satisfiability problem http://en.wikipedia.org/wiki/Boolean_satisfiability_problem

But SAT is about one boolean expression and about assigning values to boolean variables to satisfy it. Seems like SAT is fundamentally different and not a good approach in solving this problem. What if we want to visualize the boolean expressions and see the relationships between them, i.e. which ones are in conflict. Conflict between two boolean expressions means one expression has c_i and the other ex-

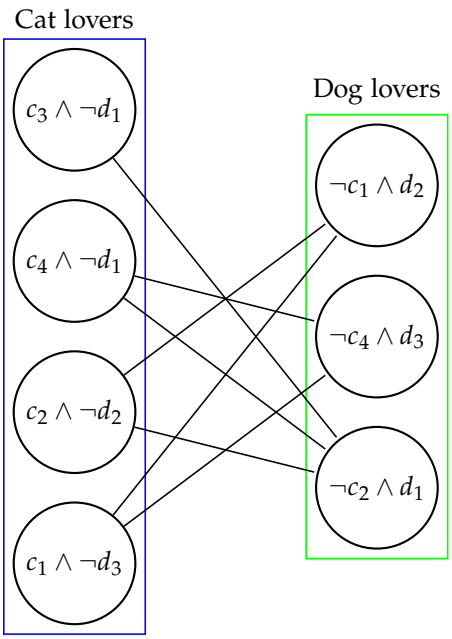


Figure 4.1: Votes form a bipartite graph. A graph is **bipartite** if the vertex set is partitioned into two subsets (blue and green in this case) such that no vertices in a subset are adjacent.

pression has $\neg c_i$ or one has d_j and the other $\neg d_j$. A good way to do that is with a graph as in Figure 4.1. The nodes in the graph are the boolean expressions and edges connect boolean expressions that are in conflict.

It becomes apparent that the graph is bipartite with cat lovers on one side and dog lovers on the other. That is good because a lot of graph algorithms are much simpler and work faster if the graphs are bipartite. But what algorithm should we use? We need to find the biggest subset of nodes in the graph that are not in conflict.

Sometimes it's easier to compute the complement of what we want: the smallest subset of nodes that are involved in conflicts. Removing these nodes and the edges they touch should leave us with a graph with only nodes and no edges, i.e. only votes without conflicts. Because we strive to remove the smallest subset of conflicting nodes we are left with the biggest subset of votes without conflicts.

The subset of nodes that are involved in conflicts is a vertex cover [3] for our bipartite graph.

We need to compute a minimum vertex cover. This will be a good excuse to learn about network flows in graphs, maximum flows and minimum cuts. This delightful detour will eventually bring us to maximum matchings[4] and then finally to minimum vertex covers.

We begin with **network flows** in graphs. We work with a directed graph $G = (V, E)$ that has two special vertices s and t called **source** and **target**. No edge goes into *source* and no edge comes out of *target*.

[3] A **vertex cover** is a subset of nodes such that each edge in a graph is incident to at least one vertex in the subset.

[4] A **matching** is a subset of edges such that no two edges in the subset share a vertex.

We also have a function $c : E \to \mathbb{R}_{\geq 0}$ that assigns a non-negative capacity to each edge. The graph G together with source s and target t and capacity function c form a **network** $(G = (V, E), s \in V, t \in V, c)$.

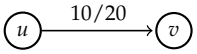


Figure 4.2: In figures we annotate an edge with flow and capacity as shown here. In this case $f(u \to v) = 10$ and $c(u \to v) = 20$. If only one number is annotating the edge then it's the capacity.

Definition 4.1. A function $f : E \to \mathbb{R}_{\geq 0}$ is a **flow** through network (G, s, t, c) if f satisfies the following constraints:

- *capacity constraint*: flow along an edge cannot exceed the capacity of the edge
$$\forall e \in E : f(e) \leq c(e)$$
- *conservation constraint*: incoming flow into a vertex (except for source and target) equals outgoing flow from the vertex
$$\forall v \in V \setminus \{s, t\} : \sum_u f(u \to v) = \sum_w f(v \to w)$$

Source s generates flow and target t consumes flow. The **value** of flow f, denoted $|f|$, is defined as

$$|f| = \sum_w f(s \to w) = \sum_v f(v \to t)$$

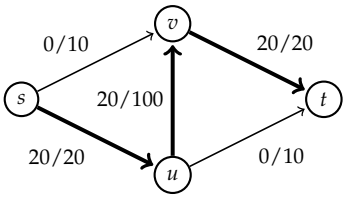


Figure 4.3: Example network flow. Here $|f| = 20$ and the whole flow is pumped along the path $s \to u \to v \to t$. In this exampe f **saturates** $s \to u$ and $v \to t$ and **avoids** $s \to v$ and $u \to t$.

Given a network (G, s, t, c) what is the maximum flow value that can be pumped through it? Figure 4.3 shows a flow of value 20 through an example network. It saturates the flow along one particular path and avoids the other edges. Is 20 the maximum flow value that can be achieved for this example network? Figure 4.4 shows the same network but now with a flow of value 30. Can we do better than 30? The answer is no, because that would exceed the outgoing capacity of source s or the incoming capacity of t.

For notational simplicity we assume functions f and c are defined on $V \times V$ and $f(u \to v) = c(u \to v) = 0$ if $u \to v$ is not an edge in $G = (V, E)$.

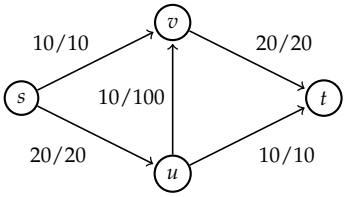


Figure 4.4: Same example network with a flow of value $|f| = 30$.

Our goal is to device an algorithm that constructs a flow with maximum value through a given network. To gauge the progress of our algorithm we need an upper bound for the maximum flow value. As said before the maximum value clearly cannot exceed the outgoing capacity of source s or the incoming capacity of t. But more generally if we sever the ties between source and target along some subset of edges such that there are no more paths from source to target then the maximum flow value cannot exceed the capacity of the cut. This seems like a useful concept to formalize.

Definition 4.2. In a network (G, s, t, c) a **cut** is a partition of the vertex set V into two subsets S and T, such that $V = S \cup T$, $S \cap T = \emptyset$ and $s \in S, t \in T$. The **capacity** of the cut (S, T), denoted $\|S, T\|$, is defined as

$$\|S, T\| = \sum_{v \in S} \sum_{w \in T} c(v \to w)$$

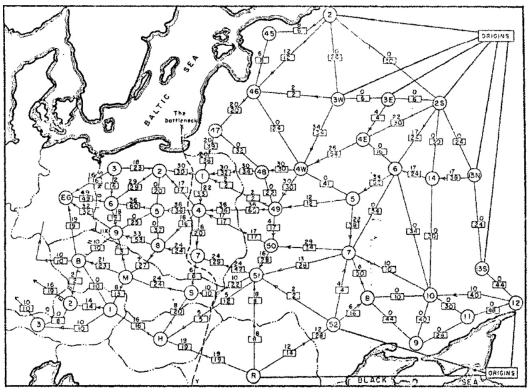


Figure 4.5: Alexander Schrijver. On the history of the transportation and maximum flow problems. 2002. URL http://homepages.cwi.nl/~lex/files/histtrpclean.pdf:
Network flows and minimum cuts played a role in the Cold War. The figure is a schematic diagram of the railway network of the Western Soviet Union and Eastern European countries, with a maximum flow of value 163,000 tons from Russia to Eastern Europe, and a cut of capacity 163,000 tons indicated as "The bottleneck".

Theorem 4.3. *With network (G, s, t, c), for any flow f and any cut (S, T) we have*

$$|f| \leq \|S, T\|$$

Furthermore equality holds if and only if f saturates every edge from S to T and avoids every edge from T to S.

Proof.

$$\begin{aligned}
|f| &= \sum_w f(s \to w) && \text{(by definition)} \\
&= \sum_w f(s \to w) - \sum_v f(v \to s) && \text{(second sum terms are all zero)} \\
&= \sum_{u \in S} (\sum_w f(u \to w) - \sum_v f(v \to u)) && \text{(flow conservation constraint)} \\
&= \sum_{u \in S} (\sum_{w \in T} f(u \to w) - \sum_{v \in T} f(v \to u)) && \text{(edges in } S \text{ cancel each other out)} \\
&\leq \sum_{u \in S} \sum_{w \in T} f(u \to w) && \text{(because } f(v \to u) \geq 0) \\
&\leq \sum_{u \in S} \sum_{w \in T} c(u \to w) && \text{(flow capacity constraint)} \\
&= \|S, T\| && \text{(by definition)}
\end{aligned}$$

□

Theorem 4.3 tells us that if we keep increasing a flow and/or decreasing a cut we should eventually meet at a maximum flow that equals a minimum cut. But given a network how do we start? A first valid flow is $\forall e \in E : f(e) = 0$. We could then try a greedy strategy. Starting with source s find the path to t with the biggest capacity[5] and pump as much flow as we can through it as illustrated in Figure 4.3. Unfortunately we are stuck at that point. We cannot pump more flow out of s on $s \to v$ because that would violate flow conservation at v (we are at the maximum outgoing flow at v). The dashed edges in Figure 4.6 show some of our options. On edges where the current flow leaves residual capacity we can pump more and on edges where there

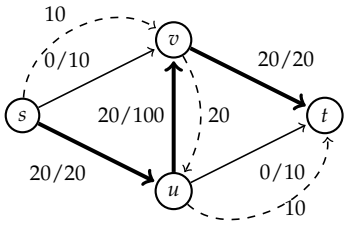


Figure 4.6: Dashed edges show how the greedy $s - t$ path flow can be augmented and reversed in order to increase overall flow.

[5] The capacity of a path is the minimum over the capacities of the edges forming the path.

is existing flow we can reverse it. Again, this concept seems worth formalizing.

Definition 4.4. A flow f in a network (G, s, t, c) induces a **residual network** (G_f, s, t, c_f) with **residual graph** G_f and **residual capacity** c_f in the following way:

- all vertices from G are vertices in G_f, also source s and target t are the same in G and G_f
- if $f(u \to v) > 0$ then G_f has an edge $(v \to u)$ with capacity

$$c_f(v \to u) = f(u \to v)$$

- if $f(u \to v) < c(u \to v)$ then G_f has an edge $(u \to v)$ with capacity

$$c_f(u \to v) = c(u \to v) - f(u \to v)$$

Figure 4.7 shows the residual network of our example network and flow. We observe that there is a simple path[6] $s \to v \to u \to t$ with capacity 10 from source s to target t in the residual graph. This path shows that there still is unused capacity for flow to be pushed from s to t. A simple path from s to t in G_f is called an **augmenting path**.

Theorem 4.5. *Given is a flow f in network (G, s, t, c). If there is an augmenting path in G_f with capacity F then the function $f' : V \times V \to \mathbb{R}_{\geq 0}$ defined as:*

$$f'(u \to v) = \begin{cases} f(u \to v) + F, & \text{if } u \to v \text{ is on the augmenting path} \\ f(u \to v) - F, & \text{if } v \to u \text{ is on the augmenting path} \\ f(u \to v), & \text{otherwise} \end{cases}$$

is a valid flow in network (G, s, t, c) with $|f'| = |f| + F$.

Proof. We need to check the capacity constraint and the conservation constraint.

Let's start with the capacity constraint. The definition of f' has three cases, so we check all three:

- Edge $u \to v$ is on the augmenting path:

$$\begin{aligned} f'(u \to v) &= f(u \to v) + F && \text{(by definition)} \\ &\leq f(u \to v) + c_f(u \to v) && \text{(by definition of } F\text{)} \\ &= f(u \to v) + c(u \to v) - f(u \to v) && \text{(by definition of } c_f\text{)} \\ &= c(u \to v) \end{aligned}$$

[6] A simple path is a path where every vertex on the path is visited only once.

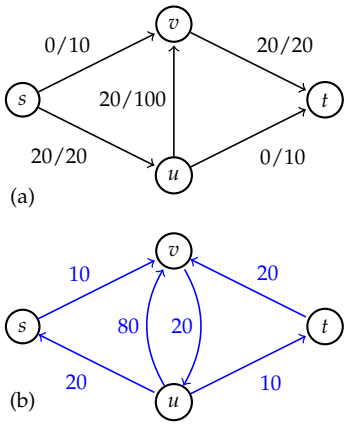


Figure 4.7:
(a) Example network with flow from Figure 4.3.
(b) Residual network (in blue) with edges annotated with their residual capacity.

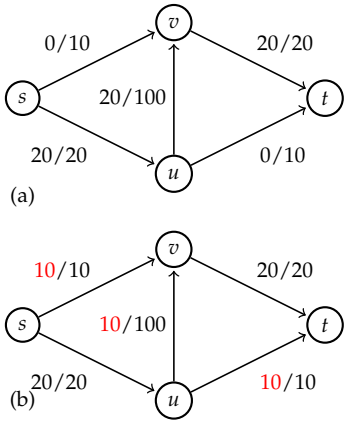


Figure 4.8:
(a) Example network with flow from Figure 4.3.
(b) Augmented flow (changed values in red) from augmenting path $s \to v \to u \to t$.

- Edge $v \to u$ is on the augmenting path:

$$\begin{aligned} f'(u \to v) &= f(u \to v) - F && \text{(by definition)} \\ &\geq f(u \to v) - c_f(u \to v) && \text{(by definition of } F\text{)} \\ &= f(u \to v) - f(u \to v) && \text{(by definition of } c_f\text{)} \\ &= 0 \end{aligned}$$

- Otherwise: In this case the flow of the edge hasn't changed so capacity constraint is satisfied.

Next is the conservation constraint. For vertices not on the augmenting path flow in and out of them hasn't changed, so conservation constraint is satisfied there. For a vertex v on the augmenting path we have four cases (since the augmenting path is simple and $v \neq s, v \neq t$):

- $u \to v$ on augmenting path and $v \to w$ on augmenting path: in this case one incoming edge into v changed by F and one outgoing changed by F, so conservation constraint holds for v
- $u \to v$ on augmenting path and $w \to v$ on augmenting path: in this case two incoming edges into v changed, one by F and the other by $-F$, so conservation constraint holds for v
- $v \to u$ on augmenting path and $w \to v$ on augmenting path: in this case one incoming edge into v changed by $-F$ and one outgoing changed by $-F$, so conservation constraint holds for v
- $v \to u$ on augmenting path and $v \to w$ on augmenting path: in this case two outgoing edges from v changed, one by $-F$ and the other by F, so conservation constraint holds for v

□

What happens when there is no augmenting path in G_f? As the Figure 4.9 hints we then have a maximum flow (in our example $|f| = 30$). The next theorem proves it.

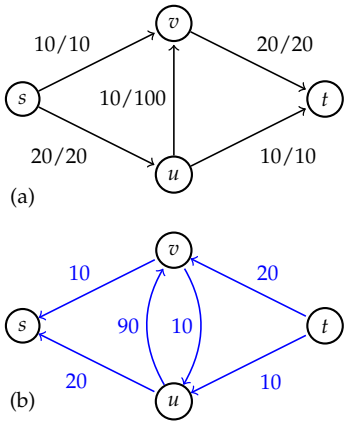


Figure 4.9:
(a) Example network with flow from Figure 4.4.
(b) Residual network (in blue) with edges annotated with their residual capacity.

Theorem 4.6. *Given is a flow f in network (G, s, t, c). If there is no augmenting path in G_f then f is a flow with maximum value.*

Proof. We define two subsets of V. The set S holds all the vertices of V that are reachable from s in G_f. Since there is no augmenting path in G_f we have $t \notin S$. We also define $T = V \setminus S$. Clearly (S, T) is a cut of our network. Also there is no G_f edge $u \to v$ with $u \in S$ and $v \in T$ because otherwise v would be reachable from somewhere in S but $v \notin S$, contradicting the definition of S. This means (by definition

of G_f) that f saturates every edge from S to T and avoids every edge from T to S. According to Theorem 4.3 we then have $|f| = \|S,T\|$ which means we have a maximum flow and minimum cut.

□

We can now piece together the following algorithm known as the **Ford-Fulkerson algorithm**:

Listing 4.1: Ford-Fulkerson algorithm

```
f = zero flow;
Gf = residual graph of f in G;

while (exists augmenting path in Gf):
   pa = choose any augmenting path;
   f = augment f with pa;
   Gf = residual graph of f in G;

return f
```

Delbert Ray Fulkerson was an American mathematician who co-developed the Ford–Fulkerson algorithm. https://en.wikipedia.org/wiki/D._R._Fulkerson

Theorem 4.7. *If the network has capacities in $\mathbb{N}_{\geq 0}$ then the Ford-Fulkerson algorithm terminates and returns the maximum flow in the network.*

Proof. We prove by induction that $f : V \times V \to \mathbb{N}_{\geq 0}$: The base case is the zero flow which is in $\mathbb{N}_{\geq 0}$. Assume the current flow values are in $\mathbb{N}_{\geq 0}$. The augmenting operation adds or subtracts a positive integer value from the current flow values and conforms to capacity constraints, so it keeps the augmented flow values in $\mathbb{N}_{\geq 0}$ which completes the induction.

The augmented flow f' modifies one outgoing edge from s by $F > 0$, so by the definition of the value of a flow we have $|f'| = |f| + F$. This means that augmenting strictly increases the value of the flow. We also know that flow values have an upper bound (by Theorem 4.3 any cut capacity is an upper bound). This means the algorithm has to eventually reach the maximum flow and terminate. □

This concludes our detour into network flows[7].
We should bring it back to our problem and the associated bipartite graph of conflicting votes. We want a minimal vertex cover and we would like to use the just derived Ford-Fulkerson algorithm to compute it. So we first have to transform our undirected bipartite graph into a network.

We have an undirected bipartite graph $G(V = X \cup Y, E)$ with $X \cap Y = \emptyset$ and $E \subseteq X \times Y$ (X could be the votes of cat lovers and Y the votes of dog lovers in our problem or vice versa). We add a source s and a target t and construct a network (G', s, t, c) in the following way:

- vertex set of G' is $X \cup Y \cup \{s, t\}$
- $\forall u \in X$ add a directed edge $s \to u$ into edge set of G'
- $\forall v \in Y$ add a directed edge $v \to t$ into edge set of G'
- $\forall \{u, v\}$ undirected edge in G with $u \in X$ and $v \in Y$ add a directed edge $u \to v$ into edge set of G'
- unit capacity[8]: $\forall (u \to v) \in$ edge set of $G' : c(u \to v) = 1$

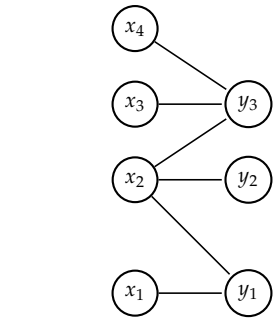
With a network (G', s, t, c) constructed from a bipartite graph G as described above (an example is shown in Figure 4.10) we have an equivalence between a matching in the bipartite graph and a flow in the network. The next theorem states this.

Theorem 4.8. *A matching M in G induces a flow f in G' such that $|f| = |M|$. Conversely a flow in G' induces a matching M in G such that $|M| = |f|$.*

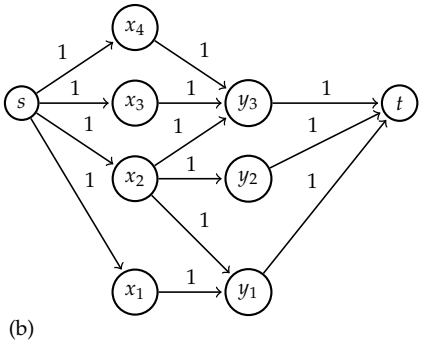
Proof. ($\Rightarrow$) We have a matching M in G, i.e. a subset of edges that don't share a vertex. From the construction of G' it follows that each of the edges in M can be extended to paths from s to t which will only meet in s and t. We define a function f that gives unit values to the edges along these paths and zero value to all other edges. We claim that f is a valid flow. It only assigns zero or unit values so it does satisfy the capacity constraint in G'. The paths don't intersect except in s and t (because M is a matching), so for any vertex along the path there is exactly one incoming edge with unit value and one outgoing edge with unit value. The rest of the edges have value zero so don't play a role in conservation. This then means that the conservation constraint is satisfied also and f is a flow. Each edge in M corresponds to one of the paths, so there are $|M|$ edges outgoing from s that have unit value. Hence $|f| = |M|$.
($\Leftarrow$) We have a flow f in G'. The flow either saturates or avoids an edge. We define the subset M of edges that are saturated by f and are

[7] We have just scratched the surface of the topic on network flows and algorithms computing maximum flows (an area of active research). For example by making smart choices when choosing the augmenting path we can improve the runtime of the algorithm (also we haven't analyzed the runtime). What happens when the capacities are not in $\mathbb{N}_{\geq 0}$. For details on all this and more see:
Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321295358
Jeff Erickson. *Algorithms, Etc*. 2015. URL http://jeffe.cs.illinois.edu/teaching/algorithms/.

[8] Unit capacity has the advantage that a flow either saturates the edge or avoids it.



(a)



(b)

Figure 4.10:
(a) Bipartite graph
(b) Network constructed from it.

between X and Y. We claim that M is a matching in G. Suppose it is not a matching. Then there exists a vertex that is shared by two edges in M. If this vertex is in X then it means it has two outgoing edges of unit value but only one incoming edge of unit value (from s). If this vertex is in Y then it means it has two incoming edges of unit value and only one outgoing edge of unit value (to t). In either case this is a contradiction to the conservation constraint of f. So M has to be a matching. The size of M is by its definition equal to the number of saturated edges from X to Y. But this number has to be equal to the number of edges of unit value going out of s (conservation constraint). Hence $|M| = |f|$. □

Theorem 4.8 let's us use the Ford-Fulkerson algorithm to compute a maximum matching in our bipartite graph G. Once we have a maximum matching we get the size of a minimum vertex cover with the following theorem, known as **König's theorem**[9]:

[9] For a short and elegant proof see: Romeo Rizzi. A short proof of König's matching theorem. *Journal of Graph Theory*, 33(3):138–139, 2000. URL https://math.dartmouth.edu/archive/m38s12/public_html/sources/Rizzi2000.pdf

Theorem 4.9. *In a bipartite graph G the size of a minimum vertex cover C equals the size of a maximum matching M.*

Proof. C is a vertex cover, so it covers all edges, which means it certainly covers a subset M of all edges. But M is a matching, so no two edges share a vertex. It follows that $|C| \geq |M|$.

From the maximum matching M we get the associated maximum flow f as described in Theorem 4.8. The residual graph G'_f of the associated network cannot have any augmenting paths.

We consider the minimum cut (S, T) associated with the maximum flow f. We define the following sets:

- $X_S = X \cap S, X_T = X \cap T$
- $Y_S = Y \cap S, Y_T = Y \cap T$
- $H = \{(u, v) \text{ edge in } G : u \in S, v \in T\}$
- $B = \{v \in Y_T : \exists u \in X_S \text{ with } (u, v) \text{ edge in } G\}$
- $D = X_T \cup Y_S \cup B$

D is a vertex cover: $X_T \subseteq D$ and $Y_S \subseteq D$, so D covers all edges that have endpoints in X_T or Y_S. The set B provides cover for H.

A vertex $u \in X_T$ is not reachable from s in G'_f. It means that f saturates $s \to u$ in G', so the saturated edge $s \to u$ crosses the (S, T) cut and counts towards $\|S, T\|$.

A vertex $v \in Y_S$ is reachable from s in G'_f. It means that f saturates $v \to t$ in G' (otherwise some vertex from T would be reachable from v and also from s in G'_f which is a contradiction). The saturated edge $v \to t$ crosses the (S, T) cut and counts towards $\|S, T\|$.

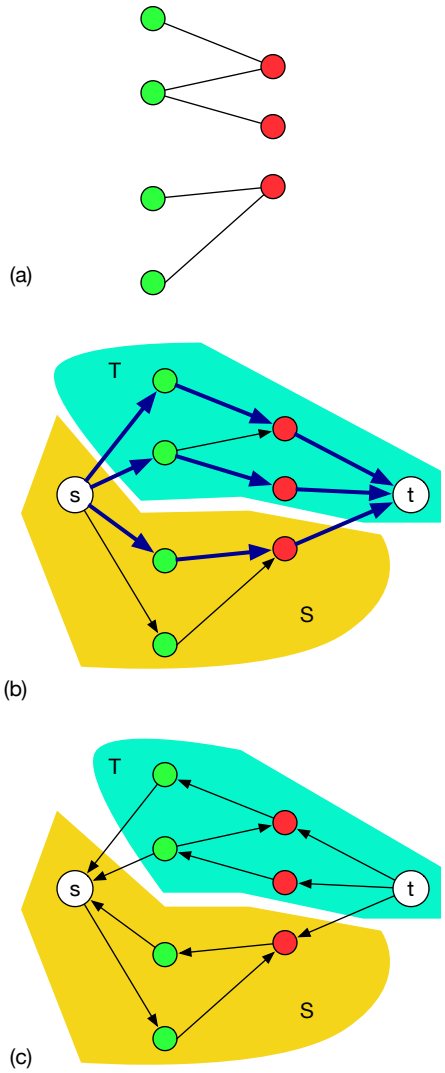


Figure 4.11:
(a) A bipartite graph $G(X \cup Y, E)$. X are green vertices, Y are red vertices.
(b) Maximum flow f (thicker arrows) and minimum cut (S, T) in the corresponding network G'. Thicker arrows between green and red vertices form the maximum matching.
(c) Same cut displayed with the corresponding residual graph G'_f.

f is a maximum flow so any edge from X_S to Y_T is saturated and counts towards $\|S, T\|$.

$$\|S, T\| = |X_T| + |Y_S| + |H|$$

Figure 4.11 shows this. In (b) there are three saturated (thick) arrows crossing the cut. The first two (counting from left to right) are due to X_T and the last one due to Y_S. In this example H is the empty set.

We have

$$|M| = |f| = \|S, T\| = |X_T| + |Y_S| + |H| \geq |X_T| + |Y_S| + |B| \geq |D|$$

D is a vertex cover and C is a minimum vertex cover, so $|D| \geq |C|$. It follows that $|C| \geq |M| \geq |D| \geq |C|$ which means $|C| = |M|$. □

This solves the problem in this note. The number of satisfied viewers is $|V| - |M|$, where V is the set of vertices in the bipartite graph G of votes with their conflicts as edges and M is a maximum matching in G computed with the Ford-Fulkerson algorithm taking advantage of the min-max duality shown in Figure 4.12.

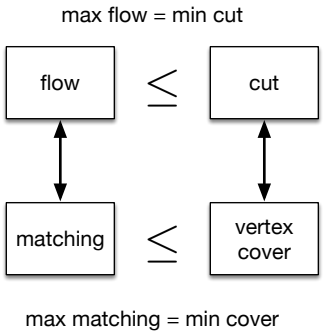


Figure 4.12: Min-max duality in bipartite graphs and corresponding networks.

5
Counting

Countable sets and counting schemes for infinite countable sets are the topics of the problem in this note.

Problem

Let $P = \{N \subset \mathbb{N} : N \text{ finite}\}$. Prove P is countable.

Let's revisit what it means for an infinite set to be countable: An infinite set M is countable if there is a bijection [1] from M to $\mathbb{N}$.

Given this definition, the problem statement is quite remarkable: the set of all the finite subsets of $\mathbb{N}$ is not "bigger" than $\mathbb{N}$.

[1] A bijection is a function that is one-to-one and onto.

Our strategy will be to start smaller and prove certain subsets of P are countable. We then expand it to P. We start by proving that the set of all subsets of $\mathbb{N}$ of size two is countable. We actually will prove something stronger, namely the set of ordered pairs of natural numbers is countable.

Theorem 5.1. *The set of ordered pairs $\mathbb{N} \times \mathbb{N}$ is countable.*

Mention puzzle 136 (Catching a Spy) from Levitin: Algorithmic Puzzles

Proof. We need a bijection from $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. There are many ways to do this [2].

The main idea we are going to use for our bijection is to order the pairs $(i, j) \in \mathbb{N} \times \mathbb{N}$ in rows, such that each pair in a row has the same value when summing the components of the pair. Figure 5.1 illustrates the idea. Row one has all pairs with components that sum up to two (in this case only one pair). Row two has all pairs with components that sum up to three, row three all pairs which sum to four, ... Notice also that in a row the pairs are sorted in increasing order of the first component.

[2] A very elegant way is described at http://www.math.upenn.edu/~wilf/website/recounting.pdf

$(1,1) \longrightarrow$ row 1

$(1,2), (2,1) \longrightarrow$ row 2

$(1,3), (2,2), (3,1) \longrightarrow$ row 3

$(1,4), (2,3), (3,2), (4,1) \longrightarrow$ row 4

...

We count the pairs from left to right in each row and go down the rows starting at the first row. For a given pair (i, j), how many pairs come before it in our counting scheme? It is in row $i + j - 1$, so there are $k : 1 \leq k < i + j - 1$ rows before it. Each row k has k pairs in it. This means there are

$$\sum_{k=1}^{i+j-2} k = \frac{(i+j-2)(i+j-1)}{2}$$

pairs in rows before our pair (i, j). There are $i - 1$ pairs before (i, j) in the same row. Therefore, our counting function is

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}, \quad f(i,j) = i + \frac{(i+j-2)(i+j-1)}{2}$$

Suppose we have two pairs $(i_1, j_1) \neq (i_2, j_2)$. We have two cases:

- $i_1 + j_1 = i_2 + j_2$, same row, then $i_1 \neq i_2$, so $f(i_1, j_1) \neq f(i_2, j_2)$
- $i_1 + j_1 \neq i_2 + j_2$, different rows, so $f(i_1, j_1) \neq f(i_2, j_2)$

This means, f is one-to-one.

To prove that f is onto, we consider an arbitrary $n \in \mathbb{N}$ and find a pair (i, j) with $f(i, j) = n$. Working backwards and assuming we have a pair (i, j) with $f(i, j) = n$, it would fall on a row $r = i + j - 1$. In each row k there are k pairs, n is on row r, so

$$\sum_{k=1}^{r-1} k = \frac{r(r-1)}{2} < n \leq \sum_{k=1}^{r} k = \frac{r(r+1)}{2}$$

Solving for r we have:[3]

[3] Note that $\frac{1+\sqrt{1+8n}}{2} - \frac{-1+\sqrt{1+8n}}{2} = 1$

$$r^2 - r - 2n < 0, \quad r^2 + r - 2n \geq 0, \quad r = \left\lceil \frac{-1 + \sqrt{1+8n}}{2} \right\rceil$$

And then

$$i = n - \frac{r(r-1)}{2}, \quad j = r - i + 1$$

This means that given an arbitrary n, there exists a pair (i, j) with $f(i, j) = n$, so f is onto.

It follows that f is a bijection and $\mathbb{N} \times \mathbb{N}$ is countable. □

A corrollary to Theorem 5.1 let's us expand the countable subsets of P even more.

Corollary. *Set of all finite sequences of length k, $\mathbb{N}^k$ is countable.*[4]

[4] $\mathbb{N}^k$ is the set of sequences of length k, or the cartesian product $\mathbb{N} \times \mathbb{N} \times \ldots \mathbb{N}$. The set of pairs is $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$.

Proof. Follows by induction on k: Assuming $\mathbb{N}^{k-1}$ is countable, then

$$\mathbb{N}^k = \mathbb{N}^{k-1} \times \mathbb{N}$$

is also countable according to Theorem 5.1. □

From the corrollary we now know[5] that the set of all subsets of $\mathbb{N}$ of size k is countable (it's a subset of $\mathbb{N}^k$). The problem in this section asks us to prove that P is countable, which means the union of all these countable sets is countable. The next theorem will prove just that.

[5] We keep using the fact that the set of all finite subsets of $\mathbb{N}$ of size k is a subset of the set of all sequences of size k. To see this impose an order on a set of size k and you get a sequence.

Theorem 5.2. *Let A_n, $n \in \mathbb{N}$ be countable sets. Then*

$$\bigcup_{n=1}^{\infty} A_n$$

is countable[6].

[6] Exercise 1.5.3 on page 30 from Stephen Abbott. *Understanding Analysis*. Springer, 2 edition, 2015. ISBN 978-1-4939-2711-1.

Proof. A_n is countable, so there exists a bijection $f_n : \mathbb{N} \to A_n$. We already know that $\mathbb{N} \times \mathbb{N}$ is countable, so there exists a bijection $g : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$.

We define $F : \mathbb{N} \to \bigcup_{n=1}^{\infty} A_n$

$$F(n) = f_i(j), \text{ where } (i, j) = g(n)$$

We claim that F is a bijection.

Take $n_1 \neq n_2$. Then $g(n_1) \neq g(n_2)$ and $(i_1, j_2) \neq (i_2, j_2)$, so

$$f_{i_1}(j_1) \neq f_{i_2}(j_2)$$

It means $F(n_1) \neq F(n_2)$ and F is one-to-one.

Now pick an arbitrary $a \in \bigcup_{n=1}^{\infty} A_n$. Then there exists $i \in \mathbb{N}$ with $a \in A_i$. [7] There also exists $j \in \mathbb{N}$ with $f_i(j) = a$. The pair (i, j) is in $\mathbb{N} \times \mathbb{N}$, so there exists $n \in \mathbb{N}$ with $g(n) = (i, j)$. It follows that $F(n) = a$ and F is onto. □

[7] We assume here the A_i are disjoint, if not we make them disjoint and their union stays the same.

6
Fibolucci

Exercise 'Fibolucci' in *Programming, The Derivation of Algorithms*[1].

[1] A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990

Problem

Write a program that calculates the function

$$f(n) = \sum_{i=0}^{n} fib(i)fib(n-i), \text{ for } n \geq 0$$

where *fib* is the Fibonacci sequence defined by:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n+2) &= fib(n+1) + fib(n), \text{ for } n \geq 0 \end{aligned}$$

To solve the Fibolucci sum we adopt the same notation used in *Programming in the 1990s*[2]: The notation of function application is the "dot" notation with name of function, followed by arguments, each separated by a dot. The notation of quantified expressions has the operator followed by the bounded variables, then a colon followed by the range for the bounded variables and ended with a colon and the actual expression. So

[2] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

$$(\sum k : i \leq k < j : x_k)$$

corresponds to the more classical mathematical notation $\sum_{k=i}^{j-1} x_k$.

For our derivation steps in predicate calculus we will use the following notation:

$$
\begin{array}{ll}
 & A \\
= & \langle \text{ reason why A equals B } \rangle \\
 & B \\
\leq & \langle \text{ reason why B is less than C } \rangle \\
 & C
\end{array}
$$

We start by finding a recursive expression for f. We will use properties of quantified expressions as covered in Chapter 3 of *Programming in the 1990s*[3]. Since $fib.(0) = 0$ we can use an equivalent definition expression for f:

[3] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

$$f(n) = (\sum i : 1 \leq i < n : fib.i\ fib.(n-i))$$

We derive:

$$
\begin{array}{ll}
 & f.(n+2) \\
= & \langle \text{ definition of } f \rangle \\
 & (\sum i : 1 \leq i < n+2 : fib.i\ fib.(n+2-i)) \\
= & \langle \text{ range split, 1-point rule} \rangle \\
 & (\sum i : 1 \leq i < n+1 : fib.i\ fib.(n+2-i)) + fib.(n+1)\ fib.(1) \\
= & \langle\ fib.(1) = 1 \rangle \\
 & (\sum i : 1 \leq i < n+1 : fib.i\ fib.(n+2-i)) + fib.(n+1) \\
= & \langle \text{ definition of } fib \rangle \\
 & (\sum i : 1 \leq i < n+1 : fib.i\ (fib.(n+1-i) + fib.(n-i))) + fib.(n+1) \\
= & \langle \text{ splitting the term} \rangle \\
 & (\sum i : 1 \leq i < n+1 : fib.i\ fib.(n+1-i)) + \\
 & (\sum i : 1 \leq i < n+1 : fib.i\ fib.(n-i)) + fib.(n+1) \\
= & \langle \text{ definition of } f \rangle \\
 & f.(n+1) + (\sum i : 1 \leq i < n+1 : fib.i\ fib.(n-i)) + fib.(n+1) \\
= & \langle \text{ range split, 1-point rule, } fib.(0) = 0 \rangle \\
 & f.(n+1) + (\sum i : 1 \leq i < n : fib.i\ fib.(n-i)) + fib.(n+1) \\
= & \langle \text{ definition of } f \rangle \\
 & f.(n+1) + f.n + fib.(n+1)
\end{array}
$$

We get the recursive definition of f:

$$
\begin{aligned}
& f.0 = 0 \\
& f.1 = 0 \\
& f.(n+2) = fib.(n+1) + f.(n+1) + f.n, \text{ for } n \geq 0
\end{aligned}
$$

It is straightforward to write a program that computes f from this recursive definition, either iteratively with a loop that step by step computes next values of f starting with $f(2)$ and remembering the last two computed values of f and of *fib* for the next computations, or in Haskell by simply declaring the above recursions for f and *fib*. This will lead to a runtime of $O(n)$. But can we do better than linear ?

Let's look again at the recursive expressions of the two functions involved, leaving out the base cases and computing one additional next value:

$$
\begin{aligned}
f.(n+2) &= \mathit{fib}.(n+1) + f.(n+1) + f.n \\
f.(n+3) &= \mathit{fib}.(n+2) + f.(n+2) + f.(n+1) \\
\mathit{fib}.(n+2) &= \mathit{fib}.(n+1) + \mathit{fib}.n \\
\mathit{fib}.(n+3) &= \mathit{fib}.(n+2) + \mathit{fib}.(n+1)
\end{aligned}
$$

The key observation we can make here is that new values of the two functions are linear combinations of previously computed values. Linear combinations implies linear applications with matrix representations from linear algebra. How many previously computed values, i.e. how far back do we need to go: we need the last computed value last and the value computed before that, so 2 previous values. Looks like we could try something in a linear space of dimension 2.

Let's try first with *fib* which is simpler and doesn't depend on f. We define the function $\mathit{Fib} : \mathbb{N} \to \mathbb{N}^2$ into the two-dimensional space $\mathbb{N}^2$:

$$
\mathit{Fib}.n = \begin{pmatrix} \mathit{fib}.n \\ \mathit{fib}.(n+1) \end{pmatrix}, \text{ for } n \geq 0
$$

For a recursive expression for *Fib* we have:

$$
\begin{aligned}
& \mathit{Fib}.(n+1) \\
= \quad & < \text{definition of } \mathit{Fib} > \\
& \begin{pmatrix} \mathit{fib}.(n+1) \\ \mathit{fib}.(n+2) \end{pmatrix} \\
= \quad & < \text{definition of } \mathit{fib} > \\
& \begin{pmatrix} \mathit{fib}.(n+1) \\ \mathit{fib}.(n+1) + \mathit{fib}.n \end{pmatrix} \\
= \quad & < \text{matrix multiplication} > \\
& \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \mathit{fib}.n \\ \mathit{fib}.(n+1) \end{pmatrix} \\
= \quad & < \text{definition of } \mathit{Fib} > \\
& \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \mathit{Fib}.n
\end{aligned}
$$

So

$$
\mathit{Fib}.(n+1) = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \mathit{Fib}.n = \ldots = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n+1} \mathit{Fib}.0
$$

The same approach can be used for f. We define a function $F : \mathbb{N} \to \mathbb{N}^4$ into the four-dimensional space $\mathbb{N}^4$:

$$F.n = \begin{pmatrix} fib.n \\ fib.(n+1) \\ f.n \\ f.(n+1) \end{pmatrix}, \text{ for } n \geq 0$$

For a recursive expression for F we have:

$$\begin{aligned}
& F.(n+1) \\
= \quad & < \text{definition of } F > \\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+2) \\ f.(n+1) \\ f.(n+2) \end{pmatrix} \\
= \quad & < \text{definitions of } fib \text{ and } f > \\
& \begin{pmatrix} fib.(n+1) \\ fib.(n+1) + fib.n \\ f.(n+1) \\ f.(n+1) + f.n + fib.(n+1) \end{pmatrix} \\
= \quad & < \text{matrix multiplication} > \\
& \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} fib.n \\ fib.(n+1) \\ f.n \\ f.(n+1) \end{pmatrix} \\
= \quad & < \text{definition of } F > \\
& \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} F.n
\end{aligned}$$

and

$$F.(n+1) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} F.n = \ldots = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}^{n+1} F.0$$

Calculating $F.n$ also calculates $f.n$ so if we can calculate $F.n$ faster than linear we also solve the original problem faster than linear. $F.n$ is basically an exponentiation so let's look at the exponentiation function $exp(x, n) = x^n$. The following recursive expression holds for exp:

$$exp.x.n = \begin{cases} exp.(x\ x).(n/2) & \text{if } n = 0 \bmod 2 \\ x\ exp.x.(n-1) & \text{if } n = 1 \bmod 2 \end{cases}$$

At least at every other step in the above recursion n is halved so computing $exp(x, n)$ has $O(log\ n)$ runtime which also implies $O(log\ n)$ runtime for F.

Before we write the actual code for computing F let's first see if we can find a more compact representation for the powers of matrix A involved in the computation:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

We are searching for patterns in the powers of A:

$$A^2 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \end{pmatrix}, \; A^3 = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 1 & 2 & 1 & 2 \\ 2 & 5 & 2 & 3 \end{pmatrix}, \; A^4 = \begin{pmatrix} 2 & 3 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 2 & 5 & 2 & 3 \\ 5 & 10 & 3 & 5 \end{pmatrix}$$

We make the conjecture that A^k for any natural k is of the form:

$$A^k = \begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix}, \text{ for some } a, b, c, d, e, f \in \mathbb{N} \tag{6.1}$$

and prove this by induction. The base case for $k = 1$ is established with values $(0, 1, 0, 0, 0, 1)$ for (a, b, c, d, e, f). Assuming that the conjecture holds for A^k we look at A^{k+1} and get:

$$A^{k+1} = A^k \, A = \begin{pmatrix} b & a+b & 0 & 0 \\ a+b & a+2b & 0 & 0 \\ d & b+c+d & b & a+b \\ f & a+b+e+f & a+b & a+2b \end{pmatrix}$$

so A^{k+1} has the same form as stated in the conjecture if we substitute $(b, a+b, d, b+c+d, f, a+b+e+f)$ for (a, b, c, d, e, f). This proves conjecture (6.1).

It means that in our program we can use a tuple representation (a, b, c, d, e, f) of 6 values instead of the whole 16 values to represent the powers of A. We need to define multiplication in this tuple space consistent with the matrix multiplication:

$$
\begin{aligned}
(a,b,c,d,e,f)(a',b',c',d',e',f') = \\
&(aa' + bb', \\
&\ ab' + b(a' + b'), \\
&\ ca' + db' + ac' + be', \\
&\ cb' + d(a' + b') + ad' + bf', \\
&\ ea' + fb' + bc' + (a + b)e', \\
&\ eb' + f(a' + b') + bd' + (a + b)f')
\end{aligned}
$$

We read this definition off the matrix multiplication:

$$
\begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix}
\begin{pmatrix} a' & b' & 0 & 0 \\ b' & a'+b' & 0 & 0 \\ c' & d' & a' & b' \\ e' & f' & b' & a'+b' \end{pmatrix}
$$

The last expression we need is:

$$
A^n F.0 = \begin{pmatrix} a & b & 0 & 0 \\ b & a+b & 0 & 0 \\ c & d & a & b \\ e & f & b & a+b \end{pmatrix}
\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} =
\begin{pmatrix} b \\ a+b \\ d \\ f \end{pmatrix}
$$

so we are interested in d which corresponds to the $F.n$ coordinate of the vector.

Putting all the pieces together we get the final Haskell program:

Listing 6.1: Haskell code

```haskell
type Tuple6Ints = (Int, Int, Int, Int, Int, Int)

tmul :: Tuple6Ints -> Tuple6Ints -> Tuple6Ints

tmul (a, b, c, d, e, f) (a', b', c', d', e', f') =
    (a * a' + b * b',
     a * b' + b * (a' + b'),
     c * a' + d * b' + a * c' + b * e',
     c * b' + d * (a' + b') + a * d' + b * f',
     e * a' + f* b' + b * c' + (a + b) * e',
     e * b' + f* (a' + b') + b * d' + (a + b) * f')

fibexp :: Tuple6Ints -> Int -> Tuple6Ints

fibexp tuple n | n == 0 = error "undefined"
               | n == 1 = tuple
               | n `mod` 2 == 0 =
                   fibexp (tuple `tmul` tuple)
                          (n `div` 2)
               | n `mod` 2 == 1 =
                   tuple `tmul` (fibexp tuple (n - 1))
               | otherwise = error "wrong input"

fourth :: Tuple6Ints -> Int

fourth (a, b, c, d, e, f) = d

fibolucci :: Int -> Int

fibolucci n | n == 0 = 0
            | otherwise =
                fourth (fibexp (0, 1, 0, 0, 0, 1) n)
```

7
Grasshopper jumping

INDUCTION and integer inequalities are the topics of this note[1].

[1] For an extension to signed jumps see Géza Kós. On the grasshopper problem with signed jumps. *The American Mathematical Monthly*, 118:877–886, 2010. URL https://arxiv.org/abs/1008.2936

Problem

Let $a_1, a_2, \ldots, a_n$ be distinct positive integers and let M be a set of $n-1$ positive integers not containing $s = a_1 + a_2 + \ldots + a_n$. A grasshopper is to jump along the real axis, starting at the point 0 and making n jumps to the right with lengths $a_1, a_2, \ldots, a_n$ in some order. Prove that the order can be chosen in such a way that the grasshopper never lands on any point in M.

We use induction on n and we use the problem as our induction hypothesis with one modification: set M has at most $n-1$ elements.

The base case $n = 2$ is trivial.

Let $A = \{a_i : 1 \le i \le n\}$ and $M = \{m_i : 1 \le i < n\}$. Assume $a_1 < a_2 < \ldots < a_n$ and $m_1 < m_2 < \ldots < m_{n-1}$. For the induction step we have several cases.

Case: $a_n \in M$

There is an $l : 1 \le l < n : m_l = a_n$.

If $l = n-1$: there is an index k for which $a_k \notin M$. Then the order $\{k, n, \ldots\}$ never lands on any point in M because $a_k + a_n > m_{n-1}$.

If $l < n-1$: Define $M' = \{m_1, m_2, \ldots, m_{l-1}\} \cup \{m_{l+1} - a_n, \ldots, m_{n-1} - a_n\}$. Use integers $a_1, \ldots, a_{n-1}$ and M' as induction step to get an order $a_{\pi(1)}, \ldots, a_{\pi(n-1)}$ with $\pi \in S_{n-1}$.

$a_{\pi(1)} \notin M'$ and $a_{\pi(1)} < a_n$, so $a_{\pi(1)} \notin M$.

$a_{\pi(1)} \notin \{m_{l+1} - a_n, \ldots, m_{n-1} - a_n\}$, so $a_{\pi(1)} + a_n \notin \{m_{l+1}, \ldots, m_{n-1}\}$. Also $a_{\pi(1)} + a_n > a_n$ so $a_{\pi(1)} + a_n \notin \{m_1, m_2, \ldots, m_{l-1}\}$. That means $a_{\pi(1)} + a_n \notin M$.

We continue with similar reasoning with the rest: $a_{\pi(1)} + a_n + a_{\pi(2)} \notin M$ because $a_{\pi(1)} + a_{\pi(2)} \notin \{m_{l+1} - a_n, \ldots, m_{n-1} - a_n\}$, so $a_{\pi(1)} + a_n +$

$a_{\pi(2)} \notin \{m_{l+1}, \ldots, m_{n-1}\}$ and $a_{\pi(1)} + a_n + a_{\pi(2)} > a_n$ etc.

This means $\{\pi(1), n, \pi(2), \ldots, \pi(n-1)\}$ is a valid order.

Case: $a_n \notin M$

If there is an $m_i < a_n$ then we can use the induction step with integers $a_1, a_2, \ldots, a_{n-1}$ and set $M' = \{m_{i+1} - a_n, m_{i+2} - a_n, \ldots, m_{n-1} - a_n\}$ to find an order and prepend a_n to that order.

If not, then $\forall\, 1 \leq i < n : m_i > a_n$.

$\sum_{j=1}^{n-1} a_j \geq m_1$ because otherwise we could have used order $\{1, 2, \ldots, n\}$.

We have $a_1 < a_n < m_1$ and $\sum_{j=1}^{n-1} a_j \geq m_1$, so there exists an $1 \leq l < n-1$ such that $s' = \sum_{j=1}^{l} a_j < m_1$.

Define $M' = \{m_2 - a_n, m_3 - a_n, \ldots, m_{n-1} - a_n\}$ and use M' with the integers $a_1, a_2, \ldots, a_{n-1}$ in an induction step which gives us an order $\pi \in S_{n-1}$.

Since $a_{\pi(1)} < m_1$ and $\sum_{j=1}^{n-1} a_{\pi(j)} \geq m_1$ there exists an $1 < l \leq n-1$ such that $\sum_{j=1}^{l-1} a_{\pi(j)} < m_1$ and $\sum_{j=1}^{l} a_{\pi(j)} \geq m_1$.

We look at the order $\{\pi(1), \ldots, \pi(l-1), n, \pi(l), \ldots, \pi(n-1)\}$ and claim it is a valid order.

Indeed $\sum_{j=1}^{l-1} a_{\pi(j)} < m_1$, so jumps $\{\pi(1), \ldots, \pi(l-1)\}$ won't encounter anything from M. We also have

$$\sum_{j=1}^{l-1} a_{\pi(j)} + a_n > \sum_{j=1}^{l} a_{\pi(j)} \geq m_1$$

which means $\{\pi(1), \ldots, \pi(l-1), a_n\}$ will avoid m_1. It will also avoid anything from $M \setminus \{m_1\}$ because $\{\pi(1), \ldots, \pi(l-1)\}$ avoids anything from M'. The rest of the order is already bigger than m_1 and avoids $M \setminus \{m_1\}$ by induction.

8
Groovy numbers

Problem

$x \in \mathbb{R}$ is said to be a groovy number iff $\exists\ n \in \mathbb{N}$ such that $x = \sqrt{n} + \sqrt{n+1}$. Prove that if x is groovy, then $\forall r \in \mathbb{N} :\ x^r$ is groovy.

Binomial Expansion

In this section we explore a property of the binomial power expansion

$$(a+b)^r = \sum_{k=0}^{r} \binom{r}{k} a^{r-k} b^k$$

We define $\mathbb{N}_r = \{k \in \mathbb{N}_0 : 0 \leq k \leq r\}$ and its partition into two subsets $\mathbb{N}_r = \mathbb{E}_r \cup \mathbb{O}_r$, with $\mathbb{E}_r = \{k \in \mathbb{N}_r : k = 2u, u \in \mathbb{N}_0\}$ and $\mathbb{O}_r = \{k \in \mathbb{N}_r : k = 2u+1, u \in \mathbb{N}_0\}$. We then partition the binomial power expansion into two sums:

$$(a+b)^r = \sum_{k=0}^{r} \binom{r}{k} a^{r-k} b^k = \sum_{k \in \mathbb{E}_r} \binom{r}{k} a^{r-k} b^k + \sum_{k \in \mathbb{O}_r} \binom{r}{k} a^{r-k} b^k$$

Let

$$E(a,b,r) = \sum_{k \in \mathbb{E}_r} \binom{r}{k} a^{r-k} b^k \text{ and } O(a,b,r) = \sum_{k \in \mathbb{O}_r} \binom{r}{k} a^{r-k} b^k$$

Then

$$\begin{aligned}(a^2-b^2)^r &= (a+b)^r (a-b)^r \\ &= (E(a,b,r) + O(a,b,r))(E(a,-b,r) + O(a,-b,r))\end{aligned}$$

But

$$E(a,-b,r) = E(a,b,r) \text{ and } O(a,-b,r) = -O(a,b,r)$$

so

$$\begin{aligned}(a^2-b^2)^r &= (a+b)^r(a-b)^r \\ &= (E(a,b,r)+O(a,b,r))(E(a,-b,r)+O(a,-b,r)) \\ &= (E(a,b,r)+O(a,b,r))(E(a,b,r)-O(a,b,r)) \\ &= E(a,b,r)^2 - O(a,b,r)^2\end{aligned}$$

We therefore proved

Lemma 8.1.

$$(a^2-b^2)^r = E(a,b,r)^2 - O(a,b,r)^2$$

Solution

Using lemma 8.1 with $a=\sqrt{n}$ and $b=\sqrt{n+1}$, we get

$$(-1)^r = E(\sqrt{n},\sqrt{n+1},r)^2 - O(\sqrt{n},\sqrt{n+1},r)^2 \qquad \text{(L)}$$

Lemma 8.2.

$$\begin{aligned}&E(\sqrt{n},\sqrt{n+1},r)^2 \in \mathbb{N}, \\ &O(\sqrt{n},\sqrt{n+1},r)^2 \in \mathbb{N}\end{aligned}$$

Proof. We will look at two cases: r even and r odd.
Case 1. For $r=2u$ even we have

$$\begin{aligned}E(\sqrt{n},\sqrt{n+1},2u) &= \sum_{k=0}^{u}\binom{2u}{2k}(\sqrt{n})^{2u-2k}(\sqrt{n+1})^{2k} \\ &= \sum_{k=0}^{u}\binom{2u}{2k}(\sqrt{n})^{2(u-k)}(\sqrt{n+1})^{2k} \\ &= \sum_{k=0}^{u}\binom{2u}{2k}n^{u-k}(n+1)^k\end{aligned}$$

so $E(\sqrt{n},\sqrt{n+1},r)\in\mathbb{N}$, and therefore $E(\sqrt{n},\sqrt{n+1},r)^2\in\mathbb{N}$.

$$
\begin{aligned}
O(\sqrt{n},\sqrt{n+1},2u) &= \sum_{k=0}^{u-1}\binom{2u}{2k+1}(\sqrt{n})^{2u-2k-1}(\sqrt{n+1})^{2k+1} \\
&= \frac{\sqrt{n+1}}{\sqrt{n}}\sum_{k=0}^{u-1}\binom{2u}{2k+1}(\sqrt{n})^{2(u-k)}(\sqrt{n+1})^{2k} \\
&= \frac{\sqrt{n+1}}{\sqrt{n}}\sum_{k=0}^{u-1}\binom{2u}{2k+1}n^{2(u-k)}(n+1)^{k} \\
&= \sqrt{n(n+1)}\sum_{k=0}^{u-1}\binom{2u}{2k+1}n^{2(u-k)-1}(n+1)^{k}
\end{aligned}
$$

so $O(\sqrt{n},\sqrt{n+1},r)^2 \in \mathbb{N}$.
Case 2. $r = 2u+1$ is handled in a similar fashion by factoring out $\sqrt{n}$ and $\sqrt{n+1}$ with the remainder $\in \mathbb{N}$.

□

From lemma 8.2 and equation (L) it follows that $E(\sqrt{n},\sqrt{n+1},r)^2$ and $O(\sqrt{n},\sqrt{n+1},r)^2$ are consecutive natural numbers. Let

$$m = min(E(\sqrt{n},\sqrt{n+1},r)^2, O(\sqrt{n},\sqrt{n+1},r)^2) \in \mathbb{N}$$

Then

$$x^r = (\sqrt{n}+\sqrt{n+1})^r = \sqrt{m}+\sqrt{m+1}$$

9
Devil's chessboard

Hamming codes are used to solve the problem[1] in this note.

You, your friend, and the Devil play a game. You and the Devil are in the room with a chess board with 64 tokens on it, one on each square. Meanwhile, your friend is outside of the room. The token can either be on an up position or a down position, and the difference in position is distinguishable to the eye. The Devil mixes up the positions (up or down) of the tokens on the board and chooses one of the squares and calls it the magic square. Next, you may choose one token on a square and flip its position. Then, your friend comes in and must guess what the magic square was by looking on the squares on the board.[2]

[1] Michael Tong. Devil's chessboard. 2013. URL https://brilliant.org/discussions/thread/the-devils-chessboard/

[2] Details:

1. You **may** flip a token. As in, you are not forced to flip a token; you may choose to not flip a token.
2. You can't just tell your friend what square it is. Or point to it. Or text him it. Or... you get the point.
3. Your friend knows the strategy as well (you tell him beforehand).
4. If you don't get it right, the Devil takes your soul. High stakes.

Problem

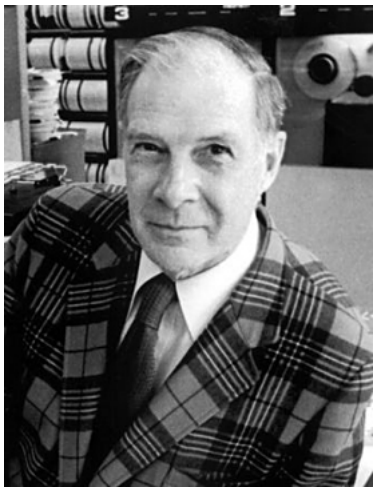
Show that there is a winning strategy such that your friend can always know what square the magic square is.

There might be solutions that exploit the chessboard geometry with its black and white fields. We will ignore the chessboard angle though and use this problem as an excuse to dive into the topic of linear codes. We will solve the problem by treating the token information as a 64-bit word and we will devise a winning strategy that involves a Hamming[3] code (a type of perfect linear code).

But first lets introduce linear codes. We operate in the field $\mathbb{F}_q$ of integers modulo a prime q.

Definition 9.1. A **linear code** C of words of length n is a subspace of the vector space $\mathbb{F}_q^n$. Let $\dim C = k$, then we say that C is a $[n, k]_q$ linear code.

Given a basis $\{\boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_k}\}$ of C, we can build a matrix $G \in \mathbb{F}_q^{k \times n}$



[3] Richard Hamming was one of the founders of modern coding theory. http://en.wikipedia.org/wiki/Richard_Hamming

using the $\boldsymbol{c_i}$ basis vectors as rows. Then C is the row space of G and G is called a **generator matrix** of C. We have[4]

$$C = \{\boldsymbol{x}G : \boldsymbol{x} \in \mathbb{F}_q^k\},$$

so a code C is made from all linear combinations of the row vectors of its generator matrix.

Let G' be the row reduced echelon form of G. By definition G has full row rank, so G' has only nonzero rows. If $G' = \left[\, I_k \mid A_{k\times(n-k)} \,\right]$ for identity matrix I_k and some matrix A then the generator matrix G' is in **standard form**[5]. Row operations preserve the row space, so G' also generates C.

Definition 9.2. Given a $[n,k]_q$ linear code C, matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ is a **parity check matrix** for C, if $C = nullspace(H) = \{\boldsymbol{c} \in \mathbb{F}_q^n : H\boldsymbol{c}^T = 0\}$.

Theorem 9.3. *Given a $[n,k]_q$ linear code C and a generator matrix $G = \left[\, I_k \mid A_{k\times(n-k)} \,\right]$ for C in standard form, then $H = \left[\, -A^T_{(n-k)\times k} \mid I_{n-k} \,\right]$ is a parity check matrix[6] for C.*

Proof. Let $\boldsymbol{c} \in C$ be a code word from C. Then there exists an $\boldsymbol{x} \in \mathbb{F}_q^k$ such that $\boldsymbol{c} = \boldsymbol{x}G$. We have

$$\begin{aligned}
H\boldsymbol{c}^T &= H(\boldsymbol{x}G)^T \\
&= \left[-A^T_{(n-k)\times k} | I_{n-k}\right] \left(\boldsymbol{x}\left[I_k | A_{k\times(n-k)}\right]\right)^T \\
&= \left[-A^T_{(n-k)\times k} | I_{n-k}\right] \left[\frac{I_k}{A^T_{(n-k)\times k}}\right] \boldsymbol{x}^T \\
&= (-A^T + A^T)\boldsymbol{x}^T \\
&= 0
\end{aligned}$$

This means that $C \subseteq nullspace(H)$. We have $\dim C = k$ and

$$\dim nullspace(H) = n - rank(H) = n - n + k = k,$$

so $C = nullspace(H)$ and H is a parity check matrix for C.

□

What can we do if the generator matrix is not in standard form? Swapping columns in the generator matrix does not preserve the row space, so the linear code generated with the modified matrix is clearly not the same as the original code, but it is an equivalent code[7].

Definition 9.4. The **Hamming distance** $d(\boldsymbol{x},\boldsymbol{y})$ between two vectors $\boldsymbol{x},\boldsymbol{y} \in \mathbb{F}_q^n$ is the number of positions in which the vectors differ. With $\boldsymbol{x} = x_1x_2\ldots x_n$ and $\boldsymbol{y} = y_1y_2\ldots y_n$ we have

$$d(\boldsymbol{x},\boldsymbol{y}) = |\{i : 1 \le i \le n : x_i \neq y_i\}|$$

[4] We treat vectors as row vectors in this section. That means that $\boldsymbol{x} \in \mathbb{F}_q^k$ is a matrix $\mathbb{F}_q^{1\times k}$.

[5] Not every generator matrix can be row reduced to the standard form. For example

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

cannot.

[6] With G in standard form this theorem let's us construct a parity check matrix very easily. Also worth noting that in standard form we generate a code word from a message $\boldsymbol{x} \in \mathbb{F}_q^k$ by appending $n-k$ parity check bits to the message with $\boldsymbol{x}G$. We check if the transmitted and received word $\boldsymbol{y} \in \mathbb{F}_q^n$ is a valid code word by verifying $H\boldsymbol{y}^T = 0$. If true then the first k positions of $\boldsymbol{y}$ are the original message $\boldsymbol{x}$.

[7] A $[n,k_1]_q$ linear code C_1 is equivalent to a $[n,k_2]_q$ linear code C_2 if there is a permutation $\pi \in S_n$ such that when π is applied to the coordinate indices of all the code words from C_1, it produces all the code words from C_2. Equivalent linear codes have the same dimension $k_1 = k_2$.

The **Hamming weight** $w(\boldsymbol{x})$ is the number of positions that differ from zero:

$$w(\boldsymbol{x}) = |\{i : 1 \leq i \leq n : x_i \neq 0\}| = d(\boldsymbol{x}, \boldsymbol{0})$$

We will use the following properties of Hamming distances:

Lemma 9.5.

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n : d(\boldsymbol{x}, \boldsymbol{x}) \geq 0$$
$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n : d(\boldsymbol{x}, \boldsymbol{y}) = 0 \Leftrightarrow x = y$$
$$\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n : d(\boldsymbol{x}, \boldsymbol{y}) = d(\boldsymbol{y}, \boldsymbol{x})$$
$$\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{F}_q^n : d(\boldsymbol{x}, \boldsymbol{y}) \leq d(\boldsymbol{x}, \boldsymbol{z}) + d(\boldsymbol{z}, \boldsymbol{y})$$

Proof of Lemma 9.5
The first three properties are obvious from the definition of Hamming distance. For the last property let i be an index where $\boldsymbol{x}$ and $\boldsymbol{y}$ differ, so $x_i \neq y_i$. For vector $\boldsymbol{z}$ we can have the following cases for position i:

$$z_i = x_i \Rightarrow z_i \neq y_i$$
$$z_i = y_i \Rightarrow z_i \neq x_i$$
$$z_i \neq x_i \wedge z_i \neq y_i$$

In each of these cases the contribution of z_i to $d(\boldsymbol{x}, \boldsymbol{z}) + d(\boldsymbol{z}, \boldsymbol{y})$ is at least one, whereas on the left side position i contributes one to $d(\boldsymbol{x}, \boldsymbol{y})$. A similar analysis holds for indices i where $x_i = y_i$. □

Definition 9.6. The **minimum distance** of C is:

$$d(C) = \min\{d(\boldsymbol{x}, \boldsymbol{x'}) : \boldsymbol{x}, \boldsymbol{x'} \in C \wedge \boldsymbol{x} \neq \boldsymbol{x'}\} = \min\{w(\boldsymbol{x}) : \boldsymbol{x} \in C\}$$

The minimum distance is important enough that we add it to the characteristic notation of a linear code: $[n, k, d]_q$ is a linear code over field $\mathbb{F}_q$ with bit strings of length n, code dimension k and minimum distance between code words d.

The next lemma establishes a connection between the minimum distance of a linear code and one of its parity check matrix.

Lemma 9.7. *The minimum distance of a code C equals the minimum number of linearly dependent columns in one of its parity check matrices.*

Proof of Lemma 9.7
Let H be a parity check matrix of $[n, k, d]_q$ linear code C. There must be a code word $\boldsymbol{c}$ with $w(\boldsymbol{c}) = d$. $\boldsymbol{c}$ belongs to the nullspace of H, so

$$H\boldsymbol{c}^T = \boldsymbol{0}$$

But $H\boldsymbol{c}^T$ is a linear combination of column vectors of H, with d nonzero coefficients, so the column vectors in this linear combination are linearly dependent. □

So far we have worked with fields $\mathbb{F}_q$ of any prime q. Now we switch to the binary world $q = 2$ and $\mathbb{F}_2$. Our vectors are bit strings. We transmit these bit strings over a binary symmetric channel.

Definition 9.8. In a **binary symmetric channel** each bit sent has the same probability $p < \frac{1}{2}$ of being received incorrectly.

We send a code word $\boldsymbol{x} \in C$ from a $[n, k]_2$ linear code C over a binary symmetric channel and receive a bit string $\boldsymbol{y}$. If there were no transmission errors, then $y = x$. If there were errors, we want to find the most likely code word $\boldsymbol{x}$ that was transmitted given the errors in $\boldsymbol{y}$.

One decoding strategy[8] would be to choose a code word $\boldsymbol{x}$ with minimum Hamming distance over all code words from C to received bit string $\boldsymbol{y}$. This type of decoding is called *nearest neighbor decoding*. The chosen $\boldsymbol{x}$ is not always unique.

[8] Finding an appropriate code word for the transmitted bit string is called *decoding*. Finding the most likely code word is called *maximum likelihood decoding*.

Theorem 9.9. *In a binary symmetric channel with error probability $p < \frac{1}{2}$ the nearest neighbor decoding is a maximum likelihood decoding.*

Proof. Given a bit string $\boldsymbol{y} \in \mathbb{F}_2^n$ received through the channel, let $P_{\boldsymbol{y}}(\boldsymbol{x})$ be the probability that the code word $\boldsymbol{x}$ was sent when $\boldsymbol{y}$ was received. Because the channel is a binary symmetric channel, we have

$$P_y(x) = p^{d(x,y)}(1-p)^{n-d(x,y)}$$

Consider two code words x and x' such that $d(x,y) \leq d(x',y)$. Because $p < \frac{1}{2}$, we then have $P_y(x) \geq P_y(x')$. It follows that

$$\max_{x \in C} P_y(x) = \min_{x \in C} d(x,y)$$

so the likeliest code word is the nearest neighbor to y. □

For the rest of this section we use nearest neighbor decoding. We want to know if we can detect and possibly correct a transmission with errors. Let's define clearly what we mean by that. A transmission is a pair $(x,y) \in C \times \mathbb{F}_2^n$, where a code word x was sent and a bit strings y was received. It has $d(x,y)$ transmission errors. The nearest neighbor decoding $nnd(y)$ finds a code word (not necessarily unique) closest to y. The following holds by definition:

$$d(y, nnd(y)) = \min_{c \in C} d(y,c)$$

If no errors occurred in the transmission, then $x = y$ and also $d(x,y) = 0$ and $nnd(y) = x$. If errors in the transmission occurred we want to:

E.1 detect that errors happened, i.e. establish that $y \notin C$.

E.2 correct the errors, i.e. establish $nnd(y) = x$.

The next theorem describes the conditions for **E.1**.

Theorem 9.10. *Given a $[n,k,d]_2$ linear binary code C, we can detect that any transmission with up to e errors was erroneous if and only if $d > e$.*

Proof. ($\Rightarrow$) Let (x,y) be a transmission with $d(x,y) \leq e < d$ errors. Assume $y \in C$. Then $d(x,y) \leq e < d$ is a contradiction to d being the minimal distance of C. It follows that $y \notin C$.
($\Leftarrow$) We can detect that any transmission with up to e errors was erroneous. Assume $d \leq e$. Then there exist two code words $x \neq x'$ such that $d(x,x') \leq e$. Now consider transmission (x,x'). It's impossible to detect that it had errors because x' is a code word. This is a contradiction with the fact that we can detect that any transmission with up to e errors was erroneous. So $d > e$. □

For **E.2** we have this theorem:

Theorem 9.11. *Given a $[n,k,d]_2$ linear binary code C, we can correct any transmission with up to e errors if $d > 2e$.*

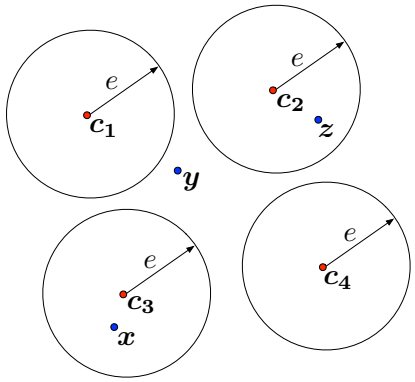


Figure 9.1: A Hamming sphere for code word c with radius e is the set $\{x : d(x,c) \leq e\}$. In this figure the spheres don't overlap, so vectors (blue dots) that fall within a sphere can be error-corrected to code words (red dots).

Proof. Let $(\boldsymbol{x}, \boldsymbol{y})$ be a transmission with $d(\boldsymbol{x}, \boldsymbol{y}) \leq e$ errors and $d > 2e$. Assume $nnd(\boldsymbol{y}) \neq \boldsymbol{x}$. Then $d(\boldsymbol{y}, nnd(\boldsymbol{y})) \leq e$ (otherwise $\boldsymbol{x}$ would be closer than $nnd(\boldsymbol{y})$ to $\boldsymbol{y}$). We have

$$d(\boldsymbol{x}, nnd(\boldsymbol{y})) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, nnd(\boldsymbol{y})) \leq e + e = 2e$$

which contradicts $d > 2e$. So $nnd(\boldsymbol{y}) = \boldsymbol{x}$.

□

Theorems 9.10 and 9.11 tell us that a large minimum distance $d(C)$ allows us to detect and correct more errors. But a large minimum distance between code words also limits the number of code words. The following theorem puts an upper bound on the number of code words given a minimum distance.

Theorem 9.12. *Given a $[n, k, 2t+1]_2$ linear binary code C, we have*

$$|C| \leq \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}$$

*This upper bound is called **Hamming bound**.*

Proof. Given a bit string $\boldsymbol{x}$ and an integer $i \leq n$, there are $\binom{n}{i}$ ways to choose the i positions at which $\boldsymbol{x}$ and and another bit string $\boldsymbol{y}$ differ. So there are $\binom{n}{i}$ bit strings $\boldsymbol{y}$ with $d(\boldsymbol{x}, \boldsymbol{y}) = i$. This means there are

$$\sum_{i=0}^{t} \binom{n}{i}$$

bit strings $\boldsymbol{y}$ with $d(\boldsymbol{x}, \boldsymbol{y}) \leq t$.

On the other hand, a bit string $\boldsymbol{y}$ with $d(\boldsymbol{y}, \boldsymbol{x}) \leq t$ to a code word $\boldsymbol{x}$ cannot have the same $d(\boldsymbol{y}, \boldsymbol{x'}) \leq t$ to a different code word $\boldsymbol{x'}$ because then

$$d(\boldsymbol{x}, \boldsymbol{x'}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{x'}) \leq t + t \leq 2t$$

which is a contradiction to $d(C) = 2t + 1$.

So for each code word, we have at most $\sum_{i=0}^{t} \binom{n}{i}$ bit strings with Hamming distance $\leq t$ and we cannot have the same bit strings near two different code words. We have 2^n bit strings, so

$$|C| \sum_{i=0}^{t} \binom{n}{i} \leq 2^n$$

□

A binary linear code that achieves equality in the Hamming bound 9.12 is called a **perfect code**.

We are now ready to define Hamming codes.

Definition 9.13. A Hamming code $\mathfrak{H}_r$ of order r (where r is a positive integer) is a binary linear code with the parity check matrix with columns that are all the $2^r - 1$ nonzero bit strings of length r.

Changing the order of the columns in the parity check matrix produces equivalent codes with the same minimum distance. So for easier analysis we now consider Hamming codes with parity check matrix in standard form, ie the last r columns form the identity matrix I_r, so $H = \left[A_{r\times(n-r)} \mid I_r \right]$, with $n = 2^r - 1$. From theorem 9.3 we then know the generator matrix is $G = \left[I_{n-r} \mid -A^T_{(n-r)\times r} \right] = \left[I_{n-r} \mid A^T_{(n-r)\times r} \right]$, since we operate in $\mathbb{F}_2$. We can see that $\dim \mathfrak{H}_r = n - r$. What is the minimum distance of $\mathfrak{H}_r$? All columns are nonzero and distinct, so no two columns are linearly dependent[9]. But consider the linear combination of the three columns

$$[1, 1, 0, \ldots, 0]^T + [1, 0, 0, \ldots, 0]^T + [0, 1, 0, \ldots, 0]^T = \mathbf{0}^T$$

They are linearly dependent. From lemma 9.7 it follows that $d(\mathfrak{H}_r) = 3$, so $\mathfrak{H}_r$ is a $[2^r - 1, 2^r - 1 - r, 3]_2$ binary linear code. According to theorem 9.11 it can correct transmissions with one error.

[9] Again, this is in $\mathbb{F}_2$. The sum of two distinct columns is always nonzero, so a linear combination that is zero has to have coefficients zero, hence linearly independent.

Theorem 9.14. *$\mathfrak{H}_r$ is a perfect code.*

Proof. The generator matrix has full row rank, so we need all linear combinations of the rows to get all the code words. This are binary words, so there are 2^{n-r} distinct linear combinations. It means $|\mathfrak{H}_r| = 2^{n-r}$.

Inserting into formula of theorem 9.12, we get[10]

$$2^{n-r} \sum_{i=0}^{1} \binom{n}{i} = 2^{n-r}(1 + n) = 2^{n-r}(1 + 2^r - 1) = 2^{n-r}2^r = 2^n$$

□

[10] With $t = 1$, because $d(\mathfrak{H}_r) = 3$.

This concludes our dive into linear codes and Hamming codes. Let's return to our problem and solve it using Hamming codes. The state of the chessboard is a binary word of length 64. We use $r = 6$, so Hamming code $\mathfrak{H}_6$. The word length is $2^6 - 1 = 63$. We agree that the devil choosing bit 64 is a special case which we handle later. For now imagine the chessboard as a 63-bit binary word and the devil only choosing a magic field between 1 and 63.

The winning strategy can be summarized as follows: the first player needs to modify the 63-bit word (by flipping at most one bit) in such a way that the magic field is the one bit error of a code word in $\mathfrak{H}_6$. Then the second player only has to come in, decode[11] the modified chessboard and point to the corrected error which is the same magic field.

Is this always possible? We know that Hamming codes are perfect codes, so any 63-bit word is at most one bit away from a code word. We have the following cases for the initial state of the chessboard:

- It happens to be a code word in $\mathfrak{H}_6$. Then the first player flips the magic field bit, producing an error there.
- It happens to be a 63-bit word that is a one bit error at the magic field. The first player doesn't flip any bit in this case.
- It happens to be a 63-bit word with a one bit error different from the magic field.

The last case needs a little thinking. Assume H is the parity check matrix for our $\mathfrak{H}_6$ Hamming code and assume the state of the chessboard is $\boldsymbol{x}$, which is one bit error from a code word $\boldsymbol{c_1}$. Also let $1 \leq m \leq 63$ be the magic field bit and $\boldsymbol{e_m}$ the unit vector with bit m set. The one bit error is different from the magic field, so $\boldsymbol{x} - \boldsymbol{c_1} \neq \boldsymbol{e_m}$. Let $\boldsymbol{y} = \boldsymbol{x} - \boldsymbol{e_m}$, which is also one bit away from a code word $\boldsymbol{c_2}$, with error bit k. So $\boldsymbol{y} = \boldsymbol{c_2} + \boldsymbol{e_k}$.

Now consider $\boldsymbol{x} - \boldsymbol{e_k}$:

$$H(\boldsymbol{x} - \boldsymbol{e_k}) = H(\boldsymbol{y} + \boldsymbol{e_m} - \boldsymbol{e_k}) = H(\boldsymbol{y} - \boldsymbol{e_k}) + H\boldsymbol{e_m} = H\boldsymbol{c_2} + H\boldsymbol{e_m} = H\boldsymbol{e_m}$$

So $\boldsymbol{x} - \boldsymbol{e_k}$ has one bit error at the magic field, which is what we want. Flipping bit k on the initial chessboard $\boldsymbol{x}$ achieves that.

In all three cases the modified chessboard is one bit away from a code word with the error at the magic field and the chessboard was modified by flipping at most one bit. The players agree that if the chessboard is a code word instead, then the devil chose bit 64 as the magic field, which handles the special case. Modifying the chessboard to get a code word can also be done by flipping at most one bit. This scales to any chessboard with size a power of two.

[11] Decoding is done as follows: $\boldsymbol{x}$ needs to be decoded. It is one bit away from a code word $\boldsymbol{c}$ with error at bit k. Let $\boldsymbol{e_k}$ be the unit vector with bit k set. So $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{e_k}$ and

$$H\boldsymbol{x} = H(\boldsymbol{c} + \boldsymbol{e_k}) = H\boldsymbol{e_k}$$

Since $\boldsymbol{e_k}$ is be the unit vector with bit k set, $H\boldsymbol{e_k}$ is column k from the parity check matrix H. To decode we calculate $H\boldsymbol{x}$ and look to see which column in H the result is. To save the lookup step we can be even more elegant. Instead of the parity check matrix in standard form, we choose a parity check matrix where column k is the bit representation of k. Instead of lookup we just reverse the bit representation back to the integer k.

What follows is a Mathematica session illustrating the strategy. We use a parity check matrix with column k the bit representation of integer k. This simplifies decoding as remarked in the side note 11 above.

The function *hamming* generates the parity check matrix for a Hamming code with a given r.

```
In[1]:= hamming[r_Integer] := Transpose[Table[IntegerDigits[i, 2, r], {i, 1, 2^r - 1}]]
```

For example

```
In[2]:= hamming[4] // MatrixForm
```

```
Out[2]=
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

We define r and the corresponding Hamming code h for our chessboard

```
In[3]:= r = 6;
```

```
In[4]:= h = hamming[6];
```

The function *pos* returns the unit vector with the error bit set from decoding the specified word.

```
In[5]:= pos[w_] := With[{s = Mod[h.w, 2]}, UnitVector[2^Length[s] - 1, FromDigits[s, 2]]]
```

Function *friendOne* implements the strategy part for the first friend. Given the initial state of the chessboard *cb* and a magic field *mf*, it returns a modified chessboard.

```
In[6]:= friendOne[cb_List, mf_Integer] := Module[{em, y, ey},
        em = UnitVector[2^r - 1, mf]; y = Mod[cb - em, 2]; ey = pos[y];
        z = Mod[cb - ey, 2]
        ]
```

Function *friendTwo* implements the strategy part for the second friend: decoding the specified chessboard and returning the index of the error bit which is also the magic field.

```
In[7]:= friendTwo[cb_List] := Position[pos[cb], 1][[1,1]]
```

This next function is returning random initial states for the chessboard.

```
In[8]:= rw := RandomInteger[1, {2^r - 1}]
```

We can now simulate one game with the devil.
cb is the initial (random) state of the chessboard.

```
In[9]:= cb = rw;
```

The magic field is some integer, the devil chose 23.

```
In[10]:= mf = 23;
```

The first friend enters the room, modifies the chessboard according to *friendOne*. The returned value is the modified chessboard.

```
In[11]:= cb2 = friendOne[cb, mf];
```

Let's check that the Hamming distance between initial and modified chessboard is at most one.

```
In[12]:= HammingDistance[cb, cb2]
Out[12]= 1
```

The second friend comes in and decodes with *friendTwo*, getting 23.

```
In[13]:= friendTwo[cb2]
Out[13]= 23
```

10
Maximum subsequence

Problem

Given a sequence of integer numbers $x_0, x_1, \ldots, x_{N-1}$ (not necessarily positive) find a subsequence $x_i, \ldots, x_{j-1}$ such that the sum of numbers in it is maximum over all subsequences of consecutive elements.

We adopt the same notation used in *Programming in the 1990s* [1] and *Programming, The Derivation of Algorithms*[2]: The notation of function application is the "dot" notation with name of function, followed by arguments, each separated by a dot. The notation of quantified expressions has the operator followed by the bounded variables, then a colon followed by the range for the bounded variables and ended with a colon and the actual expression. So

$$(\sum k : i \leq k < j : x_k)$$

corresponds to the more classical mathematical notation $\sum_{k=i}^{j-1} x_k$.
For our derivation steps in predicate calculus we will use the following notation:

$$\begin{array}{ll} & A \\ = & \{\text{reason why A equals B}\} \\ & B \\ \leq & \{\text{reason why B is less than C}\} \\ & C \end{array}$$

If all the numbers are positive then the maximum sum is the sum of the whole initial sequence. If all the numbers are negative then the maximum sum is 0 (by definition 0 is the sum over an empty range). So the interesting case is a sequence with positive and negative numbers in it.

[1] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

[2] A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990

We hope to find an algorithm that visits every number in the sequence only once, so with runtime $O(n)$. Let's introduce some notation: Let's introduce some notation[3] :

[3] Our problem can be stated as finding $f.N$ given $x_i \in \mathbb{Z}, 0 \leq i < N, N \in \mathbb{N}$.

$$f.n = (MAXi, j : 0 \leq i \leq j \leq n : s.i.j)$$

with

$$s.i.j = (\sum k : i \leq k < j : x_k).$$

We will use properties of quantified expressions as covered in Chapter 3 of *Programming in the 1990s*[4].

[4] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

$$\begin{array}{ll} & f.N \\ = & < \text{definition of f} > \\ & (MAXi, j : 0 \leq i \leq j \leq N : s.i.j) \\ = & < \text{range nesting} > \\ & (MAXj : 0 \leq j \leq N : (MAXi : 0 \leq i \leq j : s.i.j)) \\ = & < \text{defining } p.j = (MAXi : 0 \leq i \leq j : s.i.j) > \\ & (MAXj : 0 \leq j \leq N : p.j) \\ = & < \text{range split, 1-point rule} > \\ & (MAXj : 0 \leq j < N : p.j) \ max \ p.N \\ = & < \text{definition of f} > \\ & f.(N-1) \ max \ p.N \end{array}$$

We now have a recursive expression for f, which still depends on a newly introduced function p. Let's see if we can get a recursive expression for p too:

$$\begin{array}{ll} & p.N \\ = & < \text{definition of p} > \\ & (MAXi : 0 \leq i \leq N : s.i.N) \\ = & < \text{range split, 1-point rule} > \\ & (MAXi : 0 \leq i < N : s.i.N) \ max \ s.N.N \\ = & < \text{definition of s and s.N.N = o by definition of sum over empty range} > \\ & (MAXi : 0 \leq i < N : (\sum k : i \leq k < N : x_k)) \ max \ 0 \\ = & < \text{range split in sum} > \\ & (MAXi : 0 \leq i < N : (\sum k : i \leq k < N-1 : x_k) + x_{N-1}) \ max \ 0 \\ = & < + \text{ distributes over max} > \\ & (x_{N-1} + (MAXi : 0 \leq i < N : (\sum k : i \leq k < N-1 : x_k)) \ max \ 0 \\ = & < \text{definition of p} > \\ & (x_{N-1} + p.(N-1)) \ max \ 0 \end{array}$$

So $f.N = f.(N-1) \ max \ p.N$ and $p.N = (x_{N-1} + p.(N-1)) \ max \ 0$. The base cases are $f.0 = 0$ and $p.0 = 0$.

Armed with these recursive relations we can provide a Haskell program that solves the problem:

Listing 10.1: Haskell code

```
maxSum :: [Int] -> (Int, Int)

maxSum (x:xs) = let (a, b) = maxSum xs
                    c = x + b
                in (max c (max a 0), max c 0)

maxSum [] = (0, 0)
```

The maxSum function calculates the tuple $(f.N, p.N)$.

11
Minkowski Sum & Well-spaced triples

Fast Fourier Transform and using it to speed up polynomial multiplication is the topic of the two problems[1] in this note.

[1] Jeff Erickson. Algorithms — Extended Dance Remix: Fast Fourier Transforms. https://jeffe.cs.illinois.edu/teaching/algorithms/notes/A-fft.pdf, 2021. [Online; accessed 07-May-2022]

Problem

Given two sets of integers $X \subset \mathbb{Z}$ and $Y \subset \mathbb{Z}$, compute the size of the Minkowski sum: $X + Y = \{x + y : x \in X, y \in Y\}$ in $O(n \log n)$ time.

Solution. A pretty straightforward way of calculating the Minkowski sum is to generate all possible pairs (that is a nested loop, so $O(n^2)$) and then also making sure the resulting values form a set, so only occur once. This can be achieved by storing the values as we go in a balanced binary search tree. For each value the cost would be $O(\log n)$, so the straightforward solution has a runtime of $O(n^2 \log n)$ if using a binary search tree or $O(n^2)$ if using a hashtable.

Can we do better? This problem is an exercise in the FFT chapter of Jeff Erickson's Algorithms book. So the answer is: yes we can. To do so, we remember that multiplying two polynomials given in coefficient form can be done in $O(n \log n)$ using the Fast Fourier Transform.

But what polynomials should we consider? Let's explore polynomial multiplication with a couple of examples:

$$\begin{aligned}(1+x)(x^2+x^5) &= 1(x^2+x^5) + x(x^2+x^5) \\ &= x^2 + x^5 + x^3 + x^6\end{aligned}$$

$$\begin{aligned}(2x+x^4)(1+3x^3) &= 2x(1+3x^3) + x^4(1+3x^3) \\ &= 2x + 6x^4 + x^4 + 3x^7\end{aligned}$$

Notice how we multiplied each monomial[2] from the first polynomial with each mononial from the second polynomial. In the second example it is also visible that we haven't yet collected together all the monomials of degree four of the multiplication result to better demonstrate that each monomial from the first polynomial is multiplied with each monomial from the second polynomial.

When we multiply two monomials, their coefficients get multiplied and their exponents get added. Each monomial from one polynomial is paired with each monomial from the other polynomial in an operation (multiplication) and in that operation the exponents are added. This strongly suggests[3] that we should define a polynomial from one of the given sets of integers by making the members of the set be the exponents of the monomials that form the polynomial.

For example: if $X = \{2, 5, 6\}$ then the corresponding polynomial could be $x^2 + x^5 + x^6$. The polynomial coefficients have been arbitrarily chosen to all be one [4].

Thus we define the two polynomials $p_X(x)$ and $p_Y(x)$ from the given integer sets X and Y:

$$p_X(x) = \sum_{i \in X} x^i$$
$$p_Y(x) = \sum_{j \in Y} x^j$$

and we multiply them

$$p_X(x)p_Y(x) = \sum_{i \in X} \sum_{j \in Y} x^{i+j}$$

The exponents of the monomials of the polynomial product are the members of the Minkowski sum $X + Y$. The size of $X + Y$ is the number of monomials [5].

This gives us a way to calculate the size of $X + Y$ in $O(n \log n)$ because we can do the polynomial multiplication using FFT in $O(n \log n)$. □

Problem

Given is a bit string $B[1 \ldots n]$. A well-spaced triple is a triple (i, j, k) of indices such that $1 \leq i < j < k \leq n$ and $B[i] = B[j] = B[k] = 1$. Detect in $O(n \log n)$ time if bit string B contains a well-spaced triple.

[2] A monomial is an individual term of a polynomial A polynomial is a sum of monomials. In our example the monomials of $1 + x$ are 1 and x. The monomials of $x^2 + x^5$ are x^2 and x^5.

[3] In the problem each member of the first set is paired with each member of the second set in an operation (addition).

[4] There is one small wrinkle in this scheme. In the problem the given sets are sets of integers, so they can be negative. We cannot have monomials with negative exponents. For now let us assume X and Y only contain non-negative integers with the promise that at the end of this solution we will address how to drop this assumption.

[5] As promised: what can we do when X and Y contain negative integers. Let $d > 0$ be a constant such that both $d + X = \{d + a : a \in X\}$ and $d + Y = \{d + b : b \in Y\}$ have only non-negative members.

We define the polynomials slightly differently:

$$p_X(x) = \sum_{i \in X} x^{i+d}$$
$$p_Y(x) = \sum_{j \in Y} x^{j+d}$$

and we multiply them

$$p_X(x)p_Y(x) = \sum_{i \in X} \sum_{j \in Y} x^{i+j+2d}$$

Again, the size of $X + Y$ is the number of monomials.

Solution. Again the brute-force solution is easy to describe: for each middle index in the triple we consider all possible distances to left and right indices and check the condition. This is a nested loop with runtime $O(n^2)$.

We will try to improve on the brute-force by again employing polynomial multiplication even though it is not an obvious choice. After all we are only given one bit string, not two bit strings and multiplication needs two operands. But maybe we can derive a polynomial from the bit string and then square that polynomial which would be a polynomial multiplication.

Let's explore what would happen if we use the given bit string B directly as a coefficient vector of a polynomial

$$p_B(x) = \sum_{i=0}^{n-1} B[i+1]x^i$$

Squaring $p_B(x)$ we get

$$\begin{aligned}(p_B(x))^2 &= p_B(x)p_B(x) \\ &= (\sum_{i=0}^{n-1} B[i+1]x^i)(\sum_{j=0}^{n-1} B[j+1]x^j) \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} B[i+1]B[j+1]x^{i+j}\end{aligned}$$

For all $0 \leq j < n$ monomial x^j is present in $p_B(x)$ if $B[j+1] = 1$. If monomial x^j is present in $p_B(x)$ then monomial x^{2j} is present in $(p_B(x))^2$. This is because monomial x^j pairs with itself in the polynomial multiplication. So the coefficient of monomial x^{2j} in $(p_B(x))^2$ is at least one (namely $B[j+1]B[j+1] = 1$). Can it be larger than one? It would mean that some other monomial pairing has exponent sum equal to $2j$. Let i and k be the two indices for which $B[i+1] = 1$, $B[k+1] = 1$ and $i + k = 2j$. But $i + k = 2j$ is equivalent to $k - j = j - i$ which means that (i, j, k) form a well-spaced triple. This (i, k) monomial pairing appears twice in the polynomial multiplication (once for monomial x^i from the left and monomial x^k from the right and once reversed with monomial x^k from the left and monomial x^i from the right). That means that the well-spaced triple contributes the value two to the coefficient of x^{2j} in $(p_B(x))^2$ and we have found our criteria for detecting well-spaced triples: if $(p_B(x))^2$ has any monomials with even degree and coefficients greater or equal to three, then B has a well-spaced triple.

This gives us a way to detect well-spaced triples in $O(n \log n)$ because we can do the polynomial multiplication using FFT in $O(n \log n)$.

□

12
No consecutive integers

INTEGER EQUATIONS and multisets are the topics of the problem [1] in this note.

[1] Variation of Problem 1-72. on page 45 in N. Loehr. *Combinatorics*. Discrete Mathematics and Its Applications. CRC Press, 2017. ISBN 9781498780278

Problem

Determine the number of subsets of size k from set $\{1, 2, \ldots, n\}$ that do not contain consecutive integers.

The number of subsets of size k from set $\{1, 2, \ldots, n\}$ (without any constraints) is given by the binomial coefficient $\binom{n}{k}$. Each subset of size k can be represented as a word of length n from alphabet $\{\bigstar, []\}$ with k $[]$'s and $n-k$ $\bigstar$'s: if i is in the subset then the corresponding word has a $[]$ at position i otherwise it has a $\bigstar$ at position i. This representation is clearly a bijection. The constraint of no consecutive integers in a subset implies no adjacent $[]$'s in the corresponding word[2].

[2] For example given set $\{1, 2, 3, 4\}$ the subset $\{2, 4\}$ corresponds to $\bigstar\,[]\,\bigstar[]$. The subset $\{1, 2\}$ has consecutive integers and corresponds to $[]\,[]\,\bigstar\bigstar$. The reason why we chose $[]$ to indicate inclusion into a subset will become clear soon.

We will now associate words from $\{\bigstar, []\}^n$ with other combinatorial objects: the integer equations.

Definition 12.1. Given fixed integers $m > 0$ and $t \geq 0$ a sequence $(z_1, z_2, \ldots, z_m)$ is an **integer equation** if $\forall i : 1 \leq i \leq m : z_i \in \mathbb{N}_0$ and

$$\sum_{i=1}^{m} z_i = t$$

The number t is called the **target** of the integer equation.

Note that these are sequences and order matters. From an integer equation $(z_1, z_2, \ldots, z_m)$ we construct a $\{\bigstar, []\}^{t+m-1}$ word in the following way: start with z_1 number of $\bigstar$'s, then a $[]$, then z_2 number of $\bigstar$'s, then a $[]$ and so on finishing with the z_m number of $\bigstar$'s which are **not** followed by a $[]$. The word will contain exactly t $\bigstar$'s and they will need exactly $m-1$ $[]$ separators to know which stars belong to which z_i. It's easy to verify that this encoding is also a bijection[3].

[3] As an example let $m = 5$ and target $t = 10$. The sequence $(1, 2, 1, 3, 3)$ is an integer equation since

$$1 + 2 + 1 + 3 + 3 = 10$$

and it corresponds to the word

$$\bigstar\,[]\,\bigstar\bigstar\,[]\,\bigstar\,[]\,\bigstar\bigstar\bigstar\,[]\,\bigstar\bigstar\bigstar$$

This in turn corresponds to the subset $\{2, 5, 7, 11\}$ of set $\{1, \ldots, 14\}$.

Lemma 12.2. *If we set $t = n - k$ (the number of ★'s) and from $t + m - 1 = n$ we get $m = k + 1$ (the word length) then we can associate subsets of size k from set $\{1, 2, \ldots, n\}$ with integer equations $(z_1, z_2, \ldots, z_{k+1})$ for target $n - k$. The constraint of not having consecutive integers in the subsets translates to integer equations $(z_1, z_2, \ldots, z_{k+1})$ where $z_i > 0$ except for z_1 and z_{k+1} (the first and the last in the sequence). This follows from the encoding not allowing adjacent []'s so there need to be ★'s separating the []'s.*

According to this association if we can count the number of integer equations with all but the first and last z_i strictly positive then we also have the number of subsets with no consecutive integers. To get there we will first count the number of anagrams, then the number of multisets, then the number of integer equations and finally the number of integer equations with all but the first and last z_i strictly positive. In what follows we will use n and k for other things before we bring it back in the end to our initial problem.

Let's start this journey with anagrams. Let $\{s_1, s_2, \ldots, s_k\}$ be an alphabet of distinct symbols. We can build words with these symbols, for example $s_2 s_2 s_1 s_3 s_3 s_1$. As a notational convenience $s_i s_i s_i \ldots s_i = s_i^j$ if s_i appears j consecutive times in a word, so the example would be $s_2^2 s_1 s_3^2 s_1$.

Definition 12.3. A word is an **anagram**[4] of $s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k}$ with $n_i > 0$ if it is a word containing exactly n_i number of s_i symbols for each $1 \leq i \leq k$. We denote with $\mathcal{A}(s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k})$ the set of all anagrams of $s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k}$.

[4] For example given word a^2b the words *aba* and *baa* are anagrams of it. The word *abb* is not.

Theorem 12.4. *Given the set of anagrams $\mathcal{A}(s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k})$ let $n = \sum_{i=1}^{k} n_k$. Then*

$$|\mathcal{A}(s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k})| = \binom{n}{n_1, n_2, \ldots, n_k}$$

where $\binom{n}{n_1, n_2, \ldots, n_k}$ is the multinomial coefficient[5].

[5] The multinomial coefficient is defined as

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{\prod_{i=1}^{k} n_i!}$$

Proof. We have n positions in our word that we need to fill with symbols. We are going to make the following choices: first we choose n_1 positions from those n positions where we fill in the symbol s_1. Then we choose the n_2 positions from the remaining unfilled positions where we fill in s_2 and so on. In total we make k such choices and the number of remaining unfilled positions at each stage is independent of the previous choices, so the multiplication rule applies. For our first symbol s_1 we have $\binom{n}{n_1}$ possibilities, for our second symbol we have $\binom{n-n_1}{n_2}$ possibilities and so on. Because of the multiplication rule the total number of choices is the product of all these binomial coefficients, so

$$|\mathcal{A}(s_1^{n_1} s_2^{n_2} \ldots s_k^{n_k})| = \prod_{i=1}^{k} \binom{n - (\sum_{j=1}^{i-1})}{n_i}$$

Expanding[6] the binomial coefficients on the right-hand side into factorials according to the binomial coefficient definition and simplifying the expression gives us the desired result. □

[6] After the binomial coefficients are expanded the product becomes a telescoping product that simplifies to exactly the multinomial coefficient.

We move on to **multisets**. Informally multisets are sets (order does not matter) where each element can appear more than once. So given a set A (the alphabet) a multiset is a tuple of A together with a function $\mu : A \mapsto \mathbb{N}$ that determines how often an element $a \in A$ appears in the multiset. For notational convenience we will use curly braces and list elements (with exponents if they appear more than once). For example $\{a^2, b, c^4\}$ is a multiset where a appears twice, b once and c four times. Note that order does not matter, so $\{a^2, b, c^4\}$ is the same multiset as $\{b, a^2, c^4\}$. The size of the multiset is the number of elements in it with elements appearing more than once counted accordingly, so

$$|(A, \mu)| = \sum_{a \in A} \mu(a)$$

Theorem 12.5. *The number of multisets of size k from an alphabet set of size n is*[7]

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

[7] For example with alphabet set $\{a, b\}$ the multisets of size two are $\{a^2\}$, $\{b^2\}$, $\{a, b\}$, so there are three of them.

Proof. We will do an encoding of multisets to anagrams similar to what we did at the beginning of this section with ★'s and []'s. To avoid confusion with that previous encoding in this proof we will use the symbols ∘ and |.

Let $A = \{a_1, a_2, \ldots, a_n\}$ be our alphabet. For a multiset (A, μ) with size k we define the following word[8] with symbols $\{\circ, |\}$:

$$\circ^{\mu(a_1)} | \circ^{\mu(a_2)} | \ldots | \circ^{\mu(a_n)}$$

[8] The multisets from the previous example would be encoded as follows:

$$\{a^2\} \mapsto \circ\circ|$$
$$\{b^2\} \mapsto |\circ\circ$$
$$\{a, b\} \mapsto \circ|\circ$$

The first circles denote how often a_1 is in the multiset. They are separated by a | from the circles that denote how often a_2 is in the multiset and so on. In total there are k circles because the multiset has size k and there need to be $n-1$ separators because the alphabet has size n and the circles for each element need to be kept apart. It's easy to see that we have defined a bijection from the set of multisets of size k with alphabet of size n to the set of anagrams $\mathcal{A}(\circ^k|^{n-1})$. From theorem 12.4 we already know how to count the size of $\mathcal{A}(\circ^k|^{n-1})$ and with the bijection it proves this theorem. □

Our next stop are the number of integer equations. Given m and t how many integer equations $(z_1, z_2, \ldots, z_m)$ for target t are there?

Theorem 12.6. *The number of integer equations $(z_1, z_2, \ldots, z_m)$ for target t is*

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

Proof. We will associate a multiset with each integer equation[9]. The multiset will contain the element i z_i many times, for $1 \leq i \leq m$. Again it can be checked that this defines a bijection. These multisets belong to the set of multisets of size t from an alphabet of size m and theorem 12.5 counts them. By the bijection rule we have proven this theorem. □

[9] For example with $m = 5$ and target $t = 10$ the integer equation $(1, 2, 1, 3, 3)$ would correspond to multiset $\{1, 2^2, 3, 4^3, 5^3\}$.

We are almost done. In the beginning of this section we encoded our subsets without consecutive integers as integer equations with all but the first and last summand strictly positive. So we need to count these types of integer equations with this constraint.

Theorem 12.7. *The number of integer equations $(y_1, y_2, \ldots, y_m)$ for target t with $y_i > 0$ for all $1 < i < m$ is*

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

Proof. For an integer equation $(y_1, y_2, \ldots, y_m)$ we have $\sum_{i=1}^{m} y_i = t$ and $y_i > 0$ for all $1 < i < m$. So we can write

$$y_1 + \sum_{i=2}^{m-1} (y_i - 1) + y_m = t - (m - 2)$$

This shows that we can transform the integer equations with the strictly positive constraints into normal integer equations without constraints but with a new target. This again is a bijection. We know how to count these from theorem 12.6. The new target is $t - m + 2$. Plugging it in we get

$$\binom{t - m + 2 + m - 1}{m - 1} = \binom{t+1}{m-1}$$

□

Using 12.7 and $t = n - k$ and $m = k + 1$ as described by our association 12.2 of subsets of size k without consecutive integers from set $\mathbb{N}_n$ to integer equations with all but the first and last strictly positive terms, we are finally able to solve the problem in this section. The answer is $\binom{n-k+1}{k}$.

13
Paying a dollar

Problem

In how many combinations of half-dollars, quarters, dimes, nickels and pennies can you pay out one dollar ? You can assume you have enough coins for any combination and any coins of one denomination are indistinguishable.

We will look at tuples (h, q, d, n, p) where h is the number of half-dollars, q the number of quarters, d the number of dimes, n the number of nickels and p the number of pennies. We want to consider all tuples (h, q, d, n, p) such that:

$$50h + 25q + 10d + 5n + p = 100$$

We want to determine how many such tuples exist. Let's establish what the possible values for h, q, d, n and p can be:

Denomination	Possible Values	Values range size
h	0, 1, 2	3
q	0, 1, 2, 3, 4	5
d	0, . . . , 10	11
n	0, . . . , 20	21
p	0, . . . , 100	101

Clearly h, q, d, n and p cannot take values outside of the ones listed because it would violate the requirement: $50h + 25q + 10d + 5n + p = 100$.

The multiplicity principle tells us there are $3 * 5 * 11 * 21 * 101 = 349965$ distinct tuples from the possible values.

But not all of them fulfill $50h + 25q + 10d + 5n + p = 100$. To figure out how many of them do let's introduce the following notation:

$$\#(h,q,d,n,p)_x = |\{(h,q,d,n,p) : 50h + 25q + 10d + 5n + p = x\}|$$

so $\#(h,q,d,n,p)_x$ is the number of tuples of half-dollars, quarters, dimes, nickels and pennies that add up to x. We are looking for $\#(h,q,d,n,p)_{100}$.

We also use $\#(q,d,n,p)_x$ for tuples of quarters, dimes, nickels and pennies that add up to x, $\#(d,n,p)_x$ for dimes, nickels and pennies that add up to x etc.

The half-dollar denomination has the smallest values range size so it's probably in our favor to start with it and break down the problem into smaller problems from there. It is clear that

$$\#(h,q,d,n,p)_{100} = \#(q,d,n,p)_{100} + \#(q,d,n,p)_{50} + \#(q,d,n,p)_0$$

because $\#(q,d,n,p)_{100}$ comes from h taking value 0, $\#(q,d,n,p)_{50}$ from h taking value 1 and $\#(q,d,n,p)_0$ from h taking value 2. Continuing down this path and breaking down the q cases:

$$\begin{aligned}\#(q,d,n,p)_{100} &= \#(d,n,p)_{100} + \#(d,n,p)_{75} + \\ &\quad \#(d,n,p)_{50} + \#(d,n,p)_{25} + \#(d,n,p)_0 \\ \#(q,d,n,p)_{50} &= \#(d,n,p)_{50} + \#(d,n,p)_{25} + \#(d,n,p)_0 \\ \#(q,d,n,p)_0 &= \#(d,n,p)_0\end{aligned}$$

This is getting tedious though. Maybe we can find a closed-form formula for $\#(d,n,p)_x$. Let's try to find one for $\#(n,p)_x$ first.

Lemma 13.1. *If $x = 5y$ then*

$$\#(n,p)_x = y + 1$$

Proof. n can take values in range $0,\ldots,y$ and for each value of n there is only one possible value of $p = x - 5n$. □

Lemma 13.2. *If x is a multiple of 5 then*

$$\#(d,n,p)_x = \begin{cases}(y+1)^2 & \text{if } x = 10y \\ (y+1)(y+2) & \text{if } x = 10y+5\end{cases}$$

Proof. Let's deal with the case $x = 10y$ first.

$$\begin{aligned}
\#(d,n,p)_x &= \sum_{k=0}^{y} \#(n,p)_{x-10k} \\
&= \sum_{k=0}^{y} \#(n,p)_{10(y-k)} \\
&= \sum_{k=0}^{y} \#(n,p)_{10k} \\
&= \sum_{k=0}^{y} (2k+1) \\
&= (y+1)^2
\end{aligned}$$

The case $x = 10y + 5$ is established in a similar fashion. □

We can now use the two lemmas to compute the number of combinations:

$$\begin{aligned}
\#(h,q,d,n,p)_{100} &= \#(d,n,p)_{100} + \#(d,n,p)_{75} + \\
&\quad 2\#(d,n,p)_{50} + 2\#(d,n,p)_{25} + 3\#(d,n,p)_0 \\
&= 11^2 + 8*9 + 2*6^2 + 2*3*4 + 3 \\
&= 292
\end{aligned}$$

14
Penn & Teller Full Deck of Cards

Counting words with constraints is the topic of the problem in this note. This problem was posed by a coworker at a lunch discussion.

Problem

When you go see the Penn & Teller Magic Show in Las Vegas you can get a random card from the **Perfectly Ordinary Deck of Cards** at the entrance. How many times do you have to see the show to collect the full deck.

We assume the supply of cards at the entrance is endless and thoroughly shuffled. This allows us to work with a probability model of drawing with replacement where each card is equally likely to be drawn with probability $\frac{1}{52}$. At each visit we draw a card[1]. After k visits we have built up a sequence of cards which we model as a word $w_k = (c_1, c_2, \ldots, c_k)$ of size k from an alphabet of size 52 ($\forall c_i : c_i \in \{1, \ldots, 52\}$). Our random variable X is the number of visits needed to achieve a full deck. The probability $P(X = k)$ means that it took k visits to achieve the full deck.

The key observation is this: if it took k visits to achieve full deck then at visit $k-1$ the corresponding word of cards $w_{k-1} = (c_1, c_2, \ldots, c_{k-1})$ is missing just one card and all the other cards appear at least once in the word[2].

Let $\mathcal{A} = \{1, 2, \ldots, 52\}$ be our alphabet and $L_k(\mathcal{A}) = \{(c_1, c_2, \ldots, c_k) : \forall i : 1 \leq i \leq k : c_i \in \mathcal{A}\}$ the set of all the words of length k with letters (cards) from the alphabet $\mathcal{A}$. Let $M_k(c)$ be the set of words of length k where letter c does not occur in the word and every other letter occurs at least once:

[1] You can get two cards at each visit to the show in Las Vegas. Drawing only one card is a simplification to keep the expressions smaller. The case with two cards is similar but the expressions get a little bigger because you have more cases of the card sequence right before the visit that achieves full deck. We will point out the differences at the end of this note.

[2] It might be easier to see this with a concrete card. Imagine you are drawing an ace of spades at visit k and getting the full deck. This means that before the visit k you are still missing the ace of spades. If not, then drawing the ace of spades at visit k wouldn't complete the deck. Getting the full deck at visit k also means that you are not missing any other cards before the visit k, otherwise if one of the other cards would be missing then drawing an ace of spades again wouldn't complete the deck.

$$M_k(c) = \{(c_1, c_2, \ldots, c_k) : (\forall i : 1 \leq i \leq k : c_i \in \mathcal{A} \wedge c_i \neq c) \wedge (\forall d \in \mathcal{A} \setminus \{c\} : \exists i : 1 \leq i \leq k : c_i = d)\}$$

So now let us assume that at visit k we draw letter c and get the full deck. From our previous argument above we know that then the word w_{k-1} we built in the previous $k-1$ visits has to be in $M_{k-1}(c)$.

The probability that at visit k we draw letter c and get the full deck is thus the probability of drawing card c times the probability that $w_{k-1} \in M_{k-1}(c)$. It follows that:

$$P(X = k) = \sum_{c \in \mathcal{A}} \frac{1}{|\mathcal{A}|} \frac{|M_{k-1}(c)|}{|L_{k-1}(\mathcal{A})|}$$

As we will see below, $|M_{k-1}(c)|$ is the same for all cards c, so for some fixed card c_0 we have $\forall c \in \mathcal{A} : |M_{k-1}(c)| = |M_{k-1}(c_0)|$. Then

$$P(X = k) = \frac{|M_{k-1}(c_0)|}{|L_{k-1}(\mathcal{A})|} \sum_{c \in \mathcal{A}} \frac{1}{|\mathcal{A}|} = \frac{|M_{k-1}(c_0)|}{|L_{k-1}(\mathcal{A})|}$$

We already know that

$$|L_{k-1}(\mathcal{A})| = |\mathcal{A}|^{k-1}$$

What is left to do to compute $P(X = k)$ is count $|M_{k-1}(c_0)|$. To avoid carrying around the $k-1$ we will count $|M_k(c_0)|$ instead and adjust afterwards.

How do we compute $|M_k(c_0)|$? We have two constraints on the words in $w_k = (c_1, c_2, \ldots, c_k) \in M_k(c_0)$:

$$\text{constraint } C_1 : \forall i : 1 \leq i \leq k : c_i \in \mathcal{A} \wedge c_i \neq c_0$$
$$\text{constraint } C_2 : \forall d \in \mathcal{A} \setminus \{c_0\} : \exists i : 1 \leq i \leq k : c_i = d$$

Constraint C_1 is easy to satisfy: we just use the alphabet without letter c_0: $\mathcal{A}_1 = \mathcal{A} \setminus \{c_0\}$.

For constraint C_2 we could try to eliminate all the subsets of $L_k(\mathcal{A}_1)$ with words missing one letter from $\mathcal{A}_1$, all the subsets with words missing two letters from $\mathcal{A}_1$ and so on all the way to subsets with words missing all but one letter from $\mathcal{A}_1$. Something like this:

$$|M_k(c_0)| = |L_k(\mathcal{A}_1)| - \sum_{\mathcal{B} \subset \mathcal{A}_1} |L_k(\mathcal{A}_1 \setminus \mathcal{B})|$$

But we have to be careful here. The subsets that we aim to eliminate are not disjoint and this would lead to overcounting[3]. So instead the correct way to count this is:

$$|M_k(c_0)| = \sum_{i=0}^{|\mathcal{A}_1|-1} (-1)^i \sum_{\mathcal{B}\subset\mathcal{A}_1, |\mathcal{B}|=i} |L_k(\mathcal{A}_1 \setminus \mathcal{B})|$$

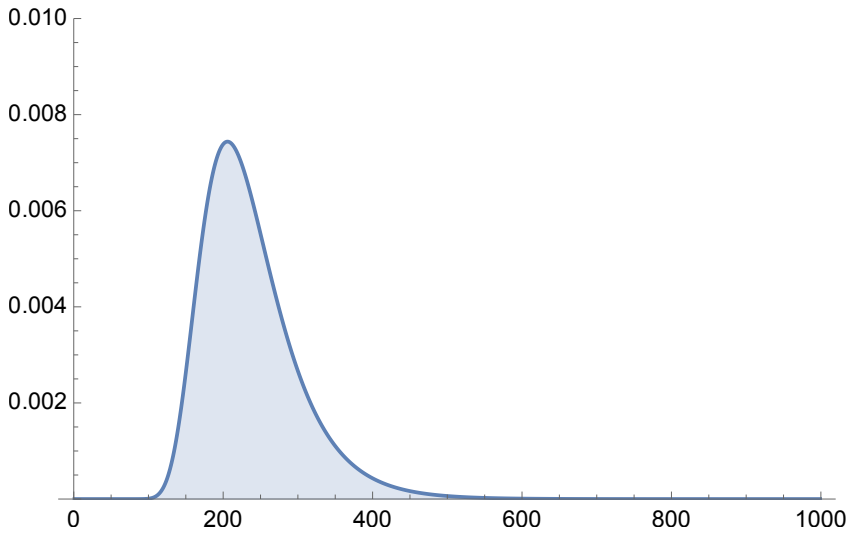
There are $\binom{|\mathcal{A}_1|}{i}$ subsets $\mathcal{B}$ of size i and for each $|L_k(\mathcal{A}_1 \setminus \mathcal{B})| = (|\mathcal{A}_1| - i)^k$. It follows that

$$|M_k(c_0)| = \sum_{i=0}^{|\mathcal{A}_1|-1} (-1)^i \binom{|\mathcal{A}_1|}{i} (|\mathcal{A}_1| - i)^k$$

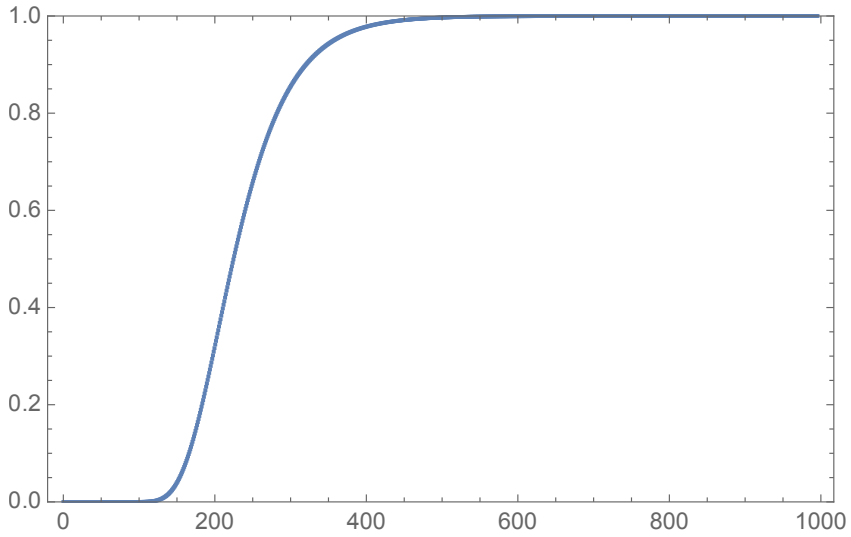
We have all the pieces now. We can adjust back to $k-1$ and also use $|\mathcal{A}| = 52$ to make a nice, closed formula for $P(X = k)$:

$$P(X = k) = \frac{1}{52^{k-1}} \sum_{i=0}^{50} (-1)^i \binom{51}{i} (51 - i)^{k-1}$$

Plugging this into Mathematica we can see the distribution and the cumulative distribution:



[3] Subsets with two missing letters are also subsets with one missing letter. Subsets with three missing letters are also subsets with two missing letters and subsets with one missing letter. And so on. It is indeed a use case for the inclusion-exclusion principle in combinatorics:
Wikipedia. Inclusion–exclusion principle — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Inclusion%E2%80%93exclusion%20principle&oldid=1086507513, 2022. [Online; accessed 07-May-2022].



It looks like around 400 visits will most likely get you a full deck. It might be cheaper to buy the full deck from the **Penn & Teller** online store.

It is always a good idea to double-check our result with a simulation[4]. The Python program below runs trials and records the number of visits necessary for a full deck:

[4] This is how I discovered a bug in my initial calculation. The listing is from my coworker that suggested the problem.

Listing 14.1: Simulation

```
import random

def trial():
    cards = [False] * 52
    visits = 0
    while not all(cards):
        visits += 1
        cards[random.randint(0, 51)] = True
    return visits

def main():
    num_trials = 100000
    trials = [trial() for _ in range(0, num_trials)]
    trials.sort()
    print(f"n={num_trials}")
    print(f"p0:   {trials[0]}")
    print(f"p25:  {trials[num_trials >> 2]}")
    print(f"p50:  {trials[num_trials >> 1]}")
    print(f"p75:  {trials[(num_trials >> 2) * 3]}")
    print(f"p100: {trials[num_trials - 1]}")

if __name__ == "__main__":
    main()
```

The results confirm that at least we are not orders of magnitude off:

```
n=100000
p0:   95
p25:  190
p50:  225
p75:  269
p100: 822
```

As promised, what are the differences when two cards are drawn at each visit. The constraints on the words right before the visit that achieves full deck are a little bit more complicated: there could be one or two cards missing. When one card is missing, the missing card could come in once or twice on the last visit. That's more or less it. Working out a closed formula for this is left as an exercise to the reader.

15
Points on circle

Problem

N distinct points, numbered from 0 onwards, are located on a circle (in the rest of this problem all point numbers are taken **mod**N). Point $i+1$ is the clockwise neighbor of point i. An integer array, $dist[0 \ldots N)$, is given such that $dist.i$ is the distance (along the circle) between points i and $i+1$. Derive a program to determine whether four of these points form a rectangle.

We adopt the same notation used in *Programming in the 1990s* [1] and *Programming, The Derivation of Algorithms*[2]: The notation of function application is the "dot" notation with name of function, followed by arguments, each separated by a dot. The notation of quantified expressions has the operator followed by the bounded variables, then a colon followed by the range for the bounded variables and ended with a colon and the actual expression. So

$$(\sum k : i \leq k < j : x_k)$$

corresponds to the more classical mathematical notation $\sum_{k=i}^{j-1} x_k$.
For our derivation steps in predicate calculus we will use the following notation:

$$\begin{array}{ll} & A \\ = & \{\text{reason why A equals B}\} \\ & B \\ \leq & \{\text{reason why B is less than C}\} \\ & C \end{array}$$

We are asked to solve S in

$\|[$

[1] Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990

[2] A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990

con $N : \ int;\ \{N \geq 4\}$
  $dist(i : 0 \leq i < N) : \ int;\ \{\forall i : 0 \leq i < N : dist.i > 0\}$
var $r : bool$;
  S
$\{r : r \equiv (\exists$ 4 points that form a rectangle$)\}$
]||

Let's first develop a more manageable postcondition. Evidently four points that form a rectangle is equivalent to two pairs of diametral opposing points. We introduce a function for the set of all indices from point x to point y in clockwise direction along the circle:

$$I : [0, \ldots, N) \to [0, \ldots, N) \to 2^{[0,\ldots,N)}$$
$$I.x.y := \begin{cases} [x, \ldots, y) & , \ x \leq y \\ [x, \ldots, N) \bigcup [0, \ldots, y) & , \ x > y \end{cases}$$

Let C be the circumference of the circle. We define function

$$f : [0, \ldots, N) \to [0, \ldots, N) \to int$$
$$f.x.y := C - 2(\sum i : i \in I.x.y : dist.i)$$

We want to find the number of diametral opposing pairs of points:

||[
  con $N : \ int;\ \{N \geq 2\}$
    $dist(i : 0 \leq i < N) : \ int;\ \{\forall i : 0 \leq i < N : dist.i > 0\}$
  var $r : int$;
    S
  $\{r : r = (\#\, x, y : 0 \leq x < N,\ 0 \leq y < N : f.x.y = 0)\}$
]||

Lemma 15.1. *The function f is increasing in its first argument and decreasing in its second argument.*

Proof. f is increasing in its first argument:

$$\begin{array}{ll} & f.(x+1).y \\ = & \{\text{definition of } f\} \\ & C - 2(\sum i : i \in I.(x+1).y : dist.i) \\ = & \{I.(x+1).y = I.x.y \setminus \{x\}\} \\ & C - 2((\sum i : i \in I.x.y : dist.i) - dist.x) \\ = & \{\text{definition of } f\} \\ & f.x.y + 2dist.x \\ > & \{dist.x > 0\} \\ & f.x.y \end{array}$$

f is decreasing in its second argument:

$$\begin{array}{ll} & f.x.(y+1) \\ = & \{\text{definition of } f\} \\ & C - 2(\sum i : i \in I.x.(y+1) : dist.i) \\ = & \{I.x.(y+1) = I.x.y \cup \{y\}\} \\ & C - 2((\sum i : i \in I.x.y : dist.i) + dist.y) \\ = & \{\text{definition of } f\} \\ & f.x.y - 2dist.y \\ < & \{dist.y > 0\} \\ & f.x.y \end{array}$$

□

Looking at the postcondition

$$\{r : r = (\#\, x, y : 0 \leq x < N,\ 0 \leq y < N : f.x.y = 0)\}$$

we define the function

$$G.a.b = (\#\, x, y : a \leq x < N,\ b \leq y < N : f.x.y = 0)$$

and we will maintain the invariants:

$$\begin{array}{lcl} P_0 & : & G.0.0 = r + G.a.b \\ P_1 & : & 0 \leq a \leq N \\ P_2 & : & 0 \leq b \leq N \end{array}$$

The initial values $r, a, b := 0, 0, 0$ satisfy the invariants and

$$a = N \vee b = N \Rightarrow G.a.b = 0 \Rightarrow r = G.0.0$$

establishes the postcondition, so we can stop when $a = N \vee b = N$.
So far we have

```
[||
    con N : int; {N ≥ 4}
        dist(i : 0 ≤ i < N) : int; {∀i : 0 ≤ i < N : dist.i > 0}
    var a, b, r : int;
    a, b, r := 0, 0, 0;
    do a ≠ N ∧ b ≠ N
       S
    od
    {r : r = G.0.0}
||]
```

We need to increment a, b and maintain the invariants:

$$\begin{array}{ll} & G.a.b \\ = & \{\text{definition of } G\} \\ & (\#\, x,y : a \le x < N,\ b \le y < N : f.x.y = 0) \\ = & \{\text{range split } x = a\} \\ & G.(a+1).b + (\#y : b \le y < N : f.a.y = 0) \\ = & \{f \text{ is decreasing in second argument (15.1), and assume } f.a.b < 0\} \\ & G.(a+1).b \end{array}$$

so $f.a.b < 0 \Rightarrow G.a.b = G.(a+1).b$. Similarly

$$\begin{array}{ll} & G.a.b \\ = & \{\text{definition of } G\} \\ & (\#\, x,y : a \le x < N,\ b \le y < N : f.x.y = 0) \\ = & \{\text{range split } y = b\} \\ & G.a.(b+1) + (\#x : a \le y < N : f.x.b = 0) \\ = & \{f \text{ is increasing in second argument (15.1), and assume } f.a.b > 0\} \\ & G.a.(b+1) \end{array}$$

so $f.a.b > 0 \Rightarrow G.a.b = G.a.(b+1)$. Also for the case $f.a.b = 0$ we have

$$\begin{array}{ll} & r + G.a.b \\ = & \{\text{definition of } G\} \\ & r + (\#\, x,y : a \le x < N,\ b \le y < N : f.x.y = 0) \\ = & \{\text{range split } x = a\} \\ & r + G.(a+1).b + (\#y : b \le y < N : f.a.y = 0) \\ = & \{f \text{ is decreasing in second argument (15.1), and assume } f.a.b = 0\} \\ & (r+1) + G.(a+1).b \end{array}$$

Our program becomes

$$\begin{array}{l} [\!|\, [\\ \quad \textbf{con } N : \mathit{int};\ \{N \ge 4\} \\ \qquad \mathit{dist}(i : 0 \le i < N) : \mathit{int};\ \{\forall i : 0 \le i < N : \mathit{dist}.i > 0\} \\ \quad \textbf{var } a, b, r : \mathit{int}; \\ \quad a, b, r := 0, 0, 0; \\ \quad \textbf{do } a \ne N \wedge b \ne N \\ \qquad \textbf{if} \\ \qquad \Box\ f.a.b > 0 \to b := b + 1 \\ \qquad \Box\ f.a.b < 0 \to a := a + 1 \\ \qquad \Box\ f.a.b = 0 \to a, r := a + 1, r + 1 \\ \qquad \textbf{fi} \\ \quad \textbf{od} \\ \quad \{r : r = G.0.0\} \\ ]\, |\!| \end{array}$$

We cannot have f in the program text so the last thing we have to do is eliminate f. We do this by introducing a new variable $c : \mathit{int}$

and maintaining the additional invariant $P_3 : c = f.a.b$. Lemma 15.1 already showed us the expressions for f when the first or the second argument increase, so our final program looks like this[3]

[3] The program is bound by the function $2N - a - b$ so it is $O(N)$. The solution is an example of the slope search technique.

```
||[
    con N : int; {N ≥ 4}
        dist(i : 0 ≤ i < N) : int; {∀i : 0 ≤ i < N : dist.i > 0}
    var a, b, c, r : int;
    a, b, c, r := 0, 0, C, 0;
    do a ≠ N ∧ b ≠ N
        if
        □ c > 0 → b, c := b + 1, c − 2dist.b
        □ c < 0 → a, c := a + 1, c + 2dist.a
        □ c = 0 → a, c, r := a + 1, 2dist.a, r + 1
        fi
    od
    {r : r = G.0.0}
]||
```

16
Prison Cells

Problem

A prison has n cells with all cell doors shut initially. The warden is a little weird so he walks the whole row of cells and opens every cell door. Then he walks the whole row again and shuts every other cell door. Then he walks the whole row again and opens every third door then walks the row again and shuts every 4th door etc. You can assume that the doors are numbered 0 to $(n-1)$ and the warden always starts at zero and walks them in order. Which doors will stay open when the warden is done ?

Each time the warden walks the row of cells he toggles the state (open or close) of some of the cells. It is clear then that the number of toggles to one cell determines if it is open or closed in the end. In the beginning each cell door is closed so if the number of toggles is even then it stays closed, if it is odd then it is open at the end.

The goal then is to calculate the number of toggles for a cell. The cells are numbered 0 to $(n-1)$ so lets try to calculate the number of toggles for cell k. The first time the warden walks the row of cells he toggles each cell including our cell k. The second time he toggles cells $0, 2, 4, \ldots$. That means he toggles cell k if k is even. The third time around he toggles cells $0, 3, 6, \ldots$ so he toggles cell k if k is a multiple of 3. If we continue we see that the cell k gets toggled on the warden's d walk if k is a multiple of d or said differently if d divides k.

It follows that the number of toggles $T(k)$ for cell k is

$$T(k) = \sum_{d|k} 1.$$

This is already pretty good but for the expression above it's not so

obvious for which k $T(k)$ will be even and for which it will be odd. So we will make a short excursion into basic number theory in the hopes that we can transform the expression into something more revealing.

A little number theory

We say that two integers m and n are *relatively prime* if the only common divisors are ± 1 and we write $(m, n) = 1$ in that case.

Definition 16.1. A function $f : \mathbb{N} \to \Omega$ with Ω a field is said to be **weakly multiplicative** if

$$\forall\ m, n \in \mathbb{N} : (m, n) = 1 \ \Rightarrow\ f(mn) = f(m)f(n).$$

Theorem 16.2. *If f is a weakly multiplicative function then so is the function*

$$g(n) = \sum_{d|n} f(d).$$

Proof. Let $m_1, m_2 \in \mathbb{N}$ with $(m_1, m_2) = 1$. Let's define two sets

$$S_1 = \{d\ :\ d \mid m_1 m_2\},\ S_2 = \{d_1 d_2\ :\ d_1 \mid m_1 \wedge d_2 \mid m_2\}.$$

It is obvious that $S_2 \subseteq S_1$. On the other hand

$\forall x \in S_1 \rightsquigarrow x \mid m_1 m_2$ (by definition)
Let $k = (x, m_1)$, so $x = yk, m_1 = zk$, for some $y, z \in \mathbb{N}$ and $(y, z) = 1$
$x \mid m_1 m_2 \rightsquigarrow yk \mid zkm_2 \rightsquigarrow y \mid m_2$ because $(y, z) = 1$
This means $x = yk \in S_2$ because $y \mid m_2 \wedge k \mid m_1$.

So we have $S_1 = S_2$. We can now write

$$
\begin{aligned}
& g(m_1 m_2) \\
= & \quad < \text{definition of g} > \\
& (\sum d : d \mid m_1 m_2 : f(d)) \\
= & \quad < \text{index sets } S_1 = S_2 \text{ so we change bounded variables} > \\
& (\sum d_1, d_2 : d_1 \mid m_1 \wedge d_2 \mid m_2 : f(d_1 d_2)) \\
= & \quad < \text{f is weakly multiplicative and } (d_1, d_2) = 1 > \\
& (\sum d_1, d_2 : d_1 \mid m_1 \wedge d_2 \mid m_2 : f(d_1) f(d_2)) \\
= & \quad < \text{nesting} > \\
& (\sum d_1 : d_1 \mid m_1 : (\sum d_2 : d_2 \mid m_2 : f(d_1) f(d_2))) \\
= & \quad < \text{multiplication distributes over addition} > \\
& (\sum d_1 : d_1 \mid m_1 : f(d_1)(\sum d_2 : d_2 \mid m_2 : f(d_2))) \\
= & \quad < \text{definition of g} > \\
& (\sum d_1 : d_1 \mid m_1 : f(d_1) g(m_2)) \\
= & \quad < \text{multiplication distributes over addition} > \\
& (\sum d_1 : d_1 \mid m_1 : f(d_1)) g(m_2) \\
= & \quad < \text{definition of g} > \\
& g(m_1) g(m_2).
\end{aligned}
$$

which proves the theorem. □

The theorem tells us that the function $T(k)$ which is the number of toggles for cell k

$$T(k) = \sum_{d|k} 1.$$

is in fact a weakly multiplicative function because the function inside the sum (the constant function 1) is trivially a weakly multiplicative function.

A more detailed solution

If we use the unique prime factorization of k

$$k = p_1^{a_1} p_1^{a_1} \ldots p_h^{a_h}$$

and use the fact that $(p_i^{a_i}, p_j^{a_j}) = 1$ we get

$$T(k) = \prod_{i=1}^{h} T(p_i^{a_i}).$$

But it's easy to see that $T(p_i^{a_i}) = a_i + 1$ so we have

$$T(k) = \prod_{i=1}^{h} (a_i + 1).$$

When is $T(k)$ even ? When any of the a_i are odd. To find out if a cell is open or closed do the prime factorization and look at the exponents of the primes. If any of them is odd then the cell stays closed.

17
0-1 Sequences

Counting inversions is the topic of the problem [1] in this note.

[1] Tung Kam Chuen. 0-1 sequences. 2016. URL https://open.kattis.com/problems/sequences

Problem

You are given a sequence, in the form of a string with characters '0', '1', and '?' only. Suppose there are k '?'s. Then there are 2^k ways to replace each '?' by a '0' or a '1', giving 2^k different 0-1 sequences (0-1 sequences are sequences with only zeroes and ones).
For each 0-1 sequence, define its number of inversions as the minimum number of adjacent swaps required to sort the sequence in non-decreasing order. In this problem, the sequence is sorted in non-decreasing order precisely when all the zeroes occur before all the ones. For example, the sequence 11010 has 5 inversions. We can sort it by the following moves: $11010 \to 11001 \to 10101 \to 01101 \to 01011 \to 00111$.
Find the sum of the number of inversions of the 2^k sequences, modulo $10^9 + 7$.

There are two ways to count the necessary inversions to sort the 2^k 0-1 sequences: we could count for each '0' how many '1' to its left are marching by in the right direction on their way to being sorted. Or we could count for each '1' how many zeros to its right are marching by in the left direction on their way to being sorted.

We arbitrary choose the first way of counting the inversions.

In the sequence $\mathbf{b} = (b_0, b_1, \ldots, bn-1)$ with characters '0', '1', and '?' we will look at each position i where $\mathbf{b}[i] =$ '0' and each position i where $\mathbf{b}[i] =$ '?'.

We define $q(i)$ to be the number of question marks to the left of i and $o(i)$ to be the number of ones to the left of i:

$$q(i) = |\{j : 0 \leq j < i : \mathbf{b}[j] = \text{'?'}\}|$$
$$o(i) = |\{j : 0 \leq j < i : \mathbf{b}[j] = \text{'1'}\}|$$

Let $s(i)$ be the number of inversions coming from $\mathbf{b}[i]$. When $\mathbf{b}[i] =$ '1' we set $s(i) = 0$ so as to not overcount[2].

When $\mathbf{b}[i] =$ '0' we know that all 2^k 0-1 sequences will have $o(i)$ ones to the left of i. These definitely will count in $s(i)$. We also need to consider all ones coming from setting '?' into '1' to the left of i. There are $q(i)$ possibilities here. For each $j : 1 \leq j \leq q(i)$ we can turn j question marks into ones. We have to choose the subset of size j of positions from the set of $q(i)$ positions with question marks[3]. It follows that:

$$s(i) = 2^k o(i) + 2^{k-q(i)}(\sum_{j=1}^{q(i)} \binom{q(i)}{j} j)$$

There is a neat way to simplify the sum with the binomial above using a combinatorial proof: Given a set of people of size N, count in how many ways you can choose a team and from that team choose a leader. There are two ways to count here. In the first way count the number of ways to choose a leader: N ways. Then count the number of ways to choose the rest of the team, which is the number of subsets from the set of people without the leader, so 2^{N-1}. In the second way for each possible team size, count the number of possible teams and then count the number of possible leader in that team. Because both ways count the same things, we have:

$$N2^{N-1} = \sum_{j=1}^{N} \binom{N}{j} j$$

Applied to our $s(i)$ we get:

$$\begin{aligned} s(i) &= 2^k o(i) + 2^{k-q(i)} q(i) 2^{q(i)-1} \\ &= 2^k o(i) + 2^{k-1} q(i) \end{aligned}$$

For $\mathbf{b}[i] =$ '?' we do a similar calculation[4], with the only difference being the number of question marks to the right of i: $2^{k-q(i)-1}$ (one less than in the previous calculation, since position i is a question mark). We get:

$$s(i) = 2^k o(i) + 2^{k-2} q(i)$$

The two cases cover all the counts and we can write the following loop (in Go, leaving out the modulus optimizations):

[2] We chose to count ones marching right, passing zeros.

[3] As a convenience we label the $q(i)$ positions with question marks as position $1, 2, \ldots q(i)$.

[4] We instantiate this question mark as a zero. The case where this position gets instantiated as a one is covered by other zero positions.

```
seen_ones := 0
seen_qmarks := 0
num_inversions := 0

for i:=0; i < n; i++ {
   switch {
   case b[i] == '0':
     num_inversions = 2^k * seen_ones + 2^(k-1) * seen_qmarks
   case b[i] == '1': seen_ones++
   case b[i] == '?':
     num_inversions = 2^k * seen_ones + 2^(k-2) * seen_qmarks
     seen_qmarks++
   }
}
```

For a more complete implementation in C++, see
https://github.com/uwedeportivo/kattis/tree/main/sequences.

18
Last three digits before decimal point

Recurrence relations and modulo arithmetic are the topics of the problem [1] in this note.

[1] Cosmin Negruseri. Codejam 2008 round 1a: Problem c: Numbers. 2008. URL https://code.google.com/codejam/contest/32016/dashboard#s=p2

Problem

Find the last three digits before the decimal point for the number $(3+\sqrt{5})^n$. For example, when $n = 5$, $(3+\sqrt{5})^5 = 3935.73982\ldots$, the answer is 935. For $n = 2$, $(3+\sqrt{5})^2 = 27.4164079\ldots$, the answer is 027. The value of n is in the range $2 \leq n \leq 2000000000$.

Looking at the numbers $(3+\sqrt{5})^n$, we can see that in general they are not integers. Ideally we would like to deal with integers. This sparks the idea of introducing the complement of $(3+\sqrt{5})$ into the mix, namely $(3-\sqrt{5})$. Let's look at the binomial expansion [2] of $(3+\sqrt{5})^n$:

[2] Binomial expansion:

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$$

$$(3+\sqrt{5})^n = \sum_{i=0}^{n} \binom{n}{i} 3^i (\sqrt{5})^{n-i}$$

Compare this to the binomial expansion of $(3-\sqrt{5})^n$:

$$(3-\sqrt{5})^n = \sum_{i=0}^{n} \binom{n}{i} 3^i (-\sqrt{5})^{n-i}$$

When $n-i$ is even, then $(\sqrt{5})^{n-i}$ and $(-\sqrt{5})^{n-i}$ are integers. When $n-i$ is odd, then the binomial terms for $(\sqrt{5})^{n-i}$ and $(-\sqrt{5})^{n-i}$ in the binomial expansions cancel each other out. So it follows that

$$\forall n \in \mathbb{N} : (3+\sqrt{5})^n + (3-\sqrt{5})^n \in \mathbb{N}$$

This is encouraging, so we define for all n:

$$a_n = (3+\sqrt{5})^n$$
$$b_n = (3-\sqrt{5})^n$$
$$c_n = a_n + b_n$$

We see that $\forall n \in \mathbb{N} : 0 < b_n < 1$, so $c_n = \lceil a_n \rceil$.

Concentrating on c_n, lets try to find the hundreds digit, the tens digit and the units digit of c_n.

Consider the polynomial:

$$(x-(3+\sqrt{5}))(x-(3-\sqrt{5})) = x^2 - 6x + 4$$

It leads to the recurrence relation: $f_n = 6f_{n-1} - 4f_{n-2}$, for which any linear combination of a_n and b_n is a solution [3]. We set the initial values of f_n such that the linear combination $c_n = a_n + b_n$ is the solution: $f_0 = 2, f_1 = 6$.
Therefore c_n satisfies the recurrence:

$$c_n = 6c_{n-1} - 4c_{n-2}$$
$$c_0 = 2$$
$$c_1 = 6$$

In theory we could just use this recurrence to compute c_n and then extract the hundreds digit, the tens digit and the units digit. Unfortunately this is not feasible for large n, since c_n grows quickly to very large values. But since we only need the last three digits of the values, we don't need to compute the values completely, computing them modulo 1000 will suffice.

Fortunately according to modulo arithmetic, the recurrence relation for c_n is still valid when doing modulo 1000. Let:

$$d_n \equiv c_n \mod 1000$$

and so

$$d_n \equiv 6d_{n-1} - 4d_{n-2} \mod 1000$$

Now consider the ordered pairs $(d_n, d_{n+1}), n \in \mathbb{N}$. Because $d_n \in \{0,1,2,\ldots,999\}$, there are only 10^6 distinct pairs of (d_n, d_{n+1}) possible. So it must be that there exist two indices $i, j \in \mathbb{N}^+$ such that:

$$(d_i, d_{i+1}) = (d_j, d_{j+1})$$

From the recurrence it follows that:

$$\forall k \in \mathbb{N} : (d_{i+k}, d_{i+k+1}) = (d_{j+k}, d_{j+k+1})$$

[3] In-depth treatment of recurrence relations can be found in Chapter 10, Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley, 3rd edition, 1993. ISBN 0201549832

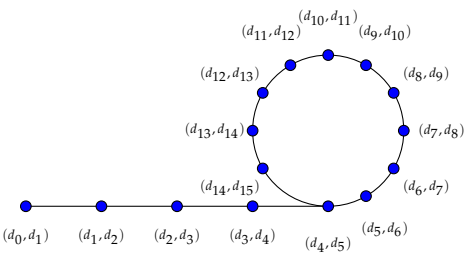


Figure 18.1: Periodic sequence of pairs preceded by a prefix of pairs.

The sequence of ordered pairs (d_n, d_{n+1}) is periodic with a period p of at most 10^6. We can construct a lookup table holding values of d_n from one period p and then compute d_n for large n by going into the lookup table at $n \mod p$.

The periodic part of the sequence doesn't necessarily start with the first pair in the sequence or with the second or the third etc... There might be a sequence prefix of ordered pairs that don't repeat before it goes into the sequence loop of repeating pairs. We write a function that computes the prefix and the period of the sequence using Floyd's cycle finding algorithm[4].

Luckily it turns out that in this case the prefix is $2, 6, 28$ and the periodic sequence has period 100.

With the lookup table we can compute d_n for any large n in constant time. d_n gives us the last three digits of c_n. Our goal though was to compute the last three digits before the decimal point of a_n.

We know $c_n = \lceil a_n \rceil$, so $c_n - 1 = \lfloor a_n \rfloor$. This means that to get the digits for a_n, we need to extract them from $d_n - 1$. Listing 18.1 has the complete Haskell implementation.

[4] Floyd's algorithm is described at https://en.wikipedia.org/wiki/Cycle_detection#Tortoise_and_hare. We use the Haskell implementation from https://wiki.haskell.org/Floyd's_cycle-finding_algorithm

Listing 18.1: Haskell code to compute last 3 digits

```
module Last3Digits(
    compute
) where

f :: Int -> Int -> Int
f a b = (6 * b - 4 * a) `mod` 1000

-- ds is our sequence of d_n
ds = 2 : 6 : zipWith (f) ds (tail ds)
-- this gives us the pairs
dps = zip ds (tail ds)

findCycle :: Eq a => [a] -> ([a],[a])
findCycle xxs = fCycle xxs xxs
  where fCycle (x:xs) (_:y:ys)
         | x == y             = fStart xxs xs
         | otherwise          = fCycle xs ys
        fCycle _      _       = (xxs,[]) -- not cyclic
        fStart (x:xs) (y:ys)
         | x == y             = ([], x:fLength x xs)
         | otherwise          = let (as,bs) = fStart xs ys in (x:as,bs)
        fLength x (y:ys)
         | x == y             = []
         | otherwise          = y:fLength x ys

tps = findCycle dps
-- ps is the prefix, cs the cycle of d_n
(ps, cs) = (map fst (fst tps), map fst (snd tps))
```

```
computeAux :: Int -> Int
computeAux n
    | n < (length ps) = ps!!n
    | otherwise       = cs!!((n - (length ps)) `mod` (length cs))

compute :: Int -> Int
compute n = computeAux n - 1
```

To check our computation we can use this Mathematica function:

```
In[14]:= last3Digits[n_Integer] := Mod[IntegerPart[(3 + Sqrt[5])^n], 1000]
```

19
How many trailing zeros in n!

Greatest dividing exponent and its properties is the topic of the problem in this note.

Problem

Write a program that calculates for an arbitrary positive integer n how many trailing zeros there are in $n!$.

Let's first try to figure out for any natural number n what the number of trailing zeros is. A useful concept here is the greatest dividing exponent [1]:

[1] Eric W. Weisstein. Greatest dividing exponent. From MathWorld—A Wolfram Web Resource. URL http://mathworld.wolfram.com/GreatestDividingExponent.html

Definition 19.1. The greatest dividing exponent $gde(n, b)$ of a base b with respect to a number n is the largest integer value of k such that $b^k \mid n$, where $b^k \leq n$.

Lemma 19.2.

$$gde(n, ab) = min(gde(n, a), gde(n, b)), \text{ with } (a, b) = 1$$

Proof. Assume $gde(n, a) \leq gde(n, b)$. Then $a^{gde(n,a)} \mid n$ and $b^{gde(n,a)} \mid n$, with $(a^{gde(n,a)}, b^{gde(n,a)}) = 1$, so $(ab)^{gde(n,a)} \mid n$. By definition of gde we then have $gde(n, a) \leq gde(n, ab)$.

We also have $(ab)^{gde(n,ab)} \mid n$, so $a^{gde(n,ab)} \mid n$. By definition of gde we then have $gde(n, a) \geq gde(n, ab)$.

It follows that $gde(n, a) = gde(n, ab)$. □

It's clear that the number of trailing zeros of n equals $gde(n, 10)$. From lemma 19.2 we are looking for $min(gde(n!, 2), gde(n!, 5))$.

Lemma 19.3.

$$gde(n!, p) = \sum_{k=1}^{\lfloor log_p n \rfloor} \left\lfloor \frac{n}{p^k} \right\rfloor, \textit{for a prime } p \leq n$$

Proof. We define the following subsets of $\{1, \ldots, n\}$:

$$M_p^k = \{i : 1 \leq i \leq n : p^k \mid i\}$$

For $k > \lfloor log_p n \rfloor$ the sets M_p^k are empty, so we only consider $k \leq \lfloor log_p n \rfloor$. Each member of one set M_p^k contributes k to $gde(n!, p)$, so the whole set contributes $k|M_p^k|$. From $p^k \mid i$ it follows that also $p^{k-1} \mid i$, so $M_p^k \subseteq M_p^{k-1}$ for $k = 2, \ldots, \lfloor log_p n \rfloor$. Being careful not to count the contributions more than once we get:

$$gde(n!, p) = \sum_{k=1}^{\lfloor log_p n \rfloor} |M_p^k|$$

With $|M_p^k| = \left\lfloor \frac{n}{p^k} \right\rfloor$ we conclude the proof. □

Lemma 19.4.

$$gde(n!, 2) \geq gde(n!, 5) \textit{ for any } n \geq 1$$

Proof. Plugging in the expression of *gde* from lemma 19.3 into the claim of this lemma we get:

$$gde(n!, 2) \geq gde(n!, 5) \Leftrightarrow \sum_{k=1}^{\lfloor log_2 n \rfloor} \left\lfloor \frac{n}{2^k} \right\rfloor \geq \sum_{k=1}^{\lfloor log_5 n \rfloor} \left\lfloor \frac{n}{5^k} \right\rfloor$$

We establish:

$$\begin{aligned} log_2 n \geq log_5 n &\Leftrightarrow log_2 n \geq log_2 n \; log_5 2 \\ &\Leftrightarrow 1 \geq log_5 2, \text{ which is true} \end{aligned}$$

For each $1 \leq k \leq \lfloor log_5 n \rfloor$ we have:

$$\left\lfloor \frac{n}{2^k} \right\rfloor \geq \left\lfloor \frac{n}{5^k} \right\rfloor$$

and for $\lfloor log_5 n \rfloor + 1 \leq k \leq \lfloor log_2 n \rfloor$ we have:

$$\left\lfloor \frac{n}{2^k} \right\rfloor > 0$$

Adding up the inequalities establishes the claim. □

From the three lemmas we found that:

$$\begin{aligned} \text{(number of trailing zeros in } n!) &= gde(n!, 10) \\ &= min(gde(n!, 2), gde(n!, 5)) \\ &= gde(n!, 5) \\ &= \sum_{k=1}^{\lfloor log_5 n \rfloor} \left\lfloor \frac{n}{5^k} \right\rfloor \end{aligned}$$

so our program needs to calculate the expression:

$$\sum_{k=1}^{\lfloor log_5 n \rfloor} \left\lfloor \frac{n}{5^k} \right\rfloor$$

The following small Haskell function does it:

Listing 19.1: Haskell code

```
gdefac :: Int -> Int

gdefac n = fst (until (\(x, y) -> y == 0)
                      (\(x, y) -> let
                                    y' = div y 5
                                  in (x + y', y'))
                      (0, n))
```

It works on tuples of numbers. It keeps dividing the second number in the tuple by 5 until zero and adding the division results together into the first number of the tuple. In the end it returns the first number in the tuple.

20
Twelve Coins

Coin weighings are the topics of the problem [1] in this note.

Problem

Of twelve coins, one is counterfeit and weighs either more or less than all the others. The others weigh the same. With a balance scale, on which one side may be weighed against the other, you are to use only three weighings to determine the counterfeit and its type (lighter or heavier).

We first present a hand-tailored solution for twelve coins.
Let M be the set of coins, $|M| = 12$. We have weighing function

$$w : M \to \{a, b\},\ a \neq b,\ a, b \in \mathbb{R}^+.$$

We have $|\{c \in M : w(c) = a\}| = 11$ and $|\{c \in M : w(c) = b\}| = 1$. We are asked to find $c_f \in M$ with $w(c_f) = b$ in three weighings.

For a subset $S \subseteq M$ we define

$$w(S) = \sum_{c \in S} w(c).$$

Let's partition M into 3 subsets S_0, S_1, S_2

$$\begin{aligned} &S_0 \cup S_1 \cup S_2 = M \\ &\forall\ 0 \leq i < 3 : |S_i| = 4 \\ &\forall\ 0 \leq i < j < 3 : S_i \cap S_j = \emptyset \end{aligned}$$

At this point we consume the first weighing:

1st weighing: compare $w(S_1)$ with $w(S_2)$

[1] Ethan Canin. *The Palace Thief Stories*, chapter Batorsag and Szerelem, page 87. Random House New York, 1994
or
Problem 1-111. on page 47 in N. Loehr. *Combinatorics*. Discrete Mathematics and Its Applications. CRC Press, 2017. ISBN 9781498780278

There are many variations of coin weighing problems. Some require identifying the type of the counterfeit (lighter or heavier), some don't. Some have more than one scale, some have more than one counterfeit, some don't state the existence of the counterfeit, some allow using scale weights or a known genuine coin. The Wikipedia page https://en.wikipedia.org/wiki/Balance_puzzle has a good overview. In this section we always want to find the counterfeit coin (we know there is exactly one counterfeit of unknown type) and its type and our scale is a balance scale with coins on both sides.

Case $w(S_1) = w(S_2)$

In this case $c_f \in S_0$. We partition S_0 into $S_0 = S_0^1 \cup S_0^3$ with $|S_0^1| = 1$ and $|S_0^3| = 3$. We also consider S_1^3, a subset of S_1 with $|S_1^3| = 3$. We consume the second weighing:

2nd weighing: compare $w(S_0^3)$ with $w(S_1^3)$

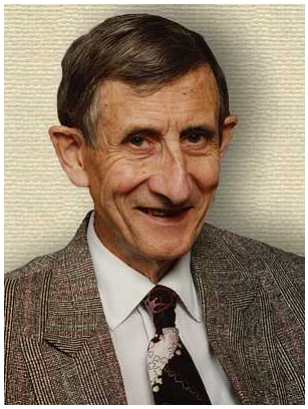
subcase 1: $w(S_0^3) = w(S_1^3)$. In this subcase $c_f \in S_0^1$ and we're done after just two weighings.

subcase 2: $w(S_0^3) > w(S_1^3)$. In this subcase S_0^3 has the counterfeit coin and $b > a$. We consume the third weighing: Let $S_0^3 = \{c_1, c_2, c_3\}$. We weigh c_1 against c_2.

3rd weighing: compare $w(c_1)$ with $w(c_2)$

If $w(c_1) = w(c_2)$ then $c_f = c_3$, if $w(c_1) > w(c_2)$ then $c_f = c_1$.

case 3: $w(S_0^3) < w(S_1^3)$. In this case S_0^3 has the counterfeit coin and $b < a$. Analog to previous case (replace heavy with light).



Freeman J. Dyson. https://en.wikipedia.org/wiki/Freeman_Dyson

Case $w(S_1) > w(S_2)$

In this case the counterfeit coin is either in S_1 or in S_2.
We consider 4 subsets:

$S_0^3 \subset S_0, |S_0^3| = 3$,

A with three coins from S_1,

B with one coin from S_2,

C with remaining coin from S_1: $C = S_1 \setminus A$.

We consume the second weighing:

2nd weighing: compare $w(S_0^3 \cup C)$ with $w(A \cup B)$

subcase 1: $w(S_0^3 \cup C) = w(A \cup B)$. In this subcase $c_f \in S_2 \setminus B$ and because $w(S_1) > w(S_2)$ we know that $b < a$. Let $\{c_1, c_2, c_3\} = S_2 \setminus B$ and we consume third weighing:

3rd weighing: compare $w(c_1)$ with $w(c_2)$

If $w(c_1) = w(c_2)$ then $c_f = c_3$, if $w(c_1) > w(c_2)$ then $c_f = c_2$.

subcase 2: $w(S_0^3 \cup C) < w(A \cup B)$. Assume $c_f \in C \subset S_1$. That would mean that $b > a$ because $w(S_1) > w(S_2)$ but that contradicts with $w(S_0^3 \cup C) = 3a + b < w(A \cup B) = 4a$. Assume $c_f \in B \subset S_2$. That

would mean that $b < a$ because $w(S_1) > w(S_2)$ but that contradicts also with $w(S_0^3 \cup C) = 4a < w(A \cup B) = 3a + b$. The only possibility remaining is $c_f \in A$. We use the ***third weighing*** analog to the previous case to find the counterfeit coin in a three-coin set using the fact that $b > a$.

subcase 3: $w(S_0^3 \cup C) > w(A \cup B)$. In this case the counterfeit coin can be either in B or in C. It cannot be in A according to a reasoning analog to previous case that leads to a contradiction. Both B and C only have one coin each so compare the coin in B with any good coin to find the counterfeit coin in this ***third weighing***.

This covers all the cases and we're done.

The solution brings up questions: how would it work for 13 coins and in general, how many weighings would be necessary for N coins. To answer these questions we present an elegant general solution[2] invented by Freeman J. Dyson.

[2] Freeman J. Dyson. Note 1931-The problem of the pennies. *Math. Gaz.*, 30:231–234, 1946
We follow an exposition of this solution by G. Shestopal.

In the hand-tailored solution for twelve coins we kept seeing a partition of coin sets into 3 subsets, two equally sized subsets that were weighed against each other and one subset that didn't participate in the current weighing. Depending on the result of the weighing and prior information from previous weighings we could narrow the search. Each weighing seems to contribute roughly three pieces of "information". A sequence of n weighings would generate 3^n amount of information that should somehow map to finding the counterfeit amongst N coins. This is not strictly accurate[3] because information deduced from weighings uses prior information from weighings before and the narrowing doesn't completely discard sets of coins but instead uses them as scale weights with known type (ie none are counterfeits). But this 3^n observation does suggest $3^n \approx N$ and point to the objective of finding a bijection between ternary codewords of length n and a set of size N.

[3] The hand-tailored solution is an **adaptive** solution: later weighings are set up according to the result of earlier weighings. Freeman J. Dyson's general solution is a **non-adaptive** solution if the number of coins is divisible by three: all the weighings are pre-determined regardless of their outcome.

Lets first consider N coins with N of the form $N = \frac{1}{2}(3^n - 3)$ for some $n \geq 2$. We will show how to find the counterfeit and its type in n weighings[4].

[4] Notice that $N = 12$ fits this form: $12 = \frac{1}{2}(3^3 - 3)$, so $n = 3$ weighings.

The set of codewords of length n from alphabet $\{0, 1, 2\}$ has size 3^n. We discard the three codewords with all digits equal: $0 \ldots 0$, $1 \ldots 1$ and $2 \ldots 2$. The set W of the remaining codewords has size $3^n - 3$. We split[5] W into $\frac{3^n-3}{2}$ pairs of complements:

[5] The expression $3^n - 3$ is the subtraction of two odd numbers, so it is even.

Definition 20.1. Two codewords $a_1 a_2 \ldots a_n \in W$ and $b_1 b_2 \ldots b_n \in W$ are **complements** of each other if

$$\forall i : 1 \leq i \leq n : a_i + b_i = 2$$

We use the notation $a_1^c a_2^c \ldots a_n^c$ for the complement of $a_1 a_2 \ldots a_n$.

Definition 20.2. The function $\delta : W \mapsto \{0,1,2\}^2$ finds the first two digits of a codeword that differ[6]

$$\delta(a_1a_2\dots a_n) = a_ia_{i+1} \text{ such that } \forall 1 \leq j < i : a_j = a_i \text{ and } a_i \neq a_{i+1}$$

[6] The function δ is well-defined because $0\dots0 \notin W$, $1\dots1 \notin W$ and $2\dots2 \notin W$.

Definition 20.3. Codeword $w \in W$ is called a **left** codeword if $\delta(w) \in \{10, 21, 02\}$ otherwise it is called a **right** codeword.

It is easy to see that the set of left codewords and the set of right codewords do not overlap and that if a codeword is a left codeword its complement is a right codeword[7]. A good mnemonic of left and right is this: $0 \to 1 \to 2$ moves digits to the right so $\{01, 12, 20\}$ corresponds to the right codewords and the right direction. Conversely $0 \leftarrow 1 \leftarrow 2$ moves digits to the left so $\{10, 21, 02\}$ corresponds to the left codewords and the left direction.

To summarize we now have $\frac{3^n-3}{2}$ pairs of codewords from W such that each pair has a left codeword and a right codeword that are complements of each other. Let P be the set of these pairs. We have $N = \frac{3^n-3}{2}$ coins. Let C be the set of coins. We pick an arbitrary bijection $\mu : C \mapsto P$ that assigns a pair to a coin, marking it with a left and a right codeword[8].

[7] For example let's prove that if $a_1a_2\dots a_n$ is a right codeword then $a_1^ca_2^c\dots a_n^c$ is a left codeword. If $a_j = a_{j+1}$ then $a_j^c = a_{j+1}^c$ and likewise if $a_j \neq a_{j+1}$ then $a_j^c \neq a_{j+1}^c$. So the index of the first two digits that differ is the same for a codeword and its complement. The complements of $\{01, 12, 20\}$ are exactly $\{21, 10, 02\}$ so the complement of a right codeword is a left codeword with differing digits at the same index as the right codeword.

We adopt the notation $\mu(c).right$ to denote the right codeword assigned to coin $c \in C$ and $\mu(c).left$ the left codeword. Also $\mu(c).right(i)$ denotes the i-th digit of the right codeword, so if $\mu(c).right = a_1a_2\dots a_n$ then $\mu(c).right(i) = a_i$. Analog $\mu(c).left(i)$ for the i-th digit of the left codeword assigned to coin c.

We define $\forall i : 1 \leq i \leq n$ and $\forall d : 0 \leq d \leq 2$ the subsets $C_i(d) \subset C$ with

$$C_i(d) = \{c \in C : \mu(c).right(i) = d\}$$

It is easy to see that $C = C_i(0) \cup C_i(1) \cup C_i(2)$ for each $1 \leq i \leq n$ and that $C_i(0) \cap C_i(1) = C_i(0) \cap C_i(2) = C_i(1) \cap C_i(2) = \emptyset$, so $C_i(0)$, $C_i(1)$ and $C_i(2)$ partition C.

Now we execute n weighings. For the i-th weighing we place the coins from $C_i(0)$ on the left pan of the scale and the coins from $C_i(2)$ on the right pan of the scale. We capture the n weighings in a codeword $x_1x_2\dots x_n \in W$: if in the i-th weighing the left pan sinks, then $x_i = 0$, if the weighing is balanced then $x_i = 1$ and if the right pan sinks then $x_i = 2$.

We claim that the weighing codeword $x_1x_2\dots x_n$ will be either the left or the right marker codeword of the counterfeit depending whether the counterfeit is lighter or heavier and prove the following theorem:

Theorem 20.4. *Let $x_1x_2\dots x_n$ be the result of the n weighings and let $c_f \in C$ be the counterfeit coin. Then $\mu(c_f).right = x_1x_2\dots x_n$ if c_f is heavier and $\mu(c_f).left = x_1x_2\dots x_n$ if c_f is lighter.*

[8] For example when $N = 12$ we could pick μ like this:

coin	left codeword	right codeword
c_1	211	011
c_2	100	122
c_3	022	200
c_4	212	010
c_5	101	121
c_6	020	202
c_7	210	012
c_8	102	120
c_9	021	201
c_{10}	221	001
c_{11}	110	112
c_{12}	002	220

$C_3(2)$ means the subset of coins for which the third digit in the right codeword is 2. In this example $C_3(2) = \{c_2, c_6, c_7, c_{11}\}$.

Proof. Assume the counterfeit c_f is lighter than the other coins (we deal with the case of c_f heavier afterwards).

There are three cases for the i-th weighing:

Case 1.(lighter) The scale is balanced, so $x_i = 1$. This means that c_f does not participate in the i-th weighing[9]: $c_f \notin C_i(0)$ and $c_f \notin C_i(2)$. Since $C_i(0)$, $C_i(1)$ and $C_i(2)$ partition C, we have $c_f \in C_i(1)$. By definition this means that $\mu(c_f).right(i) = x_i = 1$ and since right and left are complements it also means that $\mu(c_f).left(i) = x_i = 1$.

[9] Otherwise the scale wouldn't be balanced.

Case 2.(lighter) The left pan sinks, so $x_i = 0$. Here c_f participates and $c_f \in C_i(2)$ because it is lighter so on the right pan. By definition this means that $\mu(c_f).right(i) = 2$ and since $\mu(c_f).left(i)$ is the complement it also means $\mu(c_f).left(i) = x_i = 0$.

Case 3.(lighter) The pan sinks, so $x_i = 2$. In this case c_f also participates and $c_f \in C_i(0)$ because it is lighter so on the left pan. This means $\mu(c_f).right(i) = 0$ and by complement $\mu(c_f).left(i) = x_i = 2$.

For every i we have seen that $\mu(c_f).left(i) = x_i$, so $\mu(c_f).left = x_1x_2 \dots x_n$.

Now assume c_f is heavier. We proceed in analog fashion. There are three cases for the i-th weighing:

Case 1.(heavier) The scale is balanced, so $x_i = 1$. This means that c_f does not participate in the i-th weighing and we have $c_f \in C_i(1)$ and $\mu(c_f).right(i) = x_i = 1$.

Case 2.(heavier) The left pan sinks, so $x_i = 0$. Here c_f participates and $c_f \in C_i(0)$ because it is heavier so on the left pan. Then $\mu(c_f).right(i) = x_i = 0$.

Case 3.(heavier) The right pan sinks, so $x_i = 2$. In this case c_f also participates and $c_f \in C_i(2)$ because it is heavier so on the right pan. So $\mu(c_f).right(i) = x_i = 2$.

We see that for every i we have $\mu(c_f).right(i) = x_i$, so $\mu(c_f).right = x_1x_2 \dots x_n$. □

Theorem 20.4 shows that the n weighings detect the counterfeit coin and its type if there are $N = \frac{3^n-3}{2}$ coins.

If $N < \frac{3^n-3}{2}$ we have to be more careful how we mark coins with codewords[10].

Let $\pi : W \mapsto W$ be the function on the set of codewords W with $\pi(a_1a_2 \dots a_n) = b_1b_2 \dots b_n$ such that $\forall i : 1 \leq i \leq n : b_i \equiv (a_i + 1) \mod 2$.

Lemma 20.5. *The function π preserves the rightiness of codewords: if $w \in W$ is a right codeword, then $\pi(w)$ is also a right codeword.*

Proof. Let $a_1a_2 \dots a_n$ be a right codeword. By definition it means that $\delta(a_1a_2 \dots a_n) = a_ia_{i+1}$ with $\forall 1 \leq j < i : a_j = a_i$ and $a_ia_{i+1} \in \{01, 12, 20\}$. Let $\pi(a_1a_2 \dots a_n) = b_1b_2 \dots b_n$. This means that $\forall 1 \leq j < i : b_j \equiv a_j + 1 \equiv a_i + 1 \equiv b_i \mod 2$. So $\delta(b_1b_2 \dots b_n) = b_ib_{i+1}$. If $a_ia_{i+1} = 01$

[10] For example assume there are only ten coins and we arbitrarily assign the following codeword pairs:

coin	left codeword	right codeword
c_1	211	011
c_2	100	122
c_3	022	200
c_4	212	010
c_5	101	121
c_6	020	202
c_7	210	012
c_8	102	120
c_9	021	201
c_{10}	221	001

Then $C_2(2) = \{c_2, c_5, c_8\}$ and $C_2(0) = \{c_3, c_6, c_9, c_{10}\}$. This is a problem because the two sets that will be put on the scale in the two pans in the second weighing have different number of coins. We cannot deduce any information from this weighing anymore.

then $b_i b_{i+1} = 12$, if $a_i a_{i+1} = 12$ then $b_i b_{i+1} = 20$ and if $a_i a_{i+1} = 20$ then $b_i b_{i+1} = 01$. In all three cases $b_1 b_2 \dots b_n$ is also a right codeword. □

From the definition of π it is clear that $\pi^3(w) = w$. From this and lemma 20.5 it follows that π partitions the set of right codewords into subsets of size three: $\{w, \pi(w), \pi^2(w)\}$.

We now partition W into subsets of size six:

$$\{w, w^c, \pi(w), (\pi(w))^c, \pi^2(w), (\pi^2(w))^c\}$$

grouping π-generated right codewords and their left complements. The group with right codewords $00\dots01$, $11\dots12$ and $22\dots20$ we set aside and call the left-over group.

For the bijection $\mu : C \mapsto W$ we group coins in groups of three. For each group we pick a codeword group of six other than the left-over group and assign left and right complementing codewords to each coin in the group. If there are one or two coins left over from the grouping into threes, then we use the left-over codeword group to assign left and right codewords to those coins. If only one coin is left over we assign it the right codeword $11\dots12$ and if two are left over we assign the right codewords $00\dots01$ and $22\dots20$.

With μ defined this way if $N \equiv 1 \mod 3$ then the left-over coin has right codeword $11\dots12$ and will not participate in the first $n-1$ weighings. If $N \equiv 2 \mod 3$ the two left-over coins with right codewords $00\dots01$ and $22\dots20$ will both participate in every of the first $n-1$ weighings. This ensures that the left-over coins don't disrupt the first $n-1$ weighings. For a coin that is not in the left-over group it is easy to see that if it participates in a weighing then there is another coin in the same group (apply π twice to get to its right codeword) that also participates on the opposite pan. Overall we have satisfied the requirement $|C_i(0)| = |C_i(2)|$ for all $1 \le i < n$. The first $n-1$ weighings can proceed as before.

For the last weighing we have the following cases (here the solution turns adaptive, ie the setup for the last weighing depends on the outcome of the previous weighings):

The first two cases are for $N \equiv 1 \mod 3$, so one left-over coin c_l with right codeword $\mu(c_l).right = 11\dots12$.

Case 1. The first $n-1$ weighings yielded $x_1 x_2 \dots x_{n-1} = 11\dots1$ (the scale was balanced in all $n-1$ weighings). In this case we know the left-over coin is the counterfeit $c_l = c_f$ and we can just put it on one pan on the scale and any other coin on the other to determine its type.

Case 2. The first $n-1$ weighings yielded $x_1 x_2 \dots x_{n-1} \neq 11\dots1$. In this case we know the left-over coin is not the counterfeit. We can just leave it out and put $C_n(0)$ on the left pan and $C_n(2) \setminus \{c_l\}$ on the right

pan of the balance scale. The resulting complete weighing codeword will point to the counterfeit and its type.

The next cases are for for $N \equiv 2 \mod 3$, with two left-over coins c_{l0} and c_{l2} with right codewords $\mu(c_{l0}).right = 00\ldots01$ and $\mu(c_{l2}).right = 22\ldots20$.

Case 3. The first $n-1$ weighings yielded $x_1x_2\ldots x_{n-1} = 22\ldots2$. Both c_{l0} and c_{l2} participated in all $n-1$ weighings (according to their right codewords that start with $00\ldots0$ and $22\ldots2$ respectively). It means that either c_{l0} or c_{l2} is the counterfeit. In the last weighing we pit c_{l0} against any coin that is not c_{l2}. If the scale is balanced then c_{l2} is the counterfeit and it is heavier. If the scale tilts one way or the other it shows c_{l0} as a counterfeit and its type.

Case 4. The first $n-1$ weighings yielded $x_1x_2\ldots x_{n-1} = 00\ldots0$. Again both c_{l0} and c_{l2} participated in all $n-1$ weighings and again it means that either c_{l0} or c_{l2} is the counterfeit. In the last weighing we pit c_{l2} against any coin that is not c_{l0}. If the scale is balanced then c_{l0} is the counterfeit and it is heavier. If the scale tilts one way or the other it shows c_{l2} as a counterfeit and its type.

Case 5. The first $n-1$ weighings yielded $x_1x_2\ldots x_{n-1} \neq 00\ldots0$ and $x_1x_2\ldots x_{n-1} \neq 22\ldots2$. In this case both c_{l0} and c_{l2} cannot be counterfeits. In the last weighing we can just do $C_n(0)$ on the left pan and $C_n(2)$ on the right. The resulting complete weighing codeword will point to the counterfeit and its type.

This covers all the cases and shows that we can find the counterfeit coin and its type in n weighings if the number of coins $N \leq \frac{3^n-3}{2}$.

Is this optimal or does a strategy exist that needs less than n weighings?

To answer this question we want to find a lower bound for the number of weighings given N coins.

Theorem 20.6. *Given are N coins of which one is a counterfeit. If the counterfeit coin and its type can be found in n weighings using a non-adaptive strategy, then $2N \leq 3^n - 3$.*

Proof. There are n weighings producing a weighing codeword $x_1x_2\ldots x_n$ with $x_i = 0$ if left pan sinks in i-th weighing, $x_i = 1$ if i-th weighing is balanced and $x_i = 2$ if right pan sinks. We have 3^n possible codewords from n weighings.

Assume we have a non-adaptive strategy that finds the counterfeit coin and its type in n weighings. Since it is non-adaptive the information of which coin participates in which weighing on which pan is already pre-determined. Let's capture this information in a matrix $p \in \{0,1,2\}^{n\times N}$ which we call the **participation matrix**:

$$\forall 1 \leq i \leq n, 1 \leq j \leq N :$$
$$p_{ij} = \begin{cases} 0, & \text{coin } j \text{ on left pan in i-th weighing} \\ 1, & \text{coin } j \text{ not in i-th weighing} \\ 2, & \text{coin } j \text{ on right pan in i-th weighing} \end{cases}$$

The number of potential answers to the question of which coin is the counterfeit and what is its type is $2N$ (N coins and two possibilities for each coin - lighter or heavier). For notational convenience we define the index sequence $N_{\pm} = \{1, -1, 2, -2, \ldots, N, -N\}$. For all $1 \leq j \leq N$ index j means coin j is the counterfeit and it is heavier, index $-j$ means coin j is the counterfeit and it is lighter.

Given the participation matrix we define a matrix $a \in \{0, 1, 2\}^{n \times N_{\pm}}$. Cell a_{ij} tells us what result of weighing i keeps the answer that coin j is the heavier counterfeit as still a possibility. Cell $a_{i(-j)}$ tells us what result of weighing i keeps the answer that coin j is the lighter counterfeit as still a possibility. For example if $p_{ij} = 0$ then coin j is on the left pan in the i-th weighing and for coin j to be the counterfeit and heavier in the i-th weighing the left pan needs to sink, the corresponding weighing codeword component needs to be $x_i = 0$ and $a_{ij} = 0$.

In other words matrix a shows for each potential answer what needs to happen in each weighing so that the answer becomes the actual answer:

$$\forall 1 \leq i \leq n, 1 \leq j \leq N :$$
$$a_{ij} = \begin{cases} 0, & \text{if } p_{ij} = 0 \\ 1, & \text{if } p_{ij} = 1 \\ 2, & \text{if } p_{ij} = 2 \end{cases} \quad a_{i(-j)} = \begin{cases} 0, & \text{if } p_{ij} = 2 \\ 1, & \text{if } p_{ij} = 1 \\ 2, & \text{if } p_{ij} = 0 \end{cases}$$

The column j in matrix a is the weighing codeword required for making coin j and its type (from the sign of j) the actual answer.

For the strategy (represented by the participation matrix p) to work it needs to not rely on "luck", i.e. a given weighing codeword happens to be a column in matrix a. All the weighing codewords that can occur need to be columns in a exactly once.

Since the strategy is successful we know that each coin participates in at least one weighing[11], so the column vector $(1, 1, \ldots, 1)^T$ does not appear in p. This means that weighing codeword $x1x2 \ldots x_n = 11 \ldots 1$ cannot occur and we have $3^n - 1$ remaining weighing codewords that can happen.

[11] Otherwise there would be no way to find its type even if we find the counterfeit.

If $2N > 3^n - 1$ then it must be that a weighing codeword will appear more than once in a, so more than one potential answer could be the actual answer and we wouldn't know which one. So $2N \leq 3^n - 1$.

We observe that in each row i of a if $a_{ij} = 0$ then $a_{i(-j)} = 2$ and vice versa. That means that there are k_i zeros and k_i twos in row i, so

$2N - 2k_i$ ones. Also because an even number of coins participates in each weighing we have that k_i is even. So in each row we have an even number of zeros, ones and twos. There are 3^{n-1} possible weighing codewords that start with a one. That means at most 3^{n-1} columns in a can have a one in the first position. Since the number of ones is even and 3^{n-1} is odd we can have at most $3^{n-1} - 1$ ones in the first row. Similarly there are 3^{n-1} possible weighing codewords that start with a zero, so $k_1 \leq 3^{n-1} - 1$. We then have:

$$2N = 2N - 2k_1 + 2k_1 \leq 3^{n-1} - 1 + 2(3^{n-1} - 1) = 3^n - 3$$

□

Explain connection to Hamming codes for number of columns smaller than $3^n - 3$

21
Two decks of cards

Inclusion–exclusion principle and the number of derangements are the topics of the problem [1] in this section.

[1] Probability question on page ix in Preface of M. Beck and R. Geoghegan. *The Art of Proof: Basic Training for Deeper Mathematics*. Undergraduate Texts in Mathematics. Springer New York, 2010. ISBN 9781441970237

Problem

A deck of n different cards is shuffled and laid on the table by your left hand, face down. An identical deck of cards, independently shuffled, is laid at your right hand, also face down. You start turning up cards at the same rate with both hands, first the top card from both decks, then the next-to-top cards from both decks, and so on. What is the probability that you will simultaneously turn up identical cards from the two decks?

The shuffling implies equally likely outcomes so the probability is the number of outcomes with an identical card turning up divided by the number of total outcomes. The number of total outcomes is $(n!)^2$ since there are $n!$ possible shuffling outcomes of one deck (the number of permutations of S_n).

The set of outcomes where an identical card turns up seems harder to count. It feels easier to count its complement: the number of outcomes when no identical card comes up. There are $n!$ ways in which the first deck is shuffled. For a given permutation $\pi \in S_n$ of the first deck we need to count all the permutations $\rho \in S_n$ of the second deck for which $\forall i : 1 \leq i \leq n : \rho(i) \neq \pi(i)$. Let $A_\pi = \{\rho \in S_n : \forall i : 1 \leq i \leq n : \rho(i) \neq \pi(i)\}$.

Let $D_n = \{\tau \in S_n : \forall i : 1 \leq i \leq n : \tau(i) \neq i\}$. A permutations from D_n is called a **derangement**. We introduce the notation $!n = |D_n|$ for the number of derangements.

Lemma 21.1.

$$|A_\pi| = |D_n|$$

Proof. We need to present a bijection $f : D_n \to A_\pi$. We define $f(\tau) = \pi \circ \tau$. First we verify that f is well-defined, i.e. $f(\tau) \in A_\pi$.

$$\begin{aligned} &\forall i : 1 \leq i \leq n : \\ f(\tau)(i) &= (\pi \circ \tau)(i) \\ &= \pi(\tau(i)) \neq \pi(i) \\ &\text{because } \tau(i) \neq i \end{aligned}$$

Next we show that f is injective: $f(\tau_1) = f(\tau_2)$ implies $\pi \circ \tau_1 = \pi \circ \tau_2$. S_n is a group, so $\tau_1 = \tau_2$. Also f is surjective because: $\forall \rho \in A_\pi$ we have $\pi^{-1} \circ \rho \in D_n$ because $\rho(i) \neq \pi(i)$ implies $(\pi^{-1} \circ \rho)(i) \neq i$. $f(\pi^{-1} \circ \rho) = \rho$. □

From lemma 21.1 we now know that the number of outcomes when no identical card comes up is $!n \cdot n!$ and the probability requested in the problem is

$$P = 1 - \frac{!n \cdot n!}{n!^2} = 1 - \frac{!n}{n!}$$

What remains is to compute $!n$. Let us look again at the set of derangements: $D_n = \{\tau \in S_n : \forall i : 1 \leq i \leq n : \tau(i) \neq i\}$. It sometimes helps to consider the complement of a set when we have to compute its cardinality. To more precisely define the complement of D_n we will define the following subsets of S_n: $F_n(k) = \{\tau \in S_n : \tau(k) = k\}$. We then have:

$$D_n = S_n \setminus (\bigcup_{k=1}^{n} F_n(k))$$

For any given $k \in \{1, 2, \ldots, n\}$ we have $|F_n(k)| = (n-1)!$ (see footnote[2] why), so

[2] One position is fixed and the rest behave like a permutation in S_{n-1}.

$$|(\bigcup_{k=1}^{n} F_n(k))| = n(n-1)! = n!$$

But this can't be right. It would mean that $|D_n| = 0$ and $D_n = \emptyset$. But cleary D_n is not empty, for example the permutation:

$$\rho(i) = \begin{cases} i+1 & i < n \\ 1 & i = n \end{cases}$$

is a member of D_n. The problem here is that the $F_n(k)$ are not disjoint, so calculating the size of their union needs to be done more carefully. It turns out that this is a perfect use case of the inclusion-exclusion principle.

The **inclusion-exclusion principle** provides a method of counting the size of the union of subsets that are not necessarily disjoint.

We illustrate the method on a simple example of three sets A, B, C as in Figure 21.1. We would like to compute $|A \cup B \cup C|$. The expression $|A| + |B| + |C|$ would count the elements from $(A \cap B) \setminus (A \cap B \cap C)$, $(A \cap C) \setminus (A \cap B \cap C)$ and $(B \cap C) \setminus (A \cap B \cap C)$ twice and elements from $A \cap B \cap C$ three times. So $|A \cup B \cup C| < |A| + |B| + |C|$. To compensate we subtract the pairwise intersection sizes and our expression becomes $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C|$. This is almost right except for $A \cap B \cap C$ which we lost in the adjustment (it was counted three times and then subtracted three times). We add it back and get

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

In general, if we want to count $|\cup_{i=1}^{n} A_i|$ we start with $\sum_{i=1}^{n} |A_i|$ which includes pairwise intersections $A_i \cap A_j$ twice, so we exclude with $-(\sum_{1 \le i < j \le n} |A_i \cap A_j|)$. But this excludes the triple intersections so we include those with $\sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k|$. This overcounts quadruple intersections which we exclude etc. We stop with the exclusion or inclusion of the intersection of all A_i.

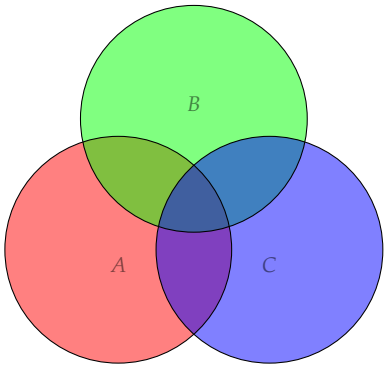


Figure 21.1: Union of three not necessarily disjoint sets.

Theorem 21.2. ***Inclusion-exclusion principle.***
Given n sets A_j, $1 \le j \le n$

$$|\bigcup_{j=1}^{n} A_j| = \sum_{k=1}^{n} (-1)^{k+1} (\sum_{J \subseteq \mathbb{N}_n,\ |J|=k} |\bigcap_{j \in J} A_j|)$$

Proof. We are going to prove the theorem by tracing the contributions of one element $a \in \bigcup_{j=1}^{n} A_j$ to the left-hand side and right-hand side of the equation. On the left it will be 1 since this is the union of sets and not multisets. For the right-hand side, we observe that there is a non-empty index set $I \subseteq \{1, 2, \ldots, n\}$ such that $\forall i \in I : a \in A_i$. The element a will contribute ± 1 from all the set intersections in which it appears, so all the terms in the sum where index set J satisfies $J \subseteq I$ (since these are intersections, a won't appear in any of the other terms). The hope is that the sum of all these ± 1 will be 1.

Let $m = |I|$. Then any index set J with size greater than m cannot be a subset of I, so the running index k of the sum only needs to go to m. For each k there are $\binom{m}{k}$ index subsets J of size k from I. In each a weighs in with 1, so the contributions of a add up to:

$$\sum_{k=1}^{m} (-1)^{k+1} \binom{m}{k}$$

We add and subtract $\binom{m}{0}$ to this sum and get

$$\begin{aligned}
\sum_{k=1}^{m}(-1)^{k+1}\binom{m}{k} &= \binom{m}{0} - \binom{m}{0} + \sum_{k=1}^{m}(-1)^{k+1}\binom{m}{k} \\
&= \binom{m}{0} - \sum_{k=0}^{m}(-1)^{k+1}\binom{m}{k} \\
&= \binom{m}{0} + \sum_{k=0}^{m}(-1)^{k}\binom{m}{k} \\
&= \binom{m}{0} + \sum_{k=0}^{m}1^{m-k}(-1)^{k}\binom{m}{k} \\
&= \binom{m}{0} + (1-1)^{m} \\
&= \binom{m}{0} = 1
\end{aligned}$$

On both sides of the equation in the theorem an arbitrary element a of the union of the sets contributes 1. This proves the inclusion-exclusion principle. □

Let us return to computing derangements. We now know how to compute $|(\bigcup_{k=1}^{n} F_n(k))|$ by using the inclusion-exclusion principle:

$$|(\bigcup_{k=1}^{n} F_n(k))| = \sum_{k=1}^{n}(-1)^{k+1}(\sum_{J\subseteq \mathbb{N}_n,\ |J|=k} |\bigcap_{j\in J} F_n(j)|)$$

For a given index set $J \subseteq \mathbb{N}_n$, $|J| = k$ the intersection contains all the permutations that are fixed in the positions $j \in J$, so the size of this intersection is $(n-k)!$. There are $\binom{n}{k}$ such index sets J of size k, so our expression becomes:

$$|(\bigcup_{k=1}^{n} F_n(k))| = \sum_{k=1}^{n}(-1)^{k+1}\binom{n}{k}(n-k)!$$

We then have

$$\begin{aligned}
!n &= n! - \sum_{k=1}^{n}(-1)^{k+1}\binom{n}{k}(n-k)! \\
&= n! + \sum_{k=1}^{n}(-1)^{k}\binom{n}{k}(n-k)! \\
&= \sum_{k=0}^{n}(-1)^{k}\binom{n}{k}(n-k)!
\end{aligned}$$

The probability of turning up identical cards from the two decks is

$$\begin{aligned}
1 - \frac{!n}{n!} &= 1 - \sum_{k=0}^{n} (-1)^k \binom{n}{k} \frac{(n-k)!}{n!} \\
&= 1 - \sum_{k=0}^{n} (-1)^k \frac{n!}{k!(n-k)!} \frac{(n-k)!}{n!} \\
&= 1 - \sum_{k=0}^{n} \frac{(-1)^k}{k!}
\end{aligned}$$

This probability converges fairly quickly to approximatively 0.7 so you have a 0.7 chance of turning up identical cards from the two decks.

22
While a

Loop invariants is the topic of the problem [1] in this note.

[1] Problem 4 on page 9 from A. Engel. *Problem-Solving Strategies*. Problem Books in Mathematics. Springer New York, 2013. ISBN 9781475789546. URL https://books.google.com/books?id=aUofswEACAAJ

Problem

We start with the state (a, b) where a, b are positive integers. To this initial state we apply the following algorithm:

```
while a > 0:
   if a < b:
      (a,b) = (2a, b - a)
   else:
      (a,b) = (a - b, 2b)
```

For which starting positions does the algorithm stop? In how many steps does it stop, if it stops? What can you tell about periods and tails?

We start with $a > 0$ and $b > 0$. We adopt the following notation: a_i, b_i are the values after $i \in \mathbb{N}_{\geq 0}$ times through the loop. Before the first time through the loop $a_0 = a$, $b_0 = b$. Let $n = a + b$.

Let's collect some invariants. We will prove all of them by induction on $i \in \mathbb{N}_{\geq 0}$.

Invariant 22.1.

$$\forall i \geq 0 : a_i + b_i = n$$

Proof. Base case $a_0 + b_0 = a + b = n$ holds by definition of n and (a_0, b_0). Assume $a_i + b_i = n$. For $a_{i+1} + b_{i+1}$ we have two cases:

Case $a_i < b_i$: Here we have $a_{i+1} = 2a_i$ and $b_{i+1} = b_i - a_i$. So

$$a_{i+1} + b_{i+1} = 2a_i + b_i - a_i = a_i + b_i = n$$

Case $a_i \geq b_i$: In this case we have $a_{i+1} = a_i - b_i$ and $b_{i+1} = 2b_i$. It follows

$$a_{i+1} + b_{i+1} = a_i - b_i + 2b_i = a_i + b_i = n$$

□

Invariant 22.2.

$$\forall i \geq 0 : b_i > 0$$

Proof. This follows almost immediately from definitions [2].

□

[2] Base case $b_0 = b > 0$ holds by definition of b. Assume $b_i > 0$. Again we have two cases. If $a_i < b_i$ then $b_{i+1} = b_i - a_i > 0$. If $a_i \geq b_i$ then $b_{i+1} = 2b_i > 0$.

Invariant 22.3.

$$\forall i \geq 0 : a_i \geq 0$$

Proof. This also follows from definitions [3].

□

[3] Base case $a_0 = a > 0$ holds by definition of a. Assume $a_i \geq 0$. Again we have two cases. If $a_i < b_i$ then $a_{i+1} = 2a_i \geq 0$. If $a_i \geq b_i$ then $a_{i+1} = a_i - b_i \geq 0$.

Invariant 22.4.

$$\forall i \geq 0 : a_i \equiv 2^i a \mod n$$

Proof. Base case $a_0 = a = 2^0 a$ trivially holds. Assume $a_i \equiv 2^i a \mod n$. For a_{i+1} we have two cases:

Case $a_i < b_i$: Here we have $a_{i+1} = 2a_i$. So

$$\begin{aligned} a_{i+1} &= 2a_i \\ &\equiv 2 \cdot 2^i a \mod n \\ &\equiv 2^{i+1} a \mod n \end{aligned}$$

Case $a_i \geq b_i$: In this case we have $a_{i+1} = a_i - b_i$. It follows

$$\begin{aligned} a_{i+1} &= a_i - b_i \\ &\equiv a_i + n - b_i \mod n \\ &\equiv a_i + a_i + b_i - b_i \mod n \\ &\equiv 2a_i \mod n \\ &\equiv 2 \cdot 2^i a \mod n \\ &\equiv 2^{i+1} a \mod n \end{aligned}$$

□

We will use these 4 invariants ($a_i \geq 0$, $b_i > 0$, $a_i + b_i = n$ and $a_i \equiv 2^i a \mod n$) to determine for which initial values a and b the loop terminates. To do so we consider $\frac{a}{n}$. Because $0 < a < n$ we know that $\frac{a}{n} \in (0, 1)$. We look at the expansion of $\frac{a}{n}$ in base 2.

Theorem 22.1. *If the expansion of $\frac{a}{n}$ is finite with k digits $d_i \in \{0, 1\}$*

$$\frac{a}{n} = \sum_{i=1}^{k} d_i 2^{-i}$$

then $a_k = 0$ and the loop terminates after k steps.

Proof. From

$$\frac{a}{n} = \sum_{i=1}^{k} d_i 2^{-i}$$

we get by multiplying both sides with $2^k n$:

$$2^k a = \sum_{i=1}^{k} n d_i 2^{k-i} \equiv 0 \mod n$$

Together with invariant 22.4 we get

$$a_k \equiv 2^k a \equiv 0 \mod n$$

and because $a_k \geq 0$, $b_k > 0$, $a_k + b_k = n$ we know that $0 \leq a_k < n$, so it must be that $a_k = 0$ and the loop terminates after at most k steps. To show that the loop terminates after exactly k steps, we need to show that $a_j > 0$ for $0 \leq j < k$. We will do this by finding a contradiction. Assume there exists a $j < k$ such that $a_j = 0$. Then it also holds that $2^j a \equiv 0 \mod n$.

From

$$\frac{a}{n} = \sum_{i=1}^{k} d_i 2^{-i}$$

we get by multiplying both sides with $2^j n$:

$$2^j a = \sum_{i=1}^{k} n d_i 2^{j-i} = \sum_{i=1}^{j} n d_i 2^{j-i} + \sum_{i=j+1}^{k} n d_i 2^{j-i} \equiv 0 \mod n$$

$2^j a \equiv 0 \mod n$, so $2^j a = nq$ for some $q \in \mathbb{Z}$. Then

$$q = \sum_{i=1}^{j} d_i 2^{j-i} + \sum_{i=j+1}^{k} d_i 2^{j-i}$$

We have $q \in \mathbb{Z}$, $\sum_{i=1}^{j} d_i 2^{j-i} \in \mathbb{Z}$, but $\sum_{i=j+1}^{k} d_i 2^{j-i} \notin \mathbb{Z}$, because $d_i \in \{0, 1\}$. This is a contradiction.

□

We arrived at a neat result: if the binary expansion of $\frac{a}{a+b}$ is finite with k digits, then the loop terminates after k steps.

What can we say if the expansion is not finite but instead has a repeating pattern with a prefix and a period (the only other option [4]) ? For starters, we can use a contradiction similar to the earlier one to prove that the loop does not terminate. Consider the infinite binary expansion:

$$\frac{a}{n} = \sum_{i=1}^{\infty} d_i 2^{-i}$$

Assume there is a k for which $a_k = 0$. Then by multiplying the expansion with $2^k n$ we get:

$$2^k a = \sum_{i=1}^{k} n d_i 2^{k-i} + \sum_{i=k+1}^{\infty} n d_i 2^{k-i} \equiv 0 \mod n$$

So for some $q \in \mathbb{Z}$ such that $2^k a = nq$ we have

$$q = \sum_{i=1}^{k} d_i 2^{k-i} + \sum_{i=k+1}^{\infty} d_i 2^{k-i}$$

The left side and the first sum on the right both belong to $\mathbb{Z}$ but the second sum does not, which is a contradiction. This means, that $\forall k : a_k > 0$ and the loop does not terminate.

[4] That is because $\frac{a}{a+b} \in \mathbb{Q}$.

Finish up this case and explain period of infinite sequence.

23
Divisible by three

Loop invariants and a constraint relaxation are used to solve this problem.

Problem

Show that an integer is divisible by three iff the sum of its digits in decimal representation is divisible by three.

The following proof is a delightful example of unconventional thinking[1].

[1] Unfortunately I don't know the origin of this proof and I don't remember where I first saw it.

We have to work with the digits in decimal representation of some integer n. Let's denote with $s(n)$ the sum of those digits. We will prove something stronger than the problem:

$$s(n) = n - 9k, \text{ for some } k \in \mathbb{Z}$$

The delightful twist that we are going to use is a relaxation of decimal representation: we will allow digits bigger than nine. We will then heal the representation back to decimal digits less than ten in a loop while maintaining $s(n) = n - 9k$ as a loop invariant.

We start with a representation with just one digit, namely n itself:

$$n = \sum_{i=0} d_i 10^i, \text{ with } d_0 = n \text{ and } \forall i > 0 : d_i = 0$$

This is not yet a valid decimal representation if $d_0 > 9$, but the loop invariant does hold: $s(n) = \sum_{i=0} d_i = n = n - 9 \cdot 0$. In a loop we now keep subtracting ten from d_0 and adding a carry-over one to d_1 until $d_0 \leq 9$. Each time through the loop d_1 increases by one and d_0 decreases by ten, so $s(n) = \sum_{i=0} d_i$ decreases by nine:

$$d_1 \leftarrow d_1 + 1$$
$$d_0 \leftarrow d_0 - 10$$
$$s(n) \leftarrow s(n) - 9$$

This means the loop invariant $s(n) = n - 9k$ is maintained during the healing of d_0. Eventually $d_0 \leq 9$. We then look at d_1 and heal it similarly, carrying over to d_2 and subtracting ten from it. Again the loop invariant holds. We repeat this for all digits until all are healed. The healing has to finish because n is a finite integer with a finite decimal representation. In the end we have a valid decimal representation and the loop invariant still holds which proves our problem.

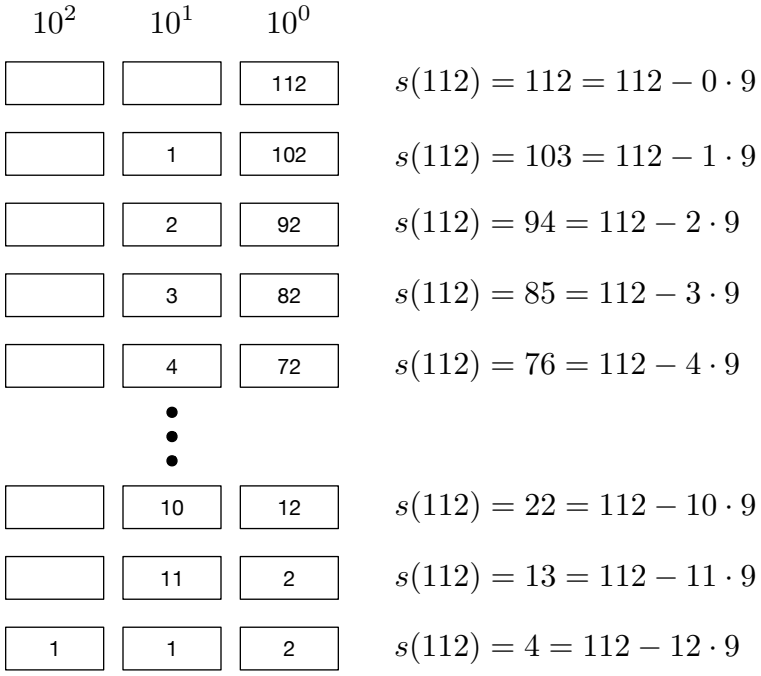


Figure 23.1: Example with $n = 112$. In each row the boxes represent the digits in the decimal representation (least significant on the right). In the first row the representation has only one digit, the number itself. Going down, each row is one step in the healing of the representation while maintaining the loop invariant $s(112) = 112 - k \cdot 9$.

24
Dutch National Flag

Problem 'Dutch National Flag' in *Programming, The Derivation of Algorithms*[1].

[1] A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990

Problem

Write a program that swaps elements of an array containing colors red, white and blue in such a way that the array's final state is in accordance with the Dutch National Flag.

We hope to solve this problem in linear time, going only once through the array in a loop.

Our array is $A[0, \ldots, n)$ with

$$\forall i : 0 \leq i < n : A[i] \in \{\blacksquare, \square, \blacksquare\}$$

The desired final state of the array has 3 contiguous regions: the red region, the white region and the blue region. Two indices r and w into the array are sufficient to show the extent of each region. We define post condition R

$$\begin{aligned} R \equiv\ & (\forall i : 0 \leq i < r : A[i] = \blacksquare) \\ \wedge\ & (\forall i : r \leq i < w : A[i] = \square) \\ \wedge\ & (\forall i : w \leq i < n : A[i] = \blacksquare) \end{aligned}$$

Our loop invariant P will be a relaxation[2] of the post condition R. We need to introduce a new index variable b to capture the notion of unprocessed region:

[2] Relaxation is a common technique to derive a useful loop invariant from a post condition. A common way to do the relaxation is to introduce a variable that in the beginning completely relaxes the condition and that then gradually changes and tightens the condition to its final desired form.

$$
\begin{aligned}
P \equiv\ & (\forall i : 0 \leq i < r : A[i] = \blacksquare) \\
& \wedge (\forall i : r \leq i < w : A[i] = \square) \\
& \wedge (\forall i : w \leq i < b : A[i] \text{ has not been processed yet}) \\
& \wedge (\forall i : b \leq i < n : A[i] = \blacksquare)
\end{aligned}
$$

This allows us to assign values to our indices r, w and b that satisfy P before the loop starts by extending the unprocessed region to be the whole array and making the red, white and blue regions empty[3]:

$$
\begin{aligned}
r &\leftarrow 0 \\
w &\leftarrow 0 \\
b &\leftarrow n
\end{aligned}
$$

[3] This is the key insight for solving the problem. Instead of having only three color regions to work with, we introduce a fourth region of unprocessed elements and we gradually shrink it. In the beginning this fourth region is the whole array and the color regions are all three empty. As the unprocessed region shrinks, the color regions start to grow in such a way that P always stays true. In the end the unprocessed region is empty and the three color regions are in their final desired state thanks to P always holding.

Our goal now is to maintain the loop invariant P while reducing the unprocessed region by processing array elements and swapping them until the unprocessed region is empty, so $b - w = 0$ or $b = w$. We then have $b = w \wedge P \Rightarrow R$. The swapping needs to happen in such a way that P always holds. We also want to make progress each time through the loop, so we want $b - w$ to get smaller each time through the loop. We achieve progress by either increasing w or decreasing b. As long as $b > w$ we go through the loop.

We will do a case analysis of the state at the region borders of processed and unprocessed regions of the array.

Let's start with a simple case: $A[w] = \square$. Then moving w one position to the right extends the white region, maintains P and shrinks $b - w$, so makes progress. We write this down as one case for the loop body:

$$\text{if } A[w] = \square \text{ then } w \leftarrow w + 1 \text{ endif}$$

Because we are inside the loop we know that $b > w$, so $A[b-1]$ exists, ie $b - 1$ is a valid index position. If $A[b-1] = \blacksquare$, then we can extend the blue region to the left and thus also shrink the unprocessed region while maintaining P:

$$\text{if } A[b-1] = \blacksquare \text{ then } b \leftarrow b - 1 \text{ endif}$$

We have a couple more cases to cover.

If $A[w] = \blacksquare$ then we can do our first swap: we swap $A[w]$ with $A[b-1]$ which will allow us after the swap to extend the blue region to its left as done in the previous case. As before this maintains P and is progress. Let's capture this for the body of our loop:

$$\text{if } A[w] = \blacksquare \text{ then } A[w] \leftrightarrow A[b-1] \,;\, b \leftarrow b-1 \text{ endif}$$

If $A[w] = \blacksquare$ then we can do the following swap: we swap $A[r]$ with $A[w]$. We can then extend the red region to its right. Whatever the white region was (empty or not), it also shifts to the right by one (as a region it hasn't changed, just the first white element might have moved to be the last element of the white region if the white region was not empty):

$$\text{if } A[w] = \blacksquare \text{ then } A[w] \leftrightarrow A[r] \,;\, r \leftarrow r+1 \,;\, w \leftarrow w+1 \text{ endif}$$

These cases[4] are sufficient to allow us to make progress while maintaining P. Because $b - w$ is finite and we reduce it each time through the loop, the loop will terminate. Our final program (in Go syntax) is:

```
r := 0
w := 0
b := 0
for w < b {
    switch {
    case A[w] == White: w = w + 1
    case A[b-1] == Blue: b = b - 1
    case A[w] == Blue: swap(A[w], A[b-1])
                       b = b - 1
    case A[w] == Red: swap(A[w], A[r])
                      r = r + 1
                      w = w + 1
    }
}
```

The loop invariant P is maintained throughout and when the loop exits, we have $b = w$ which establishes R.

[4] These cases are biased towards making progress from the left to the right. One can make similar choices that cover more cases on the right border of the unprocessed region.

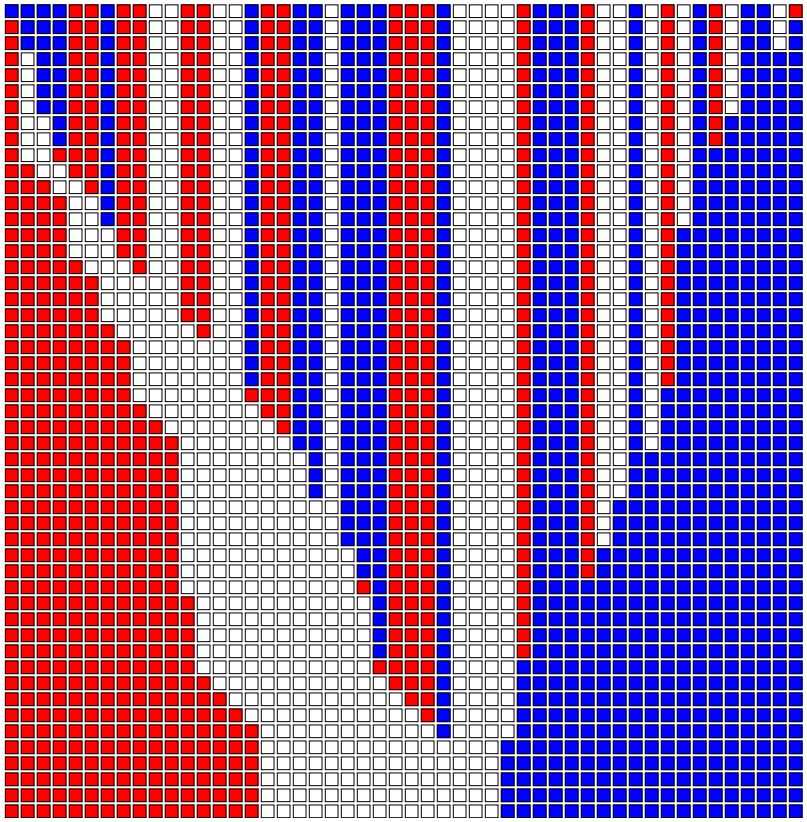


Figure 24.1: Example of processing an array. The first row is the initial state of the array. Each row below is one time through the loop. It is interesting to see that even though condition R is satisfied towards the end while $b > w$, the loop still has to process elements until $b = w$, not doing any more swaps but bringing w and b ever closer.

25
Bibliography

Stephen Abbott. *Understanding Analysis*. Springer, 2 edition, 2015. ISBN 978-1-4939-2711-1.

M. Beck and R. Geoghegan. *The Art of Proof: Basic Training for Deeper Mathematics*. Undergraduate Texts in Mathematics. Springer New York, 2010. ISBN 9781441970237.

Ethan Canin. *The Palace Thief Stories*, chapter Batorsag and Szerelem, page 87. Random House New York, 1994.

Tung Kam Chuen. 0-1 sequences. 2016. URL https://open.kattis.com/problems/sequences.

Edward Cohen. *Programming in the 1990s, An Introduction to the Calculation of Programs*. Springer-Verlag, 1990.

Freeman J. Dyson. Note 1931-The problem of the pennies. *Math. Gaz.*, 30:231–234, 1946.

A. Engel. *Problem-Solving Strategies*. Problem Books in Mathematics. Springer New York, 2013. ISBN 9781475789546. URL https://books.google.com/books?id=aUofswEACAAJ.

Jeff Erickson. *Algorithms, Etc*. 2015. URL http://jeffe.cs.illinois.edu/teaching/algorithms/.

Jeff Erickson. Algorithms — Extended Dance Remix: Fast Fourier Transforms. https://jeffe.cs.illinois.edu/teaching/algorithms/notes/A-fft.pdf, 2021. [Online; accessed 07-May-2022].

Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley, 3rd edition, 1993. ISBN 0201549832.

A. Kaldewaij. *Programming, The Derivation of Algorithms*. Prentice Hall, 1990.

Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321295358.

Géza Kós. On the grasshopper problem with signed jumps. *The American Mathematical Monthly*, 118:877–886, 2010. URL https://arxiv.org/abs/1008.2936.

N. Loehr. *Combinatorics*. Discrete Mathematics and Its Applications. CRC Press, 2017. ISBN 9781498780278.

Cosmin Negruseri. Codejam 2008 round 1a: Problem c: Numbers. 2008. URL https://code.google.com/codejam/contest/32016/dashboard#s=p2.

Romeo Rizzi. A short proof of König's matching theorem. *Journal of Graph Theory*, 33(3):138–139, 2000. URL https://math.dartmouth.edu/archive/m38s12/public_html/sources/Rizzi2000.pdf.

Günter Rote. *Crossing the Bridge at Night*. World Wide Web, http://page.mi.fu-berlin.de/~rote/Papers/pdf/Crossing+the+bridge+at+night.pdf, 2002.

Alexander Schrijver. On the history of the transportation and maximum flow problems. 2002. URL http://homepages.cwi.nl/~lex/files/histtrpclean.pdf.

Spotify. Cat vs dog. 2012. URL https://labs.spotify.com/puzzles/.

Michael Tong. Devil's chessboard. 2013. URL https://brilliant.org/discussions/thread/the-devils-chessboard/.

Eric W. Weisstein. Greatest dividing exponent. From MathWorld—A Wolfram Web Resource. URL http://mathworld.wolfram.com/GreatestDividingExponent.html.

Wikipedia. Inclusion–exclusion principle — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Inclusion%E2%80%93exclusion%20principle&oldid=1086507513, 2022. [Online; accessed 07-May-2022].

26
Index

Todo list