Algorithm Design

Giulio Paparelli, AY 22/23

Use at Your Own Risk 

Mathematical Background

Counting

Permutations

A **permutation** of a finite set S s a **ordered** sequence of all the elements in S, with each element appearing exactly once.
For example, with $S = \{a, b, c\}$ we have 6 permutations: $abc, acb, bac, bca, cab, cba$.

For a set S of n elements we have $n!$ permutations, since there are n ways to choose the first element of the sequence, $n - 1$ ways of choosing the second element and so on.

A **k-permutation** of S is an **ordered** sequence of k elements of S, with no element appearing more than once, and with $k < |S| = n$.
We can say that a classic permutation is a n-permutation of the set S.

The number of k-permutations of a n-set is:

$$n(n-1)(n-2)\dots(n-k+1) = \frac{n!}{(n-k)!}$$

Combinations

A **k-combination** of an n-set S is simply a k-**subset** of S.
For example, the 4-set $\{a, b, c, d\}$ has 6 2-combinations: ab, ac, ad, bc, bd, cd.
It is then clear that with combinations **the order do not matters**.

We can express the number of k-combinations of an n-set in terms of the number of k-permutations of an n-set.
Every k-combination has exactly $k!$ permutations of its elements, each of which is a distinct k-permutation of the n-set itself.
Thus the number of k-combinations of a n-set is the number of k-permutations divided by $k!$

From the formula seen above about k-permutations we have that:

$$k\text{-combinations} = \frac{n!}{k!(n-k)!}$$

Binomial Coefficient

The notation $\binom{n}{k}$ denotes the number of k-combinations of an n-set.
So we know that:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The binomial coefficient is symmetric in k and $n-k$:

$$\binom{n}{k} = \binom{n}{n-k}$$

Probability

We define the **probability** in terms of a **sample space** S, which is a set whose elements are called **outcomes** or **elementary events**.
An **event** is a subset of the sample space S.

The event S is called **certain event**, and the event $\emptyset$ is called the **null event**.
We say that two events A, B are **mutually exclusive** if $A \cap B = \emptyset$.

An outcome $s \in S$ also define an event $\{s\}$, which we sometimes write as just s.
By definition **all outcomes are mutually exclusive.**

Axioms of Probability

A **probability distribution** P on a sample space S is a mapping from events of S to real numbers satisfying the following axioms:

1. $P(A) \geq 0$ for any event A
2. $P(S) = 1$
3. $P(A \cup B) = P(A) + P(B)$ for any two mutual exclusive events A and B

We call $P(A)$ the **probability** of the event A.

From the above axioms we derive several results:

1. $P(\emptyset) = 0$
2. $A \subseteq B \implies P(A) \leq P(B)$
3. $P(\bar{A}) = 1 - P(A)$
4. $P(A \cup B) = P(A) + P(B) - P(A \cap B) \leq P(A) + P(B)$, for any two events A, B

Discrete Probability Distribution

A **probability distribution** is **discrete** if it is defined over a finite (or countably infinite) sample space.
Let S be a finite sample space, then for any event $A \subseteq S$ we have

$$P(A) = \Sigma_{s \in A}\, P(s)$$

since outcomes, specifically those in$A$, are mutually exclusive.

If S is finite and every outcome $s \in S$ has $P(s) = \frac{1}{|S|}$ we say that we have a **uniform probability distribution** on S.

Conditional Probability and Independence

Conditional probability formalizes the notion of having prior partial knowledge of the outcome of an experiment.
The **conditional probability** of an event A given that another event B occurs is defined as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

whenever $P(B) \neq 0$.

Two events are said **independent** if $P(A \cap B) = P(A)P(B)$.
Given two independent events A and B with $P(B) \neq 0$, then we have that $P(A|B) = P(A)$.

Bayes's Theorem

From the conditional probability and the commutative law $A \cap B = B \cap A$ it follows that for two events A and B, each with non-zero probability, we have:

$$P(A \cap B) = P(B)P(A|B) = P(A)P(B|A)$$

And solving the conditional probability we obtain that:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

which is known as the **Bayes's Theorem**.

(Discrete) Random Variables

A **(discrete) random variable** X is a function from a finite (or countably infinite) sample space S to the real numbers.
It associates a real number with each possible outcome of an experiment, which allows us to work with the probability distribution induced on the resulting set of numbers.

For a random variable X and a real number x we define the event $X = x$ to be the event $\{s \in S : X(s) = x\}$, and thus

$$P(X = x) = \Sigma_{s \in S : X(s) = x} \, P(s)$$

Example:
Say we roll a pair of 6-sided dice. There are 36 outcomes in the sample space S.
The dices are fair, so the probability distribution is uniform: each outcome $s \in S$ is equally likely with probability $\frac{1}{36}$.

Let's define the random variable X to be the maximum of the two values showing on the dice.
Now we can use X to compute the probability of events such as "the max value between the two dices is 3", by just computing $P(X = 3)$, which is $\frac{5}{36}$ as there are 5 outcomes where 3 is the maximum number.

Expected Value

The most useful summary of the distribution of a random variable is the "average" of the values it takes on.
The **expected value** of a discrete random variable X is:

$$E[X] = \Sigma_x \; x \cdot P(X = x)$$

Example:
Consider a game in which you flip two fairs coins.
You earn 3\$ for each head but lose 2\$ for each tail.
The expected value for a random variable X representing your earning is:

$$E[X] = 6 \cdot P(HH) + 1 \cdot P(HT) - 4 \cdot P(TT) = 6 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} - 4 \cdot \frac{1}{4} = 1$$

The expected value has the following properties:

1. **linearity:** $E[X + Y] = E[X] + E[Y]$
2. $E[\alpha X] = \alpha E[X]$
3. if two variables X, Y are independent and each has a defined expectations then $E[XY] = E[X]E[Y]$

When a random variable X takes on values from $\mathbb{N}$ we have a formula for its expectations:

$$E[X] = \Sigma_{i=1}^{\infty} \; P(X \geq 1)$$

Variance and Standard Deviation

The expected value of a random variable does not express how "spread out" the variable values are.
The notion of **variance** mathematically express how far from the mean a random variable's values are likely to be.
The variance of a random variable X with mean $E[X]$ is defined as:

$$Var[X] = E[X^2] - E^2[X]$$

We can also show that $E[X^2] = Var[X] + E^2[X]$.

The variance of a random variable X and the variance of αX are related:

$$Var[\alpha X] = \alpha^2 Var[X]$$

When X and Y are independent random variables we have:

$$Var[X + Y] = Var[X] + Var[Y]$$

The **standard deviation** of a random variable X, denoted by σ, is the nonnegative square root of the variance of X.

Indicator Random Variables

In order to analyze many algorithms we use **indicator random variables**.
Indicator random variables provide a convenient method for converting between probabilities and expectations.

Given a sample space S and an event A, the **indicator random variable** $I(A)$ associated with the event A is defined as:

$$I(A) = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ do not occurs} \end{cases}$$

Another notation is the following:

$$X_i = \begin{cases} 1 & \text{event } i \text{ occur with probability } p \\ 0 & \text{event } i \text{ don't occur, with probability } 1-p \end{cases}$$

We can compute the expected value of indicator variables: $E[X_i] = 1 \cdot p + 0 \cdot (1-p)$
If we define a random variable $X = \Sigma_{i=1}^{t} X_i$ we obtain that

$$E[X] = \Sigma_{i=1}^{t} E[X_i] = \Sigma_{i=1}^{t} P(X_i = 1) = t \cdot p$$

Said easy: **the expected value of an indicator random variable associated with an event is equal to the probability that event occurs.**

Las Vegas vs Monte Carlo

In the context of **probabilistic algorithms** we have two categories:

- **Las Vegas Algorithms:**
  - no errors, but the running time is probabilistic, very good w.h.p (with high probability)
  - e.g.: QuickSort
- **Monte Carlo Algorithms:**
  - 1-sided / 2-sided errors (false positive/negatives or both), running time is worst-case
  - e.g.: Karp-Rabin Fingerprint

Random Permutation

Let's see how we can randomly permute an array of n elements.
The goal is to produce a **uniform random permutation**, that is,, a permutation that is as likely as any other permutation.
To do so we need a function `rand(a,b)` that returns at random an integer in the interval $[a, b]$, where each element i has probability of being picked equal to $\frac{1}{b-a+1}$.
Mind that in practice we obtain a **pseudo-random** choice, as truly random is undecidable.

To obtain a random permutation we then use the following function:

```
void randomPermutation(int[] array){
        int n = array.length();
        for(int i = 1; i < n; i++)
                swap(array[i], array[rand(i,n)]);
}
```

It can be proved, using indicator variables, that each permutation is generated with a probability very close to $\frac{1}{n!}$

A randomized algorithm is often the simplest and most efficient way to solve a problem.
Being able to produce random permutations in a very efficient way is then crucial.

The Hiring/Headphones Problem

You need to hire a new assistant. The employment agency send you a list of candidates, and you interview them starting from the first of the list.
You must pay the employment agency a small fee to interview an applicant.
To actually hire an assistant is more costly, however, since you must first fire your current assistant and pay a substantial hiring fee to the agency.
Nonetheless you are committed to having, at all times, the best possible person for the job.
Therefore you decide that after interviewing each applicant, if that applicant is better than the current assistant, you will fire the current assistant and hire the new applicant.

This is a first example where the randomization is a way to improve the complexity, in the average case.
If we use the order given by the list the cost is predetermined and it's easy to compute, but it turns out that, in the average case, we can improve the complexity by creating a random permutation of the assistants, using the function seen before.

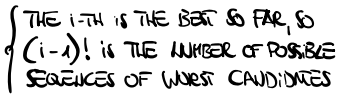
Let X be the random variable whose value equals the number of times you hire a new assistant.
We could then apply the definition of expected value $E[X] = \Sigma_{x=1}^{n} x \cdot P(X = x)$ but this calculation is quite difficult to compute.
We instead exploit the random indicator variables.

Let X_i be the indicator random variable associated with the event in which the i-th candidate is hired:

$$X_i = \begin{cases} 1 & \text{candidate } i \text{ is hired with probability } p_i \\ 0 & \text{candidate } i \text{ is not hired, with probability } 1 - p_i \end{cases}$$

We then see that $X = X_1 + X_2 + \cdots + X_N$

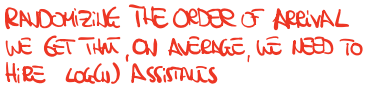
We notice that p_i is the probability that the $i - th$ candidate is better than all the previous $i - 1$ candidates (and therefore he gets hired)
We can then compute p_i

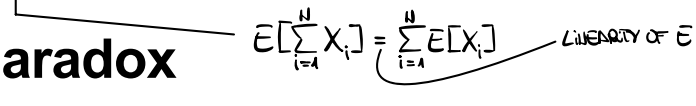


$$p_i = \frac{\text{all candidates so far but the last is the best seen}}{\text{all candidates so far}} = \boxed{\frac{(i-1)!}{i!}} = \frac{1}{i}$$

We can then use $X = \Sigma_{i=1}^{n} X_i$ and compute its expected value:

$$E[X] = \Sigma_{i=1}^{n} E[X_i] = \Sigma_{i=1}^{n} \frac{1}{i} \sim \text{ harmonic series: } \boxed{\log(n)}$$





The Birthday Paradox

The probability problem asks for the probability that, in a set of m randomly chosen people, at least two will share a birthday.
The **birthday paradox** refers to the counterintuitive fact that only 23 people are needed for that probability to exceed 50%.

We have $n = 365$ days.
Given a person i and a person j, with $i \neq j$, let's define indicator variables

$$X_{ij} = \begin{cases} 1 & i \text{ and } j \text{ have the same birthday, with probability } p \\ 0 & \text{otherwise, with probability } 1-p \end{cases}$$

Every person has probability of being born in a day k equal to $1/n$.
Then $P(i \text{ born on day k } \cap\ j \text{ born on day k}) = \frac{1}{n} \cdot \frac{1}{n}$ as the two events are independent.
Thus we have:

$$E[X_{ij}] = p = \Sigma_{k=1}^{n} \frac{1}{n}\frac{1}{n} = \Sigma_{k=1}^{n} \frac{1}{n^2} = \frac{1}{n}$$

THE SIGMA GIVES N, WE SIMPLIFY N · 1/N²

Let's now consider m as the number of people we choose.
We define $X = \Sigma_{i=1}^{m}\Sigma_{j=i+1}^{m} X_{ij}$ as the number of pairs of people among m that have the same birthday. ⟶ AKA #PEOPLE THAT SHARE A BIRTHDAY
Therefore we have:

$$E[X] = \Sigma_{i=1}^{m}\Sigma_{j=i+1}^{m} E[X_{ij}] = \binom{m}{2}\frac{1}{n} = \frac{m(m-1)}{2}\frac{1}{n}$$

And $E[X] \geq 1$ holds when $m(m-1) \geq 2n \implies m \sim \sqrt{2n}$.
Since $n = 365$ we have $m \sim 27$.

Randomized QuickSort

To show how randomization can improve drastically the performance on the average case (and the **expected case**) we show quick sort and its randomized version.

QuickSort

QuickSort applies the divide-and-conquer method for sorting a subarray $A[p : r]$

- **Divide** by partitioning (rearranging) the array $A[p : r]$ into two, possibly empty, subarrays $A[p : q-1]$ (aka the **low side**) and $A[q+1 : r]$ (aka the **high side**) such that each element in the low side of the partition is less than or equal to the **pivot** $A[q]$, which is in turn less than or equal to each element in the high side. Compute the index q of the pivot as part of this partitioning procedure.
- **Conquer** by calling quicksort recursively to sort each of the subarrays $A[p : q-1]$ and $A[q+1 : r]$
- **Combine** by doing nothing

We can then write in pseudocode the procedure that implement quicksort.
To sort an entire n-element array $A[1 : n]$ the initial call is `quicksort(A, 1, n)`

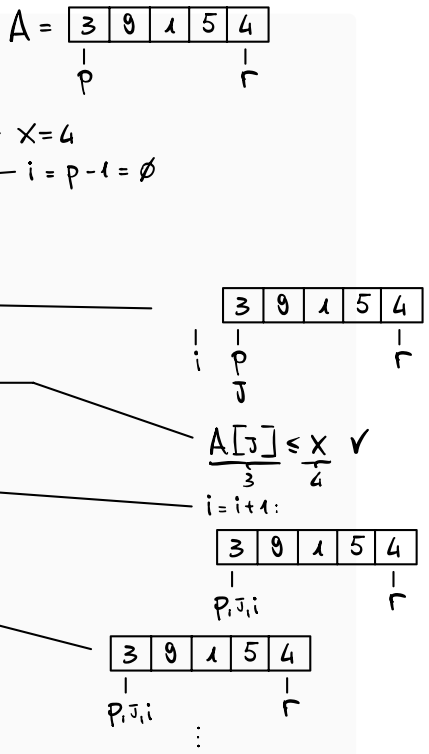
THE FIRST INDEX IS 1 AND NOT 0

```
void quicksort(int A[], p, r){
        if(p < r){
                // partition around the pivot, wich ends up in A[q]
                q = partition(A, p, r);
                quicksort(A, p, q-1); // recursively sort the low side
```

```
                quicksort(A, q+1, r); // recursively sort the high side
        }
}
```

The key of the algorithm is the **partitioning step**, which rearranges the subarray $A[p:r]$ in place, returning the index of the dividing point between the two sides of the partition.

```
int partition(int A[], p, r){
        // the pivot: the last element of the current subarray
        int x = A[r];
        // highest index into the low side
        int i = p-1;
        // process each element other than the pivot
        for(int j = p; j < r; j++)
                // does this element belong to the low side?
                if(A[j] <= x){
                        // index of a new slot on the low side
                        i = i+1;
                        // put this element there
                        swap(A[i], A[j])
                }
        // the pivot just go the right of the low side
        swap(A[i+1], A[r])
        // new index of the pivot
        return i+1
}
```



Mind that `i` is the index where we put the "current" element smaller than the pivot.

Performance of QuickSort

The running time of QuickSort depends on how balanced each partitioning is, which in turn depends on which elements are used as pivots.
If the two sides of a partition are about the same size (aka the partitioning is balanced) then the algorithm runs asymptotically as fast as MergeSort.
On the other hand, if the partitioning is unbalanced the performance are asymptotically similar to InsertionSort.

Worst-Case Partitioning

The worst-case behavior for QuickSort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 1 element.
Let us assume that this unbalanced partitioning arises in each recursive call.
The partitioning costs $\Theta(n)$ time.
Since the recursive call on an array of size 1 returns without doing anything, $T(1) = \Theta(1)$, and the recurrence for the running time is:

AN ARRAY OF SIZE 1 IS ALREADY SORTED

$$T(n) = T(n-1) + T(1) + \Theta(n) = T(n-1) + \Theta(n)$$

SORT THE PARTITION WITH N-1 ELEMENTS
SORT THE PARTITION WITH 1 ELEMENT
PARTITION COST

but we know, thanks to our assumptions, that $T(n-1) = T(n-2) + \Theta(n-1)$, and so on until we reach $T(n-n) = T(0)$.

Summing all the costs incurred at each level of the recursion we obtain an arithmetic series which evaluates to $\Theta(n^2)$.
Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$ $\square$

Randomized QuickSort

Instead of always using $A[r]$ as the pivot, the randomized QuickSort randomly chooses the pivot from the subarray $A[p:r]$, where each element has an equal probability of being choses.
It then exchanges that element with $A[r]$ before partitioning.
Because the pivot is chosen randomly we expect the split of the input array to be reasonably balanced on average.

```
randomized-partition(A, p, r)
        i = rand(p, r)
        swap(A[r], A[i])
        return partition(A, p, r)

randomized-quicksort(A, p, r)
        if p < r
                q = randomized-partition(A, p, r)
                randomized-quicksort(A, p, q-1)
                randomized-quicksort(A, q+1, r)
```

Analysis of RQS

We compute the cost of the RQS counting the number of comparison between the elements of the array: a comparison costs $\Theta(1)$, hence the number of comparisons gives the time complexity of the whole algorithm.

1. For the purpose of this analysis the output of `randomized-quicksort(A, p, r)` is $z_1 < \cdots < z_n$, we index the elements in A by their position in the sorted output, rather than the position in the input. We denote the set $\{z_i, \dots, z_j\}$ with Z_{ij}
2. When two keys z_i and z_j are compared? When one of them is chosen as pivot
3. How many times z_i and z_j are compared? At most once, since the pivot are removed from future comparisons (at the end of the partition the pivot is in its rightful place)
4. z_i and z_j are compared $\bullet$ $|A[p:q]| \geq j - i + 1$, since at least all of $z_i, \dots, z_j$ are in $A[p:q]$

THEN

We can now define:

$$X_{ij \,|\, i<j} = \begin{cases} 1 & z_i \text{ and } z_j \text{ are compared at any time} \\ 0 & \text{otherwise} \end{cases}$$

So that we can define

$$X = \Sigma_{i=1}^{n} \Sigma_{j=i+1}^{n} X_{ij}$$

as the number of comparisons in the execution of `randomized-quicksort(A, 1, n)`
This is thanks to the fact that z_i and z_j are compared at most once.

We now want to compute $E[X]$.
We know that X_{ij} are indicator variables:

$$\begin{aligned}
E[X_{ij}] = & \\
&= P(X_{ij} = 1) \\
&= P((z_i, z_j \in Z_{ij}) \cap (z_i \text{ or } z_j \text{ is the pivot for } A[p:q])), \text{and since } P(A \cap B) \leq P(B) \\
&\leq P(z_i \text{ or } z_j \text{ is the pivot for } A[p:q]) \\
&= \frac{2}{|A[p:q]|} \\
&\leq \frac{2}{j-i+1}
\end{aligned}$$

Now we can compute the expected value of X:

$$\begin{aligned}
E[X] = & \\
&= \Sigma_{i=1}^{n-1} \Sigma_{j=i+1}^{n} E[X_{ij}], \text{and since } E[X_{ij}] \leq \frac{2}{j-i+1} \\
&= \Sigma_{i=1}^{n-1} \Sigma_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq \Sigma_{i=1}^{n-1} \boxed{\Sigma_{k=1}^{n} \frac{2}{k}} \\
&= \Sigma_{i=1}^{n-1} O(\log(n)) \\
&= O(n \log(n))
\end{aligned}$$

CHANGE OF VARIABLE
$k = j - i$
$j \in [i+1, n], k \in [1, n-i]$

$\boxed{\Sigma_{k=1}^{n} \frac{2}{k}} \rightarrow O(\log(n))$

□

Karp-Rabin Fingerprint

How to check equality of files?
It can be done deterministically, but it is very slow for large files.

We can se a file as a binary string $s = 01 \ldots 1 \ldots 0$, with $w = |s|$,
s can also be seen as a huge number $\in [0, 2^n - 1]$.

Given another file/sequence s', can we check $s = s'$?
Yes, it can be done in $O(n)$ time, using the Karp-Rabin Fingerprint.

We define a fingerprint function F that produces a number for a given sequence/number s

$$F(s) = s \bmod p$$

for a randomly chosen prime p.

There is the issue of the **collision**, given two sequences/files $k_1, k_2 \in [0 \ldots 2^n - 1]$ we have this two cases:

$$F(k_1) \neq F(k_2) \implies k_1 \neq k_2$$
$$F(k_1) = F(k_2) \implies \begin{cases} k_1 = k_2 \\ k_1 \neq k_2 \text{ 1-sided error with a given probability} \end{cases}$$

1-sided **error**: $k_1 \neq k_2 \wedge F(k_1) = F(k_2)$. } ← A FALSE POSITIVE

Idea: before starting a computation we

- choose a parameter τ *sufficiently large* (see later)
- choose uniformly and randomly a prime number in $[2 \dots \tau]$
  so that the probability of error is very low.

Let's consider the case $\overbrace{k_1 \neq k_2}$ $F(k_1) = F(k_2) \implies k_1 \bmod p = k_2 \bmod p$.
We call such a prime p a **bad prime**.

$$\begin{aligned} P(\text{error}) & \\ &= P_{p \leq \tau}(k_1 \neq k_2 \wedge F(k_1) = F(k_2)) \\ &= \frac{\text{number of bad primes} \leq \tau}{\text{primes p} \leq \tau} \\ &= \frac{?}{\tau / \log(\tau)} \end{aligned}$$

KNOWN FACT

To complete the computation of $P(\text{error})$ we considering the following results:

- **Observation:** both k_1 and k_2 require n bits for being represented.
- **Theorem:** the number of distinct prime divisors on any number less than 2^n is at most n.
  - **Proof:** each prime number is greater than 1. If a number n has more than t distinct prime divisors, then $n \geq 2^t$. If n has more than t prime divisors then $n = d_1 \cdot \ldots \cdot d_{t+1}$, where each $d_i \geq 2$, hence $n \geq 2^t$.

By the previous theorem we get that for any integer k of n bits there are at most n primes in its prime factorization.
Thus: there are at most n bad primes (in common for two of the files/numbers k_1, k_2 of same length)

Therefore:

$$\begin{aligned} P(\text{error}) = & \\ &= P_{p \leq \tau}(k_1 \neq k_2 \wedge F(k_1) = F(k_2)) \\ &= \frac{\text{number of bad primes} \leq \tau}{\text{primes p} \leq \tau} \\ &= \frac{?}{\tau / \log(\tau)} \\ &\leq \frac{n}{\tau / \log(\tau)} \\ &\triangleright \text{ since } \frac{\tau}{\log(\tau)} = n^{c+1}, \text{ we choose } \tau \sim (n^{c+1} \log(n)) \text{ and we get} \\ &= \frac{1}{n^c} \end{aligned}$$

Where c is an arbitrary constant.
We can choose a τ *sufficiently large* so that c is big enough to guarantee a very low error PROBABILITY □

Pattern Matching

Let's consider the problem of pattern matching in strings. A **text** is a string $X = x_1x_2 \dots x_n$ and a **pattern** is a string $Y = y_1y_2 \dots y_m$, both over a fixed finite alphabet, such that $m \leq n$.
We restrict, without losing generality, the alphabet Σ to $\{0, 1\}$.
The pattern occurs in the text if there is a $j \in \{1, 2, \dots, n - m + 1\}$ such that for $i \in [1, m]$ we have $x_{j+i-1} = y_i$.

The pattern matching problem is that of finding an occurrence (if any) of a given pattern in the text.
The problem can be trivially solved in $O(nm)$ time trying for a match at all locations i.
We describe a Monte Carlo algorithm that achieves a running time of $O(n + m)$, and later we will present the same algorithm in a Las Vegas fashion.

Monte Carlo:
Let's define the string $X(j) = x_jx_{j+1} \dots x_{x+m-1}$ as the substring of length m in X that starts at position j.
A match occurs if there is a choice of j, for $j \in [1, n - m + 1]$, for which $Y = X(j)$.
We make the solution unique by requiring that the algorithm find the smallest value of j such that $X(j) = Y$.

We choose a fingerprint function F seen before and compare $F(Y)$ with each of the fingerprints $F(X(j))$.
An error occurs if $F(Y) = F(X(j)) \wedge Y \neq X(j)$, which is a one-sided error, a **false positive.**

We interpret the strings Y and $X(j)$ as m-bit integers, and compare their fingerprints $F_p(Y)$ and $F_p(X(j))$ instead of trying to match each symbol in the two string.

The only possible error is that we get the same fingerprint when $Y \neq X(j)$, and as before

$$
\begin{aligned}
P(\text{error}) &= \\
&= \text{probability of collision} \cdot \text{number of patterns} \\
&\leq \boxed{\frac{m}{\tau/\log(\tau)}} \cdot (n - m + 1) \\
&\leq \frac{nm}{\tau/\log(\tau)} \qquad \text{SAME AS BEFORE!} \\
&\leq \frac{n^2}{\tau/\log(\tau)} \\
&\leq \frac{1}{n^c}, \text{ with } \tau \sim n^{c+2}\log(m)
\end{aligned}
$$

Las Vegas:
To build the Las Vegas we simply do some modifications to the previous algorithm.
Whenever a match occurs between $F(Y)$ and some $F(X(j))$, we compare the strings Y and $X(j)$ in $O(m)$ time.

If this is a false match, we detect it and abandon the whole process in favor of using the brute-force $O(nm)$ time algorithm.

This means that when the fingerprints are equals but $Y \neq X(j)$ we switch to the trivial algorithm. As usual, the expected cost is the same of the Monte Carlo version, but the worst case is given by the brute-force cost.

Union Bound

The union bound, says that for any finite or countable set of events, **the probability that at least one of the events happens is no greater than the sum of the probabilities of the individual events.**
This inequality provides an upper bound on the probability of occurrence of at least one of a countable number of events in terms of the individual probabilities of the events.

$$P(A \cup B) \leq P(A) + P(B)$$

We will often refer to the union bound with UB.

Universal Hash Family

IMPORTANT, BUT NOT FOR THE ORAL EXAM

There are data structures (e.g., hash tables) and algorithms (e.g. bloom filters, see later on) that uses hash functions as key ingredient.
In those situation we need a *good* hash function.
Along with being efficiently computable, what properties a good hash function have?
How you design good hash functions?

We first define that an hash function h maps the universe U of keys to a smaller set

$$h : U \to \{0, 1, \ \ldots \ m-1\}$$

We will often use the notation $[m]$ to represent $\{0, \ \ldots, \ m-1\}$.

An "ideal" hash function h would have, for each possible input k, an output $h(k)$ that is an element randomly and independently chosen uniformly in the range $[0, m-1]$.
Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same k yields the same output $h(k)$.
We call such an ideal hash function an **independent uniform hash function**.
Such hash function do not exists but we will analyze the hashing under the assumption of independent uniform hash functions, and then present ways of achieving useful practical approximations.

Given $\mathcal{H}$ a family of hash functions, each of the form $h : U \to [m]$, we say that

- $\mathcal{H}$ is **uniform** if for any key $k \in U$ and for any slot $q \in [m]$, the probability that $h(k) = q$ is $\frac{1}{m}$
- $\mathcal{H}$ is **universal** if for any distinct keys k_1 and k_2 in U, the probability that $h(k_1) = h(k_2)$, that is the probability of having a collision, is at most $\frac{1}{m}$

More specifically, the number of h such that there is a collision for two distinct keys k_1, k_2 is $\frac{|\mathcal{H}|}{m}$.
Then:

$$P(\text{collision}) = \frac{\text{number of bad choices of } h \text{ in } \mathcal{H}}{\text{number of } h \text{ in } \mathcal{H}} \leq \frac{|\mathcal{H}|/m}{|\mathcal{H}|} = \frac{1}{m}$$

Let's now consider the family $\mathcal{H}$ of hash functions, where each $h \in \mathcal{H}$ is defined as

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

where:

- a, b are chosen at random, with:
  - $a \in \mathbb{Z}_p^+ = \{1, 2, \ \ldots, p-1\}$
  - $b \in \mathbb{Z}_p = \{0, 1, \ \ldots, p-1\}$
- m is provided by the user
- p is a prime number in $[m+1, \ \ldots, \ 2m]$

The family of all such hash functions is

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^+ \text{ and } b \in \mathbb{Z}_p\}$$

where each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$.

Since we have $p-1$ choices for a and p choices for b, the collection $\mathcal{H}_{pm}$ contains $p(p-1)$ hash functions.
We can draw, uniformly at random, a choice of $h_{a,b} \in \mathcal{H}$ with a probability $\frac{1}{|\mathcal{H}|} = \frac{1}{p(p-1)}$

Theorem: the class $\mathcal{H}_{pm}$ is universal.
Proof: Consider two distinct keys k, l from $\mathbb{Z}_p$. For a given hash function h_{ab} we let

$$r = (ak + b) \bmod p$$
$$s = (al + b) \bmod p$$

From a known theorem, since p is the same, we can subtract the congruences:

$$\begin{aligned} r - s \equiv ak + b - al - b \bmod p \rightarrow \\ \rightarrow r - s \equiv ak - al \bmod p \rightarrow \\ \rightarrow r - s \equiv a(k - l) \bmod p \end{aligned}$$

Now, by hypothesis we have:
- $a \neq 0$ as $a \in \mathbb{Z}_p^+$
- $k \neq l$
- $k, l \in [0, p-1] \implies k$ and l can not be divided by p, therefore $(k-l) \bmod p \neq 0$

Since we know that $a \neq 0$ and $(k-l) \neq 0$ modulo p we derive that $a(k-l) \bmod p \neq 0$, hence $(r-s) \bmod p \neq 0$.
We can then conclude that $r \neq s$.

Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k and l map to distinct values r and s modulo p; there are no collisions yet at the "mod p level".
Moreover, each of the possible $p(p-1)$ choices of the pair (a, b) with $a \neq 0$ yields a different resulting pair (r, s) with $r \neq s$, since we can solve for a and b, given r and s:

$$a = ((r-s)((k-l)^{-1} \bmod p)) \bmod p$$
$$b = (r - ak) \bmod p$$

where $((k-l)^{-1} \bmod p)$ denotes the unique multiplicative inverse, modulo p, of $k-l$.

Since there are only $p(p-1)$ possible pairs (r, s) with $r \neq s$, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$.
Thus, for any given pair of inputs k and l, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^+ \times \mathbb{Z}_p$, the resulting pair (r, s) is equally likely to be any pair of distinct values modulo p.

Therefore, the probability that distinct keys k and l collide is equal to the probability that $r \equiv s \pmod{m}$ when r and s are randomly chosen as distinct values modulo p.

For a given value of r, of the $p-1$ possible remaining values for s, the number of values s such that $s \neq r$ and $r \equiv s \pmod{p}$ is at most

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \star \\ &= (p-1)/m \end{aligned}$$

Where $\star$ is given by the known inequality

$$\lceil a/b \rceil \leq \lceil (a + (b-1))/b \rceil$$

The probability that s collides with r when reduced modulo m is at most $\frac{1}{m}$.

Therefore, for any pair of distinct values $k, l \in \mathbb{Z}_p$, we have

$$P(h_{ab}(k) = h_{ab}(l)) \leq \frac{1}{m}$$

so that $\mathcal{H}_{pm}$ is indeed universal □

Markov's Inequality

Markov's Inequality gives an upper bound for the probability that a non-negative function of a random variable is greater or equal to some positive constant.
Markov's inequality (and other similar inequalities) relate probabilities to expectations, and provide (frequently loose but still useful) bounds for the cumulative distribution function of a random variable.

If X is a non-negative random variable and $a > 0$, then the probability that X is at least a is at most the expectation of X divided by a.
In symbols:

$$P(X \geq a) \leq \frac{E[X]}{a}$$

Cuckoo Hashing

A **dictionary** is a data structure for storing a set of items, that support three basic operations:

- $lookup(x)$, which returns $true$ if x is in the current set, $false$ otherwise
- $insert(x)$, which adds the item x to the current set if not already present
- $delete(x)$, which removes the item x from the current set if present

A simple but not efficient way of implementing a dictionary is a linked list, where:

- $lookup(x)$ cost $O(n)$
- $insert(x)$ cost $O(n)$ as first we have to check that x is not already in the list
- $delete(x)$ cost $O(n)$

We now present the **cuckoo hashing**, **a way of implementing a dictionary where the three operations are constant worst-case.**
As always, the problem with hashing are collisions.
In cuckoo hashing they are handled by using **two different hash functions** h_1, h_2 picked at random from an universal hash family $\mathcal{H}$.
Instead of requiring that x is stored at a position $h(x)$, for any key x we have two alternatives: we insert x in $h_1(x)$ or in $h_2(x)$.

What happens when the positions $h_1(x)$ and $h_2(x)$ are already occupied?
We **throw out** the current value y at $h_1(x)$ and we store there x.
At this point we use $h_2(y)$ to find if there is a suitable alternative for y. If the alternative position for y is vacant then we are good.
Otherwise we do for y what we just did for x: throw out the current occupant z of $h_2(y)$ and try to find a place for z.
This is continued until the procedure finds a vacant position or has taken too long.
In the latter, new hash functions are chosen in $\mathcal{H}$ and the whole data structure is rebuilt, or **rehashed**.

The operations $lookup(x)$ and $delete(x)$ are clearly $O(1)$ worst-time.
$insert(x)$ creates a path from the first node $h_1(x)$ to some node j where we insert x.
The cost is then the length of the path, but insertion can also cause the rehashing of the whole table.

Insertion Cost

We choose h_1, h_2 independently and uniformly at random from the universal hash family

$$\mathcal{H} = \{((ax+b) \bmod p) \bmod m \ : p > m, a \in \mathbb{Z}_p^+, b \in \mathbb{Z}_p\}$$

In particular, given the key $x \in U$ and $i, j \in [m]$ we have that

$$P(h_1(x) = i \wedge h_2(x) = j) \leq \frac{\lceil p/m \rceil^2}{(p-1)p} \sim \frac{1}{m^2} \quad \star$$

$\star$: Take it as a known theorem.
To remember that $P(h_1(x) = i \wedge h_2(x)) \leq \frac{1}{m^2}$ you can think that

- $P(h_1(x) = i) \leq 1/m$
- $P(h_2(x) = j) \leq 1/m$
- for two are independent events A, B we have that $P(A \wedge B) = P(A)P(B)$

Now, consider the set S of keys and the hash functions h_1, h_2.
Conceptually we build an undirected graph $G = (V, E)$ where $|V| = m$ and $|E| = n$, and:

- the **vertices** are in $V = \{0, 1, \ldots, m-1\}$ and represent the table positions

- the **edges** are random: $E = \{(h_1(x), h_2(x)) : x \in S\}$
  - G is actually a **multigraph**: two vertices can be connected by multiple edges.

Consider the insertion of a new key x, which corresponds to a new edge $e = (h_1(x), h_2(x))$ in G.
At this point one of three situations may happen:

1. one of the table positions in $h_1(x)$ or $h_2(x)$ is free, and x is placed there: $O(1)$
2. the positions are taken, so the insertion follows a path in G throwing out keys till a free position is found
3. as in the previous case, but here we traverse a cycle, which means that we have to rehash the whole table.

We need to analyze **2)** and **3)**: the goal is $O(1)$ expected time.
We assume that m (number of buckets, the table's size) is $>$ than $2cn$ ($|S| = n$) for a constant $c > 2$.

Case 2: A Free Position is Eventually Found

Let i be the starting position (the position where x "should have been placed"), and j the ending position (the position where x is actually placed) in the path.
We say that k is the length of the path $i \to j$ in G.

Lemma:
For any position i, j, the probability that exists a path $i \to j$ of length ≥ 1 (which is also the shortest path from i to j) is at most $\frac{1}{c^k m}$.
Alternatively said:

$$P(\exists \text{ the path } i \to j \in G \text{ of length } k) \leq \frac{1}{c^k m}$$

Proof:
We prove it by induction on k.
base case: $k = 1$, $i \to j$ is just an edge.

$$\begin{aligned} P(\exists \text{ edge } i \to j \in G) = & \\ &= \Sigma_{x \in S} \, P[(h_1(x) = i \wedge h_2(x) = \text{J}) \vee (h_1(x) = j \wedge h_2(x) = i)] \\ &\leq n \cdot 2 \cdot \frac{1}{m^2}, \text{ thanks to } \star \\ &\leq \frac{1}{cm} \spadesuit \end{aligned}$$

where ♠ is obtained thanks to the assumption we made:

$$M > 2cN \implies N < \frac{M}{2c}$$

THEN

$$N \cdot 2 \cdot \frac{1}{M^2} < \frac{M}{2c} \cdot 2 \cdot \frac{1}{M^2} = \frac{1}{cM}$$

inductive case: $k > 1$
Here we have a path that goes through a vertex r with $r \neq i, j$ such that the successor of r is j
Alternatively said: abusing notation, consider the vertex $r = j - 1$ in the path $i \to j$

$$i \longrightarrow r \to j$$

The length of the sub-path $i \to r$ is $k-1$: here we will apply the inductive hypothesis.

$$\begin{aligned} P(\exists \text{ path } i \to j \text{ of length } k) \leq \\ &\leq \Sigma_{r \in (V-\{i,j\})}\ P(\exists \text{ path } i \to r \text{ of length } k-1 \wedge \exists \text{ edge } r \to j) \\ &= P(\exists \text{ path } i \to r \text{ of length } k-1) \cdot P(\exists \text{ edge } r \to j \mid \exists\ i \to r\ ) \circledast \end{aligned}$$

where $\circledast$ is obtained by the rule $P(A \wedge B) = P(A|B)P(B)$.

At this point we have that

- $P(\exists \text{ path } i \to r \text{ of length } k-1) \leq \frac{1}{c^{k-1}m}$ by inductive hypothesis
- $P(\exists \text{ edge } r \to j \mid \exists\ i \to k \text{ of length } k-1) \leq \frac{1}{cm}$, as this is the base case

And we conclude that

$$\circledast \leq \frac{1}{c^{k-1}m} \cdot \frac{1}{cm} = \frac{1}{c^k m^2} < \frac{1}{c^k m}\ \square$$

Now, we know that the insertion time in this case is $O(1+k)$ as

- 1 to compute the first position i
- k to follow the path from i to j

At this point we can compute the average cost for **2)** by computing the expected length of the path $i \to j$

$$\begin{aligned} O(1+E[k]) = \\ &= O(1+\Sigma_{k=1}^{n}\ k \cdot \frac{1}{c^k m}) \\ &= O(1+\frac{1}{m}) \bigstar \\ &= O(1) \end{aligned}$$

$E[X=x] = x \cdot P(X=x)$ SAME THING, SLOPPY NOTATION

SAID EASY: THE COST IS GIVEN BY 1 + THE EXPECTED LENGTH OF THE PATH: #TIMES WE RELOCATE ELEMENTS

$\bigstar$ is given by the fact that $\Sigma_{k=1}^{n}\ k \cdot \frac{1}{c^k m} = \frac{1}{m}$ is a known summatory.

Case 3: No Free Position, Rehashing

In this case a cycle appears and thus a rehashing is performed in $O(n)$ time.
We choose two new hash functions $h_1', h_2' \in \mathcal{H}$ and we reinsert all the keys from scratch.

$$\begin{aligned} P(\exists \text{ cycle in } G) =_{\text{UB}} \Sigma_{i=0}^{m-1}\ P(\exists \text{ a path from } i \text{ to } j = i) \\ &= \Sigma_{i=0}^{m-1} \Sigma_{k \geq 1}\ P(\exists \text{ path } i \to j = i \text{ of length } k) \\ &\leq \Sigma_{i=0}^{m-1} \Sigma_{k \geq 1}\ \frac{1}{c^k m}\ , \text{ as in case 2)} \\ &= \frac{1}{m} \Sigma_{i=0}^{m-1} \Sigma_{k \geq 1}\ \frac{1}{c^k} \\ &< \frac{1}{m} \Sigma_{i=0}^{m-1} \frac{1}{c-1}, \text{ as } \Sigma_{k \geq 1}\ \frac{1}{c^k} < \frac{1}{c-1} \\ &= \frac{1}{c-1} \diamondsuit \end{aligned}$$

$\diamondsuit$ is obtained since $\frac{1}{c-1}$ is independent in the Σ, hence we take it out.
The summatory becomes then $\frac{1}{c-1}\frac{1}{m}\Sigma_{i=0}^{m-1}\ 1 = \frac{1}{c-1}\frac{1}{m}m = \frac{1}{c-1}$

Since we assumed that $c > 2$ we have that $\frac{1}{c-1}$, which is the probability p of rehashing, is < 1.

So we now we know that:

- rehashing occurs with probability nearly $p = \frac{1}{c-1} < 1$
- rehashing takes $O(n)$ if it succeeds

The point is now: **how many rehashing?**

- 1 rehash with probability p
- 2 rehash with probability p^2
- ...
- t rehash with probability p^t

The expected number of rehashing is $\Sigma_{t>=1} t p^t = O(1)$ as $p < 1$.

So, about the insertion: if there is a cycle starting at position i we have $O(1)$ rehash on average, then:

$$
\begin{aligned}
P(\exists \text{ cycle starting from } i) \leq \\
&\leq P(\exists \text{ path } i \to j = i) \\
&= \Sigma_{k=1}^{n} \frac{1}{c^k m} \\
&= \frac{1}{m} \Sigma_{k=1}^{n} \frac{1}{c^k} \\
&< \frac{1}{m} \frac{1}{c-1}, \text{ as } \Sigma_{k=1}^{n} \frac{1}{c^k} < \frac{1}{c-1} \\
&= \frac{1}{m} \cdot p, \text{ as } \frac{1}{c-1} = p \\
&= \frac{p}{m}
\end{aligned}
$$

Thus we pay $O(n)$, the cost of rehashing, with probability $\frac{p}{m}$, giving $O(\frac{np}{m}) = O(1)$ expected cost
□

Bloom Filter

A bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries.
Bloom filters allow false positives but the space saving often outweigh this drawback when the probability of an error is controlled.

A **bloom filter does not store the keys!**
We have a set $S \subseteq U$ of keys, but we do not want to store them in the BF, as they are too many.
The goal is to check if $x \in S$, without storing the keys in S.

Let's take a universal hash family $\mathcal{H}$, we choose at random k hash functions $h_1, \ldots, h_k \in \mathcal{H}$ where $h_i : U \to [m]$.
We then take a bit vector B, a compact array of m bits initialized to 0s, and we do

$$x \in S \to_{BF} \forall\, B[h_i(x)] = 1$$

To check whether x is in S we simply return $B[h_1(x)] \wedge \cdots \wedge B[h_k(x)]$ $\diamondsuit$
It is clear that:

- $lookup(x)$: check the membership of a key cost $O(k)$ time
- $insert(x)$: insert an key cost $O(k)$ time
- $delete(x)$: deleting an element is not supported in this version

As before, there is the possibility of having a 1-sided error, a **false positive**:

$$x \notin S \wedge (B[h_1(x)] \wedge \ldots B[h_k(x)] = 1)$$

Let's now say that f is the probability of error:

$$f = P(\text{error}) = P(x \notin S \text{ but } \diamondsuit \text{ is true})$$

We have now two points to solve:

1. what is the error probability f
2. what are the best parameters to minimize f?

Error Probability

Suppose that the bloom filter has been built, we have B, and we take a generic position q.

step 1)
$P(B[q] = 1) = 1 - P(B[q] = 0)$

step 2)
Let's fix a function h_i, we know that $P(h_i(x) = q) = 1/m$ by definition of $h_i \in \mathcal{H}$
We know that $B[q] = 0$ when $h_i(x) \neq q$
So we have $P(h_i(x) \neq q) = 1 - P(h_i(x) = q) = 1 - \frac{1}{m}$
Alternatively said: $P(B[q] = 0 \mid h_i \text{ is chosen}) = 1 - \frac{1}{m}$

step 3)
Let's relax the previous step by removing the fixated i-th hash function:

$$\begin{aligned} P(B[q] = 0) = & \\ &= P(\text{all the } k \text{ hash functions do not give q}) \\ &= (1 - \frac{1}{m})^k \end{aligned}$$

$\forall k. h_k(x) \neq q$

step 4)
We relax the previous step: we apply the same reasoning for all the keys in S ($|S| = n$)

$$P(B[q] = 0) = [(1 - \frac{1}{m})^k]^n$$

step 5)
We can apply the result of **step 4)** in the equation of **step 1)**

$$
\begin{aligned}
P(B[q]=1) = & \\
& = 1 - P(B[q]=0) \\
& = 1 - (1 - \frac{1}{m})^{nk} \\
& = 1 - p', \text{ as we define } p' = (1 - \frac{1}{m})^{nk}
\end{aligned}
$$

We have then computed the probability of failure:

$$f = P(\text{error}) = P(x \notin S \text{ but } \diamondsuit \text{ is true}) = (1-p')^k$$

Choosing the Best Parameters

We want to **minimize** the failure probability $f = (1-p')^k$, where $p' = (1 - \frac{1}{m})^{nk}$
Remember the Taylor expansion: $(1+y) \sim e^y$ for small y.
Then we can write

$$p' = (1 - \frac{1}{m})^{nk} \sim_{y=\frac{1}{m}} (e^{-\frac{1}{m}})^{nk} = e^{-\frac{nk}{m}} = p$$ } THANKS TO TAYLOR: $p' \simeq p$

So we now try to minimize $(1-p)^k$, using the logarithm to both sides of the equation:

$$
\begin{aligned}
& 1) \log(1-p)^k = k \log(1-p) \\
& 2) \log(p) = -\frac{nk}{m}\log(e) \Rightarrow \log(p) = -\frac{nk}{m} \rightarrow k = -\frac{m}{n}\log(p)
\end{aligned}
$$

We can then put $1)$ in $2)$ and we obtain

$$k\log(1-p) = -\frac{m}{n}\log(p)\log(1-p)$$

We minimize over p, it's the best choice for p is $\frac{1}{2}$

$$
\begin{aligned}
& f \sim (1-p)^k = \frac{1}{2^k} = \frac{1}{2^{-\frac{m}{n}\log(1/2)}} = \frac{1}{2^{\frac{m}{n}\log(2)}} = (2^{-\log(2)})^{\frac{m}{n}} < 0.618^{\frac{m}{n}} \\
& k = -\frac{m}{n}\log(p) = \frac{m}{n}\log(2)
\end{aligned}
$$

observations:

- $2^{-log(2)} = 0.618$
- you have to choose m accordingly

□

Space Complexity

We have seen how Bloom Filter is a solution for a membership problem that actually do not store the keys.
The **space complexity** is: m bits plus $O(k)$ words:

- m bits to store the bitmap B
- $O(k)$ words to store k hash functions: storing the pairs (a,b) of each h_i takes k words

Let's be more careful about the space.
Consider f and do the logarithm:

$$\log(f) \sim \frac{m}{n}\log(2)\log(\frac{1}{2}) =_{(\log(1/2)=-1)} -\frac{m}{n}\log(2)$$
$$\implies m \sim \frac{n\log(f)}{\log(\frac{1}{2})\log(2)} = \frac{n\log(\frac{1}{f})}{\log(2)} \sim 1.44 \cdot n\log(\frac{1}{f}) \text{ bits}$$

We have computed both the time and space complexity of BF.

Approximate Dictionary

Consider a set S of n keys chosen from a universe U.
For a given (1-side) error probability $0 < f < 1$, we learned that Bloom Filters achieve probability f using $k \sim \frac{m}{n}\log(2)$ hash functions that map $U \to [m]$.
They take $O(k)$ time and use nearly $1.44 \cdot n\log(1/f)$ bits of space
Is it possible to do better?

Information Theory Lower Bound (IT)
Take a set $S \subseteq U$ and let $|U| = m$ and $|S| = n$.
How many sets $S \subseteq U$ of size n we can have? $\binom{m}{n}$
The information theory lower bound states that using less than $\log\binom{m}{n} \sim n\log(\frac{m}{n})$ bits cannot give a correct algorithm: indeed using less bits forces two sets S' and S'' to get the same binary representation, so the *exact* membership is impossible.

HERE ARE THE LOWER BOUNDS FOR THE MEMBERSHIP PROBLEM

- time: **trivial**, $\Theta(1)$ for both query and insertion
- space: $\geq n\log(\frac{1}{f})$, **to prove**

HERE ARE THE UPER BOUNDS FOR THE MEMBERSHIP PROBLEM

- time: $O(1)$ with 1 hash function, **to prove**
- space: $\sim n\log(\frac{1}{f})$ bits $+$ lower order terms, **to prove**

Lower Bound

OBV SAME SET, $S \subseteq S'$

Let's call D' the approximate dictionary for a set S with an error probability $1/f$.
Then let's call D the exact dictionary for some $S \subseteq S'$, and we say that

$$\frac{|S'/S|}{U} = f$$

We have that

$$\begin{aligned} S' &= \{x \in U : \text{ approximate dictionary says yes}\} \\ &= \{x \in U : D'(x) = true\} \end{aligned}$$

S are the actual elements, S' are the elements that D' recognize as part of S.
D' can be wrong.
We can have a 1-side error for S: $S \subseteq S'$ as all keys in S are accepted, plus the extra (*wrong*) keys in S/S'.

Consider this problem:

- let D' be the exact dictionary for S'
  - notice that D' is also an approximate dictionary for S
- let D be the approximate dictionary for S

Question: given D', can we get D?
Let's say that b' is the number of bits required by D'.
Then we have that $D = D' + \text{ extra bits to mark correct answers from } D'$
So we get an exact dictionary D, and then it must require at least $\log \binom{m}{n}$ by IT.

So we have:

$$|D| \geq \log \binom{m}{n}, \text{ by IT}$$
$$|D' + \text{extra bits}| \geq \log \binom{m}{n}, \text{ as } D' + \text{ extra bits is a dictionary for } S$$

If we look closely we see that

$$|D' + \text{extra bits}| = |D'| + \clubsuit \log \binom{|S'|}{|S|} \bigstar$$

$\clubsuit$: number of bits we need to mark D' to make it an exact dictionary

We then remember the initial assumptions:

$$\begin{aligned} |S'| = & \\ &= |S| + |S \backslash S'| \\ &= |S| + |U| \cdot f \star \\ &= n + fm \end{aligned}$$

$$\star : \frac{|S/S|}{|U|} = f \iff |S'/S| = fm$$

So we can resume $\bigstar$:

$$\begin{aligned} |D' + \text{extra bits}| = & \\ &= |D'| + \log \binom{|S'|}{|S|} \\ &= b' + \log \binom{n+fm}{n} \end{aligned}$$

Putting things together:

$$
\begin{aligned}
b' + \log \binom{n+fm}{n} \geq \log \binom{m}{n} = & \\
& = b' \geq \log \binom{m}{n} - \log \binom{n+fm}{n} \\
& \sim b' \geq n \log(\frac{m}{n}) - n \log(\frac{n+fm}{n}) \ \nabla \\
& \sim b' \geq n \log(\frac{m}{n}) - n \log(\frac{fm}{n}) \ \star \\
& = b' \geq n(\log(\frac{m}{n}) - \log(\frac{fm}{n})) \\
& = b' \geq n(\log(\frac{m}{n}) + \log(\frac{n}{fm})) \\
& = b' \geq n(\log(\frac{m}{n} \cdot \frac{n}{fm})) \\
& = b' \geq n \log(\frac{1}{f}) \ \square
\end{aligned}
$$

Where:

- ∇ : $\log \binom{m}{n} \sim n \log(\frac{m}{n})$
- $\star$: $\log(\frac{n+fm}{n}) \sim \log(\frac{fm}{n})$ as $\frac{n+fm}{n} = 1 + \frac{fm}{n}$

We proved the lower bound.

Upper Bound

The **upper bounds** for the membership problem:

1. time: $O(1)$ with 1 hash function:
    - We use a Succinct **Rank** Data Structure: bit-vector, m bits with n of them $= 1$
        - constant-time lookup
        - space complexity $\log \binom{m}{n} + \text{ lower order terms}$
2. space: n/f

This section is sloppy and non-complete as it was not explained during the lectures, it was delegated to personal studying.

Chebyshev's Inequality (CI)

In probability theory, Chebyshev's inequality **guarantees that no more than a certain fraction of values can be more than a certain distance from the mean.**
Let's remember the variance

$$\sigma^2 = E[X^2] - E[X]^2$$

Then we define the **Chebyshev's Inequality**:

$$P(|X - E[X]| \geq k\sigma) \leq \frac{1}{k^2}$$

Chernoff's Bound (CB)

In probability theory, a Chernoff bound is an exponentially decreasing upper bound on the tail of a random variable based on its moment generating function.
The minimum of all such exponential bounds forms the Chernoff Bound, which may decay faster than exponential.
It is especially useful for sums of independent random variables.

Let's consider a set of independent identically distributed (*iid*) variables $Y_i \in [0, 1]$

- $Y = \Sigma_{i=1}^{n} Y_i$
- $\mu = E[Y]$

Then we have the following rules, the Chernoff's Bound:

$$P(Y > \mu + \lambda) \leq e^{-\frac{\lambda^2}{2\mu+\lambda}}$$
$$P(Y < \mu - \lambda) \leq e^{-\frac{\lambda^2}{3\mu}}$$

where λ our slack parameter.

Load Balancing

Consider m servers and n jobs $(0, \dots, n-1)$, with $n >> m$
We define a **fair load:** $\frac{m}{n}$ jobs per server.

Objectives of a good load balancing strategy

- **transparency:** open source code, the mechanism is known but it can't be shut down
- **persistence:** job i always go to a small group of servers, to exploit caching, locality, ...
- **fairness:** get the average fair load for each machine

Randomization is fair

Randomly choose an hash function $h \in \mathcal{H}$.
Using h so that we have job $i \rightarrow$ server $h(i)$ gives a fair load

$$X_j = [\text{load for machine } j] = [\text{number of jobs assigned to server j}]$$

Let's also define the variable X_{ij}

$$X_{ji} = \begin{cases} 1 \text{ if job } i \text{ is assigned to server } j = h(i), \text{ with } p = \frac{1}{m} \\ 0 \text{ otherwise} \end{cases}$$

So we can compute

$$X_j = \Sigma_{i=0}^{n-1} X_{ji}$$
$$E[X_j] = \Sigma_{i=0}^{n-1} E[X_{ji}] = \Sigma_{i=0}^{n-1} P(X_{ji} = 1) = \frac{n}{m} \square$$

What about the max load?
Let's remember two rules that we will apply in the computation of the max load

- $P(X_{ji} = 1 \land X_{ji'} = 1) = P(X_{ji'} = 1)P(X_{ji'} = 1) = P(h(j) = i)P(h(j) = i')$ as $h \in \mathcal{H}$ is two-way independent

- given indicator variable X, Y, Z pair-wise independent, we have $\sigma^2(X+Y+Z) = \sigma^2(X) + \sigma^2(Y) + \sigma^2(Z)$

From the previous two points we can say that

$$\sigma^2(X_j) = \Sigma_{i=0}^{n-1}\ \sigma^2(X_{ji}) \star$$

Now, we have:

$$\begin{aligned} \sigma^2(X_j) \quad &= E[X_j^2] - E[X_j]^2 \\ &= \star\ \Sigma_{i=0}^{n-1} E[X_{ji}^2] - \Sigma_{i=0}^{n-1} E[X_{ji}]^2 \\ &= \Sigma_{i=0}^{n-1} (E[X_{ji}^2] - E[X_{ji}]^2) \\ &= n(\frac{1}{m} - \frac{1}{m^2})\ \bigstar \end{aligned}$$

Where $\bigstar$ is obtained because:

- $E[X_{ji}^2] = E[X_{ji}] = \frac{1}{m}$ by definition of random indicator variables
- $E[X_{ji}]^2 = (\frac{1}{m})^2 = \frac{1}{m^2}$

Hence we have $\sigma = \sqrt{n(\frac{1}{m} - \frac{1}{m^2})}$

Let's now set $k = \sqrt{2m}$ and use the Chebyshev's Inequality $P(|X - E[X]| \geq k\sigma) \leq \frac{1}{k^2}$:

$$P(|X_j - \frac{n}{m}| \geq \sigma\sqrt{2m}) \leq \frac{1}{2m}$$

But since $\sigma\sqrt{2m} = \sqrt{2m \cdot n(\frac{1}{m} - \frac{1}{m^2})} \sim \sqrt{2n}$, we get:

$$P(|X_j - \frac{n}{m}| \geq \sqrt{2n}) \leq \frac{1}{2m}\ \diamondsuit$$

Then we can compute the following probability:

$$\begin{aligned} P(max_j\ |X_j - \frac{n}{m}| \leq \sqrt{2n}) = \\ &= 1 - [P(\exists j : |X_j - \frac{n}{m}| \geq \sqrt{2n})] \\ &= 1 - [\Sigma_{j=\emptyset}^{m-1} P(\quad\ |X_j - \frac{n}{m}| \geq \sqrt{2n})]_{\sim},\ \text{UB} \\ &= 1 - [\Sigma_{j=0}^{m-1} \frac{1}{2m}]_{\sim},\ \text{CI} \\ &= 1 - [\frac{1}{2}]_{\leq} \\ &\geq \frac{1}{2} \end{aligned}$$

The $\geq \frac{1}{2}$, given by the $\sim$, is bound to the fact that:

$$P(\exists j : |X_j - \frac{n}{m}| \geq \sqrt{2})\ \leq_{\text{UB}}\ \Sigma_{j=1}^{m-1} P(\exists j : |X_j - \frac{n}{m}| \geq \sqrt{2})\ \leq_{\text{CI}}\ \Sigma_{j=0}^{m-1} \frac{1}{2m} = \frac{1}{2}$$

In words: the max load of a given machine is at most $\frac{n}{m} + \sqrt{2n}$ with a probability $\geq \frac{1}{2}$

We can now also apply the Chernoff's Bound (CB) $P(\gamma > \mu + \lambda) = e^{-\frac{\lambda^2}{2\mu+\lambda}}$,
where we set:

- $n = m$
- $\mu = E[X_j] = \frac{n}{m} = 1$
- $\gamma = X_j$
- $\lambda = 6\log(n)$

And we have:

$$P(X_j \geq \frac{n}{m} + 6\log(n)) \leq e^{-\frac{(6\log n)^2}{2\frac{n}{m}+6\log(n)}} \leq \ e^{-3\log(n)} = \frac{1}{n^3}\diamondsuit, \text{ w.h.p}$$

So that we can compute

$$P(max_j\ X_j \leq \frac{n}{m} + 6\log(n)) = 1 - \Sigma_{j=1}^{m}\diamondsuit \geq 1 - n \cdot \frac{1}{n^3} = 1 - \frac{1}{n^2}$$

which bounds the max load.

Azuma-Hoeffding Bound (AH)

Consider $X_1, \ldots, X_k$ *i.i.d* random vars, with

$$\mu = E[\frac{\Sigma_{i=1}^{k} X_i}{k}]$$

and with $a_i \leq X_i \leq b_i$
Then we have the Azuma-Hoeffding Bound:

$$P(|Y - \mu| \geq \epsilon) \leq 2e^{-\frac{2k^2\epsilon^2}{\Sigma_{i=1}^{k}(b_i - a_i)^2}}$$

Document Resemblance

Given a set X of elements, we have $|X|!$ permutations.
For example: $X = \{a, b, c\}$ has $3! = 6$ permutations: $abc, acb, \ldots, cba$

Each permutation **induces an order**, for example $cab \implies c < a < b$.
We define the minimum of a permutation as the first element of the permutation.
Alternatively said, each permutation can be seen as a ranking of X.

Consider a set $X \subseteq U$, and permutation $h : U \to [|U|]$.
We say a that a *permutation* family $\mathcal{H} = \{h : X \to [|X|],\ h \text{ bijective}\}$ is **min-wise independent** if

$$\forall X \subseteq U. \forall a \in X. P_{h \in \mathcal{H}}(a = min\ h(x)) = \frac{1}{|X|}$$

in words: every element has the same probability of being the minimum.

Say that $h(X)$ is a permutation of X, then we want

$$min\ h(X) = arg\_min_{a \in X}\ h(a)$$

Said easy: $min\ h(X)$ returns the $a \in X$ for which $h(a)$ is minimum.

We like the min-wise independency property because of **Jaccard's Index for Similarity**.
We can see a document as a (multi)set of ("lemmatized") words.
So, given U the set of all the possible words, consider two subsets $A, B \subseteq U$.
A, B are two files, and they can be seen as a k-permutation of U.
We define the Jaccard's Index as

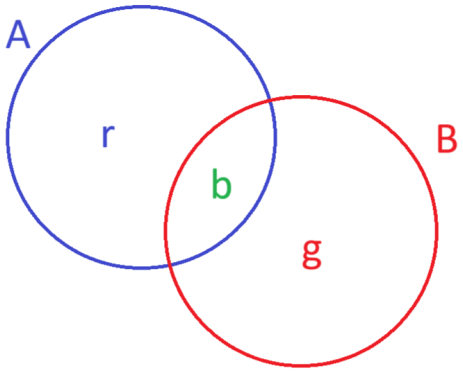
$$0 \leq J(A,B) = \frac{|A \cap B|}{|A \cup B|} \leq 1$$

After sorting A, B computing $J(A,B)$ takes $O(|A| + |B|)$ time.
In the context of Big Data, where A, B are enormous, this is a no go.

Interesting property:

$$\forall A, B \,.\, P_{h \in \mathcal{H}}(min\ h(A) = min\ h(B)) = J(A,B)$$

proof:
Consider the set $X = A \cup B$, with $|X| = r + b + g$.



So, we have that:

- $r = |A/B|$
- $b = |A \cap B|$
- $g = |B/A|$
- $|A| = r + b$
- $|B| = b + g$
- $|A \cap B| = b$
- $|A \cup B| = r + b + g$

Then:

$$\begin{aligned}
P(min\ h(A) = min\ h(B)) = & \\
&= P(\exists y \in A \cap B : y = min\ h(X)) \\
&\leq \Sigma_{y \in A \cap B}\ P(y = min\ h(x)), \text{UB} \\
&= \Sigma_{y \in A \cap B}\ \frac{1}{|X|}, \text{by def of } h \in \mathcal{H} \\
&= \frac{b}{r + b + g} \\
&= \frac{|A \cap B|}{|A \cup B|} \\
&= J(A,B)\ \clubsuit
\end{aligned}$$

Computing Jaccard is way too expensive as it requires sorting, and even after sorting is still very slow for big data problem.
To attack this problem we use the **k-min hash** technique.

It has been shown that we can "safely" replace the permutation family $\mathcal{H}$ with our universal family $\mathcal{H}$.
We then can take $h_1, \dots, h_k \in \mathcal{H}$, where we know that $\forall h_i$ ♣ holds

We compute the **sketches** of A, and B:

- $S(A) : \{min\ h_1(A), \dots,\ min\ h_k(A)\}$
- $S(B) : \{min\ h_1(B), \dots,\ min\ h_k(B)\}$

And we define the **approximate Jaccard's Index**

$$J_{\sim}(A, B) = \frac{|S(A) \cap S(B) \cap S(A \cup B)|}{|S(A \cup B)|}$$

Given $S(A), S(B)$, computing $J_{\sim}(A, B)$ takes $O(k)$ time, which is much faster than computing $J(A, B)$.
The price to pay is obvious: **it is an approximation.**

How good of an approximation?
Consider A, B, and k hash functions in $\mathcal{H}$.
Let's define an indicator variable

$$X_i = \begin{cases} 1 \text{ if } min\ h_i(A) = min\ h_i(B), \text{ with probability } p = J(A, B) \\ 0 \text{ otherwise} \end{cases}$$

X_i tells us if the minimum using h_i on A and B is the same, aka if they have the permutations have the same first element.
Summing X_i for every $i \in [k]$ tells us how many times the minimums are the same.
Then, as always for indicator variables, we have that $E[X_i] = J(A, B)$.

Let's define

$$Y = \frac{\Sigma_{i=1}^{k} X_i}{k}$$

which is an **unbiased estimator** for $J(A, B)$.
An estimator of a given parameter is said to be **unbiased** if its expected value is equal to the true value of the parameter.
In other words, an estimator is unbiased if it produces parameter estimates that are on average correct:

$$E[Y] = \frac{\Sigma_{i=1}^{k} E[X_i]}{k} = \frac{k \cdot J(A, B)}{k} = J(A, B)$$

Is this k-min-hash technique any good to compute the document resemblance?
Let's define a variable

$$
\begin{aligned}
Y' &= \Sigma_{i=1}^{k} X_i \\
&\implies \\
Y' &= kY \\
\mu &= E[Y] \\
\mu' &= E[Y'] = k \cdot E[Y] = k \cdot J(A,B)
\end{aligned}
$$

Using the **CB**, we have

$$
\begin{aligned}
P(|Y - \mu| \geq \epsilon\mu) = & \\
&= P(|Y' - \mu'| \geq \epsilon\mu'), \text{ multiplied by } k \heartsuit \\
&\leq 2 \cdot e^{-\frac{\epsilon^2\mu'}{3}}, \text{ CB} \\
&= 2e^{-\frac{\epsilon^2 k\mu}{3}}, \bigstar
\end{aligned}
$$

$\heartsuit$: if we multiply by a constant the event are the same, therefore they have the same probability.

$\bigstar$: This tells that $|Y - \mu|$ (how Y and its expected value μ are distant) is equal or greater than $\epsilon\mu$ with that probability.

Is this good?

We **want to bound this probability** by a $\delta < 1$.

Then we have

$$
2e^{-\frac{\epsilon^2 k\mu}{3}} < \delta \implies k = O(\epsilon^{-2}\log(\delta^{-1})\mu^{-1})
$$

but we know that $\mu = J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, then $\mu^{-1} \sim |A| + |B|$ for A, B very different from each other.

Then, to limit the probability we have to take k hash functions, with k large as $|A| + |B|$, which is very bad.

Let's try to bound that probability with another tool.

We use the Azuma-Hoeffding Bound: $P(|Y - \mu| \geq \epsilon) \leq 2e^{-\frac{2k^2\epsilon^2}{\Sigma_{i=1}^{k}(b_i - a_i)^2}}$

In this case:

- $X_i \in [a_i = 0, b_i = 1]$
- $\mu = E[\frac{\Sigma_{i=1}^{k} X_i}{k}]$, where $\frac{\Sigma_{i=1}^{k} X_i}{k}$ is our Y

Then:

$$
P(|Y - \mu| \geq \epsilon) \leq 2e^{-\frac{2k^2\epsilon^2}{k}} \star = 2e^{-2k\epsilon^2}
$$

$\star$: $a_i = 0, b_i = 1 \implies \Sigma_{i=1}^{k}(b_i - a_i)^2 = k$

And we want to limit that probability with δ:

$$
2e^{-2k\epsilon^2} \leq \delta \implies k = O(\epsilon^{-2}\log(\delta^{-1}))
$$

Which is good, □

the application of this technique, the network analysis, is skipped.

Count-Min Sketches for Frequent Elements

Consider a stream (possibly infinite) $a_0, \dots$ of elements $\in U$.
We want to know how many times a given element a has appeared so far in the stream.
Let's define a frequency array F such that:

$$F[a] = \text{number of times } a \text{ is appeared so far}$$

The point is that $||F|| = \Sigma_{a \in U} F[a]$ is not computable when U is huge, we do not have enough memory.

Then we set the two following goals:

1. small space occupation
2. two user-defined parameters:
   - δ, the error probability
   - ε, we cannot compute $F[]$ exactly so we give an ε-approximation

To do so we use the **count-min skectch**.
We set:

- $|U| = n$
- update of the frequency array: $F[i]$++ for every $i \in [n]$
- query: return $F[i]$
- $||F|| = \Sigma_{a \in U} F[a]$

And, given ε, δ we estimate $F_\sim[i]$ such that

$$F[i] \leq F_\sim[i] \leq F[i] + \varepsilon ||F||$$

where $F_\sim[i] \leq F[i] + \varepsilon ||F||$ holds with a probability $1 - \delta$.
The problem is that if $F[i]$ is small compared to $\varepsilon ||F||$ the estimation is useless.

CM-Sketch is a sort of Bloom Filters with Counters.
Given the input ε and δ we built a table $T = r \times c$, where:

- r are the rows, $r = \log(\delta^{-1})$
- c are the columns, $c = \frac{e}{\varepsilon}$

Each entry of the table is a counter and it is initialized to 0.
Choose uniformly and randomly $h_0, \dots, h_{r-1} \in \mathcal{H}$ where $\mathcal{H}$ is a two-way independent universal hash family, where $m = c$.
The item $i \in U$ of the stream is associated with the entries of T at positions

$$\{\langle 0, h_0(i) \rangle, \dots, \langle r-1, h_{r-1}(i) \rangle\}$$

Then we have the two operations, both costing $O(r)$:

- a new element i is seen: $F[i]$++ $= T[0][h_0(i)]$++$, \dots, T[r-1][h_{r-1}(i)]$++
- how many times i has been seen: return $F_\sim[i] = \min \star (T[0][h_0(i)], \dots, T[r-1][h_{r-1}(i)])$

Issue: $F_\sim[i] \geq F[i]$ by construction, but it could be too large!
Let j be the row that gives the minimum ($\star$):

$$F_{\sim}[i] = T[j][h_j(i)] = F[i] + X_{ji}$$

Where X_{ji} is "garbage" due to the updates $F[k]$++ of other $k \neq i$.
X_{ji} is the increasing (w.r.t $F[i]$) that the cell $T[j][h_j(i)]$ had due to the collision, for elements $k \neq i$, in the computation of $h_j(i)$.
Since, if this collision $h_j(i) = h_j(k)$ happen, it happens all the time we see k, we have an error equal to the times we have seen k, hence $F[k]$ times.
More specifically:

$$X_{ji} = \Sigma_{k \neq i \wedge h_j(i) = h_j(k)} \, F[k] = \Sigma_{k=1}^{n} \, I_{jik} \cdot F[k]$$

where I_{jik} is an indicator variable

$$I_{jik} = \begin{cases} 1 \text{ if } h_j(i) = h_j(k), \text{ with } k \neq i \\ 0 \text{ otherwise} \end{cases}$$

that is *true* if a collision did happen.

We want to show: $P(X_{ji} > \varepsilon ||F||) < \delta$ ★
We then compute:

$$E[I_{jik}] = P(\mathbb{I}_{jik} = 1) = \frac{1}{c} =_{c=\frac{e}{\varepsilon}} \frac{\varepsilon}{e}$$

and use it to compute the expected value of X_{ji}

$$\begin{aligned} E[X_{ji}] = & \\ & = \Sigma_{k=1}^{n} \, E[I_{jik}] \cdot F[k] \\ & = \Sigma_{k=1}^{n} \, \frac{\varepsilon}{e} \cdot F[k] \\ & = \frac{\varepsilon}{e} \cdot ||F|| \diamondsuit \end{aligned}$$

$\diamondsuit$ is obtained by the facts that:

- k do not appear in $\frac{\varepsilon}{e}$
- $||F|| = \Sigma_{k=1}^{n} F[k]$ by definition

From $\diamondsuit$ we get that:

$$e \cdot E[X_{ji}] = \varepsilon ||F||$$

We can then apply the **Markov's Inequality:** $P(X > z) \leq \frac{E[X]}{z}$
In this case we set:

- $z = \varepsilon ||F||$
- $X = X_{ji}$

And we get:

$$P(X_{ji} > \varepsilon ||F||) \leq_{MI} \frac{E[X_{ji}]}{\varepsilon ||F||} = \frac{E[X_{ji}]}{e \cdot E[X_{ji}]} = \frac{1}{e} \clubsuit$$

And we are done:

$$
\begin{aligned}
P(F_{\sim}[i] > F[i] + \varepsilon||F||) = \\
&= P(\forall j : P(X_{ji} > \varepsilon||F||)) \heartsuit \\
&\leq \Pi_{j=1}^{r} \frac{1}{e}, \text{ as } h_j \text{ are independently chosen, and } \clubsuit \\
&= \frac{1}{e^r} \\
&= \delta, \text{ as } \delta = \log(\frac{1}{\delta}) \text{ by def}
\end{aligned}
$$

$\heartsuit$: $F_{\sim}[i]$ is a minimum operation over the rows of T: it has to be verified for every cell!
which proves $\bigstar$, $\square$

Randomized Min-Cut Algorithm for Graphs

Consider a **(multi)graph** $G = (V, E)$ (there may be multiple edges between same nodes), where:

- $|V| = n$
- $|E| = m$

We define the **cut of the graph**

$$C = (V_1, V_2) \text{ where } V_1 \cap V_2 = \emptyset, \ V_1 \cup V_2 = V$$

And we call such V_1 and V_2 partitions of the graph G.

We then define a **cut-set:**

$$E(V_1, V_2) = \{uv \in E : (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)\}$$

A cut-set is the set of edges that goes from a partition V_1 to a partition V_2 of a cut (or vice-versa). Mind that we will often represent $E(V_1, V_2)$ as $E(C)$.

The previous definitions are useful to partition the graph G into subgraphs.
We define the **node induced graph**

$$G[V_i] = \{uv \in E : u, v \in V_i\}$$

And the **subgraph induced by the cut-set**

$$G[E(V_1, V_2)] = (V_{12}, E(V_1, V_2))$$

where $V_{12} = \{u \in V : uv \in E(V_1, V_2)\}$

Said easy, given a graph G and and a cut $C = (V_1, V_2)$ we have:

- **node induced graph:** a graph where every vertex v is in V_1 and every edge only connect vertexes of V_1
- **subgraph induced by the cut-set**: a graph where every vertex $u \in V_1$ has at least one edge that connect it to a vertex v of the other partition V_2

Every cut $C = (V_1, V_2)$ divides a graph G in three parts:

1. $G[V_1]$

2. $G[V_2]$
3. $G[E(V_1, V_2)]$

Computing the size of the subgraph induced by the cut-set is a problem:

- computing the lower bound of the size takes polynomial time
- computing the upper bound of the size is NP-hard

We say that the cut $C = (V_1, V_2)$ is a **min-cut** if

$$\forall C'. |E(C)| \leq |E(C')|$$

Said easy: the min-cut is the cut with the least amount of edges.

Observation: the min-cut is not unique. C is a min-cut
Observation: $\forall v \in V. |\overline{E(C)}| \leq deg(v)$, as otherwise we could choose a smaller cut-set (remember that $deg(v)$ is the degree of a vertex v, the number of edges from/to v)

Edge Contraction:
Consider an edge $uv \in E$.
We define the graph $G/uv = (V', E')$ where:

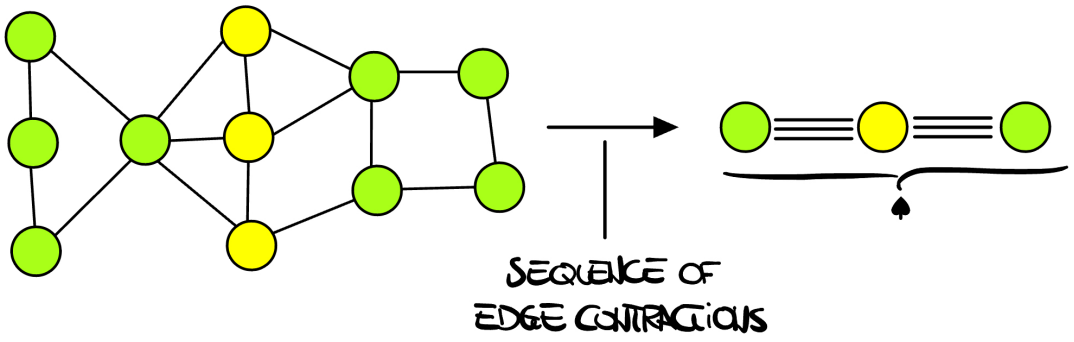
- $V' = V - \{u, v\} \cup \{\overline{uv}\}$
- $E' = E - \{uv\} \cup \{\overline{uv}z : uz \in E \vee vz \in E\}$

Contracting the edge uv means to merge the node u and the node v in a single node $\overline{uv}$, and then substitute all the edges where either u or v appear with edges of $\overline{uv}$.
Intuition: when performing the edge contraction with $\{u, v\}$, we are saying that u, v belong to the same "side" of the cut.

Relation between Edge Contractions and Cuts:
Consider a **connected graph** (every two vertices are connected by an edge) G.
Theorem A:
Any sequence of edge contractions leaving just 2 nodes gives a cut $C(V_1, V_2)$
Proof:
We prove it by induction on the number n of nodes:

- **base case:** $n = 2$
  - $V_1 = \text{one node}$
  - $V_2 = \text{the other node}$
  - we obtain the cut $C = (V_1, V_2)$
- **inductive case:** $n > 2$
  - we assume that with $n - 1$ nodes the theorem is true and we want to prove that with n nodes it takes a certain number of contractions to obtain a cut
  - we make an edge contraction and we are left with a graph with one less node than before: the graph has now $n - 1$ nodes
  - with $n - 1$ node we obtain a cut with a series of contraction by inductive hypothesis

Theorem B:
Given a cut $C = (V_1, V_2)$, it does not necessarily exists a sequence of edge contractions that leads to just 2 nodes.
Proof:



♠ : here we cannot apply any edge contraction that keeps this 2-colored (gives a cut).
Clearly, by the previous theorem, we can obtain another 2-coloring.

We can now define **the algorithm** that "guesses" the min-cut of a graph G:

```
GuessMinCut(G) {
        while(|V(G)| > 2)
                1) choose uniformly and randomly any edge uv in E
                2) G = G - uv
        return |E(G)|
}
```

Where:

- how we choose an edge $uv \in E$?
  - choose u with probability $\frac{deg(u)}{2m}$ ($2m$, the graph is connected!)
  - choose $v \in V(u)$ with probability $\frac{1}{deg(u)}$
- $|E(G)|$ is the number of multiple edges when only two nodes remain

When the previous algorithm fail?
When an edge uv is **bad**, which is a edge $uv \in E(C)$ for every min-cut C.
In other words, uv belongs to all the min-cut sets!
This is clear: `GuessMinCut` cannot find the min-cut when it removes an edge that belongs to every min-cut.

Let $k = |E(C)|$ be the size of min cut-set.
Proposition:
There are at most k bad edges in G.
Proof:
We defined $k = |E(C)|$, which is the number of edges in the min-cut.

The number of bad edges is the number of the shared edges within every min-cut, by definition of bad edges.
At most we could share k edges, but two solutions that have the same edges are the same solution (cut).
So, there are at most k bad edges, □

So we have that

$$P(uv \text{ is bad}) \leq \frac{k}{m} \bigstar$$

We also know that:

$$\begin{cases} \clubsuit \ \Sigma_{u \in V} \ deg(u) = 2m \\ \spadesuit \ \forall u \in V : deg(u) \geq k \end{cases} \implies (\heartsuit \ 2m \geq nk \iff m \geq \frac{nk}{2})$$

where:

- ♣ In a non oriented graph is it true that the sum of the degree of the vertexes is $2m$ since we count the edge uv twice: in the degree of u and in the degree of v
- ♠ Every node as degree $\geq k$, otherwise the min cut-size would be smaller than k.
  Alternatively said: otherwise I would color only one node and I would have less than k edges towards the other partition, and the min-cut would be smaller than k
- Thanks to ♣, ♠ I obtain ♡, which is true if and only if $m \geq \frac{nk}{2}$

So we can use the last point in ★

$$P(uv \text{ is bad}) \leq \frac{k}{m} \ \leq \frac{2}{nk} \cdot k = \frac{2}{n}$$

note: this argument holds for multi-edges as $deg(u)$ takes into account their multiplicities.

We then have:

$$P(n) = \text{probability that GuessMinCut(G) correctly finds a min-cut in a graph G with n nodes}$$
$$P(n) \geq \begin{cases} 1, \text{with } n = 2 \\ \star \ (1 - \frac{2}{n}) \cdot P(n-1), \ \text{with } n > 2 \end{cases}$$

Where in the case $\star$ we have:

- $1 - \frac{2}{n}$ is the probability that the chosen edge is nod a bad edge
- $(1 - \frac{2}{n}) \cdot P(n-1)$ is obtained by the formula $P(A \cap B) = P(A) \cdot P(B|A)$
  - $P(n-1)$ is the conditional probability given the choice of uv

We can expands $P(n)$ recursively, where remember that $1 - \frac{2}{n} = \frac{n-2}{n}$

$$P(n) \geq (\frac{n-2}{n}) \cdot (\frac{n-3}{n-1}) \cdot (\frac{n-4}{n-2}) \cdots \frac{1}{3} \cdot 1 = \frac{(n-2)!}{n!/2} = \frac{2}{n(n-1)}$$

Which is not great, but can be boosted (boosting the probability, we do not see it .

Diameter in Undirected Unweighted Graphs

Consider an undirected graph $G = (V, E)$, with $|V| = n$ and $|E| = m$.
We define the **distance** between two nodes $x, y \in V$ as follow:

$$d(x, y) = |\text{shortest path from } x \text{ to } y| = d(y, x)$$

Then we have the **Milgram's Experiment:**

$$\text{average distance} = \Sigma_{x,y \in V, x \neq y} \frac{d(x, y)}{\binom{n}{2}}$$

Let's then define the **diameter of a graph:**

$$\text{diameter of } G = D(G) = max_{x,y \in V,\, x \neq y}\, d(x, y)$$

Baseline Algorithm to compute the Diameter

Let's define the **eccentricity** of a node

$$ecc(u) = \max_{v \in V}\, d(u, v)$$

The eccentricity of a node u is the maximum distance among the distances with the other nodes of the graph.

The simplest way of computing the diameter D is to run the BFS from each node v in V, keeping track of the maximum distance from u, ♣
Alternatively said:

$$\begin{aligned} D = & \\ &= \max_{x \neq y} d(x, y) \\ &= \max_{x}(\max_{y \neq x} d(x, y)\ \clubsuit) \\ &= \max_{x} BFS(x) \end{aligned}$$

The cost of this process is $O(n \cdot (n + m)) = O(nm)$, where:

- n is the number of nodes
- $(n + m)$ is the cost of a single BFS

If G is sparse ($|E| = \Theta(N)$), as it is in real-world scenarios (e.g., networks), the baseline cost is $O(n^2)$, which is not usable in big-data contexts.
Mind that this is still the state of the art in the worst case.

Randomized Approach

The goal is to compute an approximation $D_\sim$ of the diameter for sparse graphs, with

$$\frac{2}{3} D \leq D_\sim \leq D$$

and doing so in $O(n\sqrt{n}\log(n))$ time with high probability.

High-Level Idea:
Consider the diameter $D = d(a, b)$ of a graph G (other diameter nodes can exists).

We have that $ecc(a) = ecc(b) = D$.

Let's also remember the **triangle inequality:** the sum of any two sides of a triangle is greater than or equal to the third side,
Making a triangle using three vertices of the graph, we have:

$$\text{TI} = d(x, y) + d(y, z) \geq d(x, z)$$

We set the parameter $k = \sqrt{n}$.
Then we define:

- $S =$ set of $O(\sqrt{n}\log(n))$ randomly selected nodes
- $w \in V$ is the farthest node from S

And we define the **Approximation Algorithm:**

1. $S =$ set of $\alpha\frac{n}{k}\log(n)$ randomly selected nodes from V, $\alpha\ cost > 2$ ♣
2. $w =$ farthest node from S ♠
3. $Z = S \cup N_k(w)$ ♡
   - $N_k(w) =$ the first k nodes traversed by $BFS(w)$, "truncated BFS"

- return $D_{\sim} = \max_{u \in Z}\ ecc(u)$

The running time of the algorithm is:

$$\clubsuit\ O(\alpha\frac{n}{k}\log(n)) + \spadesuit,\ \text{sparse G}, O(n) + \heartsuit\ O(|Z| \cdot n) =_{\alpha\ cost > 2,\ k=\sqrt{n}}\ O(n\sqrt{n}\log(n))$$

We then have the following statements:

$$\begin{aligned}
&a)\ S \text{ is a hitting set: } \forall u \in V.\, S \cap N_k(u) \neq \emptyset\ w.h.p. \\
&b)\ D = 3h + z, z \in \{0, 1, 2\}\ \diamondsuit \\
&\begin{cases} b_1)\ 2h + z \leq D_{\sim} \leq D, \text{ when } z \in \{0, 1\}\ (S) \\ b_2)\ 2h + 1 \leq D_{\sim} \leq D, \text{ when } z = 2\ (N_k(w)) \end{cases} \implies \frac{2}{3}D \leq D_{\sim} \leq D
\end{aligned}$$

$\diamondsuit$ We divided D by 3, so we have that D is $3h$ plus the reminder z, which can only be either $0, 1$ or 2.

Let's prove them:
$a)$ S is a hitting set:

$$\begin{aligned}
P(S \text{ hitting set for } N_k(u), u \in V) = \\
= 1 - P(\exists u \in V : S \cap N_k(w) = \emptyset)\ \bigstar
\end{aligned}$$

But we know that

$$p = P(\exists u \in V : S \cap N_k(w) = \emptyset) = (1 - \frac{|N_k(u)|}{n})^{|S|} = (1 - \frac{k}{n})^{|S|}$$

which is the probability that of a given element of S is not in $N_k(w)$.

We then remember that $e^x \sim 1 + x$, hence:

$$(1 - \frac{k}{n})^{|S|} \sim (e^{-\frac{k}{n}})^{|S|} = e^{-\frac{k}{n}(\alpha\frac{n}{k}\log(n))} = e^{-\alpha\log(n)} = \frac{1}{n^{\alpha}}$$

Which we can plug in ★

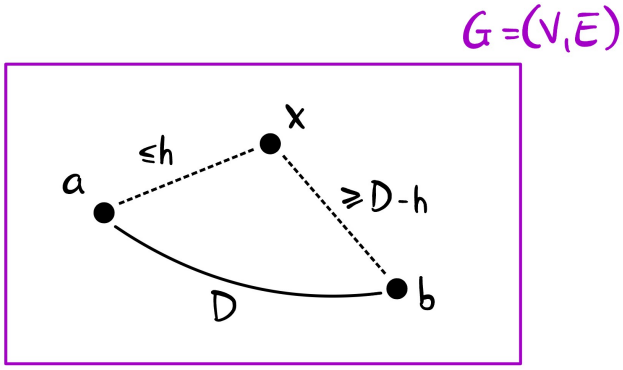
$$\begin{aligned} P(S \text{ hitting set for } N_k(u), u \in V) = \\ &= 1 - P(\exists u \in V : S \cap N_k(w) = \emptyset) \\ &=_{UB} 1 - np \\ &= 1 - \frac{n}{n^\alpha} \\ &= 1 - \frac{1}{n^\alpha - 1}, \alpha > 2 \end{aligned}$$

$b)\ D = 3h + z, z \in \{0, 1, 2\}$
We prove the two cases, starting with $b1)\ 2h + z \leq D_\sim \leq D,\ \text{when } z \in \{0, 1\}\ (S)$
Say that the diameter is $D = d(a, b)$.

Say that $d(w, S) \leq h$:

- w is the farthest node from $S \implies d(a, S) \leq h$
- take $x \in S$ closest to $a \implies ecc(x) \geq d(x, b)$ by definition of eccentricity

We use the TI (triangular inequality):



$$d(x, a) + d(x, b) \geq d(a, b) = D \implies d(x, b) \geq D - d(x, a) \geq_{d(x,a) \leq h} D - h$$

Which means that

$$ecc(x) \geq D - h = (3h + z) - h = 2h + z,\ \square$$

We now prove the second case, $b_2)\ 2h + 1 \leq D_\sim \leq D,\ \text{when } z = 2\ (N_k(w))$
$d(w, S) > h$

- if $ecc(w) \geq 2h + z$ we are happy
- hence, suppose $ecc(w) < 2h + z$
  - consider the shortest path w to $b : ecc(w) < 2h + z \implies d(a, b) < 2h + z$ otherwise we would have a bigger $ecc(w)$
  - S is an hitting set for $N_k(w)$, with high probability exists $x \in S \cap N_k(w)$
  - since we assume $d(w, S) > h \implies d(w, x) > h$
  - $\implies \exists w'$ along the path such that $d(w', w) = h$
  - $\implies d(w', b) < h + z$
  - $\implies$ we use again the TI:
    - $d(w', a) > D - d(w', b) \geq D - (h + z - 1) = (3h + z) - (h + z - 1) = 2h + 1, \square$

SETH Conjecture

I skipped a big chunk of this lecture as it was not understandable by the notes.

The **Strongly Exponential-Time Hypothesis** states that no algorithm can solve SAT in $O(2^{\alpha n'} poly(n))$ time for $\alpha \in [0, 1]$, unless $P = NP$.

SAT: Boolean Satisfiability Problem

SAT is the satisfiability problem of boolean formulas:

- n boolean vars: $x_1, \ldots, x_{n'}$
- literals: $x_i, \overline{x}_i$
- clauses are ORs of literals: $x_i \vee \overline{x}_j \vee x_k$
- CNF formulas are ANDs of clauses: $(x_i \vee \overline{x}_j \vee x_k) \wedge \ldots$
- define the assignment $\pi : [n'] \to \{0, 1\}^{n'}$
- π satisfy a CNF formula F, in symbols $\pi \vdash F$, if replacing every $x_i \in F$ with $\pi(x_i)$ gives $F = 1$

Computing π such that it satisfy a formula F takes $O(2^{n'} \cdot poly(n))$ time.
SAT is NP-complete.
Reminder: NP-complete problems are a subset of NP-hard problems and are exclusively decision problems, while NP-hard problems are not limited to decision problems and are at least as hard as NP-complete problems.

How to Attack HARD Problems

When hard means that the problem is either NP-complete or NP-hard.

We use a running example: the **Vertex Cover Problem**, which is NP-complete.
Given an undirected graph $G = (V, E)$ we say

$$S \subseteq V \text{ is a vertex cover if } \forall uv \in E : u \in S \vee v \in S$$

We can now distinguish the vertex cover problem and its decision problem:

- **minimization:** find $k_{min} : |S| = k_{min}$ such that S is a vertex cover and $\forall S'. |S'| < k_{min} \implies S' \text{ is not a vertex cover}$
- **decision:** consider VC_k, a vertex cover of size k, does exists a smaller vertex cover?

We focus on the decisional problem.

Baseline for Vertex Cover

There are $\binom{n}{k}$ subsets S of V, where $|S| = k$ and $|V| = n, |E| = m$.
We can check each subset in polynomial time, hence the total time is $O(\binom{n}{k})$, which is $O(n^{k+O(1)})$ since $\binom{n}{k} \sim n^k$.

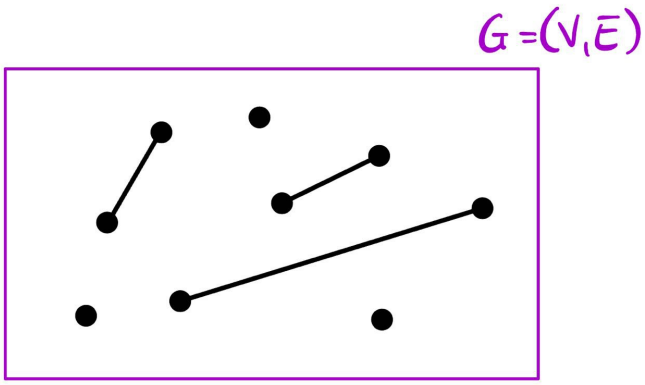
Parametrized Algorithms: Exact Exponential Algorithms

We can also solve the problem in exactly $O(f(k) \cdot poly(n))$ time, where $f(k)$ can be large.
This is useful when $k << n$, even though k can be close to n in some instances.

observation: this do not tell us nothing about P vs NP

Kernelization

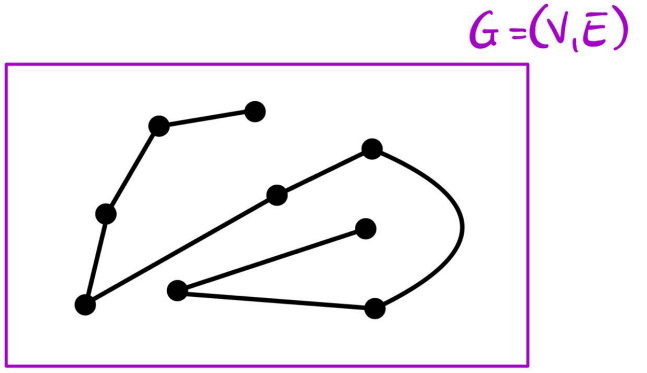
Consider the undirected graph $G = (V, E)$.
Let's consider easy cases:
Case 1:



The maximum degree of G, in this case, is 1, as every node has at most one edge.
In symbols:

$$max\_degree(G) = 1$$

Case 2:



The maximum degree of G, in this case, is 2, as every node has at most one edge.
In symbols:

$$max\_degree(G) = 2$$

Then, for both simple paths and simple cycles, we check if their number of edges is odd or even.

Consider VC_k,, an instance $\langle G, k \rangle$ to indicate that we want to check whether a vertex cover of size $\leq k$ exists in G.
We then apply the following two rules as long as possible:

- **R1)** if v is an isolated node $deg(v) = 0$ then solve $\langle G - \{v\}, k \rangle$
- **R2)** if v has degree $> k$ in G then solve $\langle G - v, k - 1 \rangle$

The reduced graph with R1) and R2) is said the kernel of G, that's why it is called kernelization.

Each rule takes $poly(n)$ time.
When it is not possible to apply R1 and R2 anymore, we apply the rule **R3)**

If: $(k < 0) \vee (G \text{ has} > k + k^2 \text{ nodes}) \vee (G > k^2 \text{ edges})$ ♣
Then: report no vertex cover for the input.
Else: G_k be the current graph (reduced with R1 and R2), solve vertex cover on $\langle G_k, k \rangle$ and return the answer.
♣: if $\langle G_k, k \rangle$ has a VC of size $\leq k$ then it must have $\leq k^2 + k$ nodes and $\leq k^2$ edges.

Let S be such a vertex cover, $|S| \leq k$.

1. $|S| \leq k, \forall u \in S : deg(u) \leq k, S \text{ is a VC} \implies |E| \leq |S| \cdot k = k^2$
2. $|S| + |G_k - S| \leq k + k^2$

And 1) + 2) $\implies poly(k)$

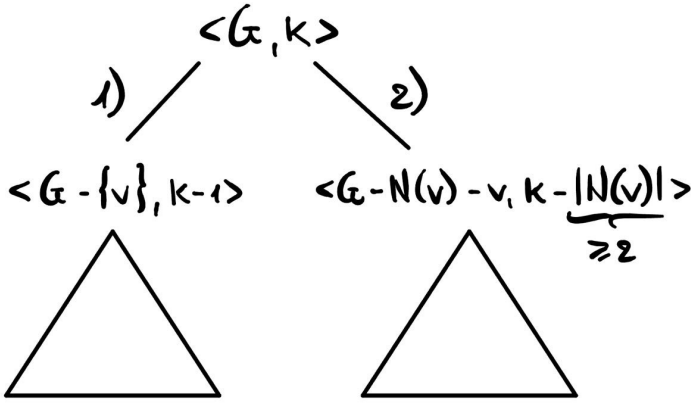
To answer $\langle G_k, k \rangle$ we run the baseline: $f(k) = (k^2)^k$.
$\implies$ total cost: $O(f(n) + poly(n))$

Branching Technique

Let's consider $\langle G, k \rangle$ as before.
Let's define v as the node in G with maximum degree. We have recursion on this degree.

base case: $deg(v) = 1$
recursive case: $deg(v) \geq 2$

1. either v belongs to the vertex cover (and we cannot claim anything for $N(v)$), or
2. all of $N(v)$ should belong to the VC



The total cost of is given by:

$$\#\text{recursive calls} \cdot poly(n)$$

Each call generates two calls $\implies$ $\#\text{calls} \leq 2 \cdot \#\text{leaves}$, the leaves are the base case: $L(k)$

$$L(k) = L(k-1)(1) + L(k - |N(v)|)(2) \leq L(k-1) + L(k-2)$$

And we notice that $L(k)$ **is Fibonacci:** $L(k) \sim \Phi^k$, $\Phi = 1.6\ldots$
The total cost is: $O(\Phi^k \cdot poly(k))$

Branching + Kernelization

We can put things together:

$$O(VC(k^2) + poly(n)) = O((k^2)^k + poly(k)) \to O(\Phi^{k^2} + poly(n)) = O(1.6^{k^2} + poly(n))$$

And, even though 1.6^{k^2} can be exponential in k, in most cases k is small.

Glimpse on Approximation

Given a graph $G = (V, E)$, recall the vertex cover problem.
By definition $S \subseteq V$ is a vertex cover if $\forall uv \in E. u \in S \vee v \in S$.
We then have VC_k, the decisional problem:

$$\exists ?\ S \subseteq V. |S| \leq k, S = VC$$

That, as we have seen, we can solve in $O(1.6^{k^2} + poly(n))$.

We then define VC^*, the problem of finding the minimum vertex cover of G

$$VC^* = k_{min} \text{ such that } \forall S \subseteq V. |S| < k \implies S \text{ is not a } VC$$

We can do binary search to find k as it is too expensive: k is at the exponent.
Better go trivially: test VC_k for $k = 1, 2, \dots$ and we will surely stop at $k = k_{min}$.

We can do the following **2-approx** by relaxing the optimality.
Instead of looking for $VC\ S^* \in V \text{ such that } |S^*| = k_{min}$ we are satisfied if we find a vertex cover $S_\sim \subseteq V \text{ such that } |S_\sim| \leq 2 \cdot k_{min}$
The procedure for finding such $VCS_\sim$ is:

```
S~ = {}
for each uv in E:
        if (u not in S~) and (v not in S~):
                S~ = S~ U {u,v}
return S~
```

where we see that:

1. $S_\sim$ is a vertex cover by design
2. $|S_\sim| \leq 2 \cdot k_{min} = 2 \cdot |S^*|$
    1. consider S^*, by definition $\forall uv \in E : u \in S^* \vee v \in S^*$
    2. consider $S_\sim$, for every two edges we have
        1. no endpoint in common
        2. each of such edge has at least a node in S^*
    3. then the number of edges is $\frac{|S_\sim|}{2}$ ($S^\sim$ contains vertexes!)
    4. then: $\frac{|S_\sim|}{2} \leq |S^*| \implies |S_\sim| \leq 2 \cdot |S^*|$

Random FPT Algorithms

The fixed-parameter approach is an algorithm design technique for solving combinatorially hard (mostly NP-hard) problems.

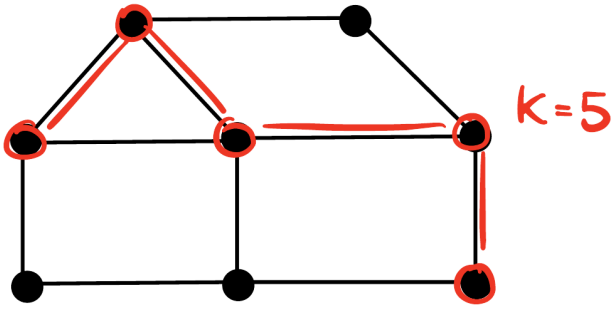
For some of these problems, it can lead to algorithms that are both efficient and yet at the same time guaranteed to find optimal solutions.

FPT: Fixed-Parameter Tractable algorithms, $O(f(k) \cdot n^c)$ time.
We have two techniques:

- **color coding**
- **randomized separation**

Color Coding

Consider an undirected graph $G = (V, E)$.
We define a k-path a path of k nodes which are all distinct.



The problem of finding the k-path is NP-complete, as finding the solution with $k = |V| = n$ is HAM.

Idea: Random Coloring
Consider a k-coloring function $X : V \to [k]$ that associate to every vertex a color.
Random k-coloring because we require:

$$\forall v \in V. \forall c \in [k].\ P(X(v) = c) = \frac{1}{k}$$

observation: it is clear that X is an hash function.
Suppose that X is chosen uniformly at random from a universal family $\mathcal{H}$.
Then we have:

$$V' \subseteq V \text{ is said colorful } \iff \forall u, v \in V', u \neq v. X(u) \neq X(v)$$

We then have:

- **proposition a)** a colorful walk of k nodes is a k-path (the opposite is not always true)
- **proposition b)** a walk of k nodes is a k-path if an only if exists a coloring X such that the walk is colorful

Consider $V' \in V$ such that $|V'| = k$

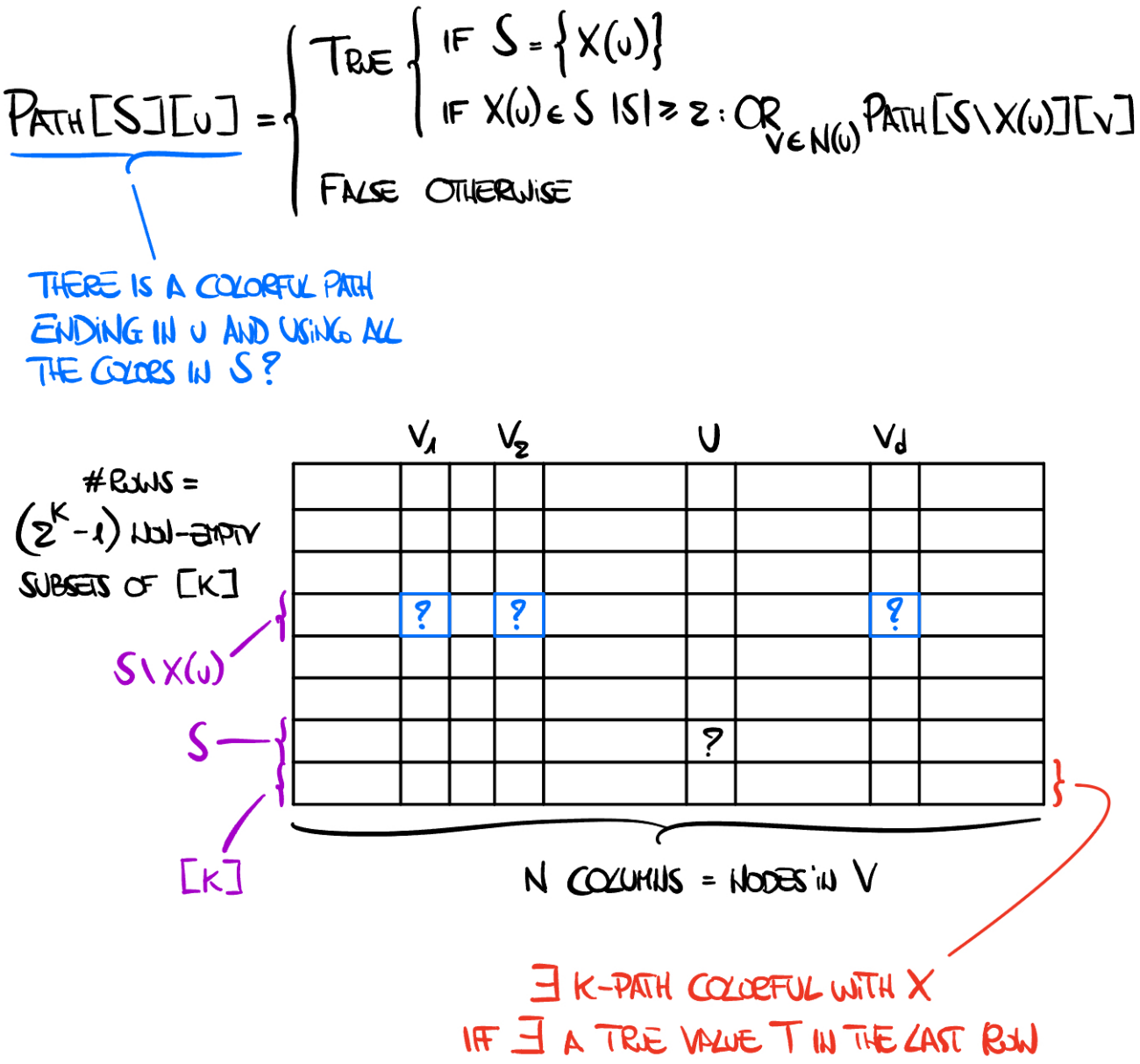
$$
\begin{aligned}
P(V' \text{ is colorful}) &= \\
&= \frac{\#\text{colorings } X \text{ that makes } V' \text{ colorful}}{\#\text{all possible colorings of } G} \\
&= \frac{k^{n-k} \cdot k!}{k^n} \ \spadesuit \\
&= \frac{k!}{k^k} \\
&\geq e^{-k} \ \clubsuit
\end{aligned}
$$

where:

- ♠ is from the following:
  - k^{n-k} is V/V'
  - $k!$ is V': choose $n, n-1, \ldots$ nodes
- ♣ $k! \geq (\frac{k}{e})^k$

Question: Given a random coloring X, how do we find colorful paths (k-paths)?
We use **dynamic programming:**

- a) any sub-path of a colorful path is a colorful path
- b) a colorful path of length j ends in a node u if and only if exists a node $v \in N(u)$ such that a colorful path of length $j-1$ ends in v and none of the nodes in that path have a color $X(u)$
- c) the set of $j-1$ colors matters, not their order or paths



The matrix `Path` has $2^k - 1$ rows and $n = |V|$ columns.
Each entry `Path[S][u]` takes $O(|N(u)|)$ time to fill.
The total cost is

$$\begin{aligned}\Sigma_u |N(u)| = \\ &= O(m) \text{ per row}, m = |E| \\ &= O(2^k \cdot m) \\ &= O(2^k \cdot n^2)\end{aligned}$$

Final Touch:
We repeat the above for e^k times, each time randomly choosing a coloring X.
We stop when we find T in the last row of the table.
We recall the union bound $\implies$ constant probability of success, then the running time is $O(e^k 2^k n^2) = O((2e)^k n^2)$, **FPT.**
What we have seen is a Monte Carlo approach.

Why Color Coding is Powerful?
Take a node u and consider all k-paths ending in u
Potentially, we may have up to n^k such paths, and a significant fraction of them are colorful for X.

For each colorful k-path, the set of colors is the same, namely $[k]$.
With **just one set of colors we represent them all.**
In general: for a j-path, where $j \in [1,k]$, we have $\binom{k}{j}$ set of colors to remember, instead of $O(n^j)$ j -paths.

Randomized Separation

Idea: random $2-$coloring X.
You split the graph into groups of the same color.

We consider the problem of finding a Subgraph Isomorphism, k-path is a special case of this problem:

- **IN:** $G=(V,E), H=(V_H,E_H)$
- **OUT:** $\hat{H}=(\hat{V},\hat{E})$ subgraph of G, with $\hat{V}\subseteq V, \hat{E}\subseteq E$, isomorphic to H
  - $\hat{H}$ is isomorphic to H if and only if $\exists$ a 1-to-1 mapping $\phi : V_H \to \hat{V}$ such that $(u,v)\in E_H \iff (\phi(u),\phi(v))\in \hat{E}$
  - checking is exponential in $|\hat{V}|=|V_H|=k$, $O(k!\cdot k^2)$

FPT Algorithm: which parameter?

- $k=|V_H| \implies \text{no FPT unless } P=NP$
- $k=|V_H|, \Delta = max\_degree(G) \implies FPT \text{ algorithm } O(2^{\Delta k}\cdot k! n^{O(1)})$ time
  - Δ is a bounded degree

Suppose that $\hat{H}$ isomorphic to H exists in G.
Let's define $\Gamma = \{(u,v)\in E/\hat{E} : u\in\hat{V}\vee v\in\hat{V}\}$
If we cut Γ, we get $\hat{H}$ as a connected connected, aka CC.

proposition: $|\Gamma|+|\hat{H}|\leq|\hat{V}|\cdot\Delta = k\cdot\Delta$
(mind that $\Gamma,\hat{H}$ are disjoint by definition)

Here the FPT algorithm:

1. check if $\Delta = max\_degree$
2. repeat $2^{\Delta\cdot k}$ times
   1. $X=$ random 2-coloring $X: V\to[2]$
   2. $G_0=$ part of G colored 0
   3. $G_1=$ part of G colored 1
      1. we have the induced subgraphs
   4. find the CCs in G_0 and in G_1, and check if any two CC is isomorphic to $\hat{H}$

With the previous algorithm we have

$$P(X \text{ gives } \hat{H}) = \frac{2}{2^{|\Gamma|+|\hat{E}|}} \geq \frac{1}{2^{\Delta\cdot k}}$$

R-Approximation

Hardness: P, NP, NP-complete

- P, problems that can be **solved** in $poly(n)$ time
- NP, problems that admit a $poly(n)$ time **certificate**
- NP-complete: every problem in NPC can be **reduced** to any other in the same class in $poly(n)$. Solve one of them and you have solved them all.

$$NP \subseteq_? P$$

To attack such hard problems:

- **exact solutions:** parametrized algorithms, FPT
- **approximated solutions:** optimization problems
  - min-cost
  - max-benefit

We see now an approximation technique, called **r-approximation**, where the algorithm provides a solution $S_\sim$ such that:

$$min = \frac{cost(S_\sim)}{OPT} \leq r$$
$$max = \frac{OPT}{cost(S_\sim)} \leq r$$

with $r > 1$.
We have already had a taste of this approximation technique with Vertex Cover: VC is a 2-approx, $cost(S_\sim) \leq 2 \cdot OPT$

Max-Cut of an Undirected Graph

Consider an undirected graph $G = (V, E)$
Remember the:

- definition of **cut**: $C = (V_1, V_2)$ where $V_1 \cap V_2 = \emptyset \wedge V_1 \cup V_2 = V$
- definition of **cut-set:** $E(V_1, V_2) = \{uv \in E : u \in V_1 \wedge v \in V_2\}$

We have already seen the min-cut. Consider that:

- computing the min-cut: min $E(V_1, V_2)$ is in the class P
- computing the max-cut: max $E(V_1, V_2)$ is in the class NPC

Three ways to get a $2-$approx for max-cut: $cost(S_\sim) = |E(V_1, V_2)| \geq \frac{1}{2} \cdot OPT$

1. **local search**
2. **greedy**
3. **randomization**

Local Search

We define a set of nodes S, and we return the cut-set $E(S, V-S)$.

The local search algorithm do the following operations:

- $S = \emptyset$, the starting cut-set is $E(\emptyset, V)$
- while $\exists u \in V$ such that if we were to move u from the current side of the cut-set to the other side we would obtain a bigger cut-set ♣
  - if $u \in S \implies S = S/\{u\}$
  - else $S = S \cup \{u\}$
- return $E(S, V-S)$

♣ Moving a vertex u from one side to the other side increases the size of the cut-set only if there are edges from/to u that now crosses the cut!

The algorithm terminates:

- $|E(S, V-S|$ increase after every iteration
- $|E(S, V-S)| \leq |E|$

And we see that the algorithm takes at most $|E|$ steps.

The algorithm provides a 2-approximation:
We do not know OPT, so we establish an upper bound $UB = |E| \geq OPT$
and we prove that $cost(S_{\sim}) \geq \frac{1}{2} \cdot UB$.
To see this we notice that after the algorithm ends, each node u has at most $\frac{d_u}{2}$ incident edges in $E(S, V-S)$ ♠, where $d_u = deg(u)$.

We prove that $|E(S, V-S)| \geq \frac{1}{2}|E|$

$$
\begin{aligned}
|E(S, V-S)| = & \\
&= \frac{\Sigma_{u \in V} \, \#\text{edges incident to u in } E(S, V-S)}{2} \\
&\geq \Sigma_{u \in V} \, \frac{d_u}{2} \text{ ♠} \\
&= \frac{\frac{2|E|}{2}}{2} \\
&= \frac{|E|}{2}
\end{aligned}
$$

♠ at least $\frac{d_u}{2}$ of the edges incident to u are in the cut-set, as otherwise changing the side of u in the cut-set would increase the size of the cut-set, but this is not possible as the algorithm terminates when that is not possible.

Greedy

We number the nodes from 1 to n (any order is fine).
We define the set S and as before we will return the cut-set $E(S, V-S)$

The greedy algorithm do the following operations:

- $S = \emptyset$, the starting cut-set is $E(\emptyset, V)$

- $\forall u = 1, 2, \dots, n$
  - put u in S and take $|E(S, V-S)|$
  - put u in $V-S$ and take $|E(S, V-S)|$
  - store the largest
- return the largest

We see that the greedy algorithm:

1. takes $n = |V|$ steps, trivial
2. is a $2-$approximation: $|E(S, V-S)| \geq \frac{|E|}{2}$ ♣

Let's prove ♣
Consider the edge $i \to j$, and without loss of generality we assume $i < j$.
The "fate" of the edge ij is decided by the color of j rather than the color of i.
The node j is "responsible" for edge $ij \iff i < j$.

We define $r_u = |\{v \in V : uv \in E \wedge u > v\}|$
In words: r_u is the set of of nodes distant 1 from u but with assigned index $> u$.
Alternatively said: r_u is the set of neighbors of u reached by an edge for which u is responsible.
claim: the number of edges for which u is responsible and belong to the final $E(S, V-S)$ is $\geq \frac{r_u}{2}$

We then can write:

$$\begin{aligned} |E(S, V-S)| = \\ &= \Sigma_{u \in V} \ \#\text{edges for which } u \text{ is responsible and belong to the cut-set} \\ &\geq \Sigma_{u \in V} \frac{r_u}{2} \ \spadesuit \\ &= \frac{|E|}{2} \end{aligned}$$

♠ We do not divide by 2 as each edge has exactly one responsible node:

$$\Sigma_u \, r_u = |E|$$

Randomization

We assign a random color using $X : V \to \{0, 1\}$ uniformly and independently.
We know that $uv \in E$ is in $E(S, V-S) \iff X(u) \neq X(v)$.

Let's define $S = \{u \in V : X(u) = 0\}$, the cut-set will be, as before, $E(S, V-S)$

Thanks to X, we know that $P(uv \in E(S, V-S) = \frac{1}{2}$
Then we define the indicator variable:

$$X_{uv} = \begin{cases} 1 \text{ if } uv \in E(S, V-S) \\ 0 \text{ otherwise} \end{cases}$$

It is then clear that $|E(S, V-S)| = \Sigma_{uv \in E} \, X_{uv}$.
And we can now compute the expected value:

$$E[|E(S, V-S)|] = \Sigma_{uv \in E} \, E[X_{uv}] = \Sigma_{uv} \, \frac{1}{2} = \frac{|E|}{2}$$

NP-Hard Problems: TSP

The **Travel Salesman Problem (TSP)**

- n cities
- $D_{ij} =$ cost (aka distance) of going from the city i to the city j
- we define a **tour**, a permutation of the all the cities

It is clear that

$$\text{cost of the tour} = (\Sigma_{i=1}^{n-1} D_{c_i,c_{i+1}}) + D_{n,1}$$

We want to minimize the cost.

TSP is NPC
There is no r-approximation, $r > 1$, for TSP unless $P = NP$.
Alternatively said: **approximating TSP is as hard as solving it optimally**

Consider a graph $G = (V, E)$ and suppose A_r is a poly-time algorithm that provides an r-approx for TSP.
We then build an algorithm A_H to solve HAM $\in NPC$, in symbols:

$$A_H(G) = \begin{cases} 1 \text{ if } G \text{ is Hamiltonian} \\ 0 \text{ otherwise} \end{cases}$$

And A_H is defined as follows:
step 1)
We set the distances of G as follows

$$D_{ij} = \begin{cases} 1 \text{ if } (i,j) \in E \\ 1 + r \cdot |V| \text{ otherwise} \end{cases}$$

step 2)
Execute A_r, and let C be the (approximated) cost returned.
step 3)
If $C \leq r \cdot |V|$ then the answer is YES, else NO.

A_r poly-time $\implies$ A_H poly-time.

- **G is Hamiltonian:** $\exists$ a tour that has all cost 1 as the Hamiltonian cycle uses edges in $E \implies C = \text{cost(tour)} = |V| \implies C \leq r \cdot \text{cost(tour)} = r \cdot |V| = r \cdot C$
- **G is not Hamiltonian:** $\forall$ tour there should be at least one distance $D_{ij} = 1 + r \cdot |V|$, as there is at least one $ij \notin E$ that is "traversed" by the tour $\implies$ any tour has cost $C \geq (|V| - 1) \cdot 1 + (1 + r \cdot |V|) > r \cdot |V|$

Metric TSP

Remember the triangular inequality (TI): $D_{ij} + D_{jk} \geq D_{ik}$
Lets give the **2-approx**.
Let $G = (V, E, D)$ be the **complete** weighted graph whose edge weights are in D

1. compute $S = MSG(G)$, with $MSG =$ **minimum spanning tree**
2. compute a tour T as follows:
    1. perform a traversal of S: each time we see a node in S for the first time, we add it to T

Where:

- **complete:** a complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge
- **minimum spanning tree:** A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices, without any cycles and with the minimum possible total edge weight.

We then have:

1. the cost of the optimal solution is greater than the cost of S, as a tour without an edge is a spanning tree
2. the computed tour T is the approximated solution: cost(T) $\leq 2 \cdot$ cost(S), this is given by the triangle inequality ♣ and by the fact that each edge weight is summed twice in $\leq 2 \cdot$ cost(S) ♠
    - ♣
    - ♠

$$1) \text{ and } 2) \implies \text{cost}(T) \leq 2 \cdot \text{cost}(S) \leq 2 \cdot OPT$$

NP-Hard Problems: Knapsack

The Knapsack problem:

- n elements
- W is the knapsack capacity
- w_i occupancy of the i-th element in the input, $i \in [n]$
- v_i is the value of the i-th element in the input, $i \in [n]$
- $S \subseteq [n]$ is feasible if $\Sigma_{i \in S}\, w_i \leq W$, (note that $\forall i.\, w_i \leq W$)

The Greedy Approach

We define $\frac{v_i}{w_i}$ as the value per unit.
We do the following hypothesis, without loss of generality:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

which intuitively means: take the i-th, with i as small as possible, as soon as possible.

Let's give a 2-approximation

- $S = \emptyset$
- $W' = W$, W' represent the residual capacity of the knapsack
- $\forall i = 1, 2, \ldots, n$

    - if $w_i \leq W'$ then $S = S \cup \{i\}, W' = W' - w_i$
- return S

The previous algorithm is a good heuristics, but it has a glitch.

$$\begin{cases} v_1 = v_2 = \cdots = v_{n-1} = 1 \wedge v_n = W - 1 \\ w_1 = w_2 = \cdots = w_{n-1} = 1 \wedge w_n = W \end{cases} \implies \forall i < n : \frac{v_i}{w_i} = 1 \wedge \frac{v_n}{w_n} < 1$$

And we then have:

- $S = \{1, \ldots, n-1\} \implies \text{cost}(S) = \Sigma_{i \in S}\, v_i = n - 1$
- the optimal solution is $S^* = \{n\}$
    - $OPT = \text{cost}(S^*) = W - 1$

And the **approximation ratio is**

$$\frac{OPT}{\text{cost}(S)} = \frac{W-1}{n-1} \sim k$$

with k arbitrarily large!

Let's fix the glitch:
Let m_G be the value returned by the greedy algorithm.
Let $v_{max} = max_{i \in [n]}\, v_i$
And, as a solution, we return $\max(m_G, v_{max})$.

The fix gives us a 2-approximation:
Let's define an upper bound $UB \geq OPT$.
Let's define j as the smallest element in $[n]$ that does not fit in W' ($j \geq 2$).
In symbols:

$$\Sigma_{i=1}^{j-1}\, w_i \leq W \text{ but } \Sigma_{i=1}^{j}\, w_i > W$$

We then use the following notation:

- $\bar{W}_j = \Sigma_{i=1}^{j-1}\, w_i$
- $\bar{V}_j = \Sigma_{i=1}^{j-1}\, v_i$

And we say that:

$$UB = \Sigma_{i=1}^{j}\, v_i = \bar{V}_j + v_j \geq OPT \spadesuit$$

$\spadesuit$ by default as UB is not a solution, the object j by def can't be placed in the knapsack.

Since $OPT < \bar{V}_j + v_j$ we have two cases:

1. $\bar{V}_j > v_j \implies OPT < 2\bar{V}_j \leq 2m_G$ as S is the greedy solution and satisfies $\{1, \ldots, j-1\} \in S$
2. $\bar{V}_j \leq v_j \implies OPT < 2v_j \leq 2v_{max}$ as $v_j \leq v_{max}$ by definition

Summing up the two previous point:

$$OPT \leq 2(max(m_G, v_{max})) = 2 \cdot \text{cost}(S) \implies \frac{OPT}{\text{cost}(S)} \leq 2, \square$$

FPTAS: Fully-Polytime Approximate Solution

A Fully Polynomial Time Approximation Scheme (FPTAS) is an algorithm for finding approximate solutions to function problems, especially optimization problems.
It takes as input an instance of the problem and a parameter ε > 0, and returns a value that is at least a certain factor times the correct value, and at most another factor times the correct value.

The key characteristic of an FPTAS is that its run-time is polynomial in the problem size and in 1/ε. This is in contrast to a general Polynomial Time Approximation Scheme (PTAS), whose run-time is polynomial in the problem size for each specific ε, but might be exponential in 1/ε.

An FPTAS is considered to be a subset of PTAS, and unless P = NP, it is a strict subset.
The difference between the two lies in their time complexity with respect to 1/ε, where an FPTAS cannot have a time complexity that grows exponentially in 1/ε, while a PTAS can.
An FPTAS is considered the strongest possible result that can be derived for an NP-hard problem

Knapsack FPTAS: DP2

Solve an instance of the knapsack problem in $O(n^2 \cdot v_{max})$ time.
Idea: scale the values v_i to $v_i^{\sim} = \lfloor \frac{v_i}{k} \rfloor$ for some suitable factor k to be fixed asap.
We then have two "instances" of the problem:

- **a)** original: v_i, w_i, W
- **b)** scaled: $v_i^{\sim}, w_i, W$

And it is true that S is feasible in **a)** $\iff$ S is feasible in **b)**.

Let's fix k so that $v_i^{\sim}$ are $poly(n)$ as we will pay $O(n^2 \cdot v_{max})$.
A good choice is

$$v_i^{\sim} = \lfloor \frac{v_i}{k} \rfloor \leq \lfloor \frac{n}{\varepsilon} \rfloor \implies \frac{v_i}{k} \leq \frac{n}{\varepsilon} \implies k = \frac{v_{max} \cdot \varepsilon}{n}$$

Then we run DP2 on **b)** in time $O(n^2 \cdot v_{max}^{\sim}) = O(\frac{n^3}{\varepsilon})$.
observation: we find the exact optimal solution $S^{\sim}$ for **b)**

Question: how good is $S^{\sim}$ in **a)**? Clearly it is feasible.
What about:

$$\Sigma_{i \in S^{\sim}} v_i \geq (1 - \varepsilon) \cdot \Sigma_{i \in S^*} v_i$$

Clearly

$$\Sigma_{i \in S^{\sim}} v_i \geq \Sigma_{i \in S^*} v_i$$

And recall that $v_i^{\sim} = \lfloor \frac{v_i}{k} \rfloor \implies \frac{v_i}{k} - 1 \leq v_i^{\sim} \leq \frac{v_i}{k}$

We then have this three facts:

1. $v_i \geq \tilde{v_i}$
2. $k\tilde{v_i} \geq v_i - k$
3. $\Sigma_{i \in S^\sim} \tilde{v}_i \geq \Sigma_{i \in S^*} \tilde{v}_i$

Then:

$$
\begin{aligned}
\text{cost}_{\text{a)}}(S^\sim) = & \\
& = \Sigma_{i \in S^\sim} v_i \\
& \geq_{1)} \Sigma_{i \in S^\sim} k\tilde{v_i} \\
& \geq_{3)} \Sigma_{i \in S^*} k\tilde{v_i} \\
& \geq_{2)} \Sigma_{i \in S^*} (v_i - k) \\
& = \Sigma_{i \in S^*} v_i - \Sigma_{i \in S^*} k \\
& \geq OPT - nk, \text{ with } nk = \frac{\varepsilon \cdot v_{max}}{n} \cdot n = \varepsilon \cdot v_{max} \leq \varepsilon \cdot OPT \text{ since } v_{max} \leq OPT \\
& \geq OPT - \varepsilon \cdot OPT \\
& = (1 - \varepsilon) \cdot OPT
\end{aligned}
$$

□