• Formulas
  - \((1 - \frac{1}{e})^n\) increases from 1/4 to 1/e as \(n\) increases from 2 up to \(\infty\), so \(1/4 \leq (1 - \frac{1}{e})^n < 1/e\)
  - \((1 - \frac{1}{e})^{n-1}\) decreases from 1/2 to 1/e as \(n\) increases from 2 up to \(\infty\), so \(1/e < (1 - \frac{1}{e})^{n-1} \leq 1/2\)
  - Define independent events
  - Define conditional probability

• Global Minimum Cut
  - Recall definition of conditional probability
  - Recall Global Minimum Cut Problem
  - Finish description of algorithm
  - Do analysis of probability of success
    * Let \(F_* = [A*, B*]\) be a globally minimum cut and let \(k\) be the number of edges in \(F_*\)
    * Let \(S_j\) be the event that the \(j^{th}\) contraction does not select an edge of \(F_*\)
    * We want \(P[S_1 \cap S_2 \cap \ldots \cap S_{n-2}]\)

\[
P[S_1 \cap \ldots \cap S_{n-2}] = P[S_{n-2}|S_1 \cap \ldots S_{n-3}]P[S_1 \cap \ldots S_{n-3}] \quad (1)
\]

\[
= P[S_{n-2}|S_1 \cap \ldots S_{n-3}]P[S_{n-3}|S_1 \cap \ldots S_{n-4}]P[S_1 \cap \ldots S_{n-4}] \quad (2)
\]

\[
= \ldots \quad (3)
\]

\[
= P[S_{n-2}|S_1 \cap \ldots S_{n-3}] \ldots P[S_2|S_1]P[S_1] \quad (4)
\]

* Now observe that, after the \(j^{th}\) contraction, there are at least \(k(n - j)/2\) edges, so the probability that an edge of \(F_*\) will be selected next (if no edge of \(F_*\) has yet been selected) is at most \(2/(n - j)\).
* So the probability that an edge of \(F_*\) is not selected on round \(j + 1\) is at least \(1 - 2/(n - j)\)
* Note that \(1 - 2/(n - j) = \frac{n-j-2}{n-j}\).
* This yields, for the probability that no edge of \(F_*\) is contracted, a value

\[
\geq \frac{n-2}{n} \frac{n-3}{n-1} \ldots \frac{n-j-2}{n-j} \ldots \frac{2}{4} \frac{1}{3} \quad (5)
\]

\[
= \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \quad (6)
\]

  - So, the algorithm fails with probability at most \(1 - 1/\binom{n}{2}\)
  - So, the probability that the algorithm fails if we run it \(\binom{n}{2}\) times is \((1 - 1/\binom{n}{2})^{\binom{n}{2}} \leq 1/e\)
  - If we run it \(\binom{n}{2} e \log n\) times, the probability of failure is at most \((1/e)^e \log n = 1/n^e\)
  - Improvements: Can be improved [Karger 2000] to run in time \(O(m \log^3 n)\)
  - This is better than any known deterministic algorithm!

• Note that this is a Monte Carlo algorithm
  - Guaranteed to run in polynomial time
  - Will never give a “false positive”
  - Will only give a false negative with low probability (a fixed probability independent of problem size)

• Recall Expected Value Definition
  - Do binary search example
Example: Memoryless Guessing

- Suppose the set \( S = \{1, \ldots, n\} \) is stored in an array \( A[] \) in some order.
- We want to create an array \( G[1..n] \), where \( G[i] \) is a "guess" as to the value of \( A[i] \).
- Assume that each \( G[i] \) is assigned a value of \( S \) uniformly at random.
- Note: \( G[] \) can contain repeated values
- What is the expected number of correct guesses? [That is, expected number of \( i \) for which \( G[i] = A[i] \)?
- Question: What is the probability space for this problem? [Answer: \( U \) is the set of all possible arrays of guesses, all equally likely.]

Assume now that before we assign a value to \( G[i] \) is a "guess" as to the value of \( A[i] \).

Let \( X_i \) be the random variable defined by \( X_i = 1 \) if \( G[i] = A[i] \); \( X_i = 0 \) otherwise

Now let \( X = \sum_{i=1}^{n} X_i \). We're interested in \( E[X] \), since \( X \) counts the number of correct guesses for any given \( G[] \)

Note that \( E[X_i] = 0Pr[X_i = 0] + 1Pr[X_i = 1] = Pr[X_i = 1] = 1/n \)

By linearity of expectation: \( E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/n = 1 \) [Regardless of the value of \( n \) !!!]

So, you might as well just make the same guess for every element of \( A[] \) and you’ll do exactly this well!

Example: Guessing with Memory

- Suppose the set \( S = \{1, \ldots, n\} \) is stored in an array \( A[] \) in some order.
- We want to create an array \( G[1..n] \), where \( G[i] \) is a "guess" as to the value of \( A[i] \).
- Assume now that before we assign a value to \( G[i + 1] \) we can look at \( A[1..i] \)

Note: \( G[] \) can still contain repeated values, but we’ll never use the values of \( A[1..i] \) in our guess for \( G[i + 1] \)

Let \( X_i \) be the random variable defined by \( X_i = 1 \) if \( G[i] = A[i] \); \( X_i = 0 \) otherwise

Now let \( X = \sum_{i=1}^{n} X_i \). We’re interested in \( E[X] \), since \( X \) counts the number of correct guesses for any given \( G[] \)

Now \( E[X_i] = 1/(n - i + 1) \)

By linearity of expectation: \( E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1/(n - i + 1) = (1/n) \sum_{j=1}^{n} 1/j = H(n) \)

But we’ve seen this before: \( H(n) \in \Theta(\log n) \), in fact \( \log(n+1) \leq H(n) \leq \log n + 1 \), so we can expect to make about \( \log n \) correct guesses.

Max 3-SAT

- Introduce MAX 3SAT
- Calculate probability of single clause being satisfied as \( 7/8 \), and so the expectation that it is satisfied is also \( 7/8 \)
- Calculate expected number of satisfied clauses as (also) \( \frac{7}{8} k \), and so \( 7/8 \) of optimal
Note surprising consequence: There’s always an assignment that achieves at least that ratio!
Now use this to design a randomized algorithm that will always find such an assignment but won’t always be fast
Repeatedly generate random assignments until success.
What’s the expected run time?
Suppose we can show that the probability that at least $\frac{7}{8}$ of the clauses are satisfied at least $p > 0$.
Then expected number of iterations is at most $\frac{1}{p}$, so if $\frac{1}{p}$ is polynomial in $n$ and $k$, we’re done.

Let $p_j$ be the probability that a random assignment satisfies exactly $j$ clauses.
Now the expected number of clauses satisfied is $\frac{7}{8}k = \sum_{j=0}^{k} j p_j$
Let $k'$ be the largest integer less than $\frac{7}{8}k$

\[ \frac{7}{8}k = \sum_{j=0}^{k} j p_j = \sum_{j=0}^{k'} j p_j + \sum_{j=k'+1}^{k} j p_j \]
\[ \leq \sum_{j=0}^{k'} j p_j + \sum_{j=k'+1}^{k} k p_j \]
\[ = k' \sum_{j=0}^{k'} p_j + k \sum_{j=k'+1}^{k} p_j \]
\[ = k'(1 - p) + kp \leq k' + kp \]

So $\frac{7}{8}k \leq k' + kp$, which gives $kp \geq \frac{7}{8}k - k' \geq 1/8$, since $\frac{7}{8}k > k'$.
Thus $p \geq \frac{1}{8k}$

Thus the expected number of repetitions is at most $\frac{1}{p} = 8k$ and the algorithm runs in expected time $O((n + k)k)$, or $O(k^2)$, assuming every variable appears in at least one clause.

This type of algorithm is called a Las Vegas algorithm.

It always yields the correct answer
Has polynomial expected run time.

Formal Definitions of Monte Carlo and Las Vegas

The class RP captures Monte Carlo algorithms for decision problems

- Runs in polynomial time
- If answer is ”no”, returns ”no”
- If answer is ”yes”, returns ”yes” with probability at least $1/2$ ($1/2$ is arbitrary; any fixed probability $p > 0$ will do).

The class ZPP captures Las Vegas algorithms for decision problems

- Always computes the correct solution (if it halts)
- Runs in expected polynomial time
- Markov’s Inequality: For $a > 0$, $P[X \geq a] \leq \frac{E[X]}{a}$

So, let $a = 2E[X]$, then $P[X \geq 2E[X]] \leq \frac{E[X]}{2E[X]} = 1/2$

So, if we run a Las Vegas algorithm for at least $2E[X]$ steps and return ”no” if it hasn’t halted, the probability that it returns the incorrect answer is at most $1/2$. Thus, it becomes a Monte Carlo algorithm.