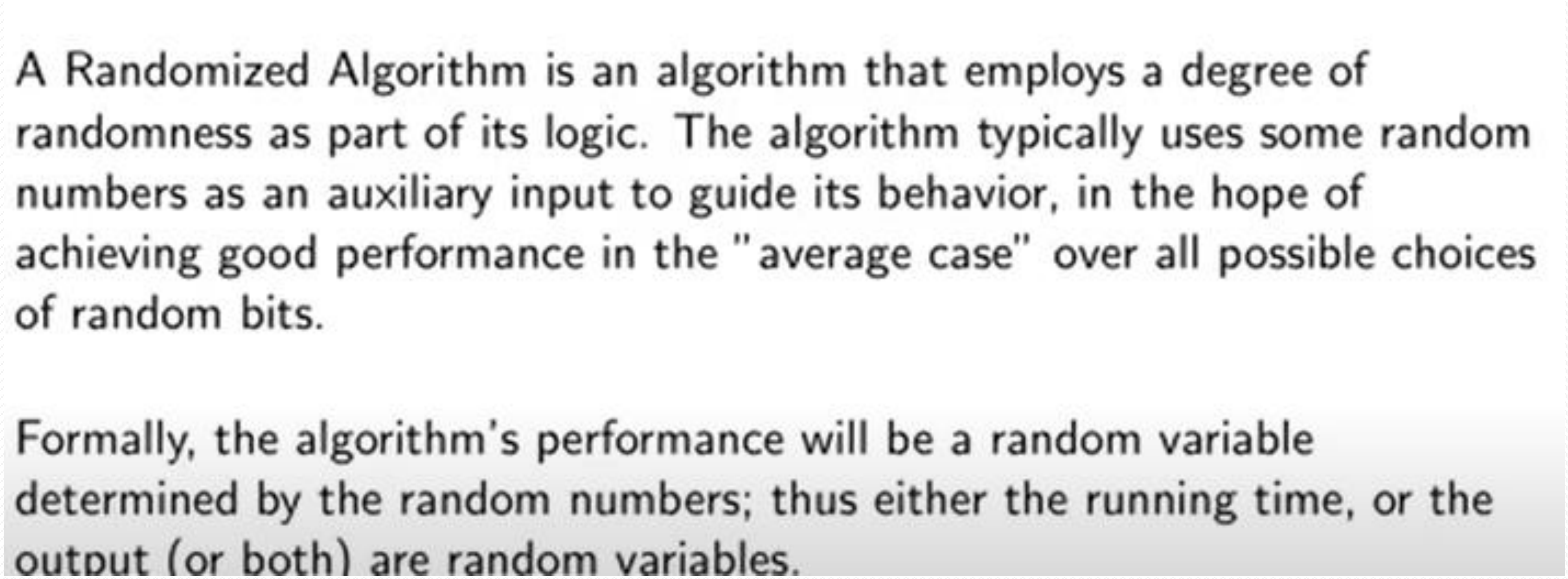
Randomized Quicksort

Ref:
https://www.youtube.com/watch?v=rfAHeilGWoE

Randomized algorithms are algorithms that use random numbers at least once during the computation to make decisions.

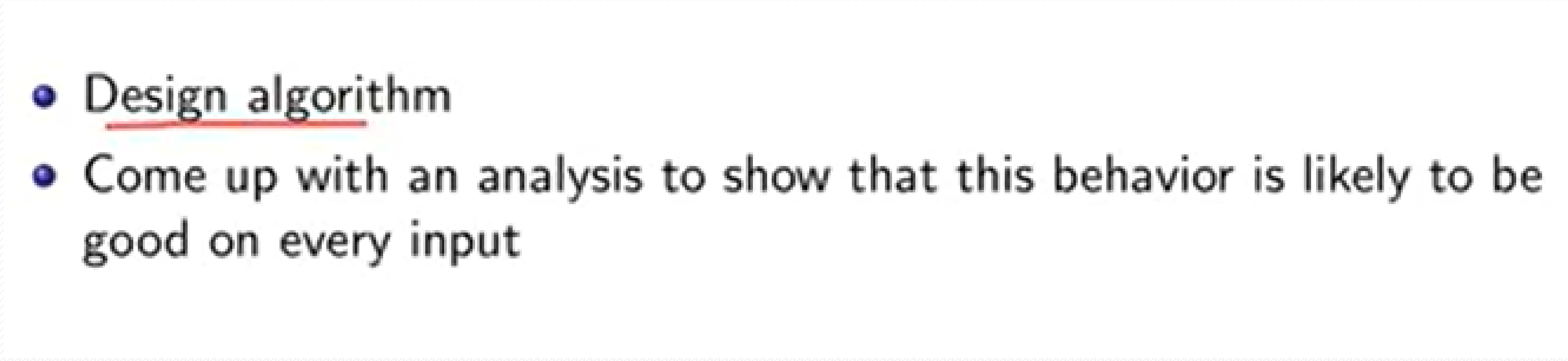
(Algorithms that use random numbers as part of logic to influence decisions during execution)

They have the advantage of simplifying problem-solving, enhancing efficiency for certain inputs, and providing better average-case performance than deterministic algorithms for certain tasks.

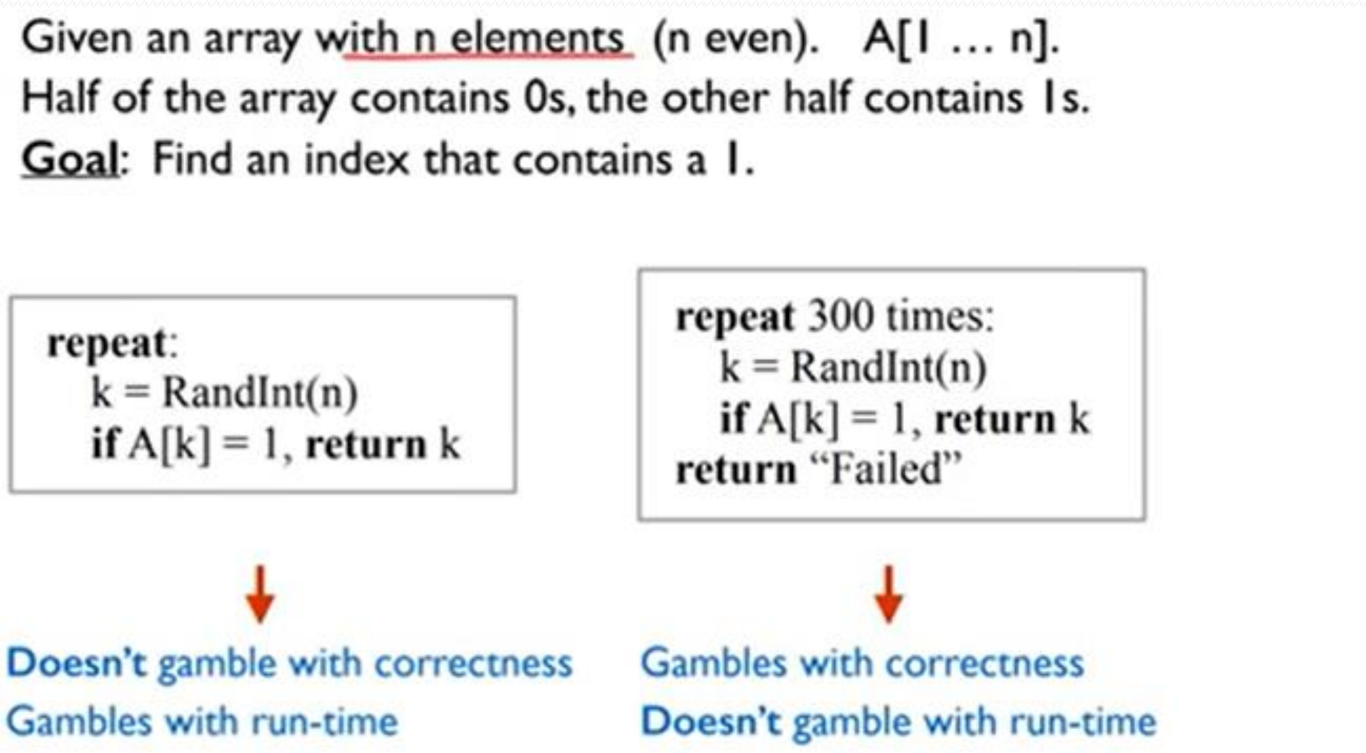


A Randomized Algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses some random numbers as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits.

Formally, the algorithm's performance will be a random variable determined by the random numbers; thus either the running time, or the output (or both) are random variables.



- Design algorithm
- Come up with an analysis to show that this behavior is likely to be good on every input



Given an array with n elements (n even). A[1 … n].
Half of the array contains 0s, the other half contains 1s.
Goal: Find an index that contains a 1.

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

Doesn't gamble with correctness
Gambles with run-time

Gambles with correctness
Doesn't gamble with run-time

always produces a correct result but runtime is more.

Randomized algorithms are typically classified into two main categories:

1. **Las Vegas Algorithms:** These algorithms always produce a correct result, but their running time may vary. The randomness only affects the efficiency.

2. **Monte Carlo Algorithms:** These algorithms run in fixed time but may produce incorrect results with a certain probability. They aim for an acceptable error rate, which can often be minimized by increasing random samples.

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

$\mathbf{Pr}[\text{failure}] = 0$

Worst-case running time: can't bound (could get super unlucky)

Expected running time: $O(1)$ (2 iterations)

This is called a Las Vegas algorithm.
(gambles with time but not correctness)

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

$$\mathbf{Pr}[\text{failure}] = 0$$

Worst-case running time: can't bound A[k] ≠ 1 on every choice of k
(could get super unlucky)

Expected running time: $O(1)$
(2 iterations)

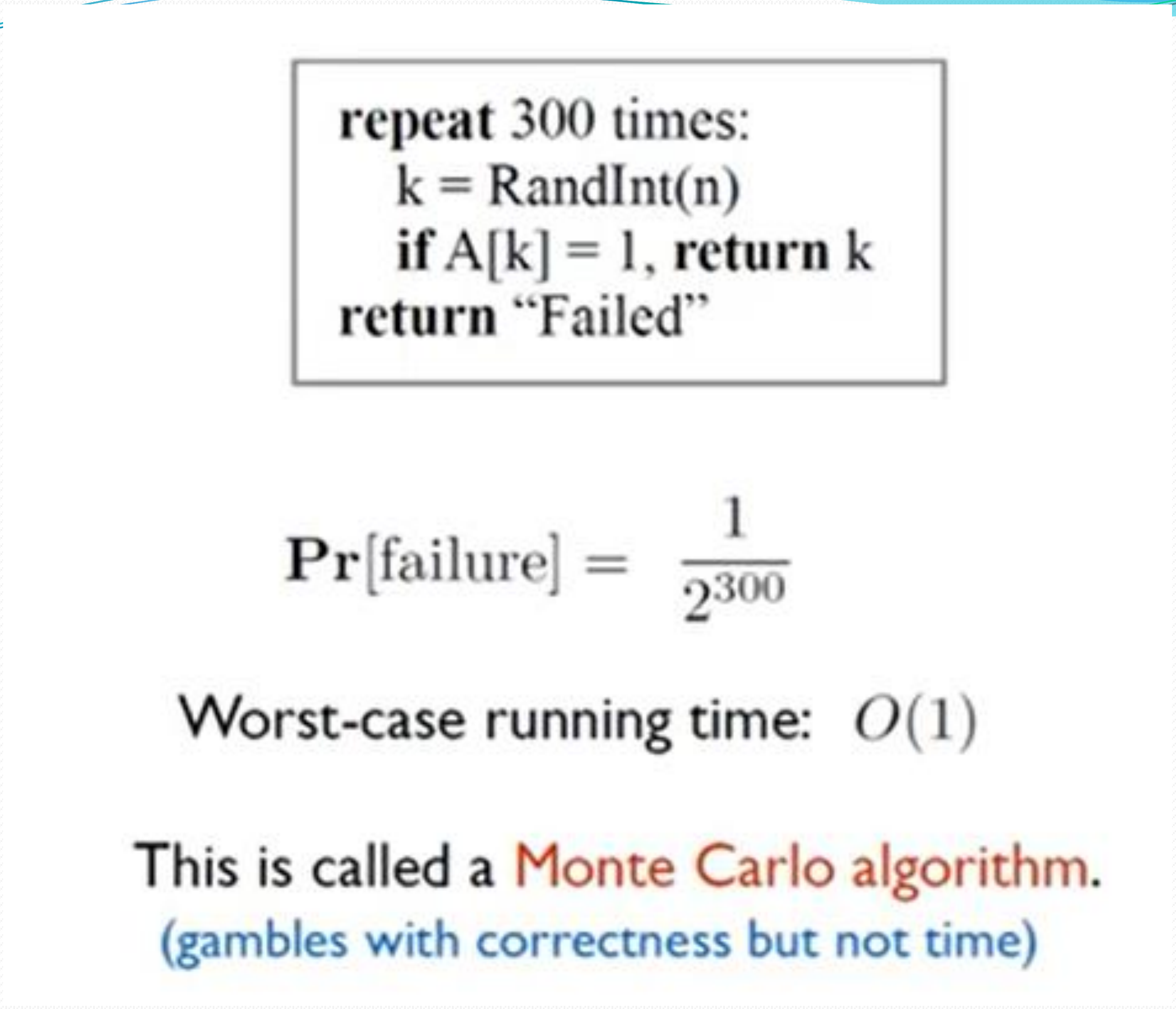
This is called a Las Vegas algorithm.
(gambles with time but not correctness)

Why O(1) Expected Running Time for Two Iterations?

If each attempt is independently likely to succeed with a probability p, then the expected number of attempts until success is 1/p.

If p=1/2 (meaning each attempt has a 50% chance of success), the expected number of iterations is 2 (since 1/(1/2)=2).

We might technically run more than one iteration, the expected running time across many cases remains constant, O(1).

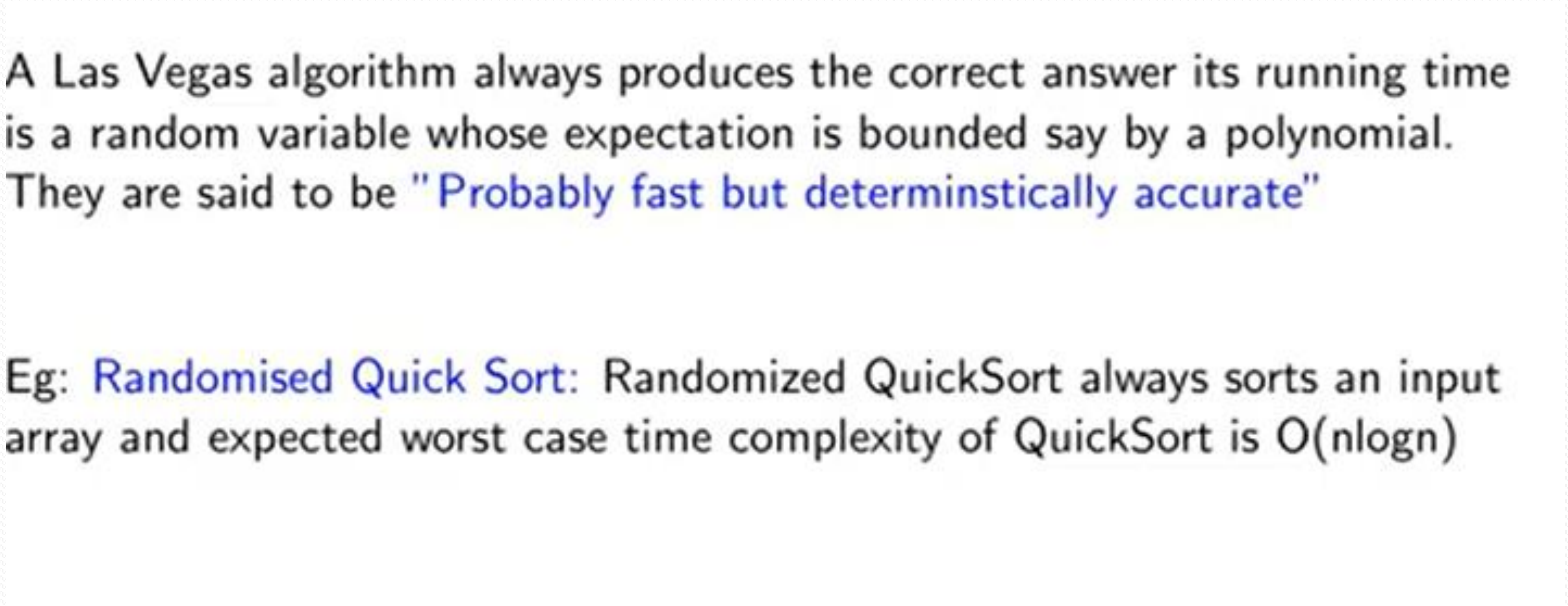


```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

$$\mathbf{Pr}[\text{failure}] = \frac{1}{2^{300}}$$

Worst-case running time: $O(1)$

This is called a Monte Carlo algorithm.
(gambles with correctness but not time)



A Las Vegas algorithm always produces the correct answer its running time is a random variable whose expectation is bounded say by a polynomial. They are said to be "Probably fast but determinstically accurate"

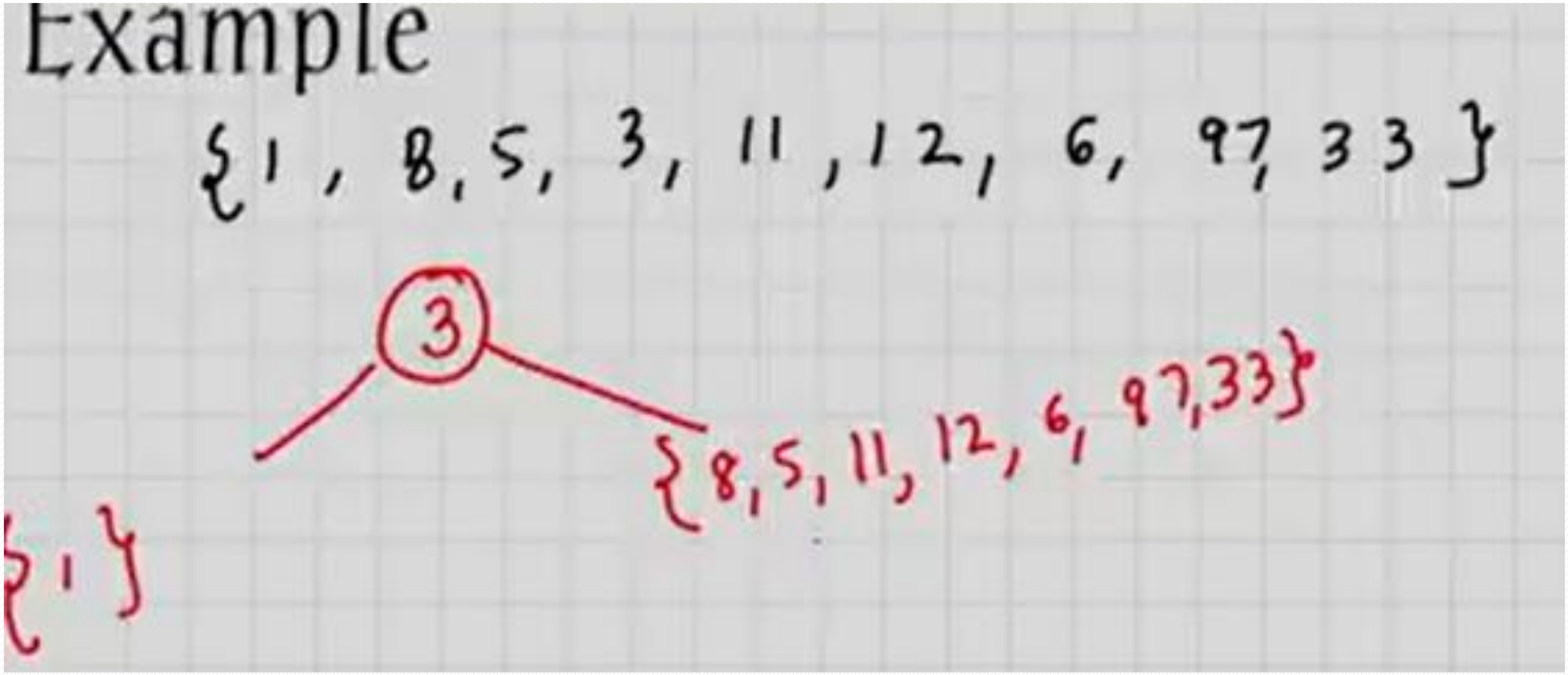
Eg: Randomised Quick Sort: Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nlogn)

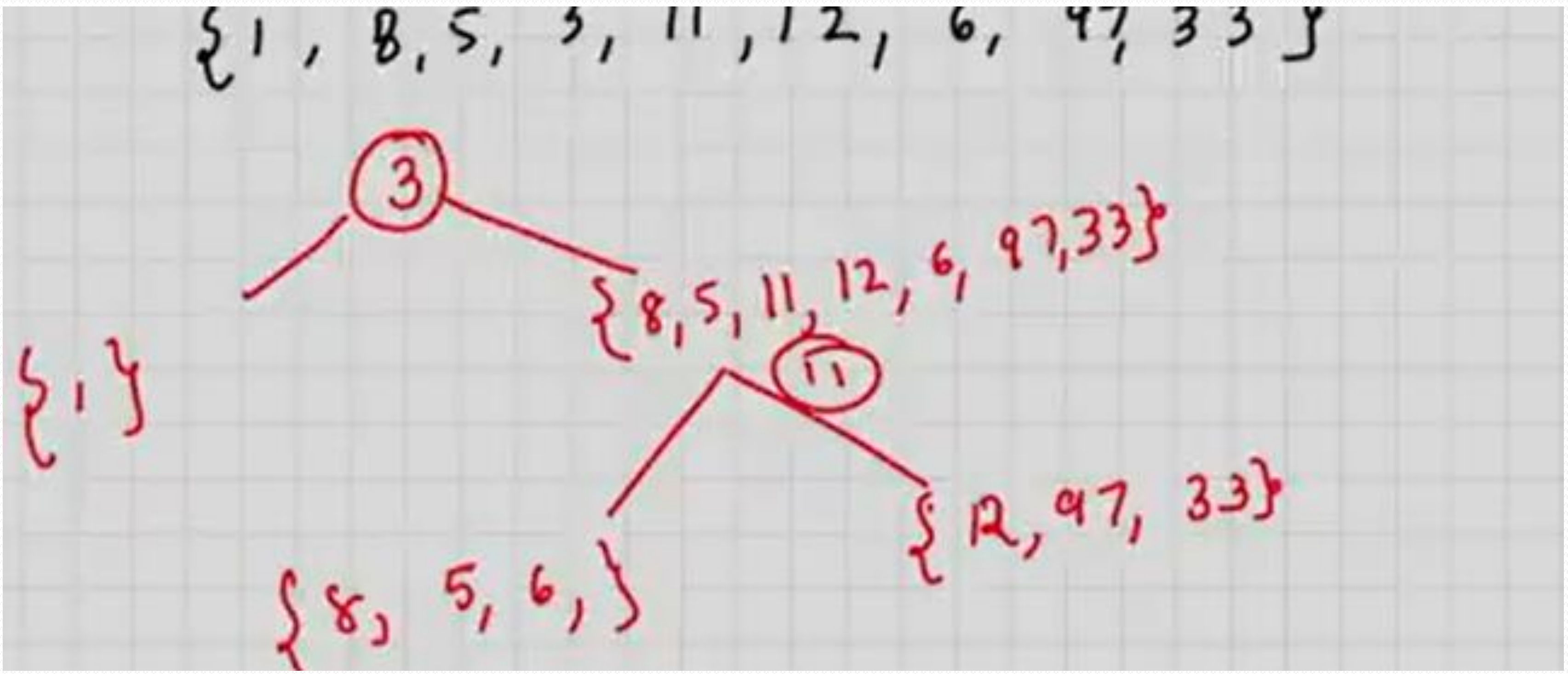
The Quick Sort Problem

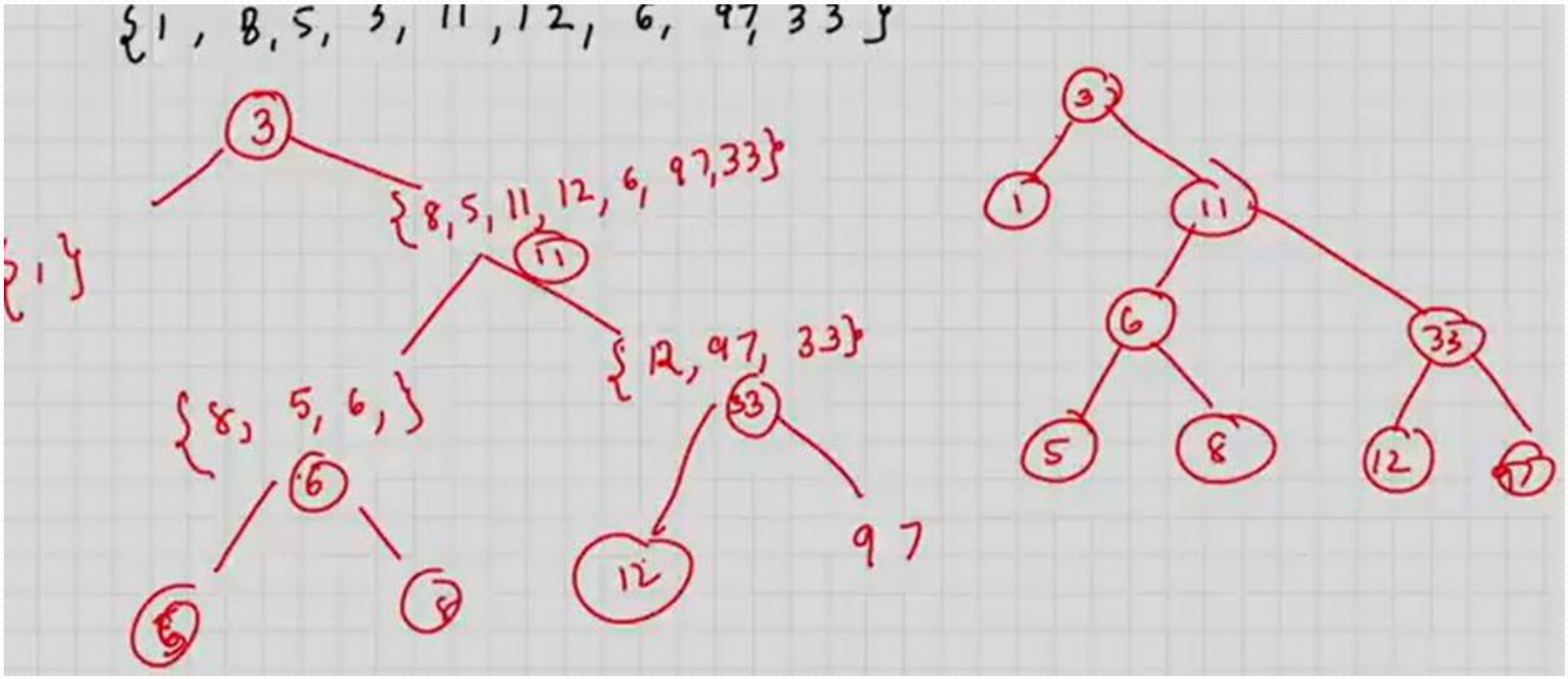
- To sort a given set of numbers
  - In traditional quick sort algorithm, we pick a particular index element as pivot for splitting.
    - Worst Case: $O(n^2)$
    - Average Case: $O(n \log n)$
  - A good pivot can be selected using median finding algorithm but the total complexity will again be $O(n^2)$.
  - So, what if we pick a random element uniformly as pivot and do the partition. We will show that this takes expected $O(n \log n)$ time.

Randomized Quick Sort algorithm

- Assuming all elements are distinct
- We pick a random element x as the pivot and partition the input set S into two sets L and R such:
    - L = numbers less than x
    - R = numbers greater than x
- Recursively sort L and R.
- Return LxR

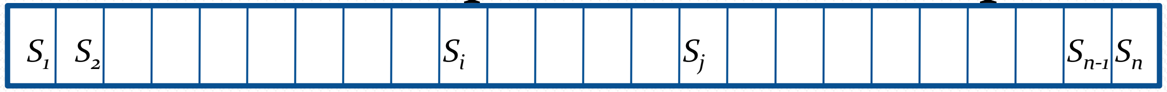

Example
{1, 8, 5, 3, 11, 12, 6, 97, 33}
3
{1}
{8, 5, 11, 12, 6, 97, 33}


{1, 8, 5, 3, 11, 12, 6, 97, 33}
3
{1}
{8, 5, 11, 12, 6, 97, 33}
11
{8, 5, 6,}
{12, 97, 33}

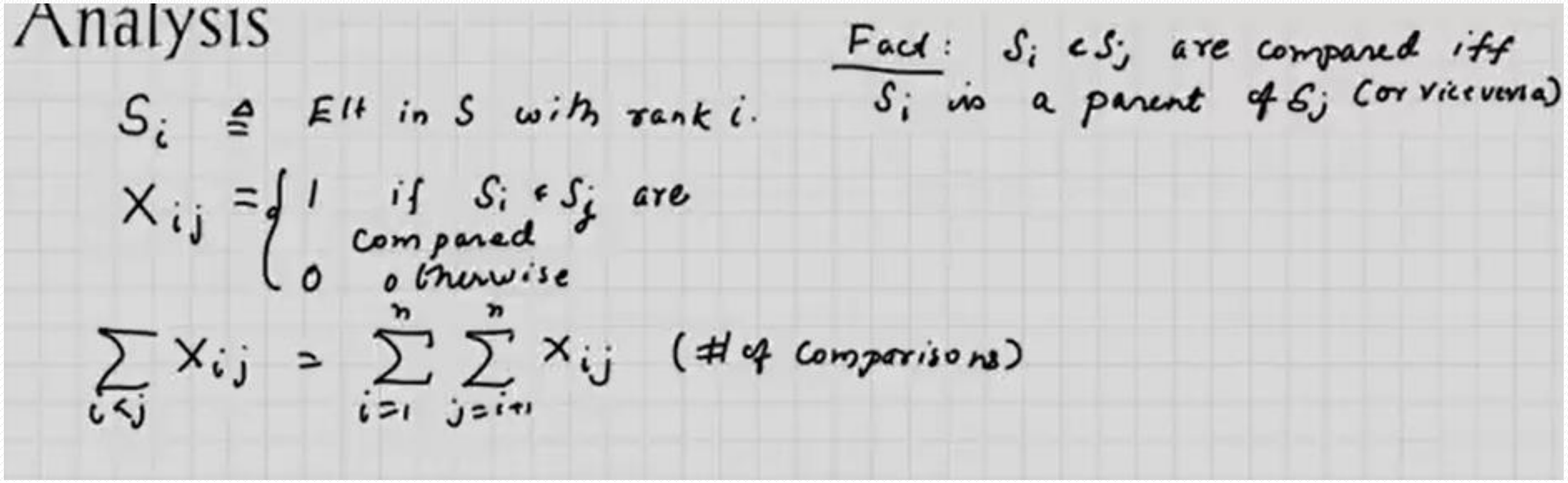

{1, 8, 5, 3, 11, 12, 6, 97, 33}
3
{1}
{8, 5, 11, 12, 6, 97, 33}
11
{8, 5, 6,}
{12, 97, 33}
6
33
5
8
12
97
3
1
11
6
33
5
8
12
97

Analysis of Randomized Quick Sort

- The running time of this algorithm is variable.
- Running time = # of comparisons in this algorithm.
- Expected Running Time, $E[X] = \sum x_i * \Pr[X = x_i]$
- Let S be the sorted sequence of the n input numbers.

$S =$ 

- Let X_{ij} = 1 if S_i and S_j are compared in the algo
  = 0 otherwise
- Running time = # of comparisons = $\sum_{i=1}^{n} \sum_{j \geq i} X_{ij}$



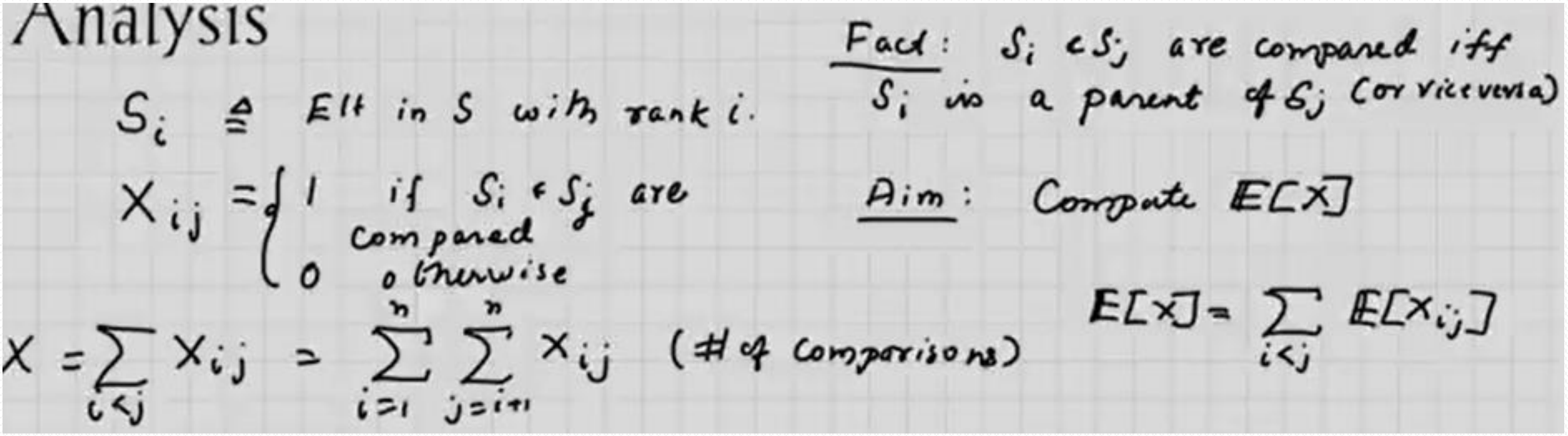
Analysis

$S_i \triangleq$ Elt in S with rank i.

$$X_{ij} = \begin{cases} 1 & \text{if } S_i \,\&\, S_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_{i<j} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \quad (\#\text{ of comparisons})$$

Fact: S_i & S_j are compared iff S_i is a parent of S_j (or viceversa)



Analysis

$S_i \triangleq$ Elt in S with rank i.

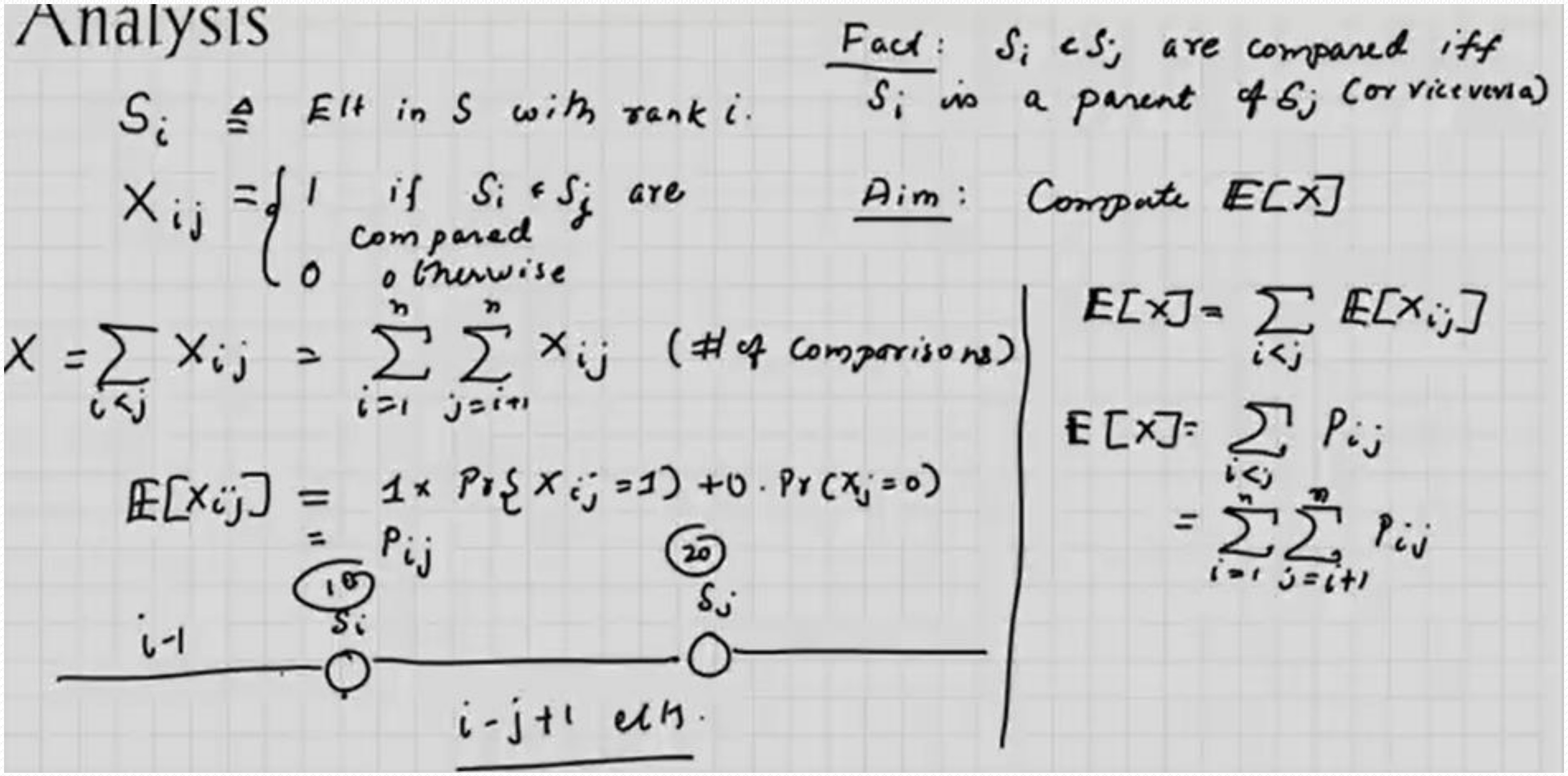
$$X_{ij} = \begin{cases} 1 & \text{if } S_i \,\&\, S_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

$$X = \sum_{i<j} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \quad \text{(\# of comparisons)}$$

Fact: S_i & S_j are compared iff S_i is a parent of S_j (or viceversa)

Aim: Compute $E[X]$

$$E[X] = \sum_{i<j} E[X_{ij}]$$



Analysis

<u>Fact</u>: S_i & S_j are compared iff S_i is a parent of S_j (or viceversa)

$S_i \triangleq$ Elt in S with rank i.

$$X_{ij} = \begin{cases} 1 & \text{if } S_i \,\&\, S_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

<u>Aim</u>: Compute $E[X]$

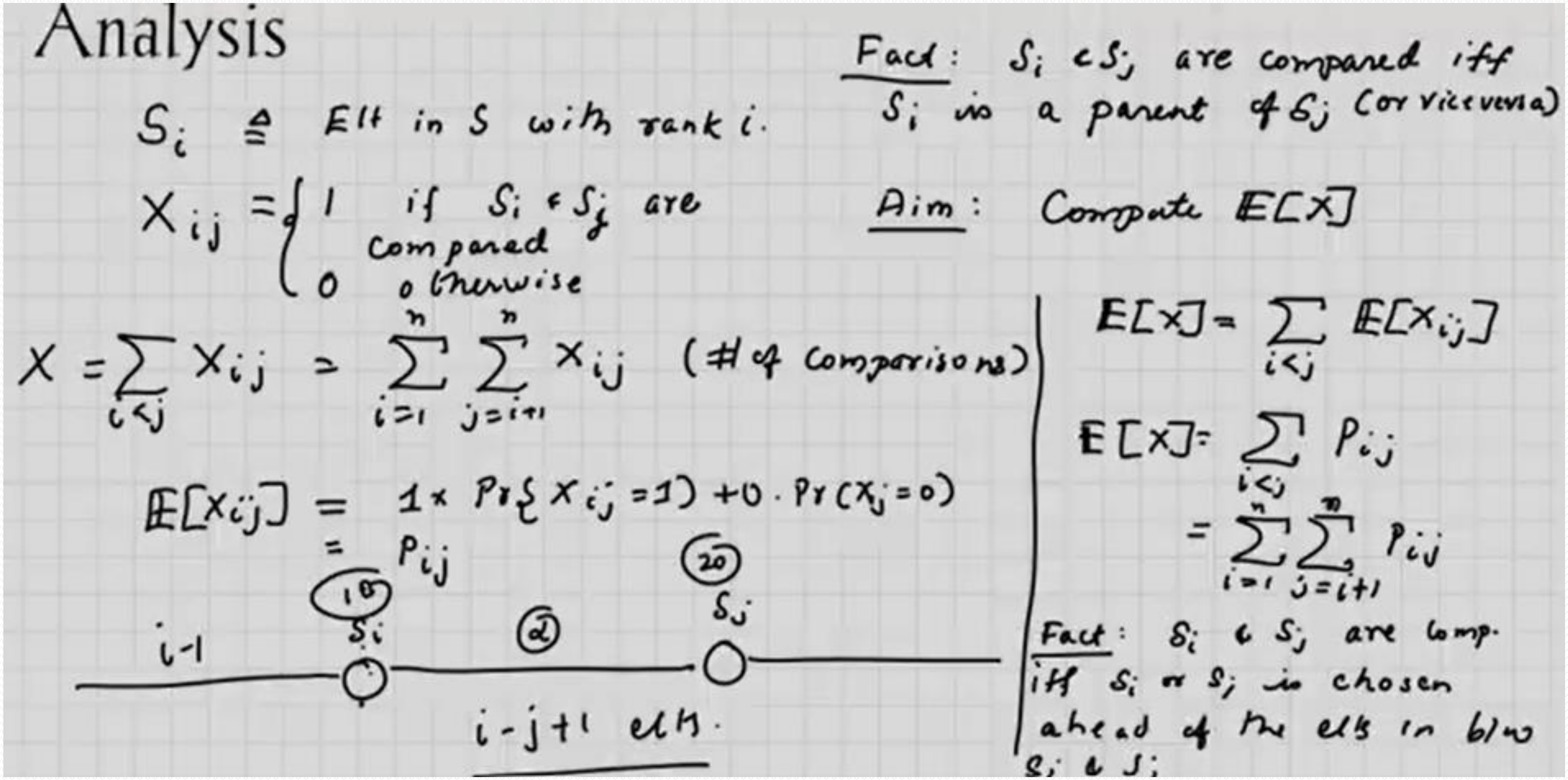
$$X = \sum_{i<j} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \quad (\#\text{ of comparisons})$$

$$E[X] = \sum_{i<j} E[X_{ij}]$$

$$E[X_{ij}] = 1 \times Pr\{X_{ij} = 1) + 0 \cdot Pr(X_j = 0)$$
$$= P_{ij}$$

$$E[X] = \sum_{i<j} P_{ij}$$
$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} P_{ij}$$

i-1
$\boxed{10}$ S_i
$\boxed{20}$ S_j
$\underline{i-j+1 \text{ elts.}}$



Analysis

Fact: S_i & S_j are compared iff S_i is a parent of S_j (or viceversa)

$S_i \triangleq$ Elt in S with rank i.

$$X_{ij} = \begin{cases} 1 & \text{if } S_i \,\&\, S_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

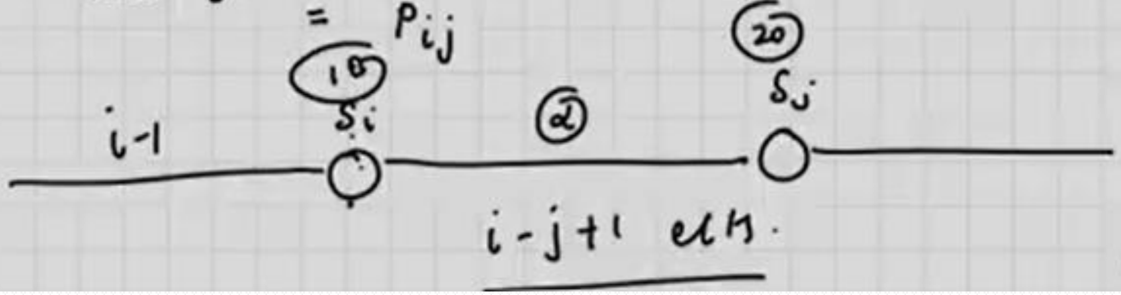
Aim: Compute $\mathbb{E}[X]$

$$X = \sum_{i<j} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} x_{ij} \quad (\text{\# of comparisons})$$

$$\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}]$$

$$\mathbb{E}[X_{ij}] = 1 \times \Pr\{X_{ij} = 1) + 0 \cdot \Pr(X_j = 0) = p_{ij}$$

$$\mathbb{E}[X] = \sum_{i<j} p_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{ij}$$



Fact: S_i & S_j are comp. iff S_i or S_j is chosen ahead of the elts in b/w S_i & S_j

Analysis cont…

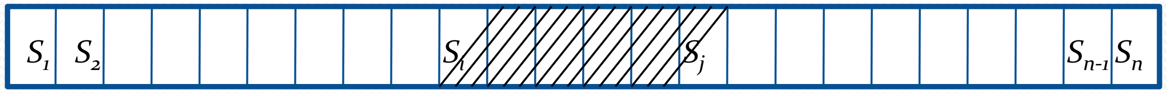
- The expected running time = $E[\sum_{i=1}^{n}\sum_{j\geq i} X_{ij}] = \sum_{i=1}^{n}\sum_{j\geq i} E[X_{ij}]$
- Now, $E[X_{ij}]$ = 1 * Pr[S_i and S_j are compared in our alg.] + 0 * Pr[S_i and S_j are not compared]
- Suppose we have a set of numbers:

  2, 7, 15, 18, 19, 23, 35

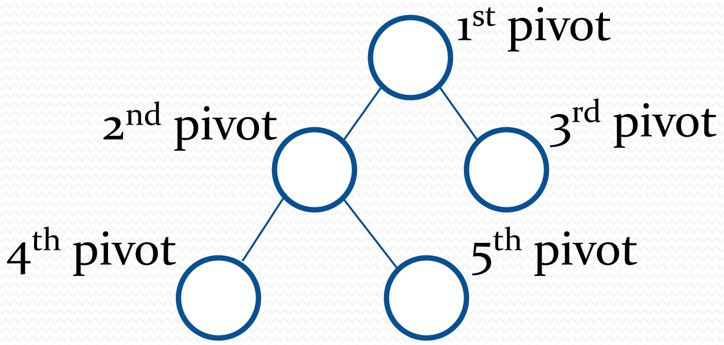
  - In this 18 and 19 will always be compared.
  - 2 and 35 will be compared only if compared at root.

Analysis cont…

- Pr[S_i and S_j are compared in our algo] = Pr[the first element chosen as pivot in set $\{s_i, s_{i+1}, \ldots., s_j\}$ is either s_i or s_j].



- To elements get compared only if they have ancestor relationship in the tree.
- Pr[Picking S_i or S_j] = $\frac{1}{j-i+1}+\frac{1}{j-i+1}=\frac{2}{j-i+1}$



Analysis cont…

- Thus the expected runtime:

$$E[\sum_{i=1}^{n}\sum_{j\geq i}^{n} X_{ij}] = \sum_{i=1}^{n}\sum_{j\geq i}^{n}\frac{2}{j-i+1} \leq \sum_{i=1}^{n}\sum_{j=1}^{n}\frac{2}{j} = O(n\log n)$$

$$\text{Since, } 1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}\approx \ln n$$

- This algorithm will always give the right answer though the running time may be different. This is an example of ***Las Vegas algorithms***.