Average-Case Analysis and Randomization

Textbook Reading

Chapter 7 & Sections 8.4, 9.2

Overview

Design principle

- Do the easy thing and hope it works for most inputs
- Make random choices and hope they're good

Problems

- Sorting (Quick Sort)
- Permuting
- Selection
- Game tree evaluation

Quick Sort Revisited

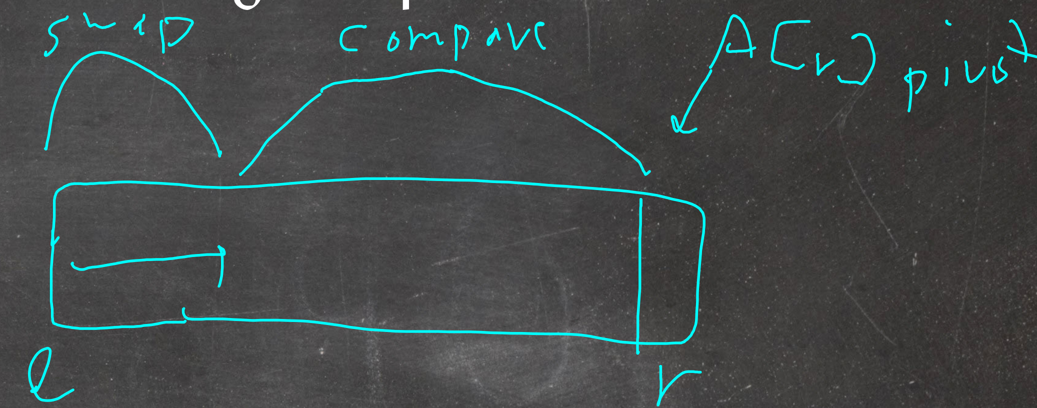
The problem with deterministic Quick Sort:

The running time is in $O(n \lg n)$, but the algorithm for finding the pivot is non-trivial (and slow).

Quick Sort Revisited

The problem with deterministic Quick Sort:

The running time is in $O(n \lg n)$, but the algorithm for finding the pivot is non-trivial (and slow).



Remedy:

Blindly use the last element as pivot.

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
        then i = i + 1
             swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Average-Case Analysis of Simple Quick Sort

Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

Average-Case Analysis of Simple Quick Sort

Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

We defined the average-case running time of an algorithm as the average of its running time over all possible inputs of size n.

Average-Case Analysis of Simple Quick Sort

Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

We defined the average-case running time of an algorithm as the average of its running time over all possible inputs of size n.

Problem: There are infinitely many different inputs of size n!

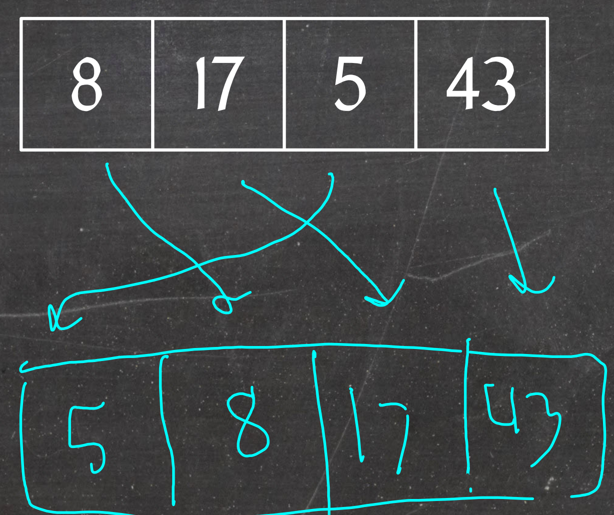
Average-Case Analysis of Simple Quick Sort

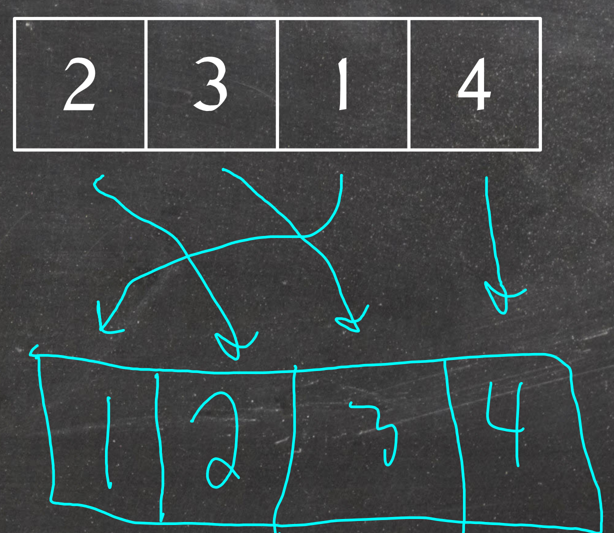
Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

We defined the average-case running time of an algorithm as the average of its running time over all possible inputs of size n.

Problem: There are infinitely many different inputs of size n!

Observation: Simple Quick Sort behaves the same on all inputs whose elements have the same relative order.





Average-Case Analysis of Simple Quick Sort

Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

We defined the average-case running time of an algorithm as the average of its running time over all possible inputs of size n.

Problem: There are infinitely many different inputs of size n!

Observation: Simple Quick Sort behaves the same on all inputs whose elements have the same relative order.

8	17	5	43

2	3	1	4

$\Rightarrow$ The input to SimpleQuickSort is a permutation π of the sorted output sequence $\langle x_1, x_2, \ldots, x_n \rangle$ we expect as the output.

Average-Case Analysis of Simple Quick Sort

Lemma: The average-case running time of SimpleQuickSort is in $O(n \lg n)$.

We defined the average-case running time of an algorithm as the average of its running time over all possible inputs of size n.

Problem: There are infinitely many different inputs of size n!

Observation: Simple Quick Sort behaves the same on all inputs whose elements have the same relative order.

8	17	5	43

2	3	1	4

$\Rightarrow$ The input to SimpleQuickSort is a permutation π of the sorted output sequence $\langle x_1, x_2, \ldots, x_n \rangle$ we expect as the output.

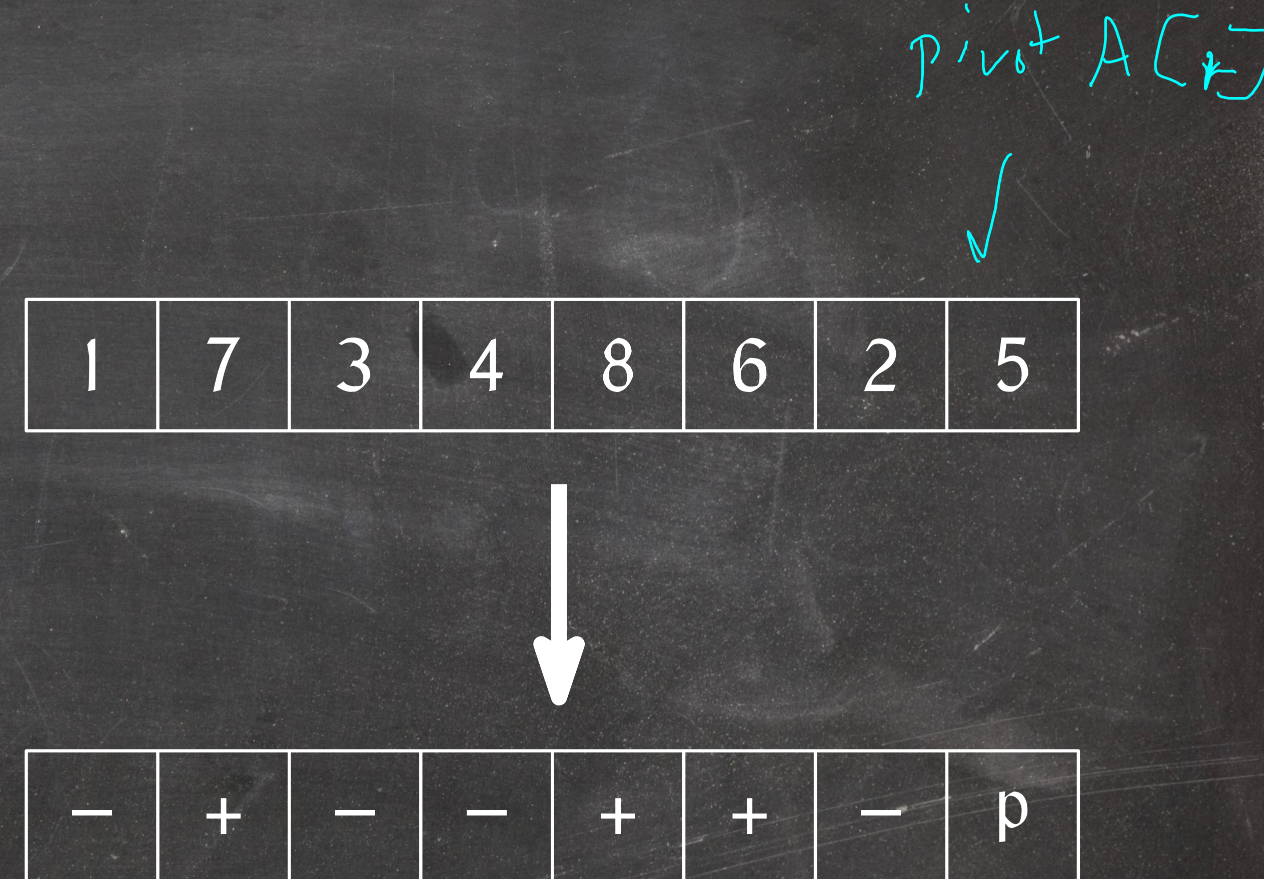
$\Rightarrow$ The average-case running time of SimpleQuickSort is the same as its expected running time on a uniformly random input permutation.

Partitioning Maintains Uniformity

Lemma: If $A[\ell\,..\,r]$ is a uniform random permutation of the elements in $A[\ell\,..\,r]$, then the two subarrays $A[\ell\,..\,m-1]$ and $A[m+1\,..\,r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

Partitioning Maintains Uniformity

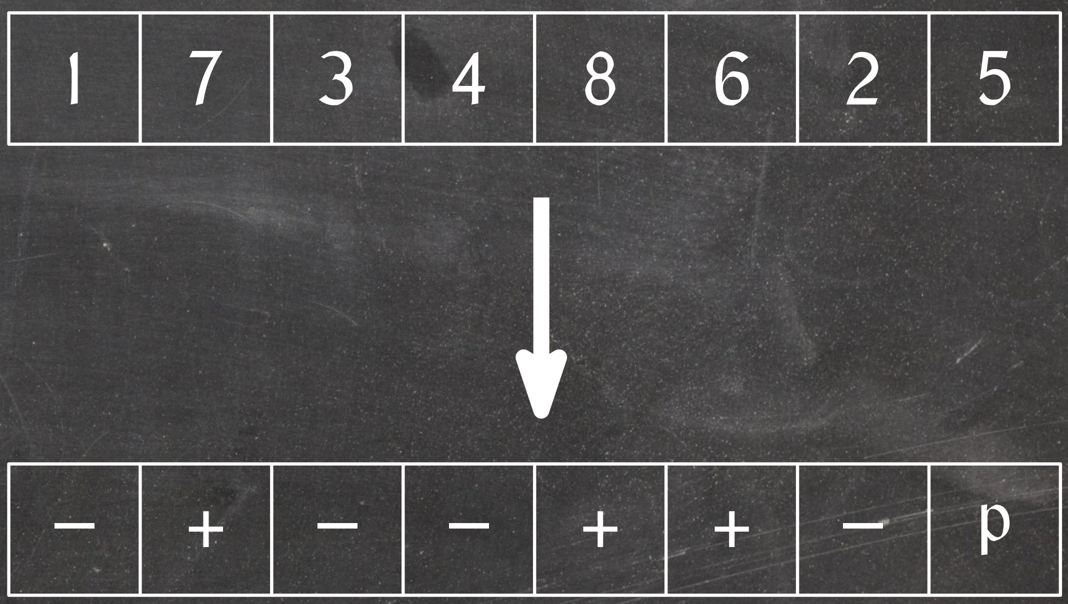
Lemma: If $A[\ell\,..\,r]$ is a uniform random permutation of the elements in $A[\ell\,..\,r]$, then the two subarrays $A[\ell\,..\,m-1]$ and $A[m+1\,..\,r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.



Partitioning Maintains Uniformity

Lemma: If $A[\ell \mathinner{.\,.} r]$ is a uniform random permutation of the elements in $A[\ell \mathinner{.\,.} r]$, then the two subarrays $A[\ell \mathinner{.\,.} m-1]$ and $A[m+1 \mathinner{.\,.} r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

The behaviour of Partition depends only on the sequence of −s and +s!

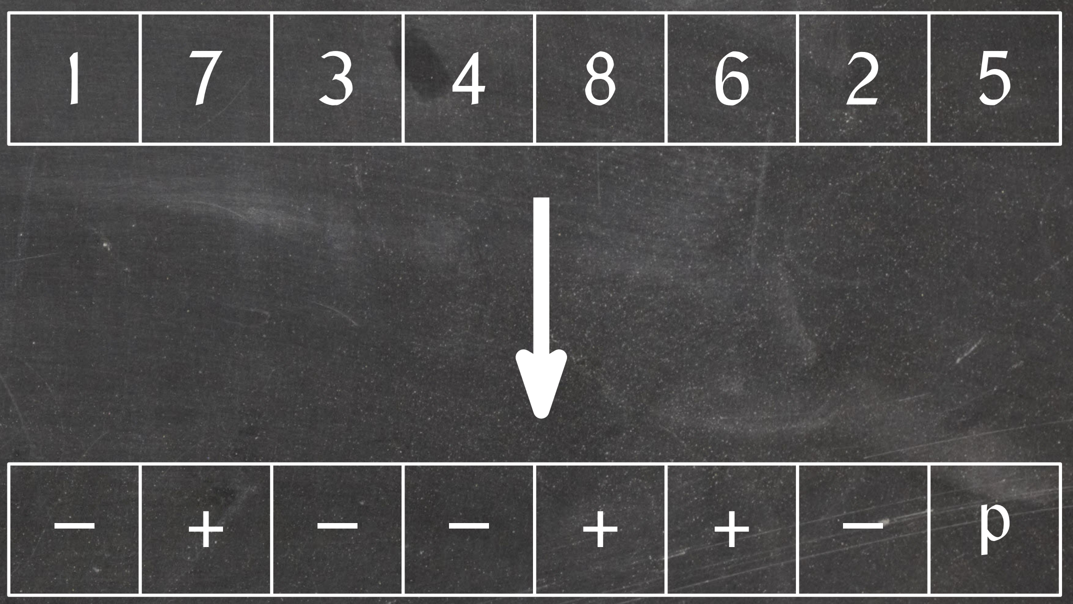


Partitioning Maintains Uniformity

Lemma: If $A[\ell \,..\, r]$ is a uniform random permutation of the elements in $A[\ell \,..\, r]$, then the two subarrays $A[\ell \,..\, m-1]$ and $A[m+1 \,..\, r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

The behaviour of Partition depends only on the sequence of −s and +s!

The −s are exactly the elements that end up in $A[\ell \,..\, m-1]$, the +s end up in $A[m+1 \,..\, r]$.



Partitioning Maintains Uniformity

Lemma: If $A[\ell \,..\, r]$ is a uniform random permutation of the elements in $A[\ell \,..\, r]$, then the two subarrays $A[\ell \,..\, m-1]$ and $A[m+1 \,..\, r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

The behaviour of Partition depends only on the sequence of −s and +s!

The −s are exactly the elements that end up in $A[\ell \,..\, m-1]$, the +s end up in $A[m+1 \,..\, r]$.

In a uniformly random permutation, any permutation of the −s or +s is equally likely.

1	7	3	4	8	6	2	5

−	+	−	−	+	+	−	p

Partitioning Maintains Uniformity

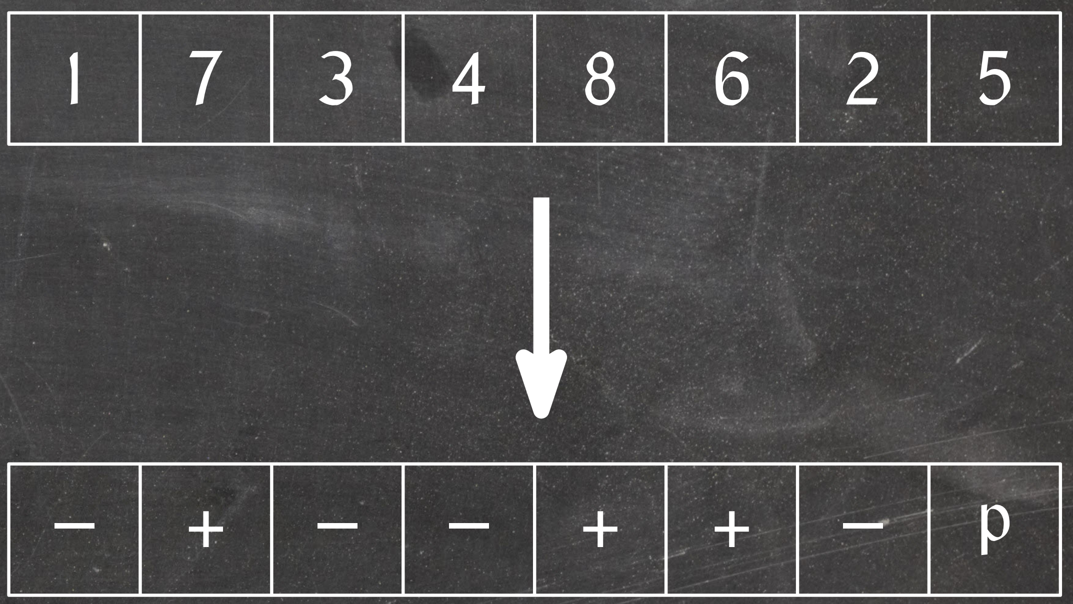
Lemma: If $A[\ell\,..\,r]$ is a uniform random permutation of the elements in $A[\ell\,..\,r]$, then the two subarrays $A[\ell\,..\,m-1]$ and $A[m+1\,..\,r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

The behaviour of Partition depends only on the sequence of −s and +s!

The −s are exactly the elements that end up in $A[\ell\,..\,m-1]$, the +s end up in $A[m+1\,..\,r]$.

In a uniformly random permutation, any permutation of the −s or +s is equally likely.

Each such permutation produces a different permutation of $A[\ell\,..\,m-1]$ or $A[m+1\,..\,r]$.



Partitioning Maintains Uniformity

Lemma: If $A[\ell \,..\, r]$ is a uniform random permutation of the elements in $A[\ell \,..\, r]$, then the two subarrays $A[\ell \,..\, m-1]$ and $A[m+1 \,..\, r]$ produced by Partition(A, ℓ, r) are also uniform random permutations of the elements they contain.

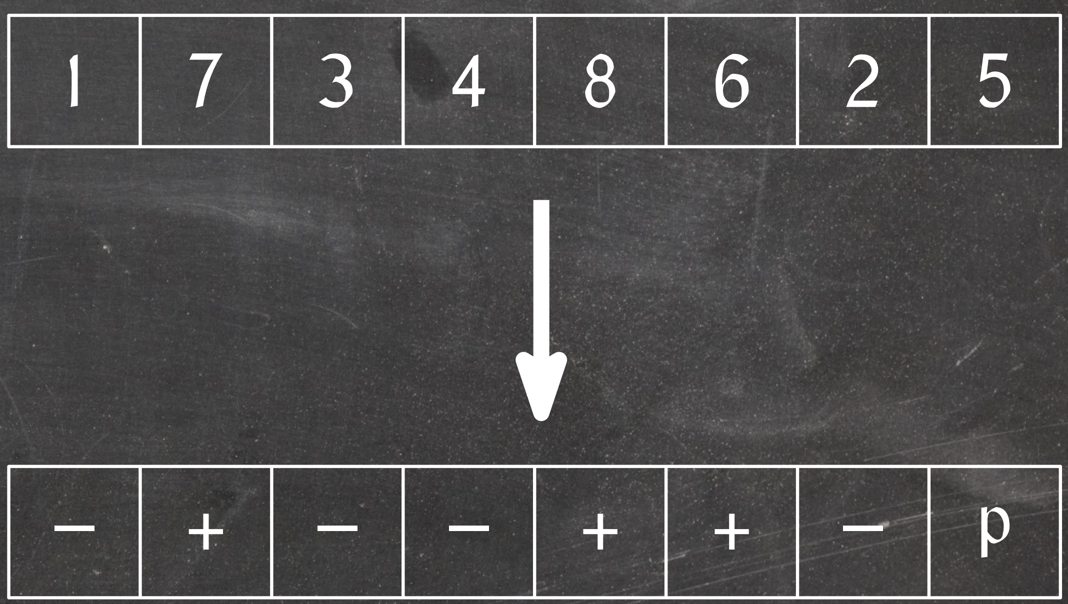
The behaviour of Partition depends only on the sequence of −s and +s!

The −s are exactly the elements that end up in $A[\ell \,..\, m-1]$, the +s end up in $A[m+1 \,..\, r]$.

In a uniformly random permutation, any permutation of the −s or +s is equally likely.

Each such permutation produces a different permutation of $A[\ell \,..\, m-1]$ or $A[m+1 \,..\, r]$.

$\Rightarrow$ $A[\ell \,..\, m-1]$ and $A[m+1 \,..\, r]$ are uniform random permutations.



Average-Case Analysis of Simple Quick Sort

Observation: The running time of SimpleQuickSort is in $O(n + C)$, where C is the number of comparisons it performs between input elements.

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
   then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
   do if A[j] ≤ A[r]
         then i = i + 1
              swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Average-Case Analysis of Simple Quick Sort

Observation: The running time of SimpleQuickSort is in $O(n + C)$, where C is the number of comparisons it performs between input elements.

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
            then i = i + 1
                 swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

- There are $O(n)$ recursive calls in total.
- The cost of each recursive call, excluding the call to Partition is constant.
- The cost of Partition is $O(1 + \#\text{ comparisons it performs})$.

Average-Case Analysis of Simple Quick Sort

Observation: The running time of SimpleQuickSort is in $O(n + C)$, where C is the number of comparisons it performs between input elements.

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
   then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
   do if A[j] ≤ A[r]
         then i = i + 1
              swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

- There are $O(n)$ recursive calls in total.
- The cost of each recursive call, excluding the call to Partition is constant.
- The cost of Partition is $O(1 + \#\text{ comparisons it performs})$.

$\Rightarrow$ It suffices to prove that $E[C] \in O(n \lg n)$.

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared at most once.

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared at most once.

- Each call SimpleQuickSort(A, ℓ, r) compares every element in $A[\ell \,..\, r-1]$ to the pivot p stored in $A[r]$.
- The pivot is not part of the recursive calls SimpleQuickSort$(A, \ell, m-1)$ and SimpleQuickSort$(A, m+1, r)$.

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
            then i = i + 1
                 swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared at most once.

- Each call SimpleQuickSort(A, ℓ, r) compares every element in $A[\ell \,..\, r-1]$ to the pivot p stored in $A[r]$.
- The pivot is not part of the recursive calls SimpleQuickSort$(A, \ell, m-1)$ and SimpleQuickSort$(A, m+1, r)$.

$$\text{Let } C_{ij} = \begin{cases} 1 & \text{if } x_i \text{ and and } x_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}.$$

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
            then i = i + 1
                    swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared at most once.

- Each call SimpleQuickSort(A, ℓ, r) compares every element in A[ℓ . . r − 1] to the pivot p stored in A[r].
- The pivot is not part of the recursive calls SimpleQuickSort(A, ℓ, m − 1) and SimpleQuickSort(A, m + 1, r).

Let $C_{ij} = \begin{cases} 1 & \text{if } x_i \text{ and and } x_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$.

$$\Rightarrow C = \sum_{i,j} C_{ij}$$

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
          then i = i + 1
               swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared at most once.

- Each call SimpleQuickSort(A, ℓ, r) compares every element in A[ℓ . . r − 1] to the pivot p stored in A[r].
- The pivot is not part of the recursive calls SimpleQuickSort(A, ℓ, m − 1) and SimpleQuickSort(A, m + 1, r).

SimpleQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
m = Partition(A, ℓ, r)
SimpleQuickSort(A, ℓ, m − 1)
SimpleQuickSort(A, m + 1, r)
```

Partition(A, ℓ, r)

```
i = ℓ − 1
for j = ℓ to r − 1
    do if A[j] ≤ A[r]
          then i = i + 1
               swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

Let $C_{ij} = \begin{cases} 1 & \text{if } x_i \text{ and and } x_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}.$

$$\Rightarrow C = \sum_{i,j} C_{ij}$$

$$\Rightarrow E[C] = E\left[\sum_{i,j} C_{ij}\right] = \sum_{i,j} E[C_{ij}]$$

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.

Average-Case Analysis of Simple Quick Sort

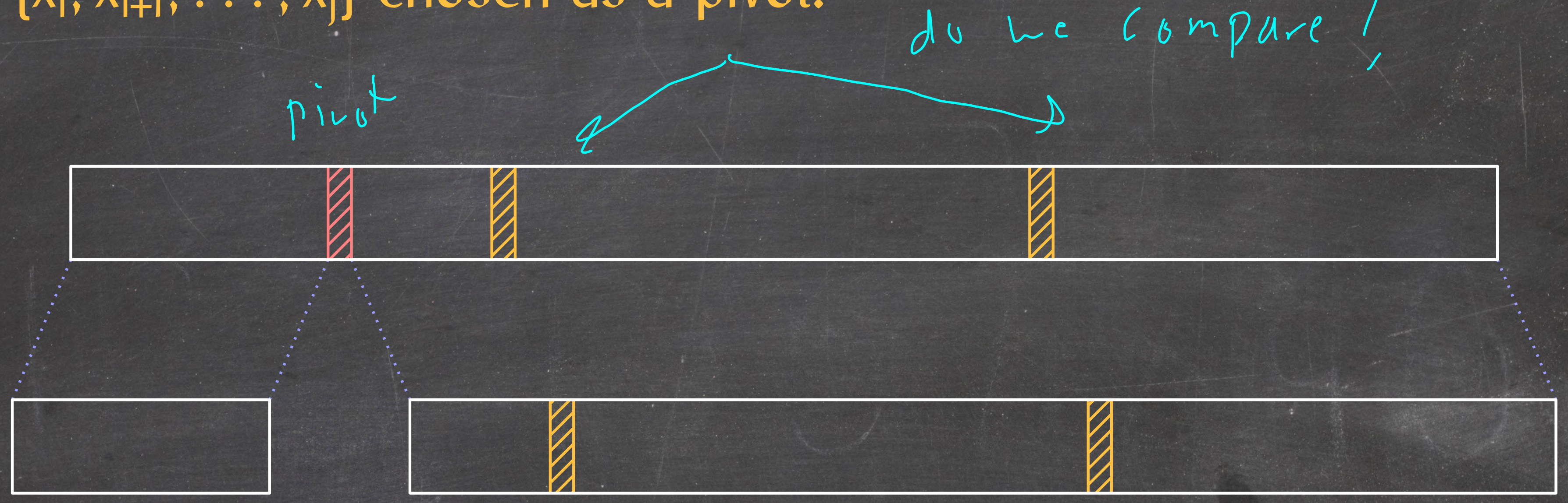
Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.

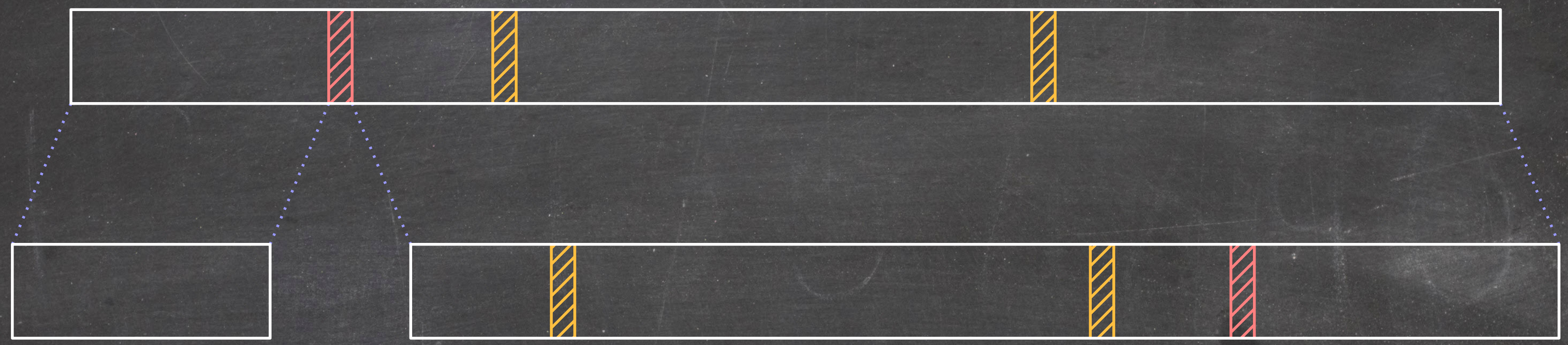
Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



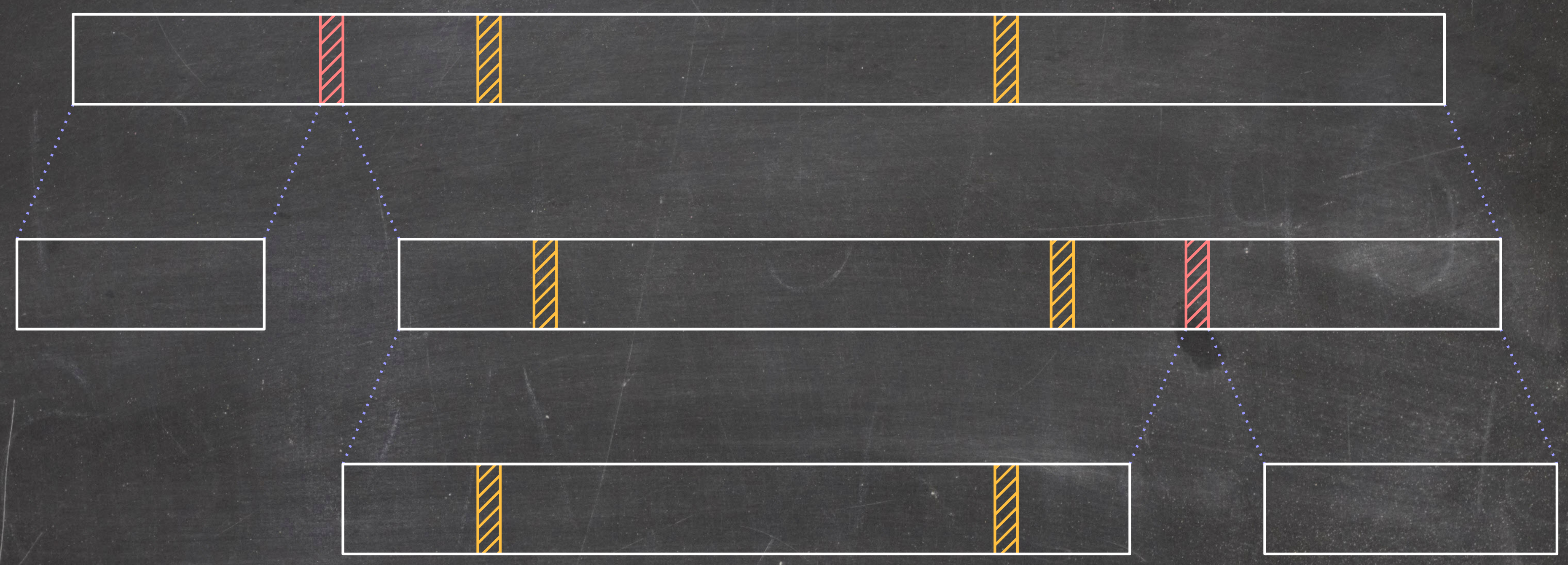
Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



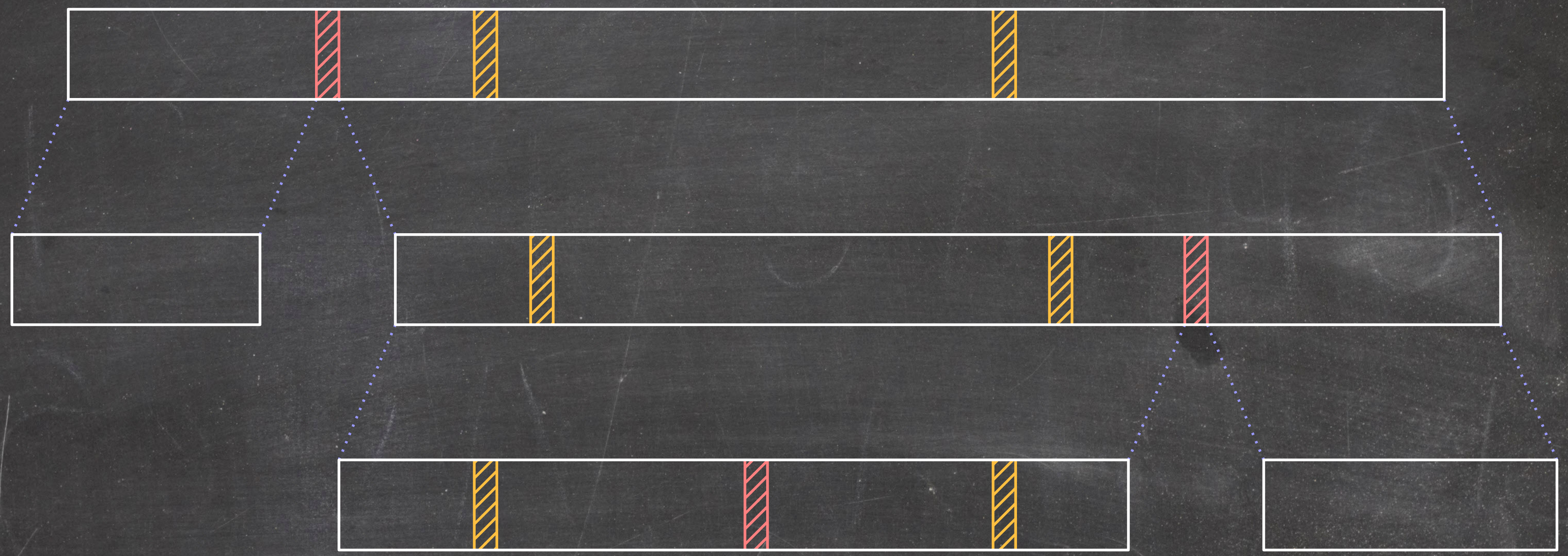
Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



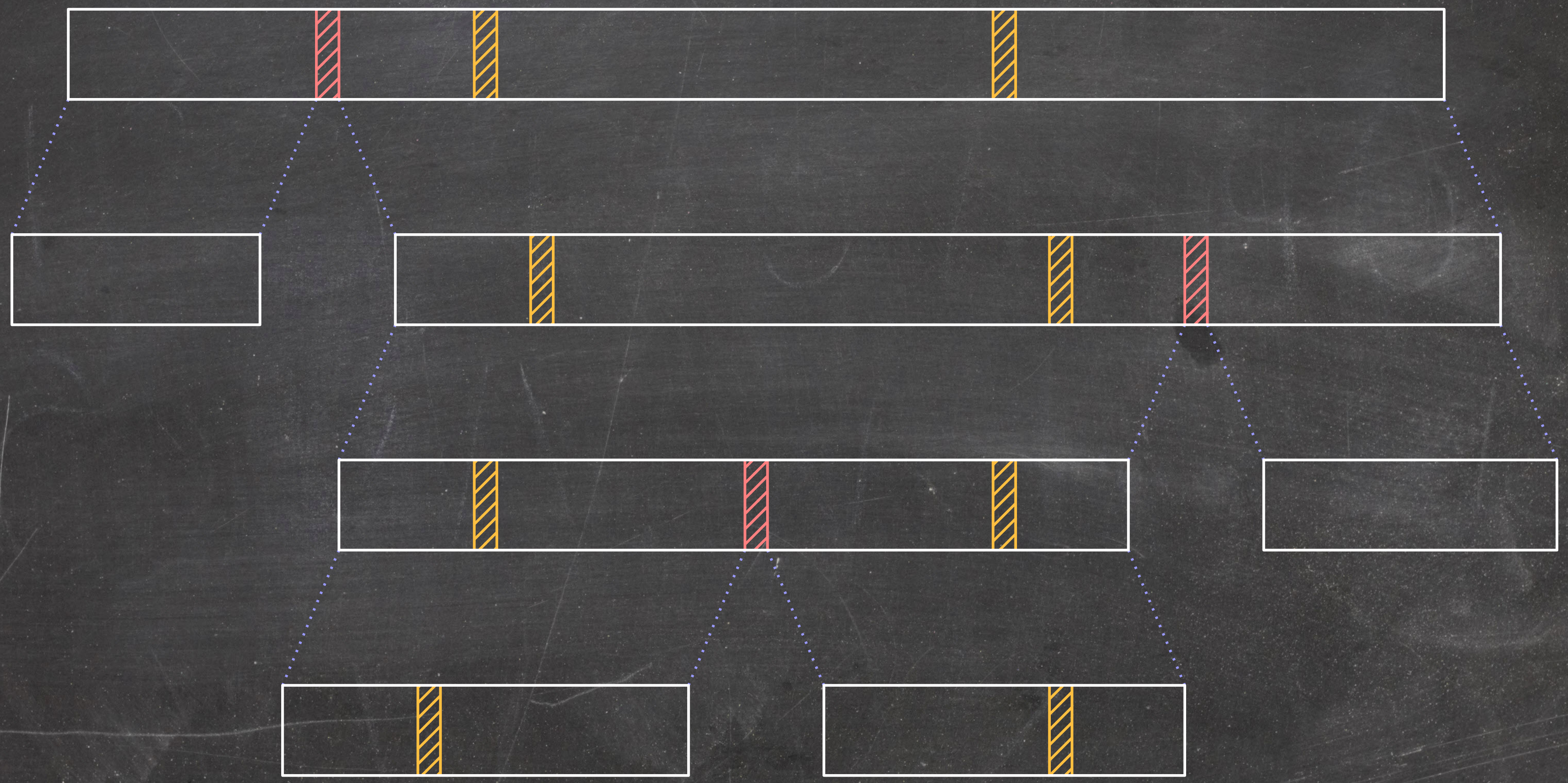
Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



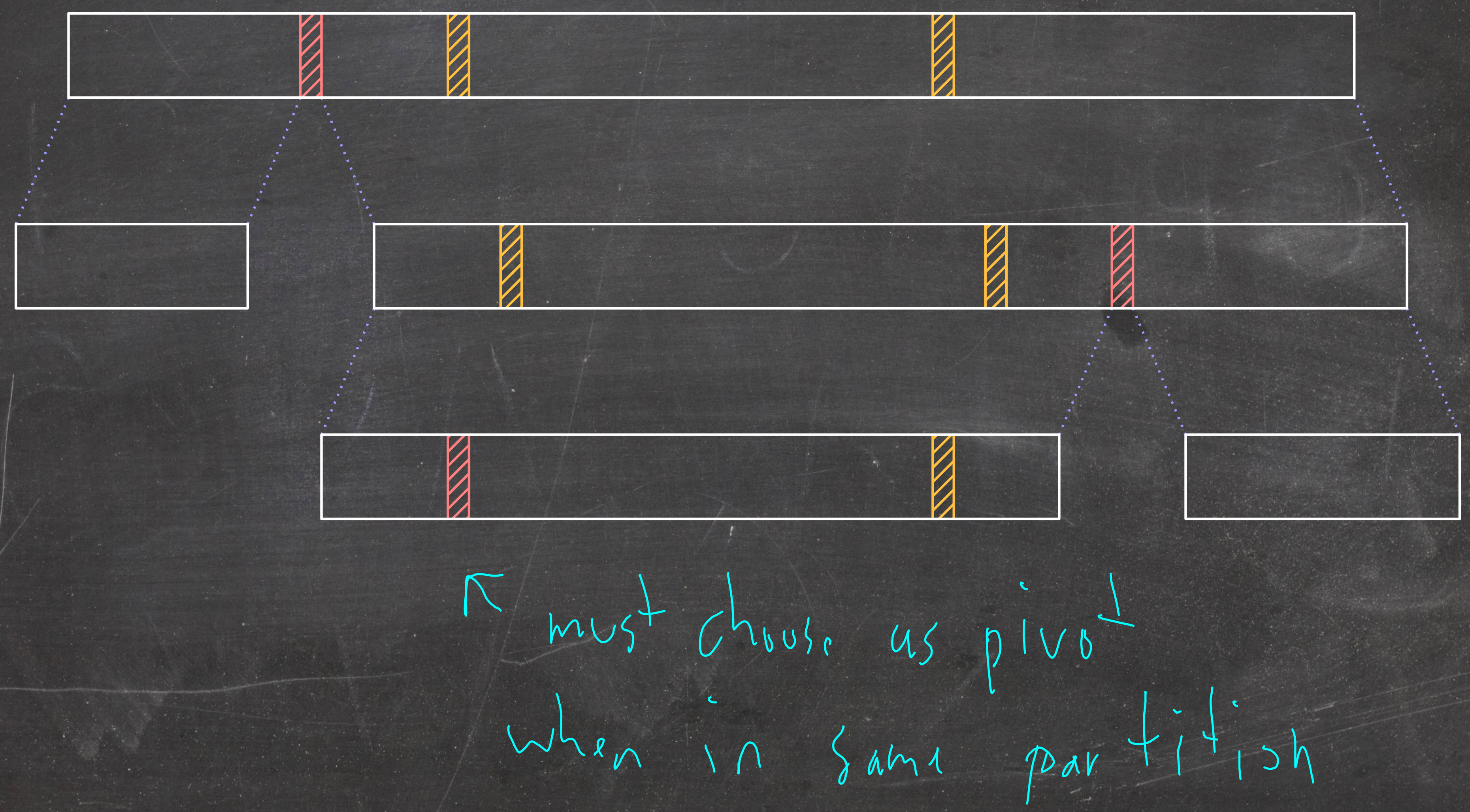
can't be compared

Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.

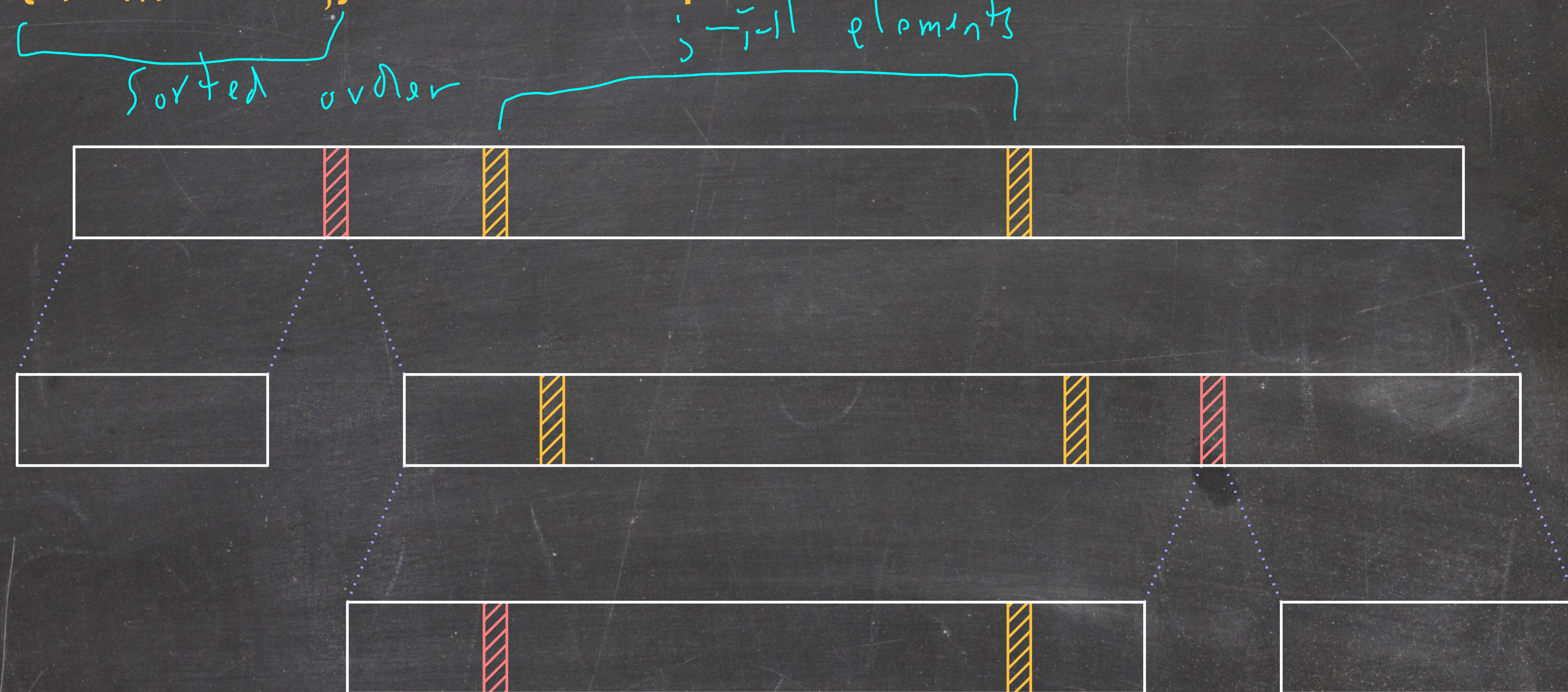
Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



Average-Case Analysis of Simple Quick Sort

Observation: Two elements x_i and x_j are compared if and only if x_i or x_j is the first element in $\{x_i, x_{i+1}, \ldots, x_j\}$ chosen as a pivot.



Corollary: $E[C_{ij}] = \dfrac{2}{j - i + 1}$.

Average-Case Analysis of Simple Quick Sort

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$$

add up the expectations

Average-Case Analysis of Simple Quick Sort

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

← value we proved

Average-Case Analysis of Simple Quick Sort

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$j = i+1, i+2, \ldots n$

i here

$\frac{2}{j-i+1}$

$= \frac{2}{1+1}, \frac{2}{2+1}, \frac{2}{3+1}, \ldots \frac{2}{n-i+1}$

set to k

Average-Case Analysis of Simple Quick Sort

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1}$$

$$< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

Average-Case Analysis of Simple Quick Sort

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

$$= 2(n-1)H_n.$$

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \text{nth Harmonic Number}$$

Average-Case Analysis of Simple Quick Sort

$$\sum_{i=1}^{n} \frac{1}{i}$$

Average-Case Analysis of Simple Quick Sort

$$\int_1^{n+1} \frac{dx}{x} < \sum_{i=1}^{n} \frac{1}{i}$$

Average-Case Analysis of Simple Quick Sort

$$\ln(n+1) = \int_1^{n+1} \frac{dx}{x} < \sum_{i=1}^{n} \frac{1}{i}$$

Average-Case Analysis of Simple Quick Sort

$$\ln(n+1) = \int_1^{n+1} \frac{dx}{x} < \sum_{i=1}^{n} \frac{1}{i} < 1 + \int_1^{n} \frac{dx}{x}$$

Average-Case Analysis of Simple Quick Sort

$$\ln(n+1) = \int_1^{n+1} \frac{dx}{x} < \sum_{i=1}^{n} \frac{1}{i} < 1 + \int_1^{n} \frac{dx}{x} = 1 + \ln n$$

Average-Case Analysis of Simple Quick Sort

$$\ln(n+1) = \int_1^{n+1} \frac{dx}{x} < \sum_{i=1}^{n} \frac{1}{i} < 1 + \int_1^{n} \frac{dx}{x} = 1 + \ln n$$

$$\Rightarrow E[C] \leq 2(n-1)H_n \in O(n \lg n)$$

Interpretation of Average-Case Analysis

Algorithms that are fast in the worst case are the gold standard but are difficult to design and often have higher constant factors than algorithms that are efficient on average.

Interpretation of Average-Case Analysis

Algorithms that are fast in the worst case are the gold standard but are difficult to design and often have higher constant factors than algorithms that are efficient on average.

Worst-case efficiency is desirable if we need performance guarantees every single time we run the algorithm.

e.g simple-quicksort

1, 2, ..., n

1, 2, ..., n-1

1, 2, ..., n-2

Interpretation of Average-Case Analysis

Algorithms that are fast in the worst case are the gold standard but are difficult to design and often have higher constant factors than algorithms that are efficient on average.

Worst-case efficiency is desirable if we need performance guarantees every single time we run the algorithm.

Algorithms that are fast on average are often simpler and on average faster than worst-case efficient algorithms.

Interpretation of Average-Case Analysis

Algorithms that are fast in the worst case are the gold standard but are difficult to design and often have higher constant factors than algorithms that are efficient on average.

Worst-case efficiency is desirable if we need performance guarantees every single time we run the algorithm.

Algorithms that are fast on average are often simpler and on average faster than worst-case efficient algorithms.

They are a good choice when we want good performance most of the time and possibly averaged over running the algorithm many times.

Interpretation of Average-Case Analysis

What exactly is the meaning of the following statement?

"The average-case running time of algorithm A is T(n)."

Interpretation of Average-Case Analysis

What exactly is the meaning of the following statement?

"The average-case running time of algorithm A is T(n)."

"If every input is equally likely, then we expect to see a running time of T(n) on average."

Interpretation of Average-Case Analysis

What exactly is the meaning of the following statement?

"The average-case running time of algorithm A is T(n)."

"If every input is equally likely, then we expect to see a running time of T(n) on average."

This assumption may not be true in some applications, invalidating the performance prediction we obtain using average-case analysis!

Interpretation of Average-Case Analysis

What exactly is the meaning of the following statement?

"The average-case running time of algorithm A is $T(n)$."

"If every input is equally likely, then we expect to see a running time of $T(n)$ on average."

This assumption may not be true in some applications, invalidating the performance prediction we obtain using average-case analysis!

Example:

SimpleQuickSort takes $\Theta(n^2)$ time on almost sorted inputs.

There are applications where the inputs to be sorted are all almost sorted.

SimpleQuickSort is a poor choice of a sorting algorithm in such applications.

Randomization

Average-case analysis is applied to a deterministic algorithm and assumes randomness in the input.

Randomization

Average-case analysis is applied to a deterministic algorithm and assumes randomness in the input.

A randomized algorithm makes no assumptions about the input and ensures randomness by making random choices.

Randomization

Average-case analysis is applied to a deterministic algorithm and assumes randomness in the input.

A randomized algorithm makes no assumptions about the input and ensures randomness by making random choices.

Since a randomized algorithm behaves differently every time it runs, there is no way to force it to exhibit its worst-case running time!

Randomization

Average-case analysis is applied to a deterministic algorithm and assumes randomness in the input.

A randomized algorithm makes no assumptions about the input and ensures randomness by making random choices.

Since a randomized algorithm behaves differently every time it runs, there is no way to force it to exhibit its worst-case running time!

The expected running time of a randomized algorithm is an expectation over the random choices the algorithm makes.

Randomization

Average-case analysis is applied to a deterministic algorithm and assumes randomness in the input.

A randomized algorithm makes no assumptions about the input and ensures randomness by making random choices.

Since a randomized algorithm behaves differently every time it runs, there is no way to force it to exhibit its worst-case running time!

The expected running time of a randomized algorithm is an expectation over the random choices the algorithm makes.

$\Rightarrow$ No more assumptions about the probability distribution. We know the distribution of the choices the algorithm makes.

Randomized Quick Sort, Take 1

The expected running time of SimpleQuickSort on a uniform random permutation is in $O(n \lg n)$.

Randomized Quick Sort, Take 1

The expected running time of SimpleQuickSort on a uniform random permutation is in O(n lg n).

So why don't we just ensure the input is a uniform random permutation?

RandomPermutationQuickSort(A)

```
RandomPermute(A)
SimpleQuickSort(A, 1, n)
```

Randomized Quick Sort, Take 1

The expected running time of SimpleQuickSort on a uniform random permutation is in O(n lg n).

So why don't we just ensure the input is a uniform random permutation?

RandomPermutationQuickSort(A)

```
RandomPermute(A)
SimpleQuickSort(A, 1, n)
```

We can compute a uniform random permutation in O(n) time in the worst case.

Randomized Quick Sort, Take 1

The expected running time of SimpleQuickSort on a uniform random permutation is in O(n lg n).

So why don't we just ensure the input is a uniform random permutation?

RandomPermutationQuickSort(A)

```
RandomPermute(A)
SimpleQuickSort(A, 1, n)
```

We can compute a uniform random permutation in O(n) time in the worst case.

Corollary: The expected running time of RandomPermutationQuickSort is in O(n lg n).

Randomized Quick Sort, Take 2

The key to the analysis of SimpleQuickSort:

Randomized Quick Sort, Take 2

The key to the analysis of SimpleQuickSort:

If the input is a uniform random permutation, then any element is equally likely to be chosen as pivot.

Randomized Quick Sort, Take 2

The key to the analysis of SimpleQuickSort:

If the input is a uniform random permutation, then any element is equally likely to be chosen as pivot.

So why don't we make sure we choose a uniform random pivot, no matter the input permutation?

RandomPivotQuickSort(A, ℓ, r)

```
if r ≤ ℓ
    then return
p = RandomNumber(ℓ, r)
swap A[p] and A[r]
m = Partition(A, ℓ, r)
RandomPivotQuickSort(A, ℓ, m − 1)
RandomPivotQuickSort(A, m + 1, r)
```

Randomized Quick Sort, Take 2

The key to the analysis of SimpleQuickSort:

If the input is a uniform random permutation, then any element is equally likely to be chosen as pivot.

So why don't we make sure we choose a uniform random pivot, no matter the input permutation?

RandomPivotQuickSort(A, ℓ, r)

```
if r ≤ ℓ
   then return
p = RandomNumber(ℓ, r)
swap A[p] and A[r]
m = Partition(A, ℓ, r)
RandomPivotQuickSort(A, ℓ, m − 1)
RandomPivotQuickSort(A, m + 1, r)
```

Lemma: The expected running time of RandomPivotQuickSort is in $O(n \lg n)$.

The analysis is 100% identical to that of SimpleQuickSort!

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Uniform Random Permutation In Linear Time

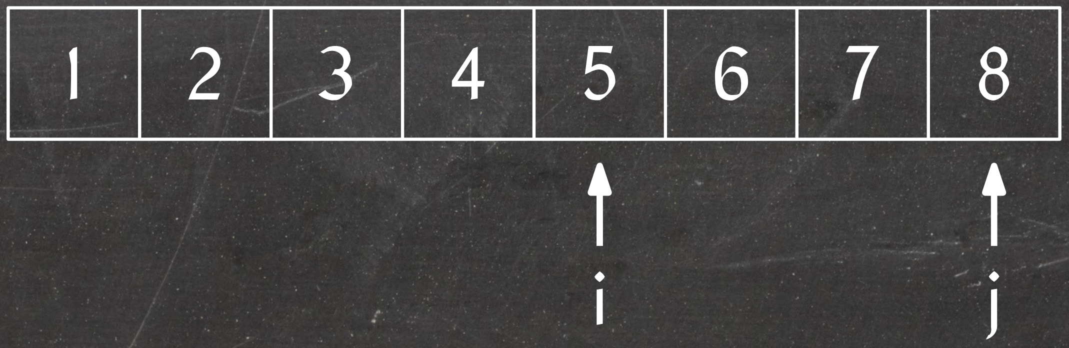
RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Uniform Random Permutation In Linear Time

RandomPermute(A)

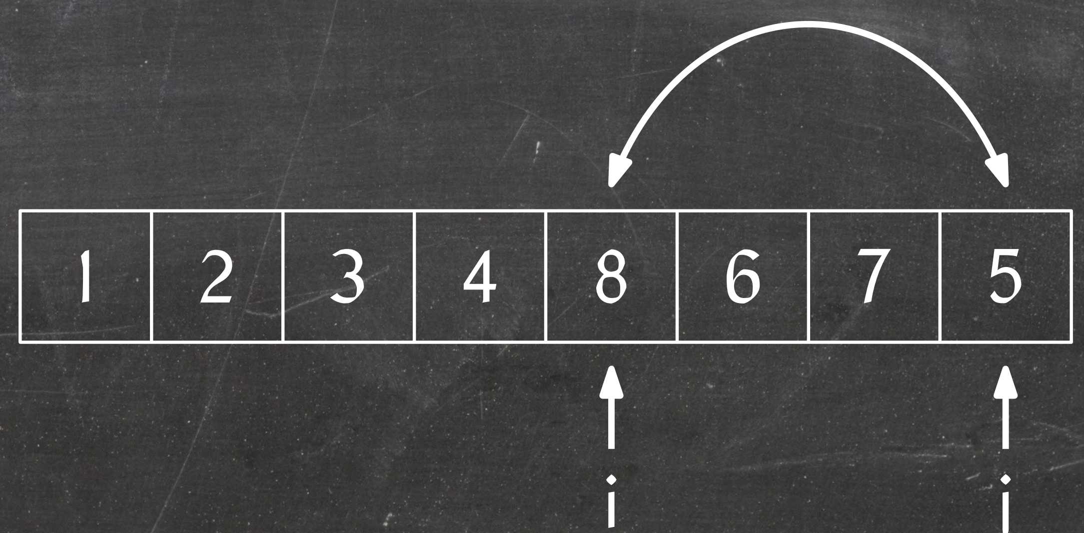
```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

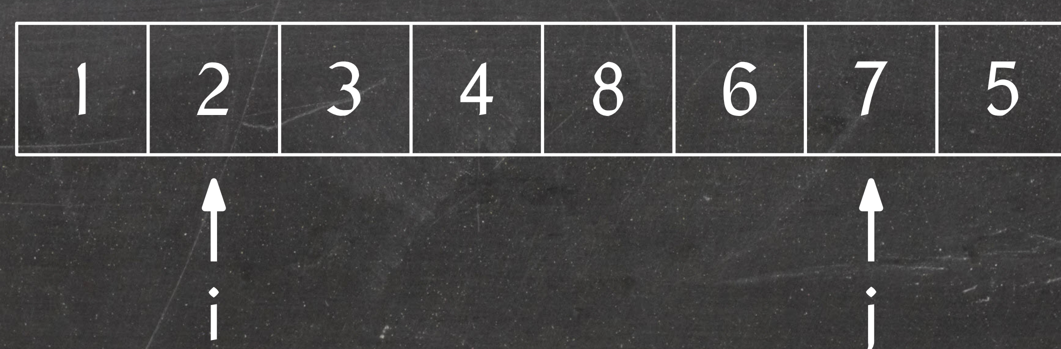
RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Uniform Random Permutation In Linear Time

RandomPermute(A)

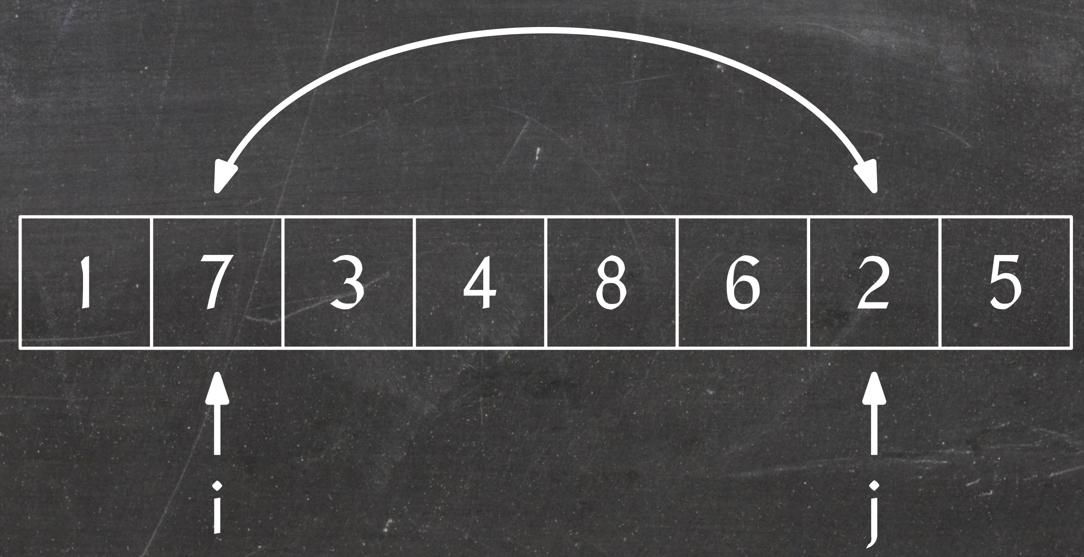
```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

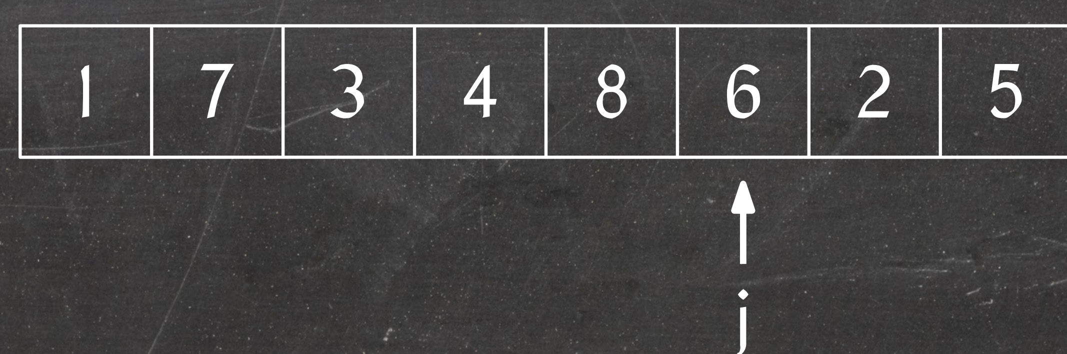
```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

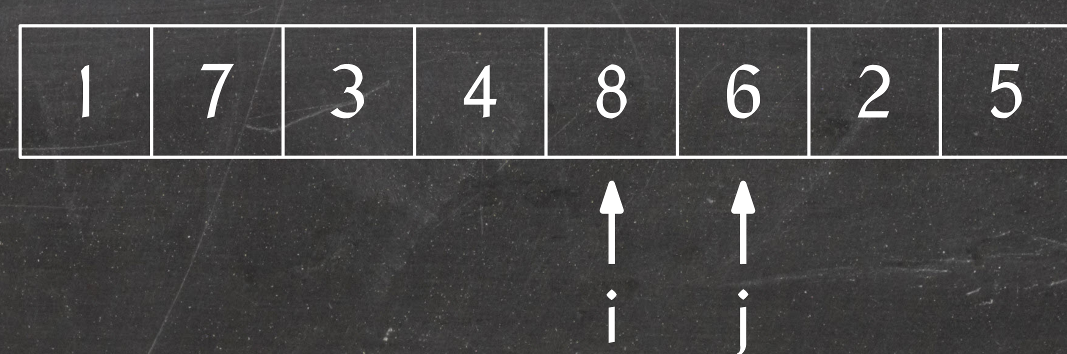
```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

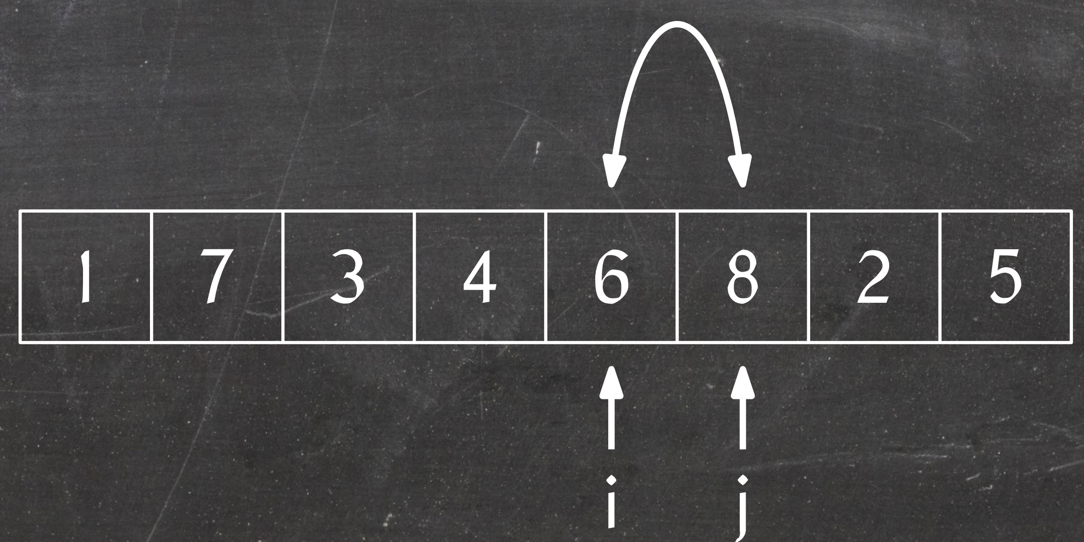
```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```



Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes O(n) time.

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes $O(n)$ time.

Lemma: RandomPermute produces each permutation of the input array A with probability $\frac{1}{n!}$.

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes $O(n)$ time.

Lemma: RandomPermute produces each permutation of the input array A with probability $\frac{1}{n!}$.

Induction on n.

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes $O(n)$ time.

Lemma: RandomPermute produces each permutation of the input array A with probability $\frac{1}{n!}$.

Induction on n.

If $n = 1$, then it produces the only possible permutation with probability $1 = \frac{1}{1!}$.

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes $O(n)$ time.

Lemma: RandomPermute produces each permutation of the input array A with probability $\frac{1}{n!}$.

If $n > 1$, then to produce the permutation $\langle x_1, x_2, \ldots, x_n \rangle$ (event E), we need to

- Place x_n into $A[n]$ (event E_1) and
- Place $x_1, x_2, \ldots, x_{n-1}$ into $A[1 \,..\, n-1]$ (event E_2).

Uniform Random Permutation In Linear Time

RandomPermute(A)

```
n = |A|
for j = n downto 2
    do i = RandomNumber(1, j)
        swap A[i] and A[j]
```

Observation: RandomPermute takes $O(n)$ time.

Lemma: RandomPermute produces each permutation of the input array A with probability $\frac{1}{n!}$.

If $n > 1$, then to produce the permutation $\langle x_1, x_2, \ldots, x_n \rangle$ (event E), we need to

- Place x_n into $A[n]$ (event E_1) and
- Place $x_1, x_2, \ldots, x_{n-1}$ into $A[1 \,..\, n-1]$ (event E_2).

So $P[E] = P[E_1 \cap E_2] = P[E_1] \cdot P[E_2 | E_1] = \frac{1}{n} \cdot \frac{1}{(n-1)!} = \frac{1}{n!}$.

Randomized Selection

recall select(k)
= find kth element from
unsorted array

RandomizedSelection(A, ℓ, r, k)

```
if r ≤ ℓ
    then return A[ℓ]
p = RandomNumber(ℓ, r)
swap A[p] and A[r]
m = Partition(A, ℓ, r)
if m − ℓ = k − 1
    then return A[m]
    else if m − ℓ ≥ k
            then RandomizedSelection(A, ℓ, m − 1, k)
            else RandomizedSelection(A, m + 1, r, k − (m + 1 − ℓ))
```

ideally want median

Randomized Selection

RandomizedSelection(A, ℓ, r, k)

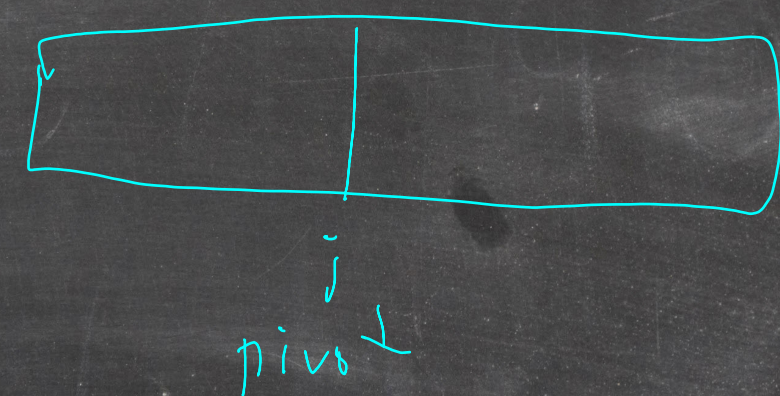
```
if r ≤ ℓ
   then return A[ℓ]
p = RandomNumber(ℓ, r)
swap A[p] and A[r]
m = Partition(A, ℓ, r)
if m − ℓ = k − 1
   then return A[m]
   else if m − ℓ ≥ k
           then RandomizedSelection(A, ℓ, m − 1, k)
           else RandomizedSelection(A, m + 1, r, k − (m + 1 − ℓ))
```

Lemma: The expected running time of RandomizedSelection is in $O(n)$.

Randomized Selection

Observation: If we choose the ith smallest element as pivot, then

$$E[T(n)] \leq O(n) + E[T(\max(n-i, i-1))].$$



Randomized Selection

Observation: If we choose the ith smallest element as pivot, then

$$E[T(n)] \leq O(n) + E[T(\max(n-i, i-1))].$$

Corollary: $E[T(n)] \leq O(n) + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(n-i, i-1))].$

Randomized Selection

Observation: If we choose the ith smallest element as pivot, then

$$E[T(n)] \leq O(n) + E[T(\max(n-i, i-1))].$$

Corollary: $E[T(n)] \leq O(n) + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(n-i, i-1))].$

Claim: $E[T(n)] \leq cn$, for some $c > 0$.

Randomized Selection

Observation: If we choose the ith smallest element as pivot, then

$$E[T(n)] \leq O(n) + E[T(\max(n-i, i-1))].$$

Corollary: $E[T(n)] \leq O(n) + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(n-i, i-1))].$

Claim: $E[T(n)] \leq cn$, for some $c > 0$.

Base case: $1 \leq n < 4$.

$T(n) \leq c \leq cn$.

Randomized Selection

Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n} \sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

Randomized Selection

Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

Randomized Selection

Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci$$

Randomized Selection

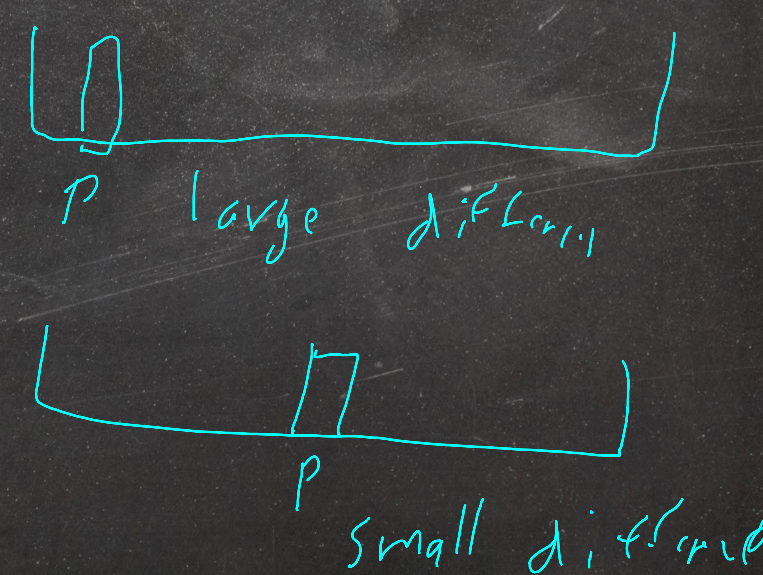
Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n} \sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

$$\leq an + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

$$\leq an + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} ci$$

$$= an + \frac{2c}{n} \left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor n/2 \rfloor - 1} i \right)$$



Randomized Selection

Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci$$

$$= an + \frac{2c}{n}\left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor n/2 \rfloor - 1} i\right)$$

$$= an + \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor(\lfloor n/2 \rfloor - 1)}{2}\right)$$

Randomized Selection

Inductive step: $n \geq 4$.

$$E[T(n)] \leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

$$\leq an + \frac{2}{n}\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci$$

$$= an + \frac{2c}{n}\left(\sum_{i=1}^{n-1} i - \sum_{i=1}^{\lfloor n/2 \rfloor - 1} i\right)$$

$$= an + \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor(\lfloor n/2 \rfloor - 1)}{2}\right)$$

$$\leq an + \frac{c}{n}\left[n(n-1) - \left(\frac{n}{2} - 1\right)\left(\frac{n}{2} - 2\right)\right]$$

Randomized Selection

Inductive step: $n \geq 4$.

$$
\begin{aligned}
E[T(n)] &\leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))] \\
&\leq an + \frac{c}{n}\left[n(n-1) - \left(\frac{n}{2}-1\right)\left(\frac{n}{2}-2\right)\right]
\end{aligned}
$$

Randomized Selection

Inductive step: $n \geq 4$.

$$
\begin{aligned}
E[T(n)] &\leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))] \\
&\leq an + \frac{c}{n}\left[n(n-1) - \left(\frac{n}{2}-1\right)\left(\frac{n}{2}-2\right)\right] \\
&= an + \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2}\right)
\end{aligned}
$$

Randomized Selection

Inductive step: $n \geq 4$.

$$
\begin{aligned}
E[T(n)] &\leq an + \frac{1}{n} \sum_{i=1}^{n} E[T(\max(i-1, n-i))] \\
&\leq an + \frac{c}{n} \left[n(n-1) - \left(\frac{n}{2} - 1\right)\left(\frac{n}{2} - 2\right) \right] \\
&= an + \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} \right) \\
&= \left(a + \frac{3c}{4} + \frac{c}{2n} \right) n
\end{aligned}
$$

Randomized Selection

Inductive step: $n \geq 4$.

$$
\begin{aligned}
E[T(n)] &\leq an + \frac{1}{n}\sum_{i=1}^{n} E[T(\max(i-1, n-i))] \\
&\leq an + \frac{c}{n}\left[n(n-1) - \left(\frac{n}{2}-1\right)\left(\frac{n}{2}-2\right)\right] \\
&= an + \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2}\right) \\
&= \left(a + \frac{3c}{4} + \frac{c}{2n}\right)n \\
&\leq cn \quad \forall c \geq 8a.
\end{aligned}
$$

Sorting in Linear Time?

Using comparisons only, as Insertion Sort, Merge Sort, Quick Sort do, it is impossible to sort faster than in $\Omega(n \lg n)$ time.

Sorting in Linear Time?

Using comparisons only, as Insertion Sort, Merge Sort, Quick Sort do, it is impossible to sort faster than in $\Omega(n \lg n)$ time.

By exploiting assumptions about the input and using element values in the algorithm, we can do better:

Sorting in Linear Time?

Using comparisons only, as Insertion Sort, Merge Sort, Quick Sort do, it is impossible to sort faster than in $\Omega(n \lg n)$ time.

By exploiting assumptions about the input and using element values in the algorithm, we can do better:

Counting sort: Sorts n integers between 1 and n in $O(n)$ time.

Sorting in Linear Time?

Using comparisons only, as Insertion Sort, Merge Sort, Quick Sort do, it is impossible to sort faster than in $\Omega(n \lg n)$ time.

By exploiting assumptions about the input and using element values in the algorithm, we can do better:

Counting sort: Sorts n integers between 1 and n in $O(n)$ time.

Radix sort: Sorts n integers between 1 and n^c in $O(cn)$ time. This is $O(n)$ if c is a constant.

Sorting in Linear Time?

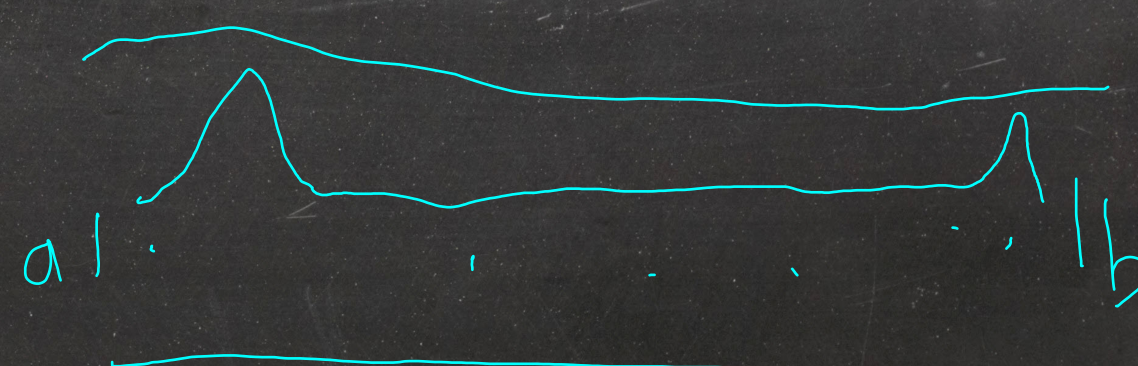
Using comparisons only, as Insertion Sort, Merge Sort, Quick Sort do, it is impossible to sort faster than in $\Omega(n \lg n)$ time.

By exploiting assumptions about the input and using element values in the algorithm, we can do better:

Counting sort: Sorts n integers between 1 and n in $O(n)$ time.

Radix sort: Sorts n integers between 1 and n^c in $O(cn)$ time. This is $O(n)$ if c is a constant.

Bucket sort: Sorts n real numbers drawn uniformly at random from an interval $[a, b)$ in expected linear time.



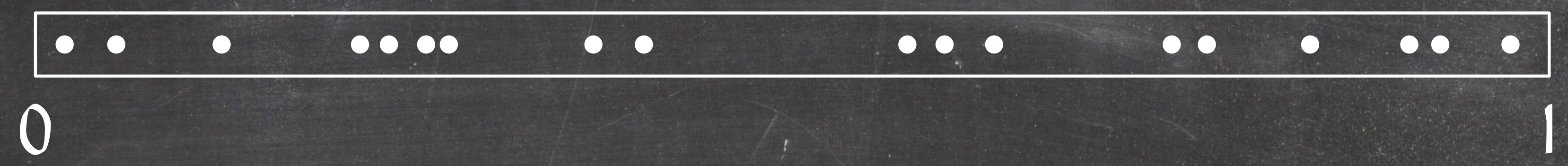
Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval [a, b).

Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval [a, b).

We can normalize this to the interval [0, 1).

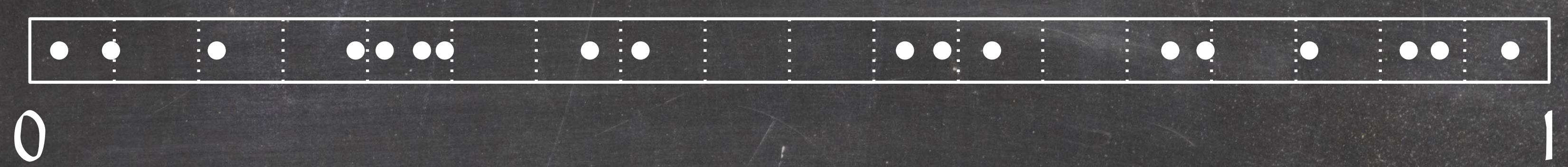


Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval [a, b).

We can normalize this to the interval [0, 1). → *by subtracting a, divide by (b − a)*

Divide [0, 1) into subintervals of length $\frac{1}{n}$.

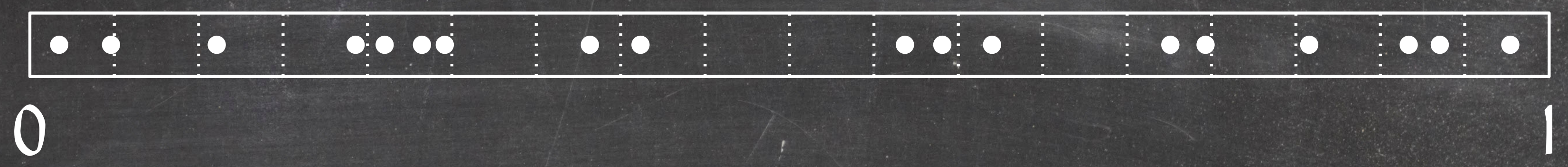


Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval [a, b).

We can normalize this to the interval [0, 1).

Divide [0, 1) into subintervals of length $\frac{1}{n}$.



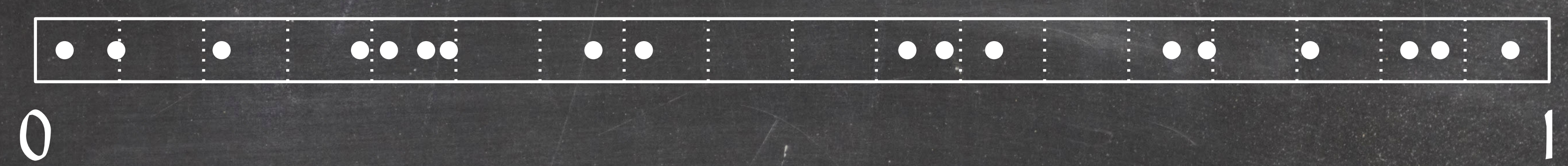
How many elements do we expect to end up in each subinterval?

Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval [a, b).

We can normalize this to the interval [0, 1).

Divide [0, 1) into subintervals of length $\frac{1}{n}$.



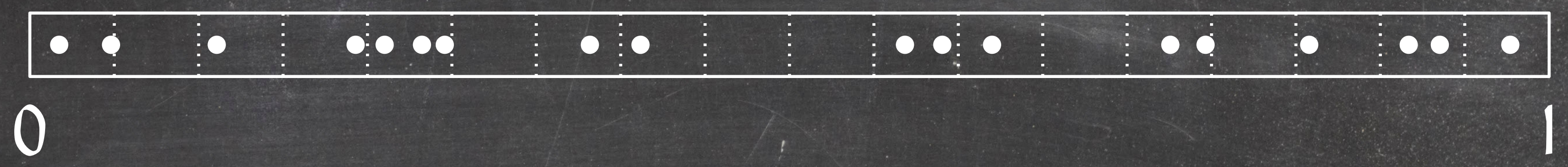
How many elements do we expect to end up in each subinterval? 1!

Bucket Sort

Assume the inputs are real numbers drawn uniformly at random from some interval $[a, b)$.

We can normalize this to the interval $[0, 1)$.

Divide $[0, 1)$ into subintervals of length $\frac{1}{n}$.



How many elements do we expect to end up in each subinterval? 1!

$\Rightarrow$ Strategy:

- Bucket items according to the subinterval they belong to.
- Sort each bucket, hopefully in constant time.
- Concatenate the sorted buckets.

Bucket Sort

preprocess to range 0,1 before

BucketSort(A)

```
n = |A|
B = an array of n empty singly-linked lists
for i = 1 to n
    do prepend A[i] to list B[1 + ⌊n · A[i]⌋]
for i = 1 to n
    do InsertionSort(B[i])
j = 0
for i = 1 to n
    do for every element x ∈ B[i]
            do A[j] = x
               j = j + 1
```

1-indexing

Bucket Sort

BucketSort(A)

```
n = |A|
B = an array of n empty singly-linked lists
for i = 1 to n
    do prepend A[i] to list B[1 + ⌊n · A[i]⌋]
for i = 1 to n
    do InsertionSort(B[i])
j = 0
for i = 1 to n
    do for every element x ∈ B[i]
            do A[j] = x
               j = j + 1
```

This is where we depart from using comparisons only!

Bucket Sort

BucketSort(A)

```
n = |A|
B = an array of n empty singly-linked lists
for i = 1 to n
    do prepend A[i] to list B[1 + ⌊n · A[i]⌋]
for i = 1 to n
    do InsertionSort(B[i])
j = 0
for i = 1 to n
    do for every element x ∈ B[i]
            do A[j] = x
               j = j + 1
```

This is where we depart from using comparisons only!

Worst-case running time: $O(n^2)$

Bucket Sort

BucketSort(A)

```
n = |A|
B = an array of n empty singly-linked lists
for i = 1 to n
    do prepend A[i] to list B[1 + ⌊n · A[i]⌋]
for i = 1 to n
    do InsertionSort(B[i])
j = 0
for i = 1 to n
    do for every element x ∈ B[i]
            do A[j] = x
               j = j + 1
```

This is where we depart from using comparisons only!

Why not Merge Sort?

Worst-case running time: $O(n^2)$

Bucket Sort

BucketSort(A)

```
n = |A|
B = an array of n empty singly-linked lists
for i = 1 to n
    do prepend A[i] to list B[1 + ⌊n · A[i]⌋]
for i = 1 to n
    do InsertionSort(B[i])
j = 0
for i = 1 to n
    do for every element x ∈ B[i]
            do A[j] = x
               j = j + 1
```

This is where we depart from using comparisons only!

Why not Merge Sort?

It only helps in the worst case.

It's more complicated.

It actually hurts when buckets are small, which is what we expect.

Worst-case running time: $O(n^2)$

Bucket Sort

Running time: $T(n) \in O\left(n + \sum_{i=1}^{n} n_i^2\right)$

n_i = the number of elements in B[i]

Bucket Sort

Running time: $T(n) \in O\left(n + \sum_{i=1}^{n} n_i^2\right)$ ← size of buckets

n_i = the number of elements in B[i]

expected size of buckets

$$E[T(n)] \in O\left(n + \sum_{i=1}^{n} E[n_i]^2\right)$$

Bucket Sort

Running time: $T(n) \in O\left(n + \sum_{i=1}^{n} n_i^2\right)$

n_i = the number of elements in B[i]

$$E[T(n)] \in O\left(n + \sum_{i=1}^{n} E[n_i]^2\right)$$

Lemma: $E[n_i^2] < 2$.

Bucket Sort

Running time: $T(n) \in O\left(n + \sum_{i=1}^{n} n_i^2\right)$

n_i = the number of elements in B[i]

$$E[T(n)] \in O\left(n + \sum_{i=1}^{n} E[n_i]^2\right)$$

Lemma: $E[n_i^2] < 2$.

Corollary: $E[T(n)] \in O(n)$.

$\leq c_1 n + c_2 2 n$

$\leq c_3 (n + 2n)$

$\leq c_3 5n = O(n)$

Bucket Sort

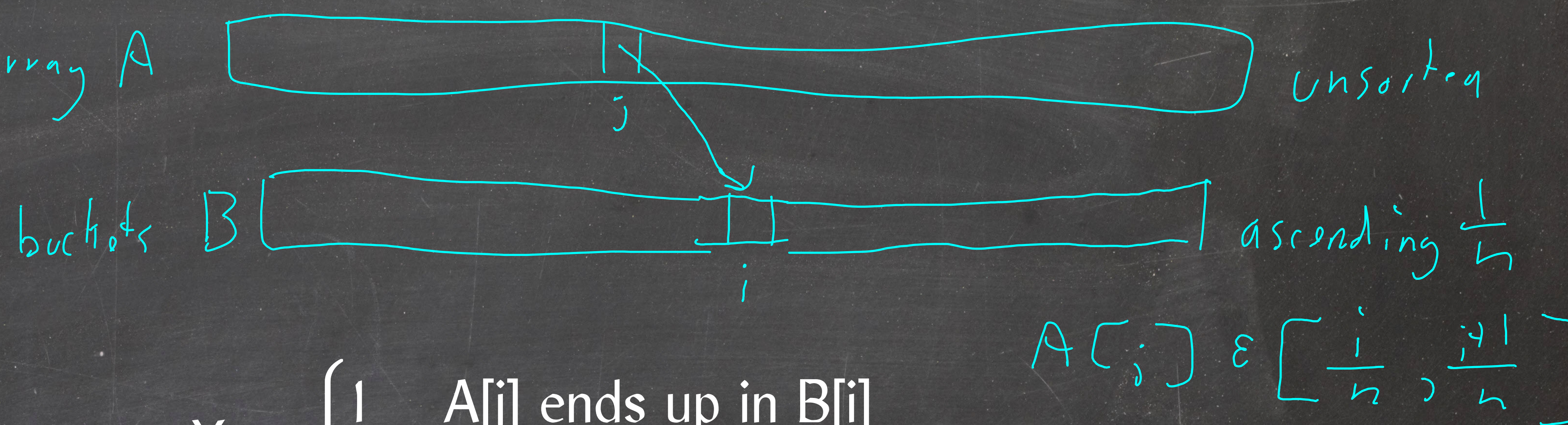
Lemma: $E[n_i^2] < 2$.

Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & \text{A[j] ends up in B[i]} \\ 0 & \text{otherwise} \end{cases}$$

Bucket Sort

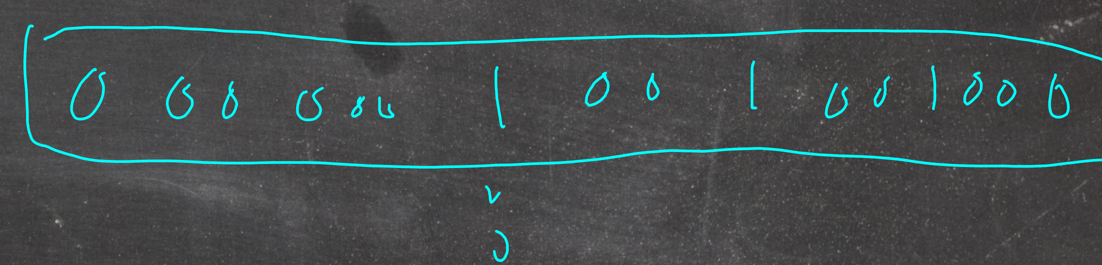


Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & A[j] \text{ ends up in } B[i] \\ 0 & \text{otherwise} \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_j$$

just for bucket i



Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & A[j] \text{ ends up in } B[i] \\ 0 & \text{otherwise} \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_j$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_j\right)^2\right]$$

Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & A[j] \text{ ends up in } B[i] \\ 0 & \text{otherwise} \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_j$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_j\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_j X_k\right]$$

Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & \text{A[j] ends up in B[i]} \\ 0 & \text{otherwise} \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_j$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_j\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_j X_k\right] = \sum_{j=1}^{n}\sum_{k=1}^{n} E[X_j X_k]$$

Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & A[j] \text{ ends up in } B[i] \\ 0 & \text{otherwise} \end{cases}$$

$$n_i = \sum_{j=1}^{n} X_j$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_j\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_j X_k\right] = \sum_{j=1}^{n}\sum_{k=1}^{n} E[X_j X_k]$$

$$= \sum_{j=1}^{n} E[X_j^2] + \sum_{j=1}^{n}\sum_{\substack{k=1 \\ k\neq j}}^{n} E[X_j]E[X_k]$$

Bucket Sort

Lemma: $E[n_i^2] < 2$.

$$X_j = \begin{cases} 1 & A[j] \text{ ends up in } B[i] \\ 0 & \text{otherwise} \end{cases}$$

$$(x_1 + x_2 + x_3 \ldots x_n)^2$$

$$n_i = \sum_{j=1}^{n} X_j$$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_j\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_j X_k\right] = \sum_{j=1}^{n}\sum_{k=1}^{n} E[X_j X_k]$$

↑ more expectations

$$= \sum_{j=1}^{n} E[X_j^2] + \sum_{j=1}^{n}\sum_{\substack{k=1 \\ k \neq j}}^{n} E[X_j]E[X_k]$$

$x_1 x_1$ $x_1 x_2$

X_j and X_j are clearly not independent. X_j and X_k are independent.

Bucket Sort

$$E[X_j] = \frac{1}{n} \cdot 1 + \left(1 - \frac{1}{n}\right) \cdot 0 = \frac{1}{n}$$

Bucket Sort

$$E[X_j] = \frac{1}{n} \cdot 1 + \left(1 - \frac{1}{n}\right) \cdot 0 = \frac{1}{n}$$

$$E[X_j^2] = \frac{1}{n} \cdot 1^2 + \left(1 - \frac{1}{n}\right) \cdot 0^2 = \frac{1}{n}$$

Bucket Sort

$$E[X_j] = \frac{1}{n} \cdot 1 + \left(1 - \frac{1}{n}\right) \cdot 0 = \frac{1}{n}$$

$$E[X_j^2] = \frac{1}{n} \cdot 1^2 + \left(1 - \frac{1}{n}\right) \cdot 0^2 = \frac{1}{n}$$

$$E[n_i^2] = \sum_{j=1}^{n} E[X_j^2] + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} E[X_j]E[X_k] = n \cdot \frac{1}{n} + \frac{n(n-1)}{n^2} < 2$$

same

different

$1 + <1$

<2

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

No!

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

No!

For Quick Sort, we relied on a random ordering of the elements.

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

No!

For Quick Sort, we relied on a random ordering of the elements.

Randomly permuting the input to guarantee this does not affect the final result of the algorithm.

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

No!

For Quick Sort, we relied on a random ordering of the elements.

Randomly permuting the input to guarantee this does not affect the final result of the algorithm.

Bucket Sort relies on the random distribution of the input values.

Randomized Bucket Sort?

For Quick Sort, we were able to eliminate assumptions about the input distribution using randomization.

Does that work for Bucket Sort?

No!

For Quick Sort, we relied on a random ordering of the elements.

Randomly permuting the input to guarantee this does not affect the final result of the algorithm.

Bucket Sort relies on the random distribution of the input values.

We can't simply change them without changing the algorithm's output.

Game Tree Evaluation

Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

Game Tree Evaluation

Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Game Tree Evaluation

Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

Game Tree Evaluation

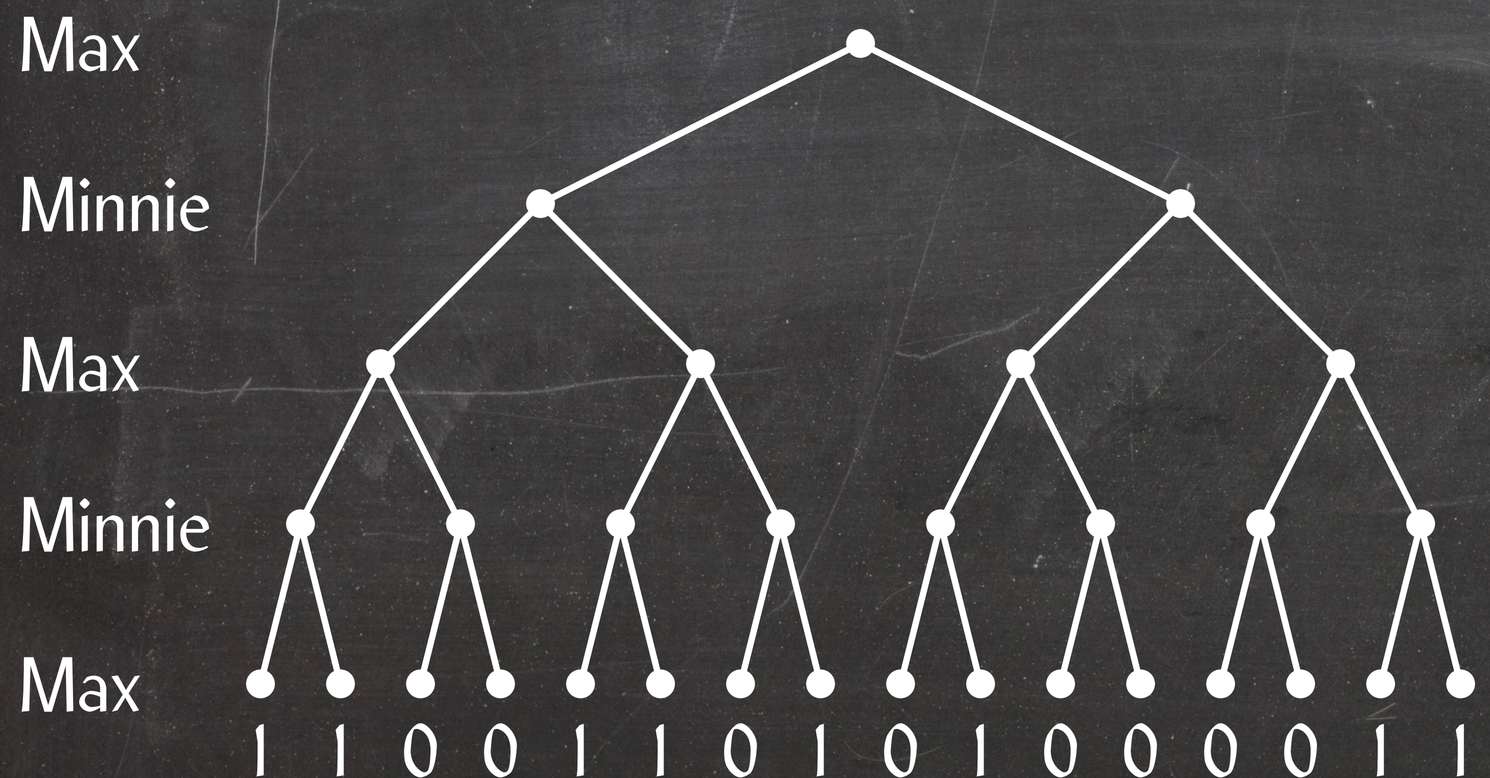
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

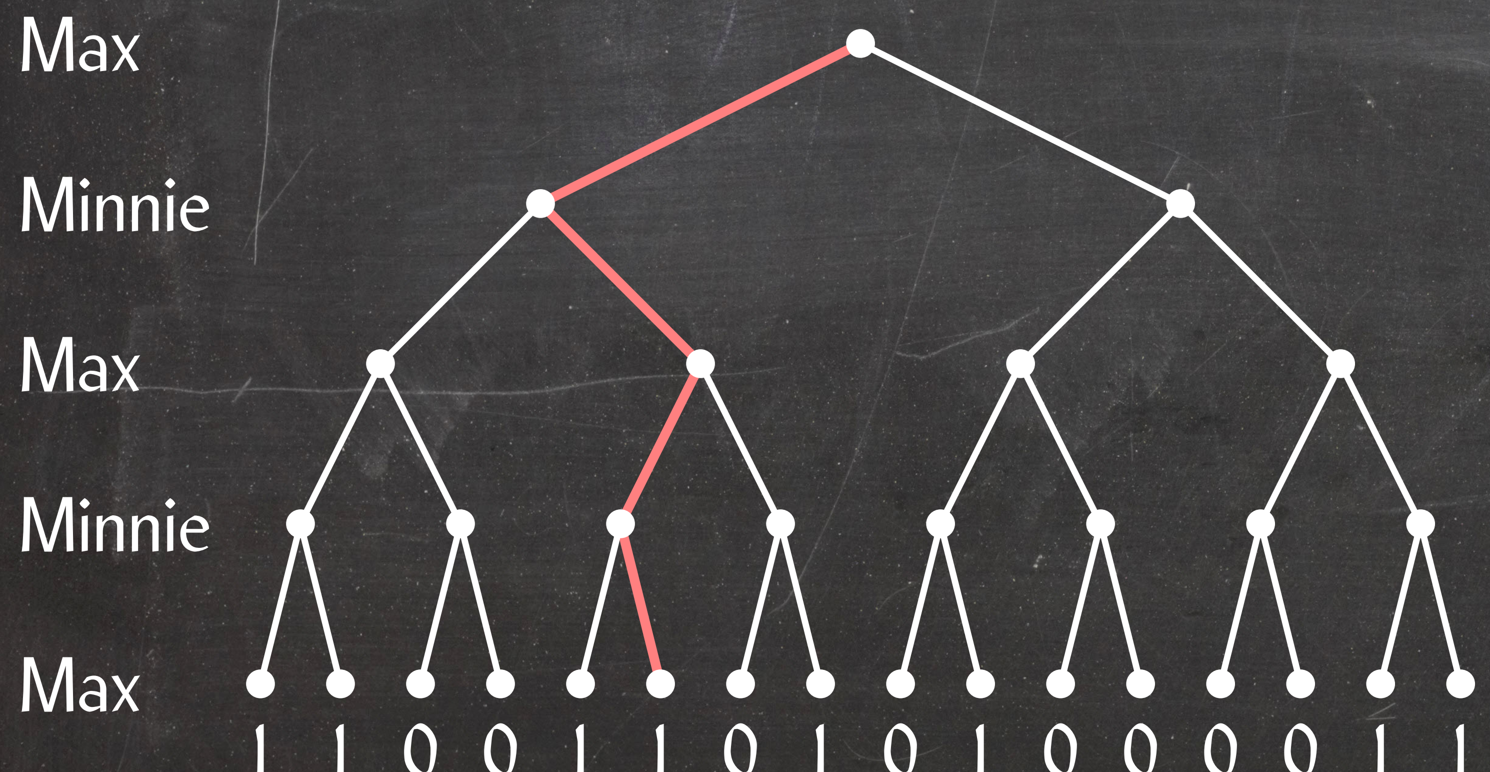
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

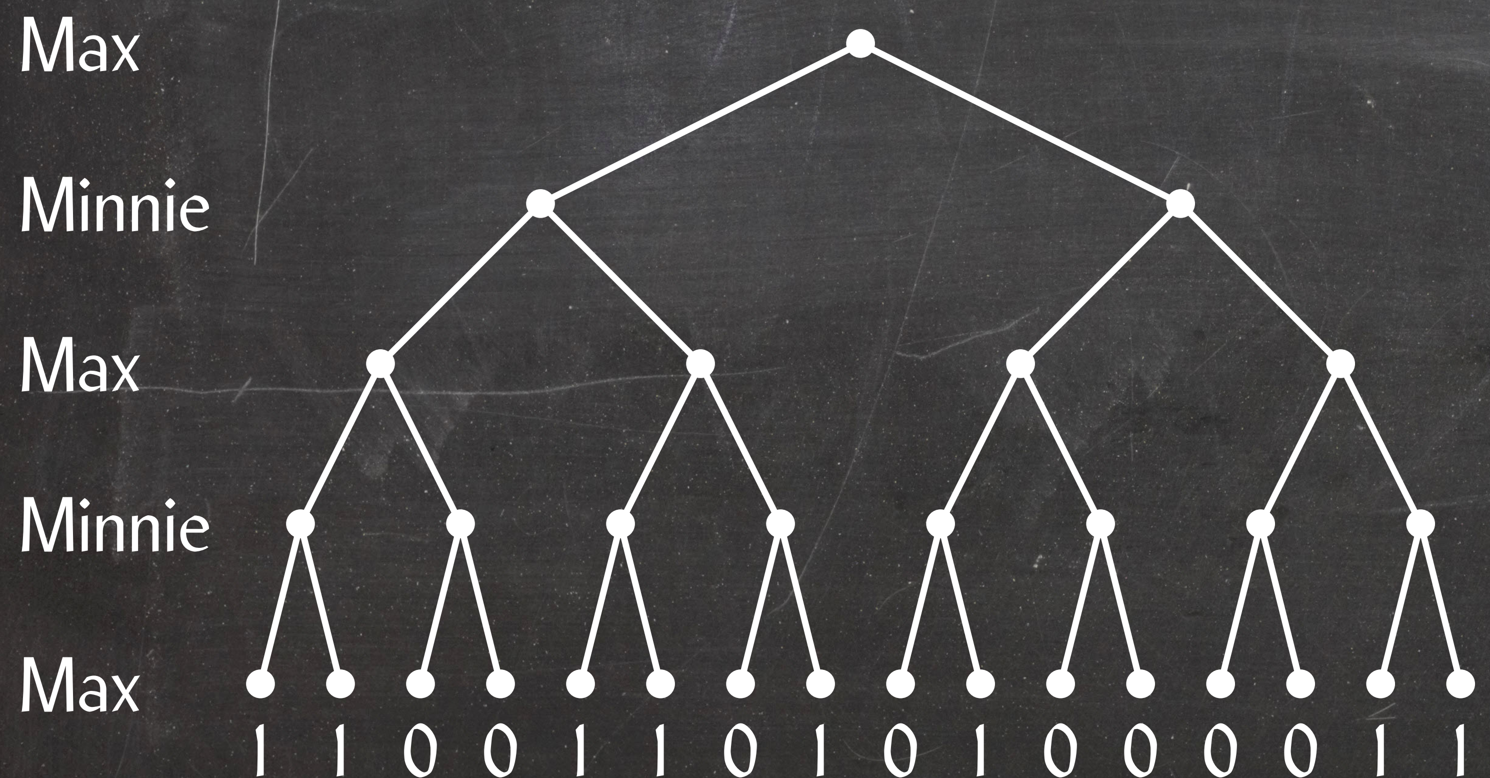
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

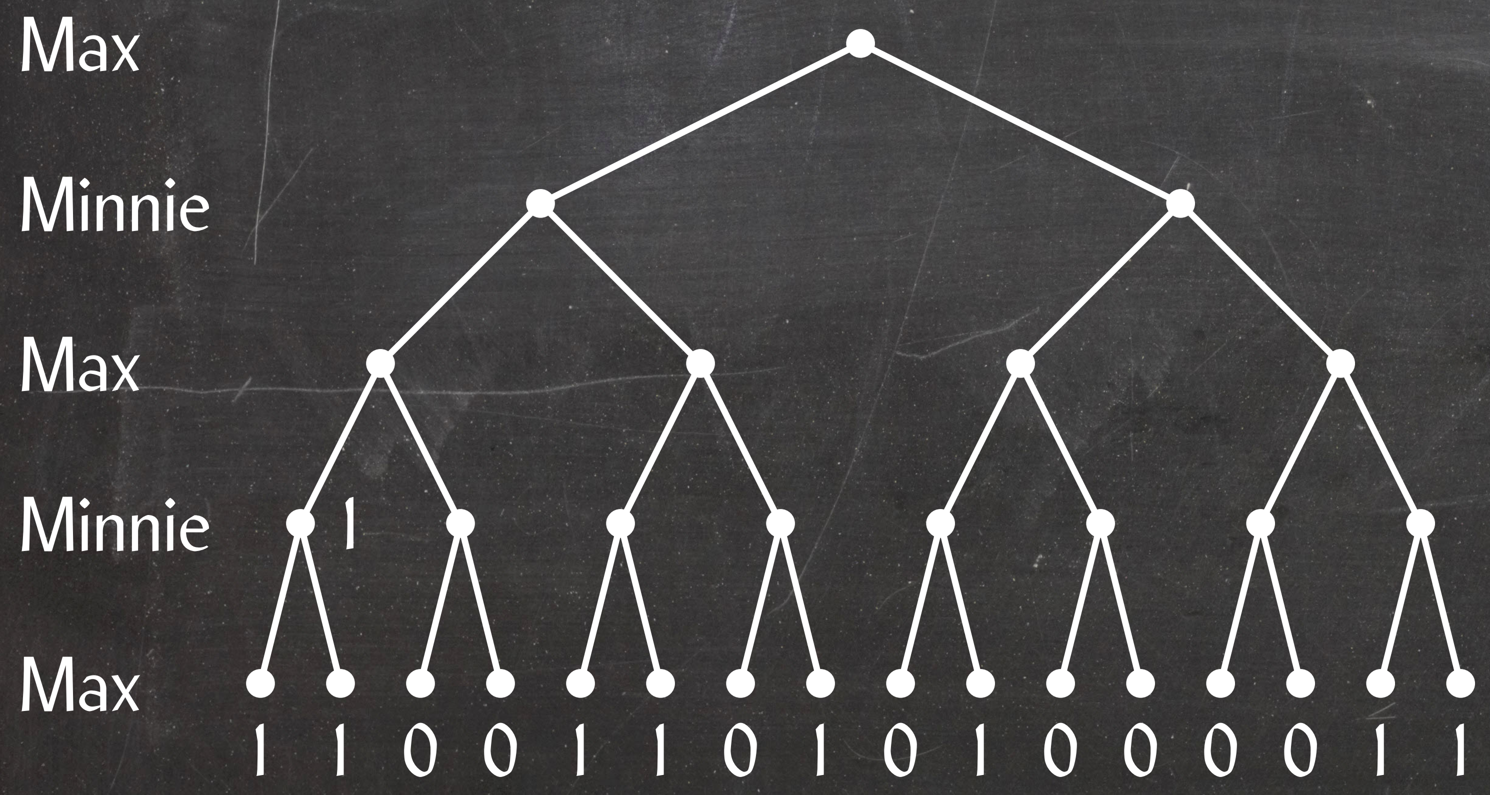
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

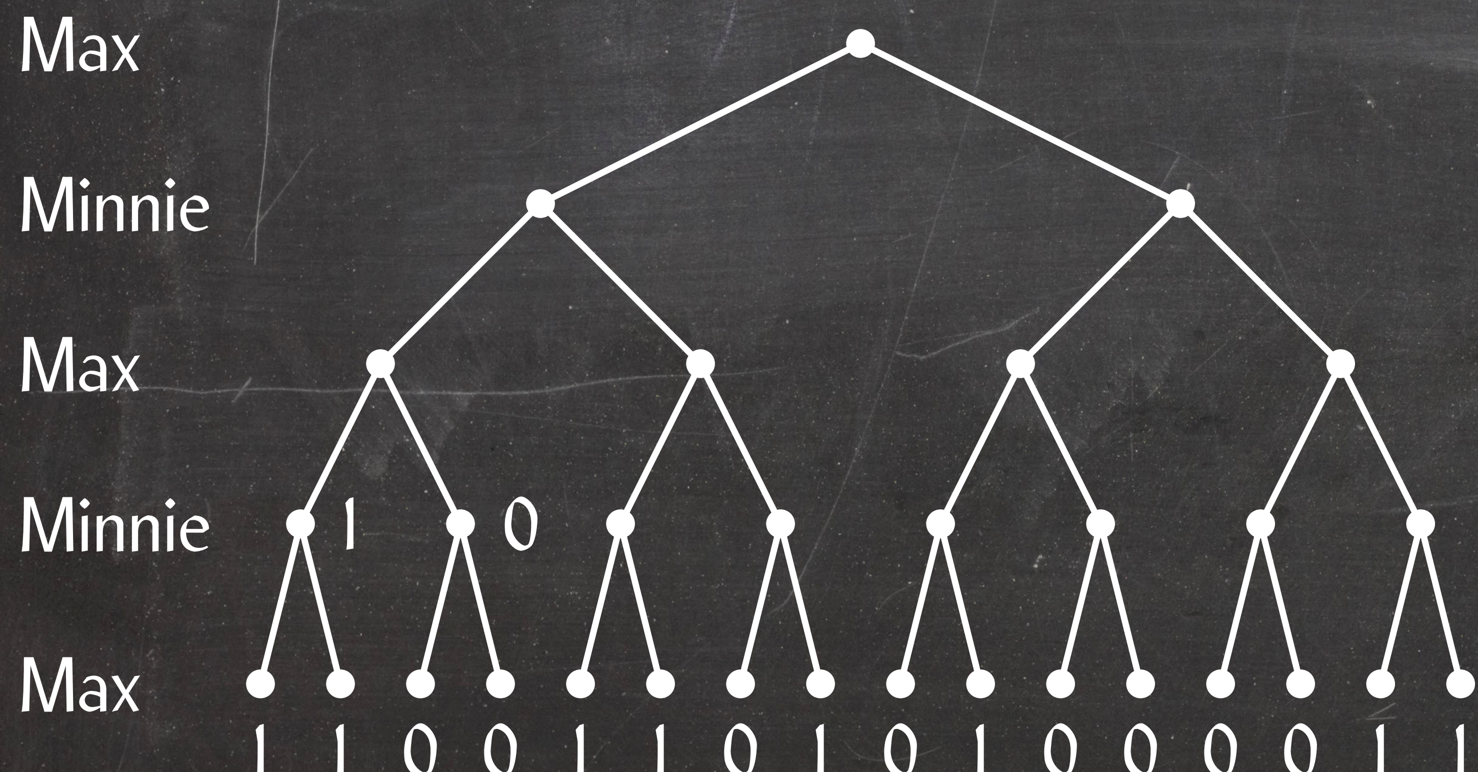
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

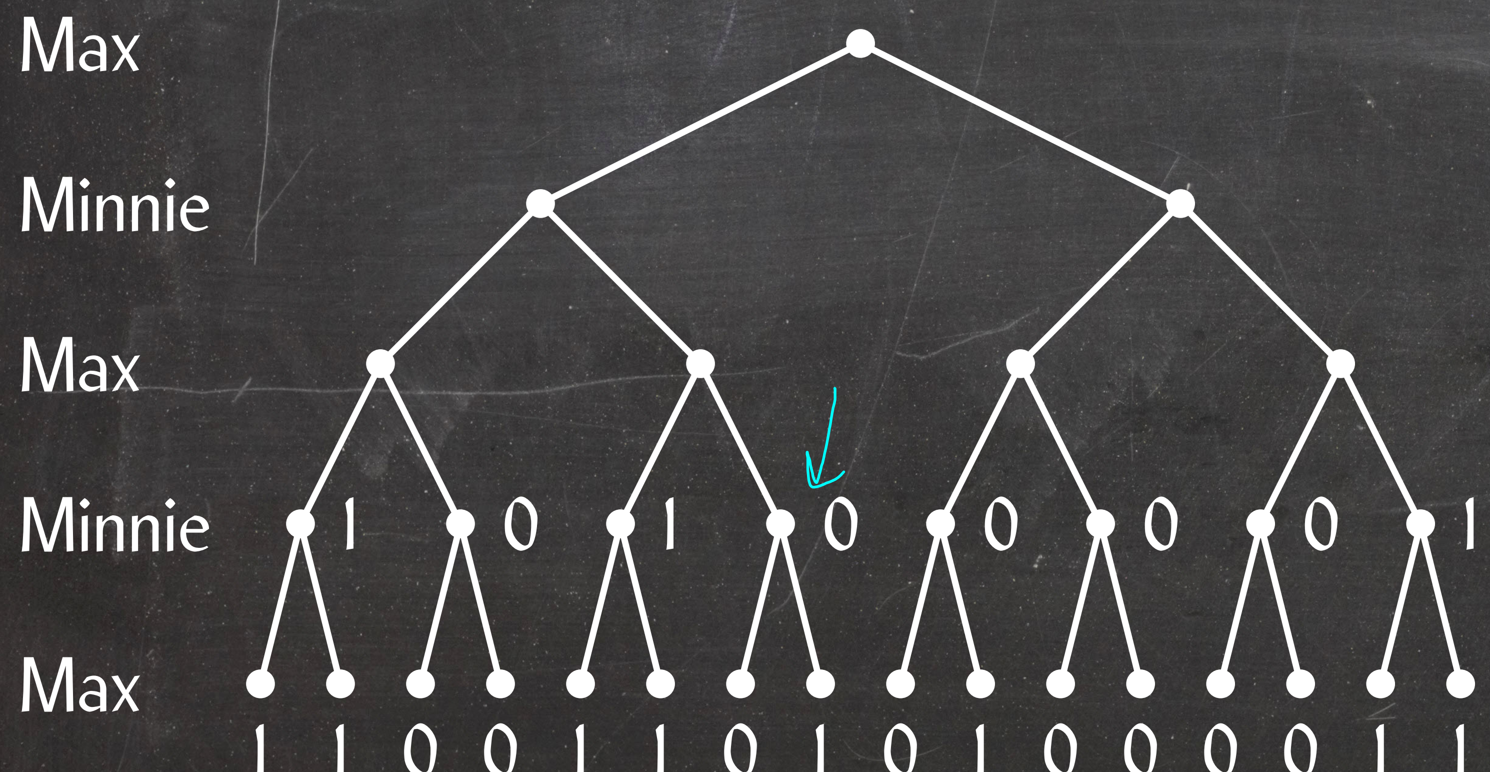
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

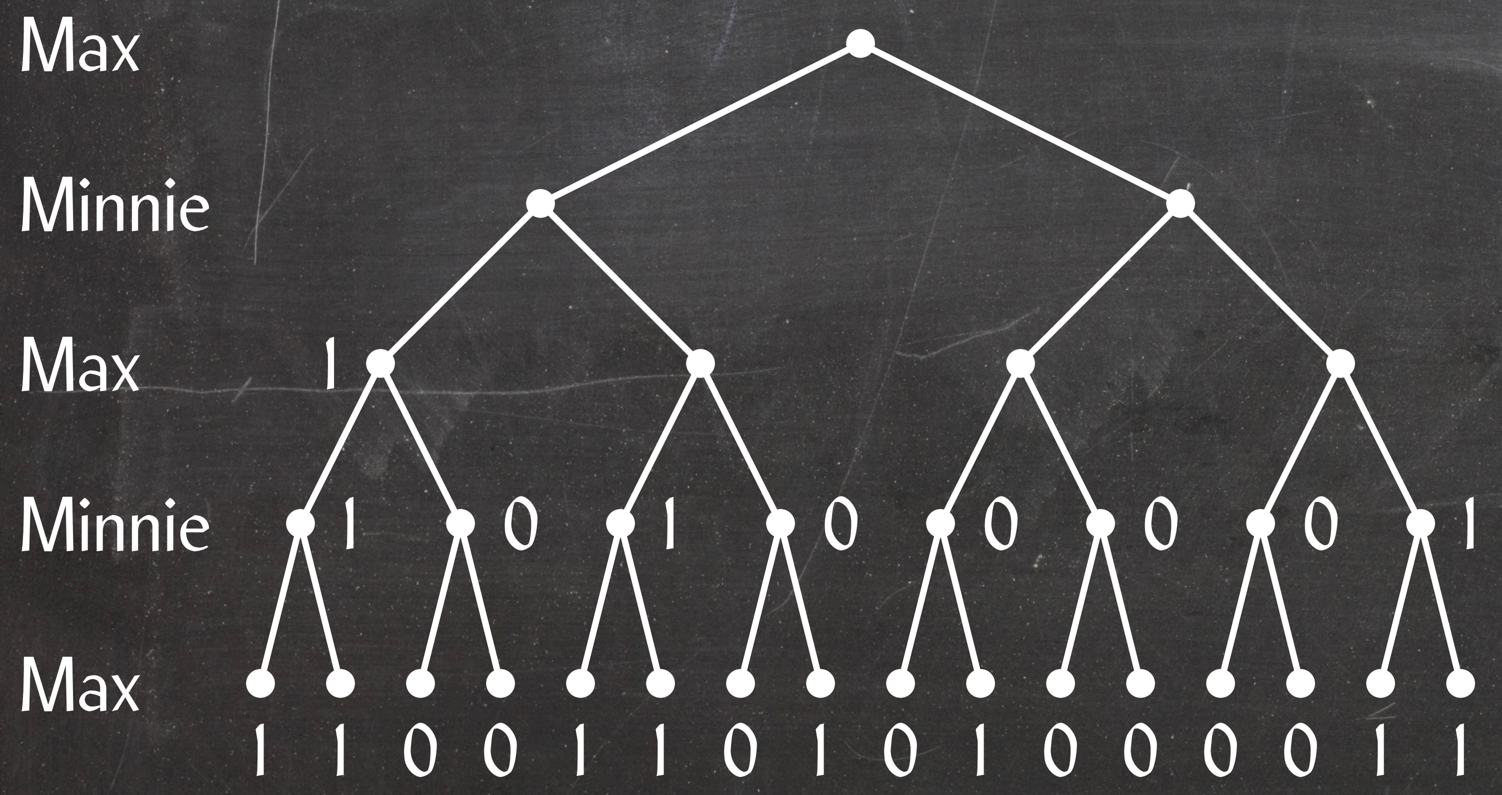
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

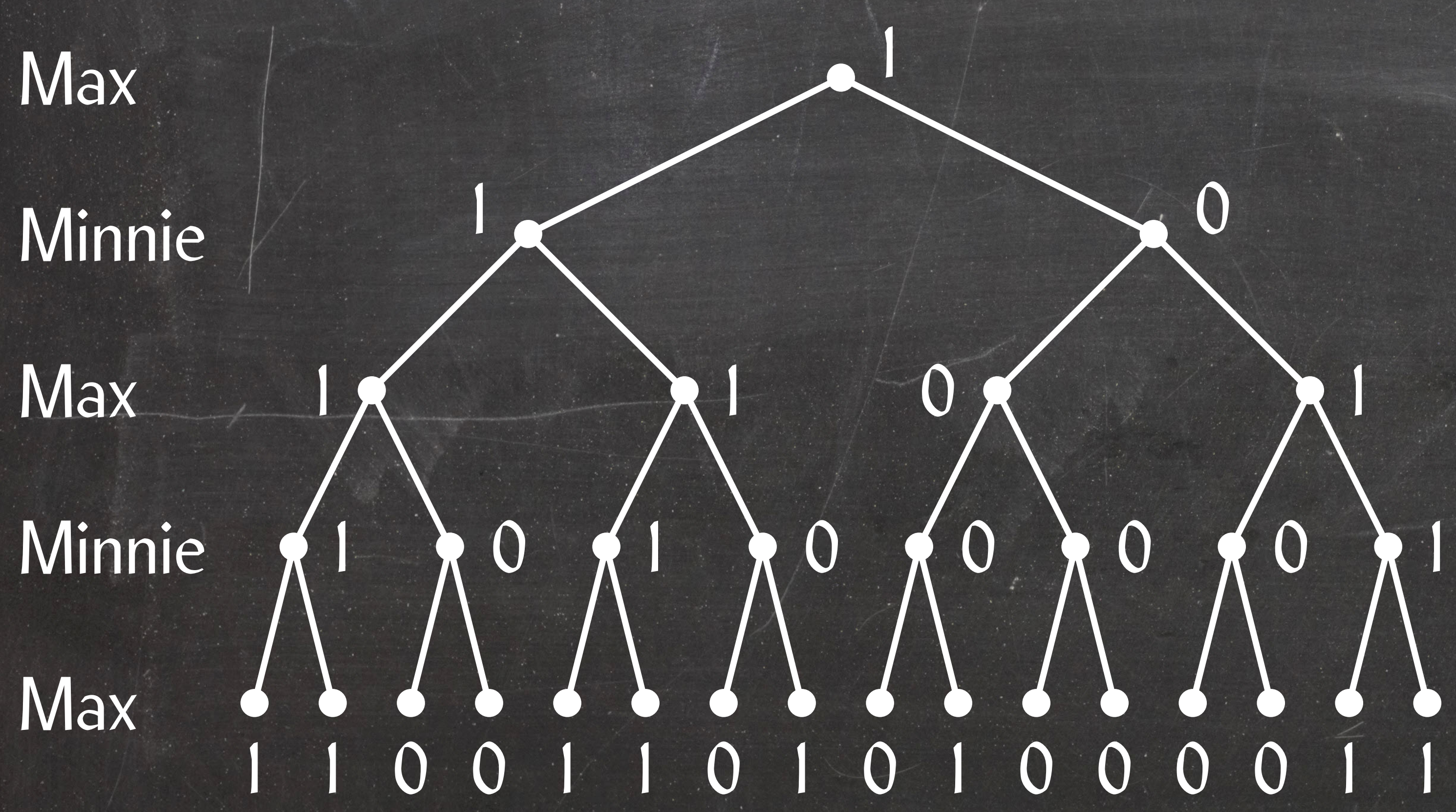
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

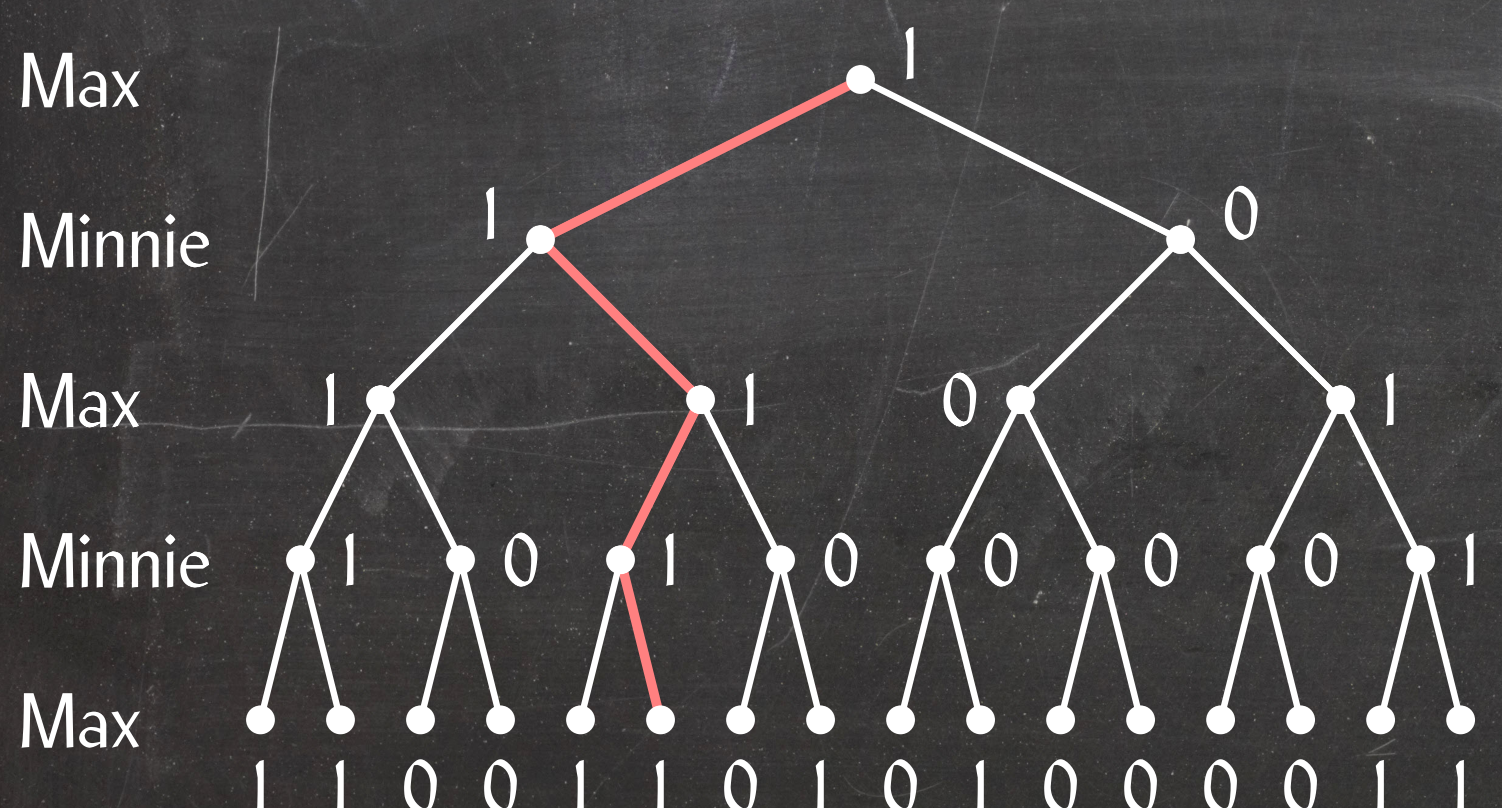
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Game Tree Evaluation

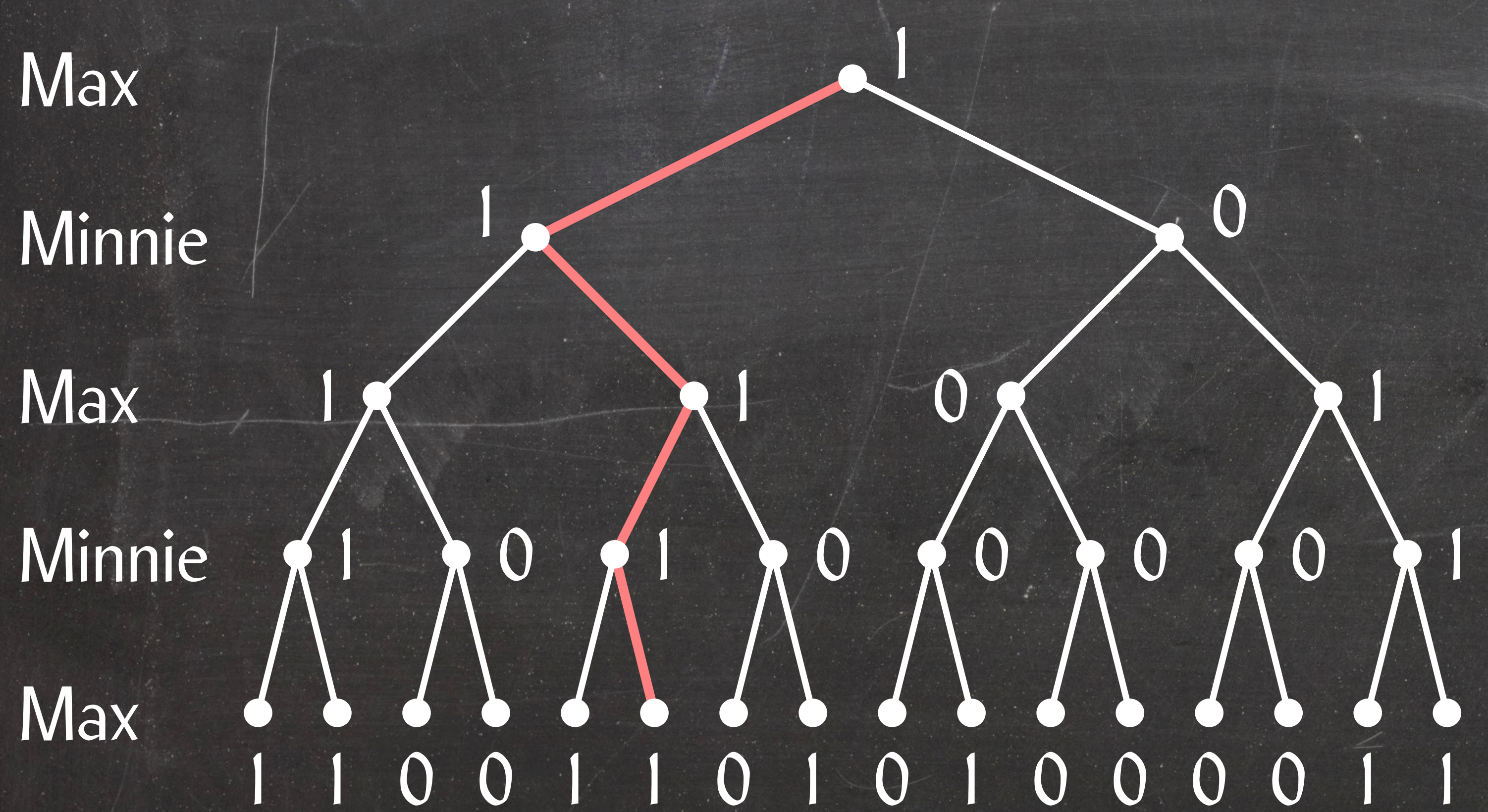
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Max-node:

$$\text{label}(v) = \max_{\text{child } w} \text{label}(w)$$

Minnie-node:

$$\text{label}(v) = \min_{\text{child } w} \text{label}(w)$$

Game Tree Evaluation

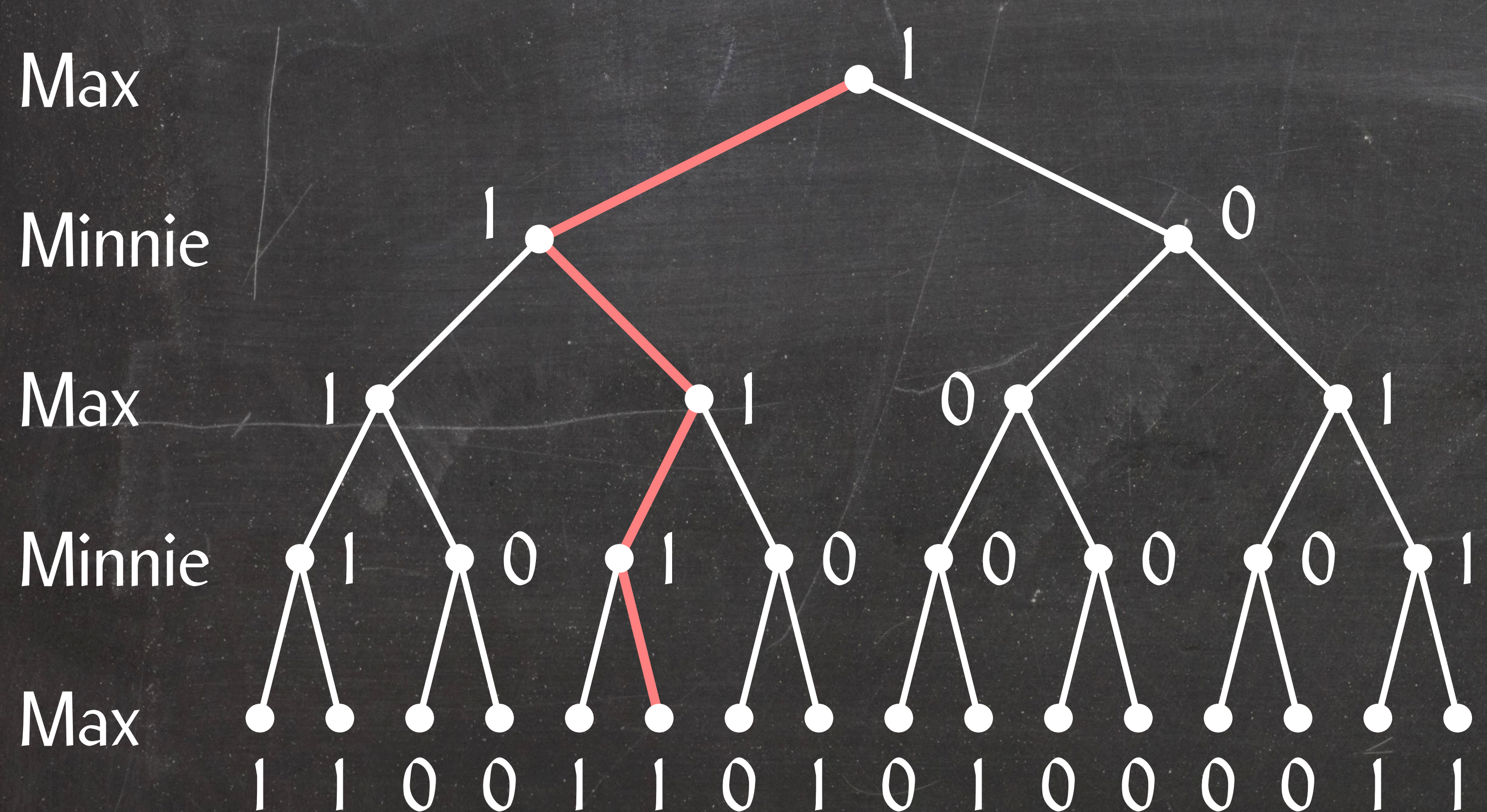
Motwani/Raghavan.
Randomized Algorithms. Section 2.1.

Consider a game where two players, Max and Minnie, take turns. Assume the game cannot end in a draw.

We label a win for Max with 1 and a win for Minnie with 0.

Max (Minnie) has a winning strategy if he can win the game no matter how Minnie (Max) plays.

We can model all possible games as a game tree:



Max-node:

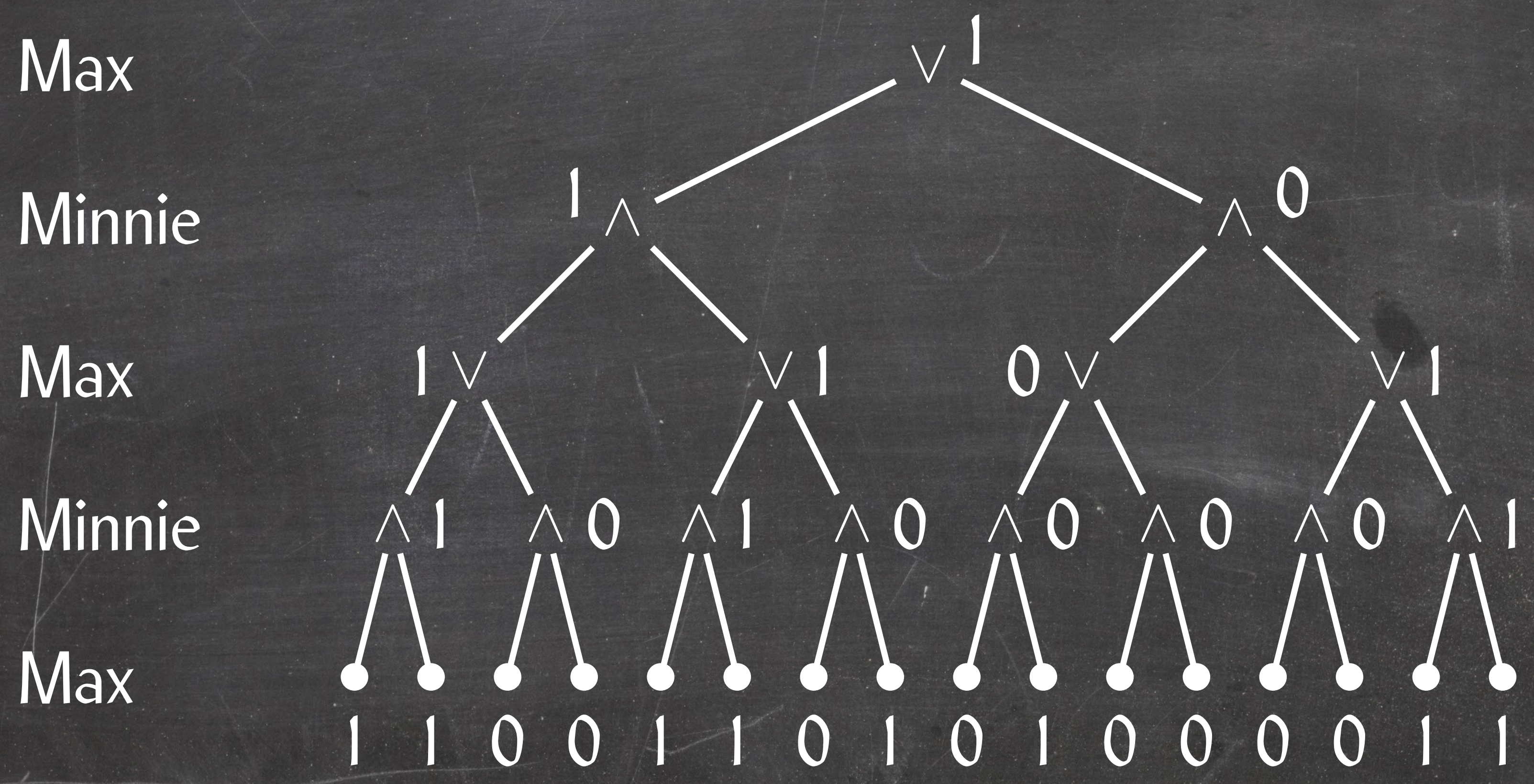
$$\text{label}(v) = \bigvee_{\text{child } w} \text{label}(w)$$

Minnie-node:

$$\text{label}(v) = \bigwedge_{\text{child } w} \text{label}(w)$$

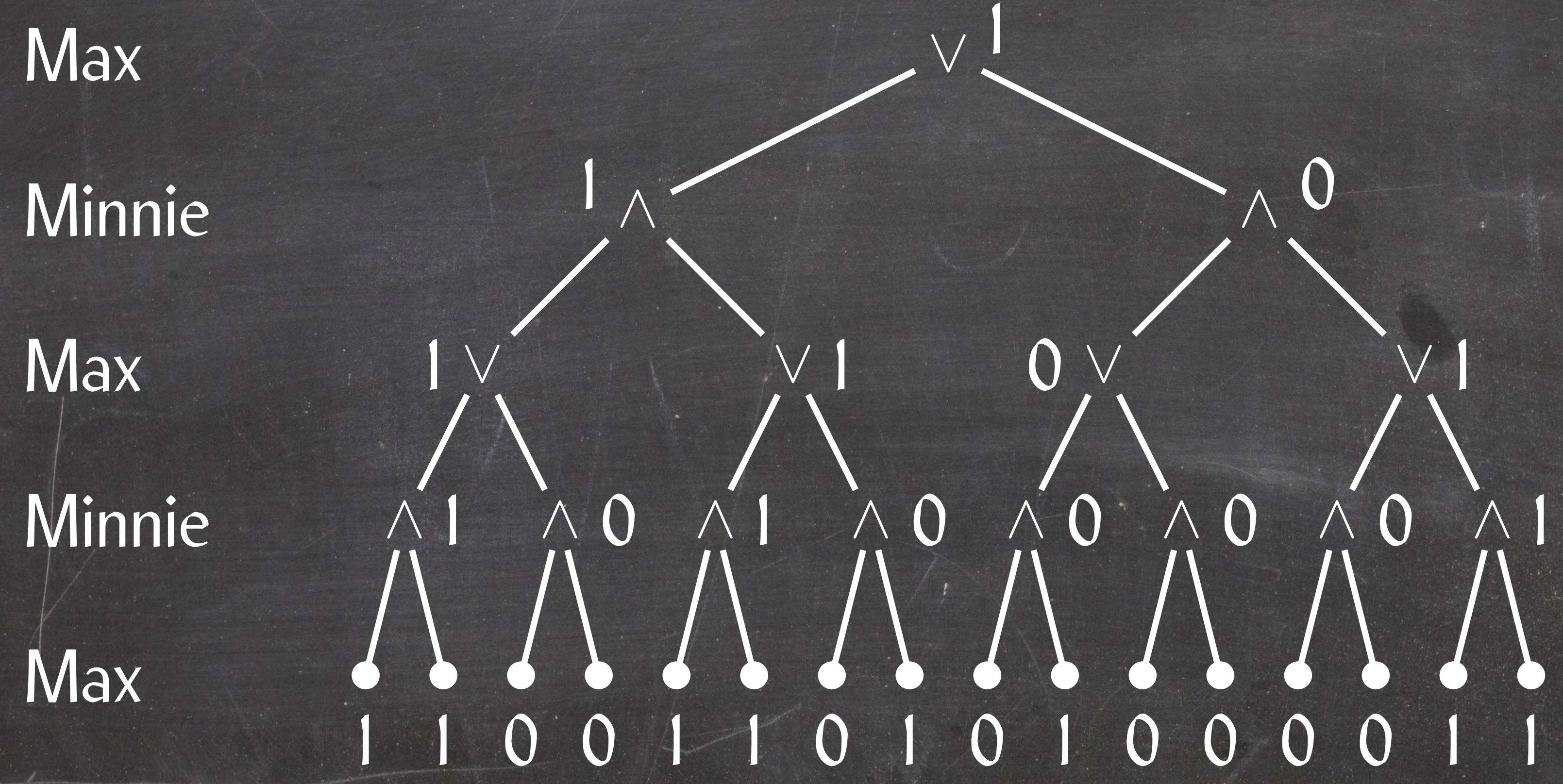
Game Tree Evaluation

Restrict ourselves to binary game trees of height 2k $\Rightarrow$ n = 2^{2k} leaves



Game Tree Evaluation

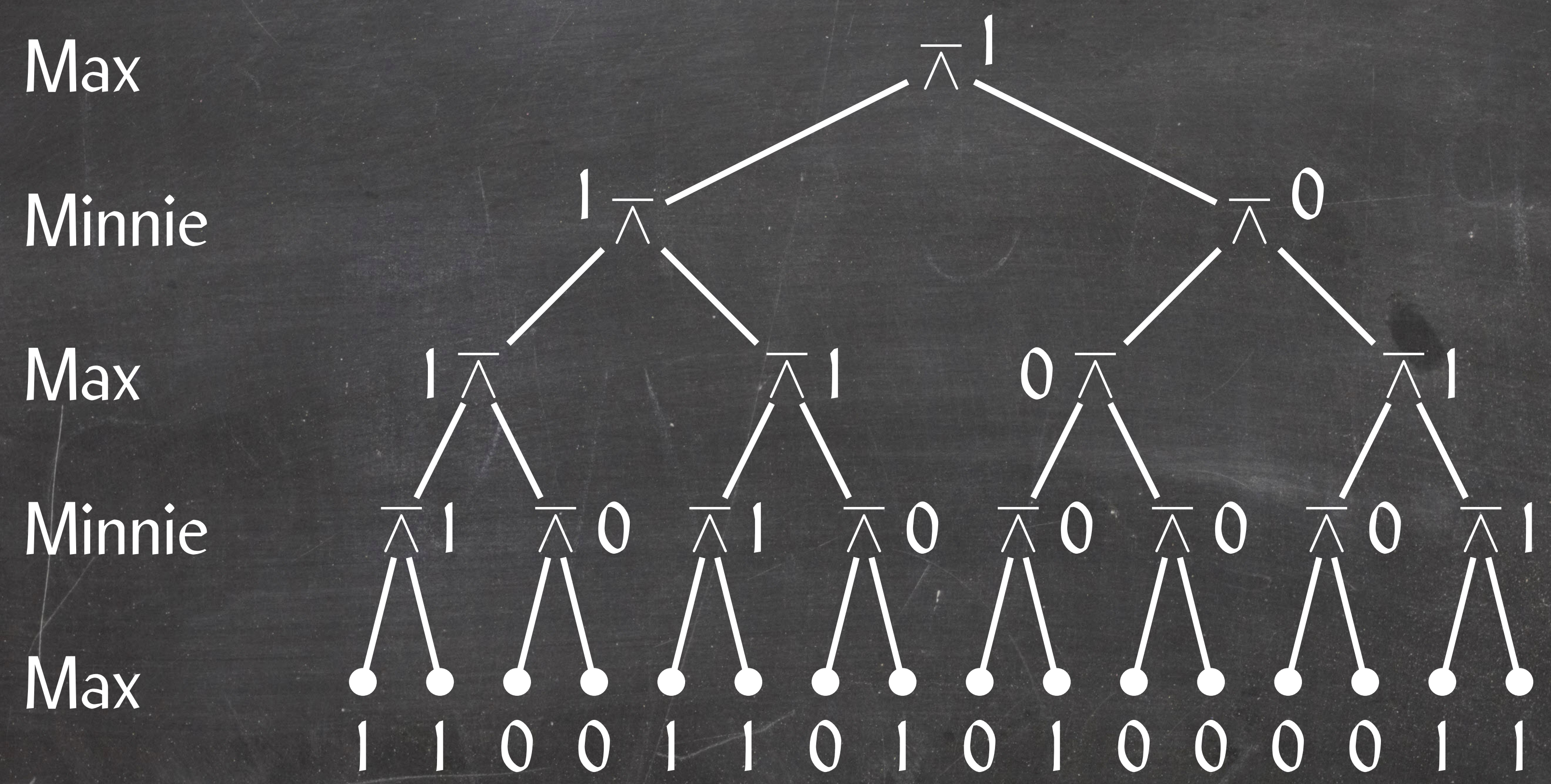
Restrict ourselves to binary game trees of height 2k $\Rightarrow$ n = 2^{2k} leaves



$$(a \wedge b) \vee (c \wedge d) = \overline{\overline{(a \wedge b)} \wedge \overline{(c \wedge d)}}$$

Game Tree Evaluation

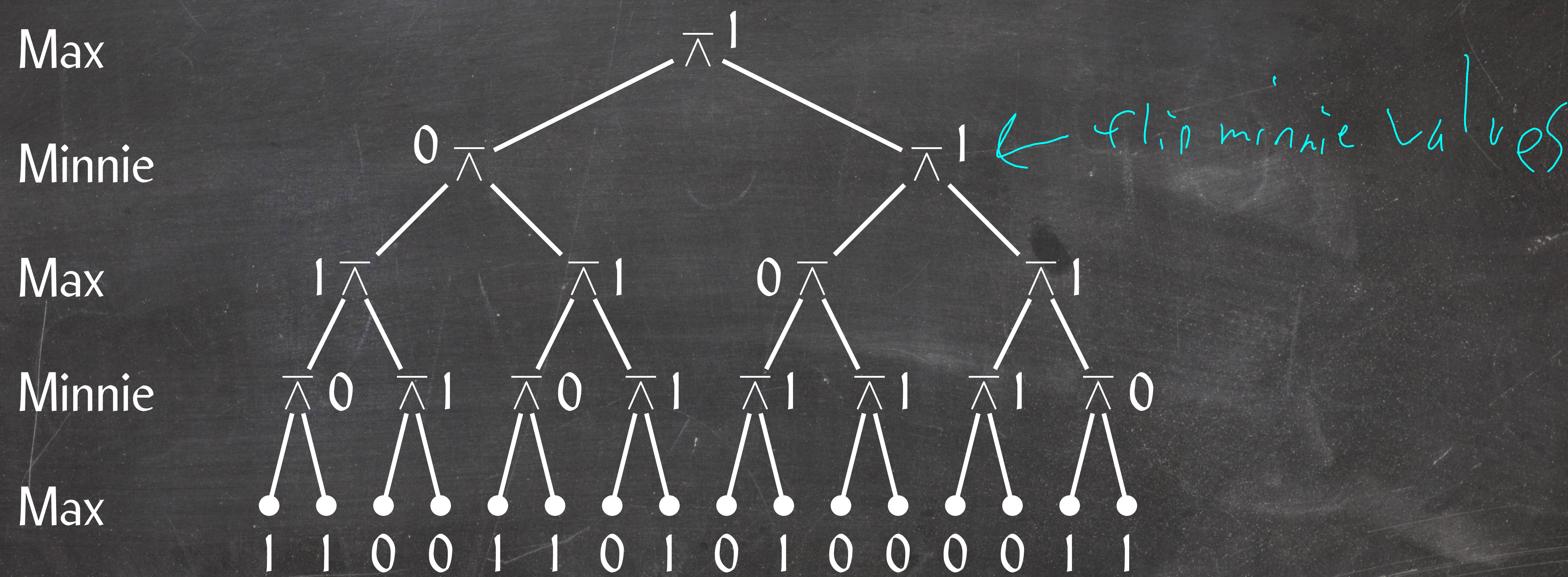
Restrict ourselves to binary game trees of height 2k $\Rightarrow$ n = 2^{2k} leaves



$$(a \wedge b) \vee (c \wedge d) = \overline{\overline{(a \wedge b)} \wedge \overline{(c \wedge d)}}$$

Game Tree Evaluation

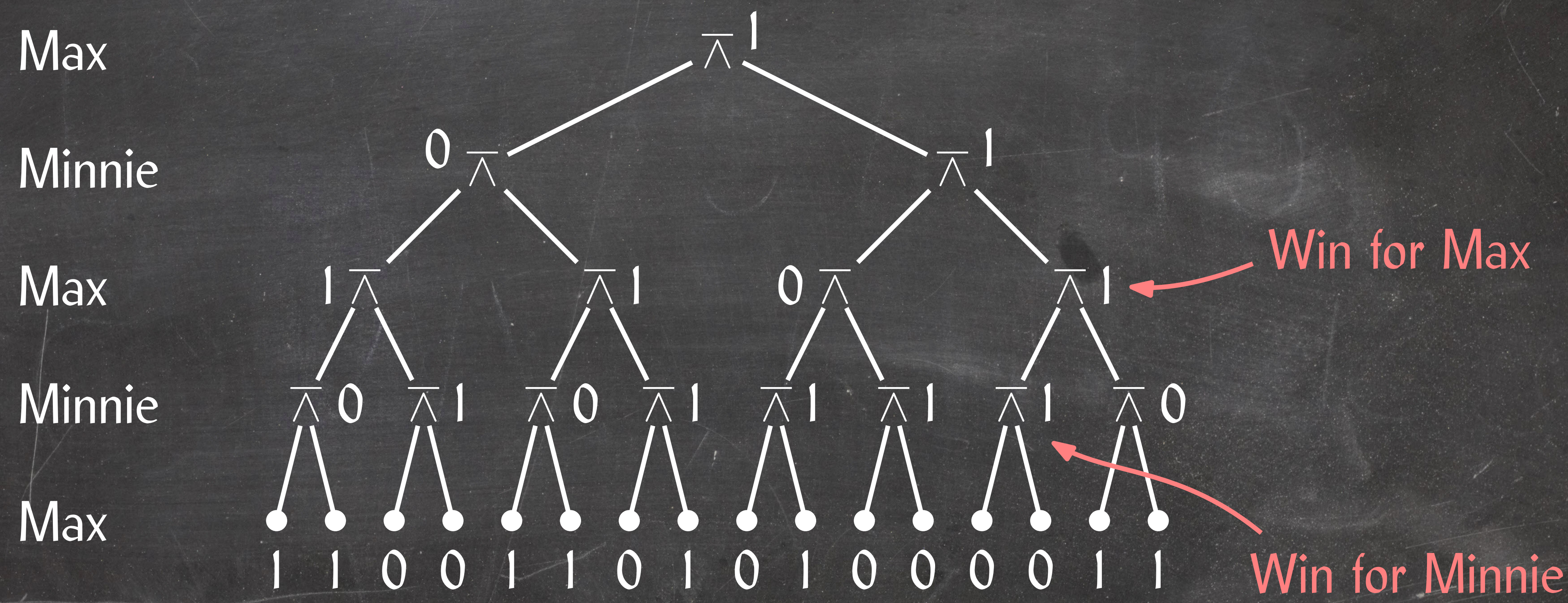
Restrict ourselves to binary game trees of height 2k $\Rightarrow n = 2^{2k}$ leaves



$$(a \wedge b) \vee (c \wedge d) = \overline{\overline{(a \wedge b)} \wedge \overline{(c \wedge d)}}$$

Game Tree Evaluation

Restrict ourselves to binary game trees of height 2k $\Rightarrow$ n = 2^{2k} leaves



$$(a \wedge b) \vee (c \wedge d) = \overline{\overline{(a \wedge b)} \wedge \overline{(c \wedge d)}}$$

Game Tree Evaluation: A Deterministic Algorithm

GameValue(v)

```
if v is a leaf
    then return its value
    else return not (GameValue(v.leftChild) and GameValue(v.rightChild))
```

Game Tree Evaluation: A Deterministic Algorithm

GameValue(v)

```
if v is a leaf
    then return its value
    else return not (GameValue(v.leftChild) and GameValue(v.rightChild))
```

- One recursive call per node
- $2n - 1$ nodes

$\Rightarrow$ Running time $O(n)$

Game Tree Evaluation: A Deterministic Algorithm

before: not (value(l) ∧ value(r))

GameValue(v)

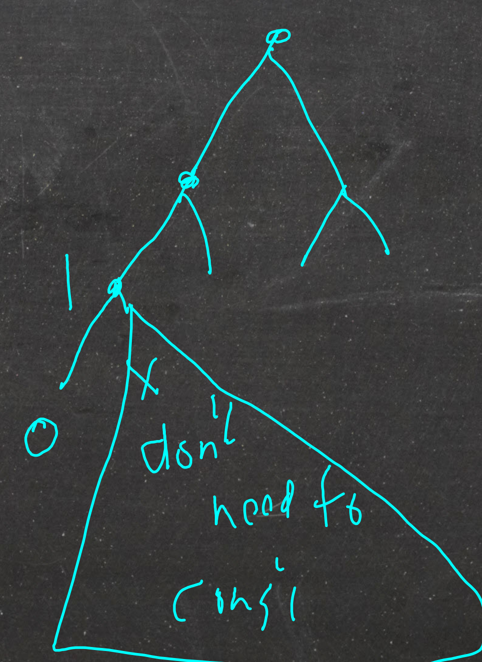
```
if v is a leaf
    then return its value
if not GameValue(v.leftChild)
    then return 1
    else return not GameValue(v.rightChild)
```

if not(value(l)) that is a win

else return not(value(r))

- One recursive call per node
- 2n – 1 nodes

⇒ Running time O(n)



Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.

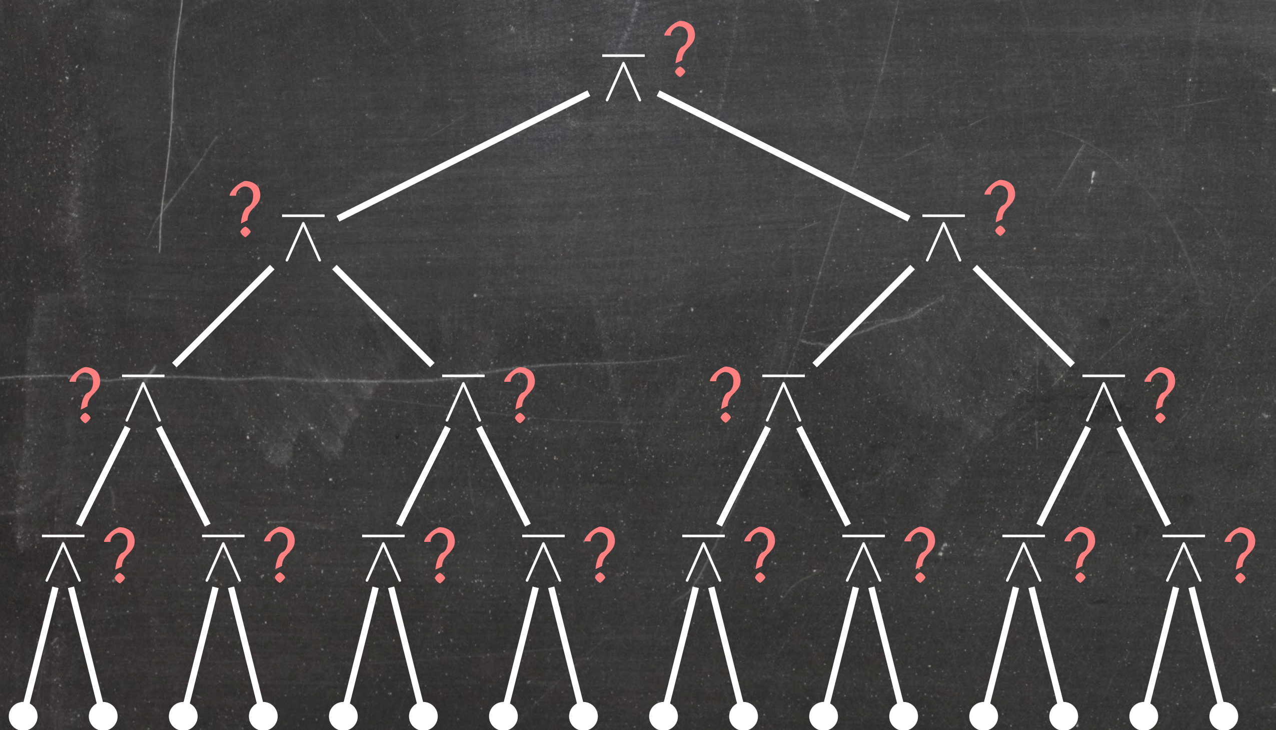
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



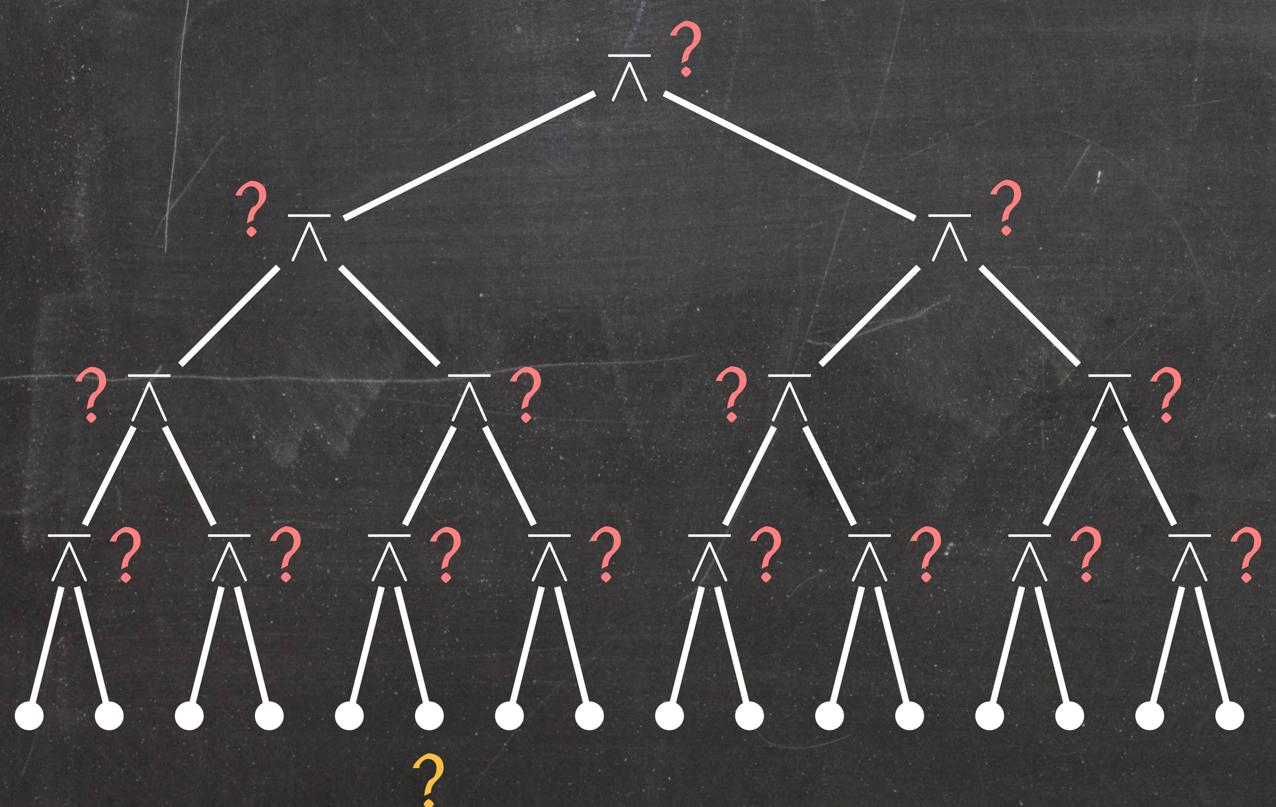
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



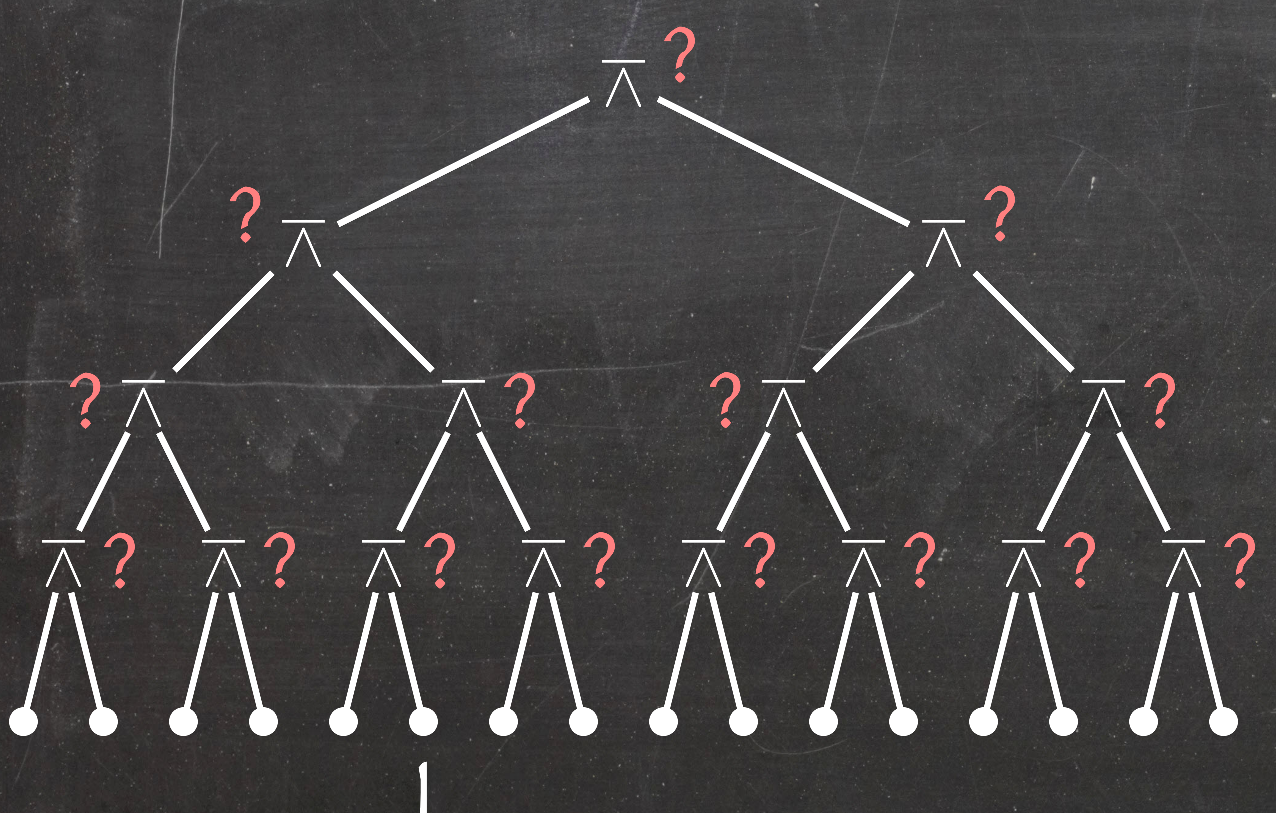
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



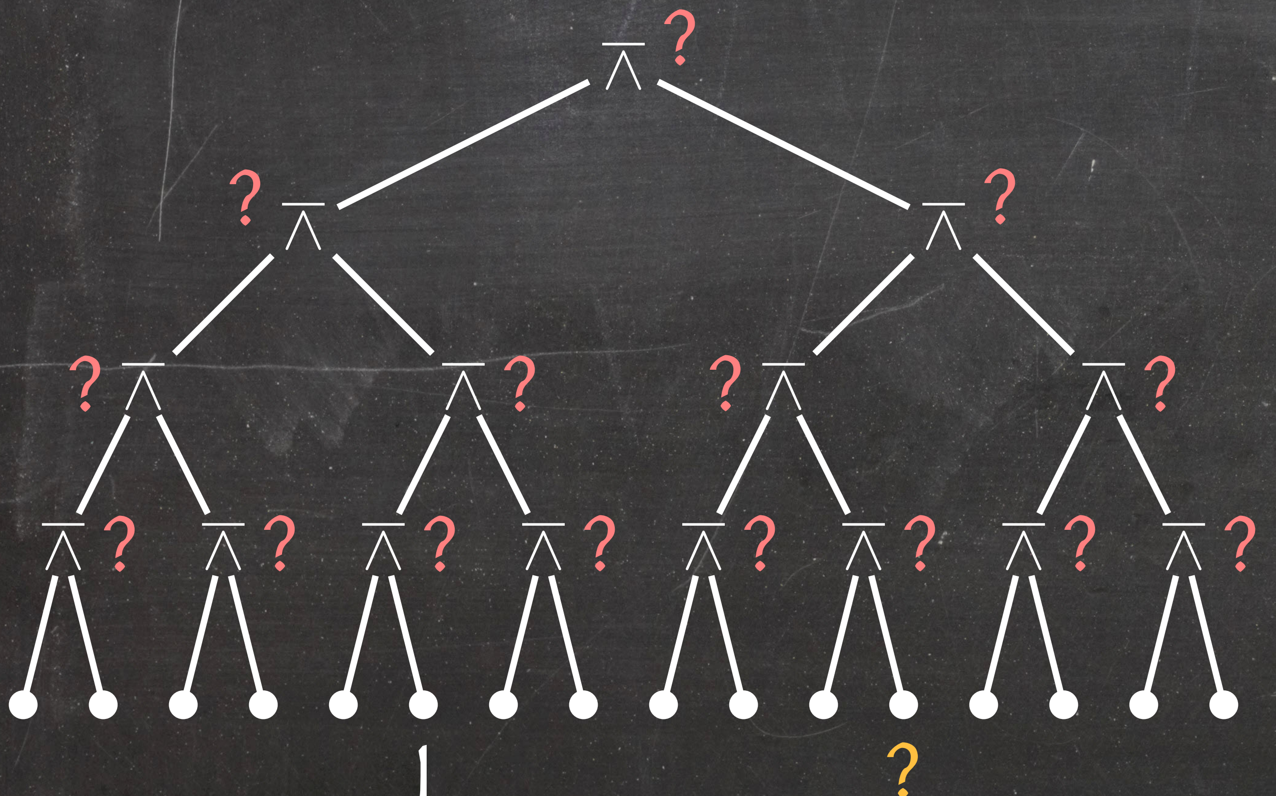
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



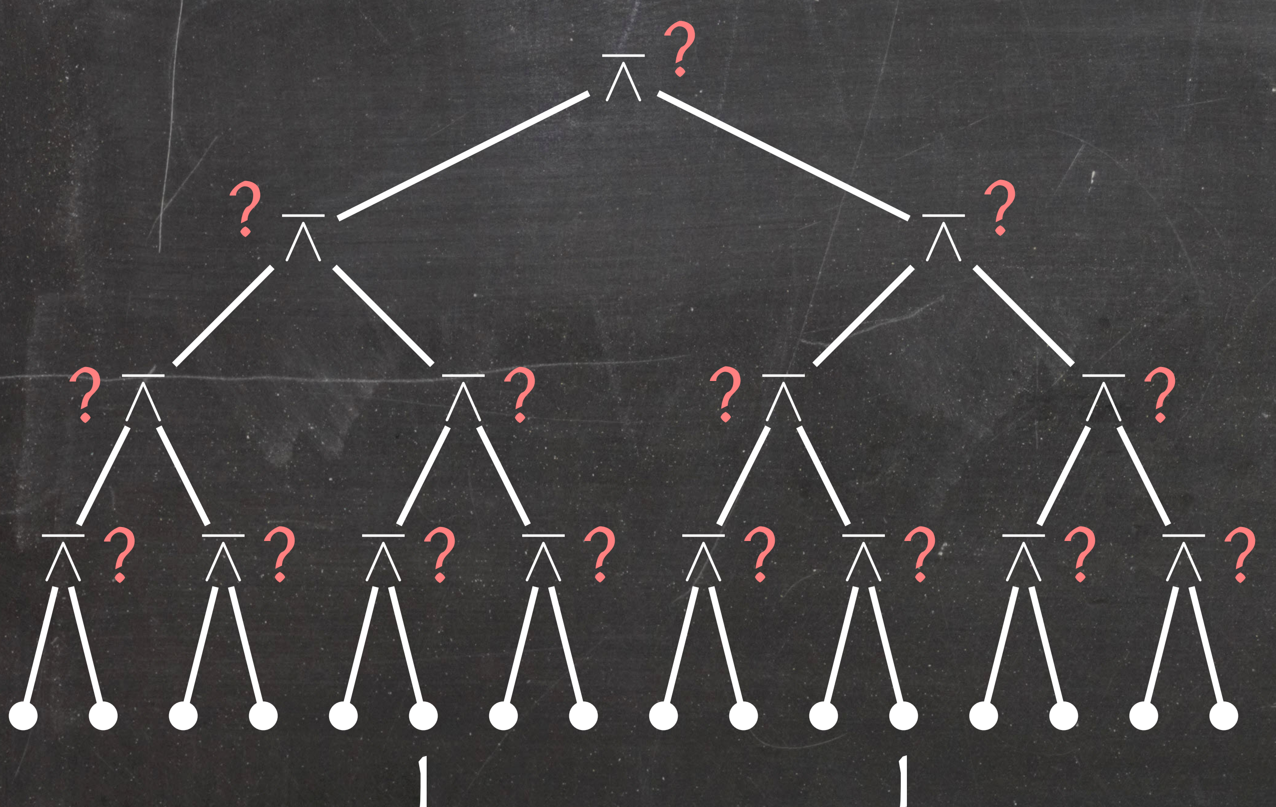
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



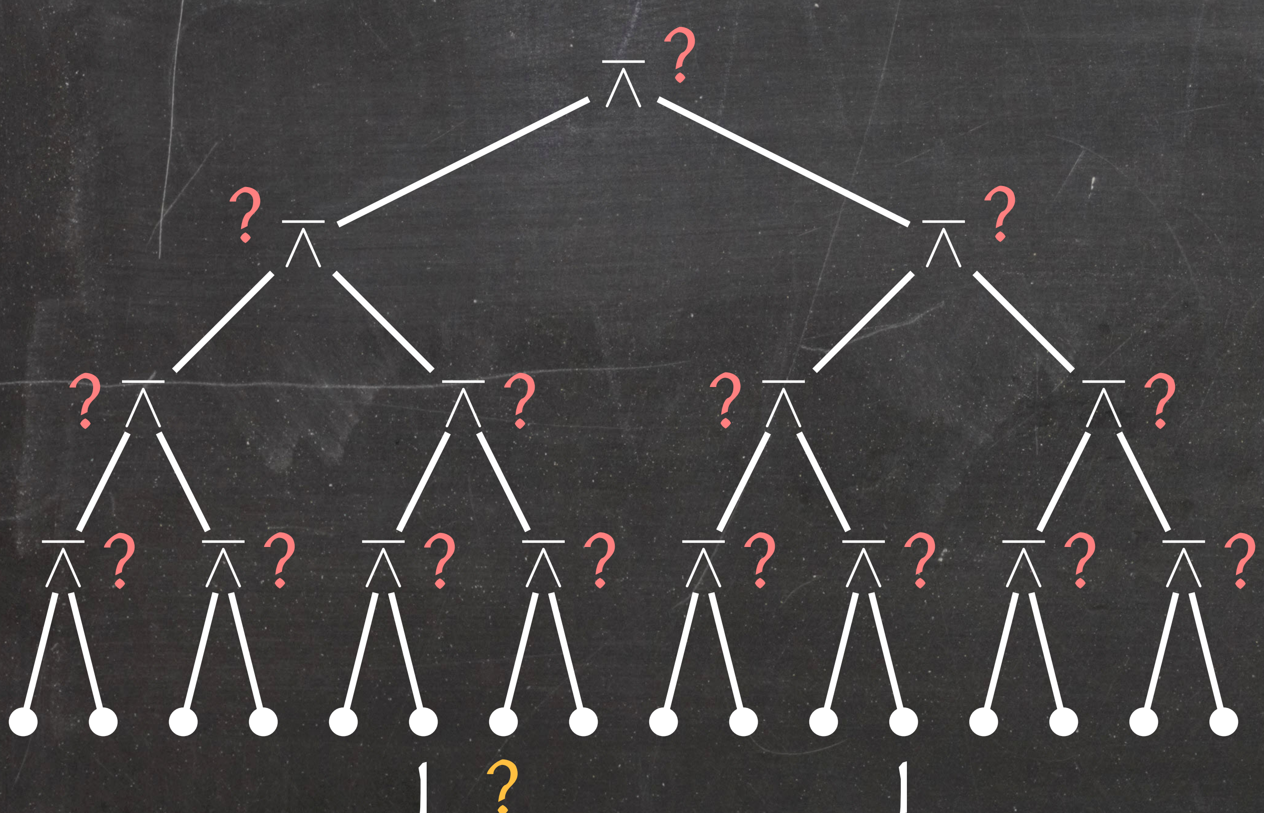
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



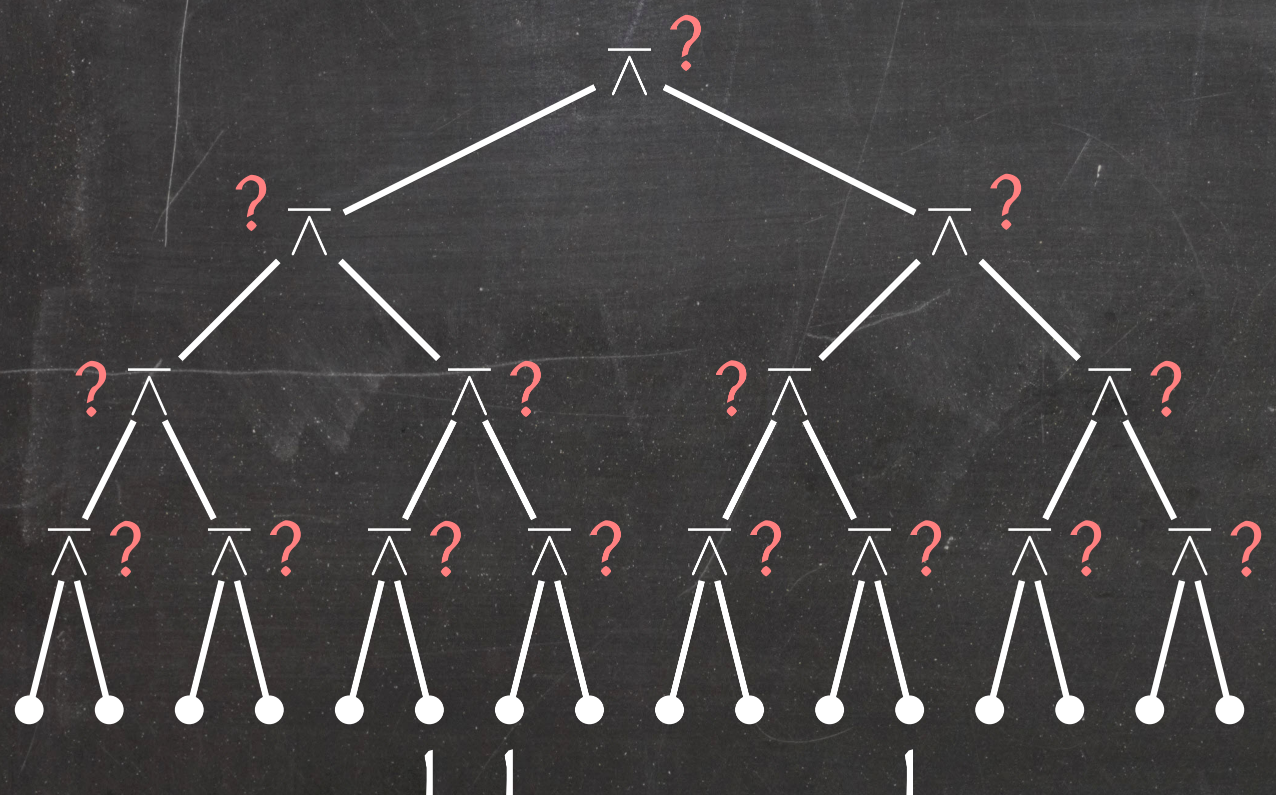
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



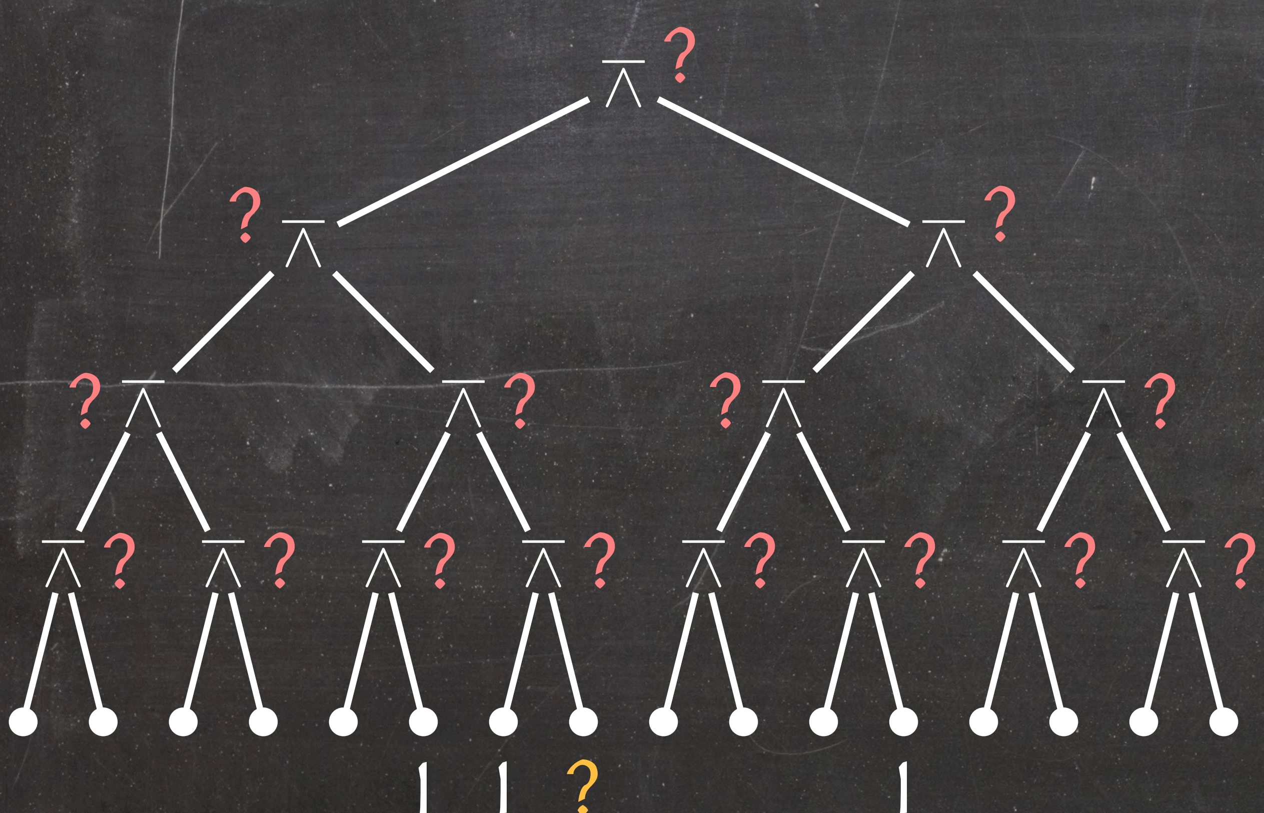
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



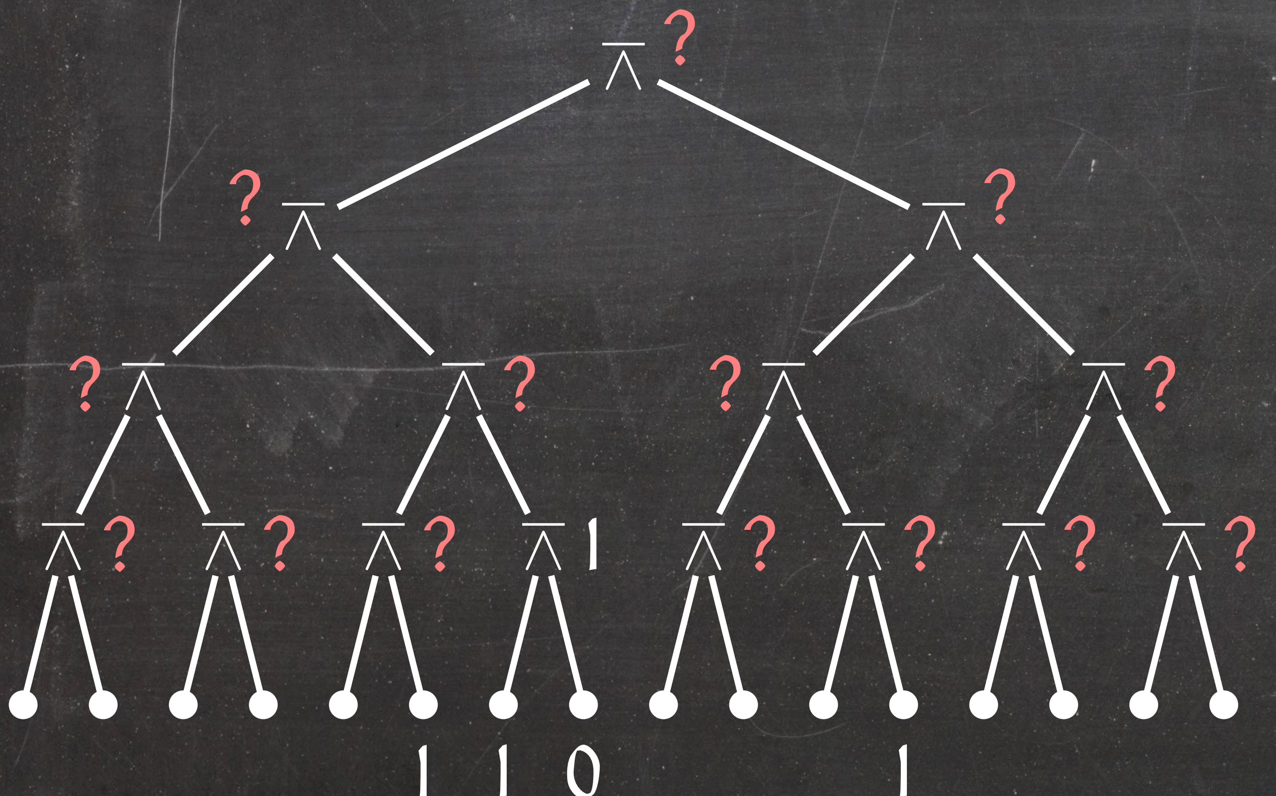
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



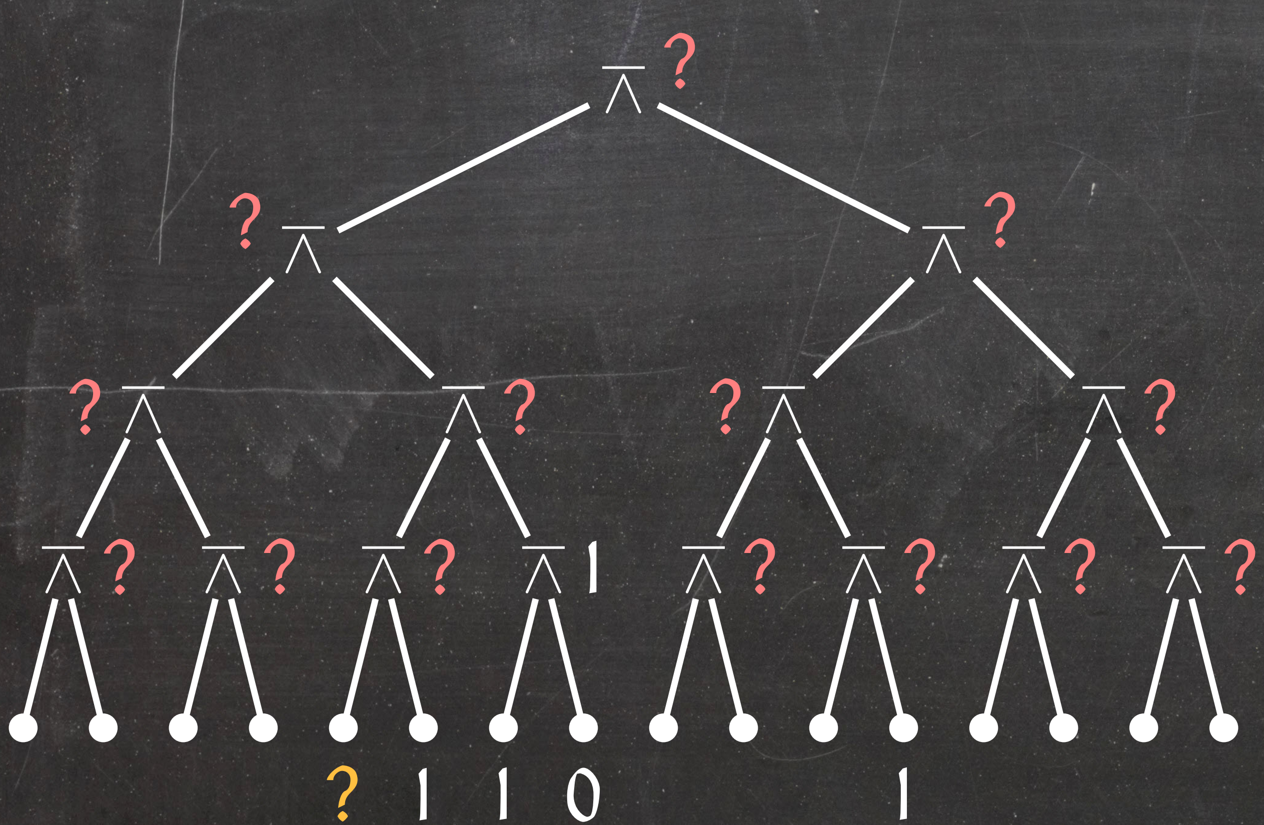
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



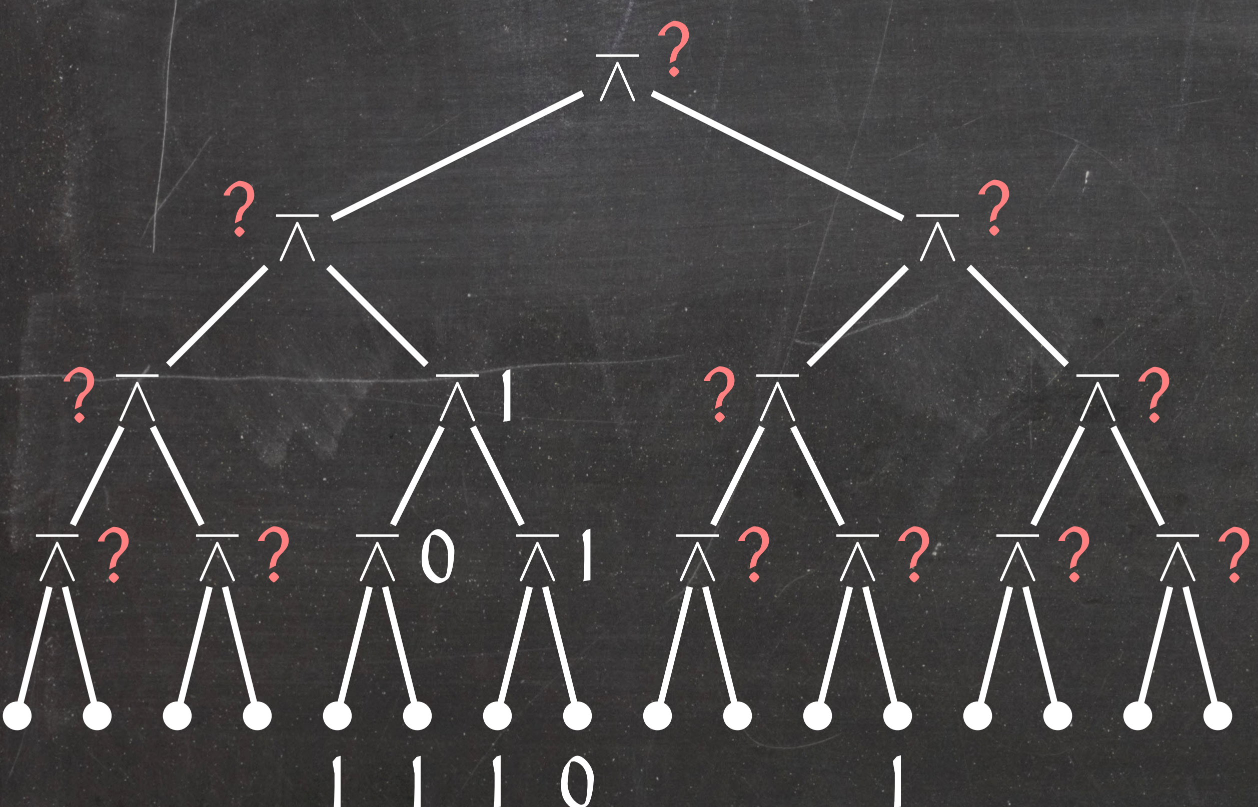
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



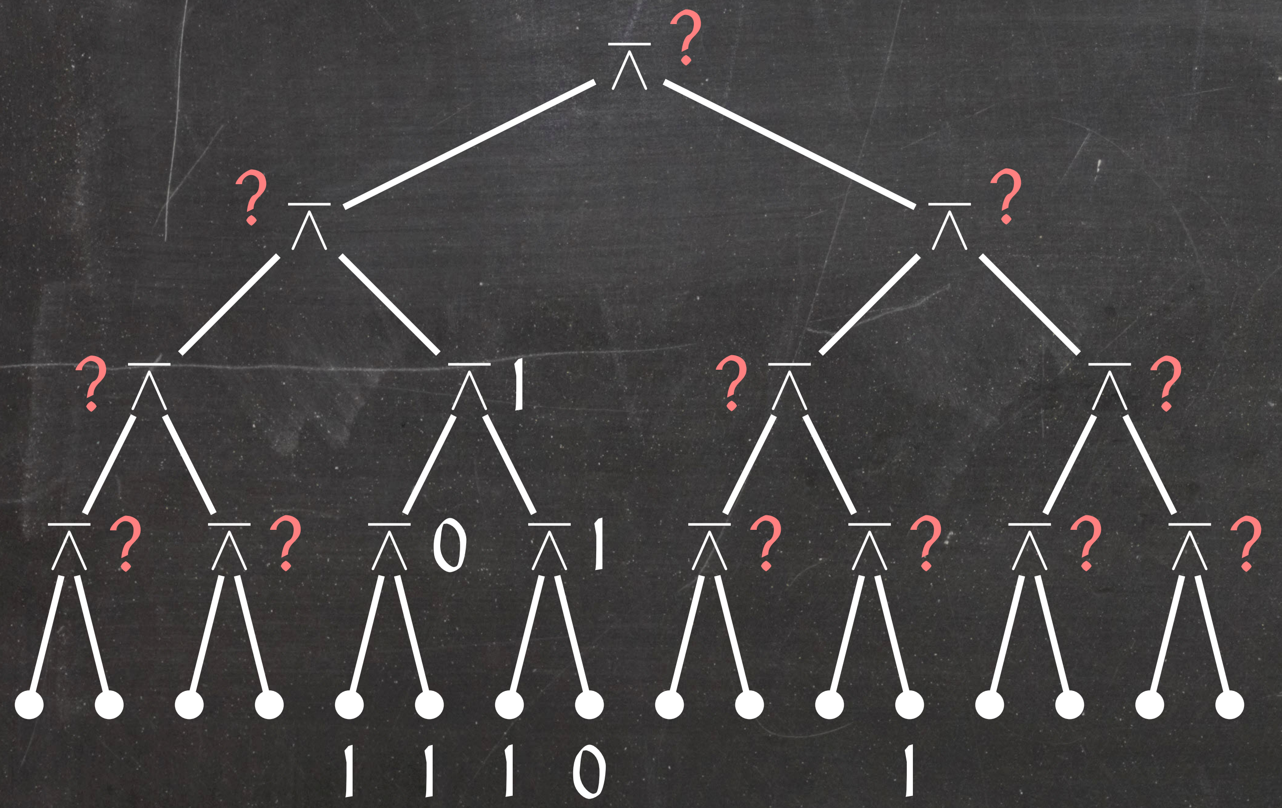
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

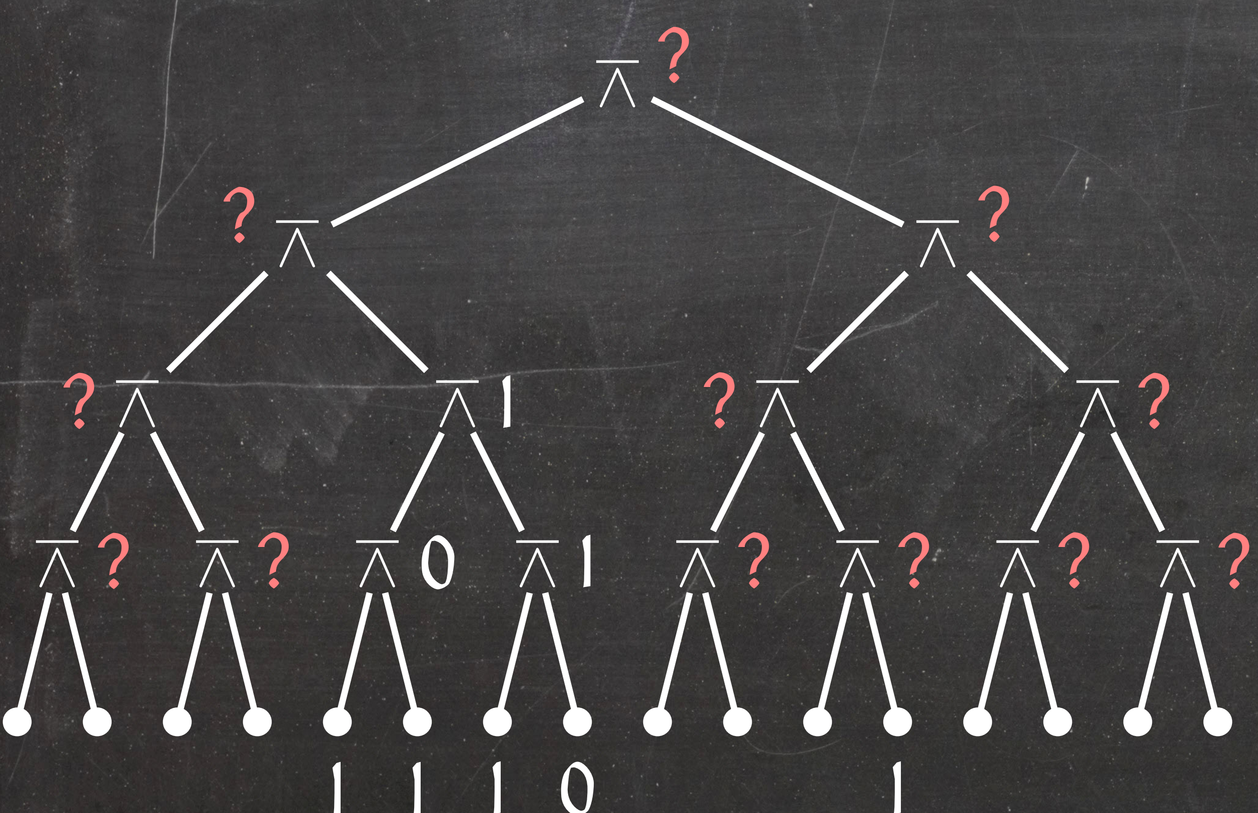
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

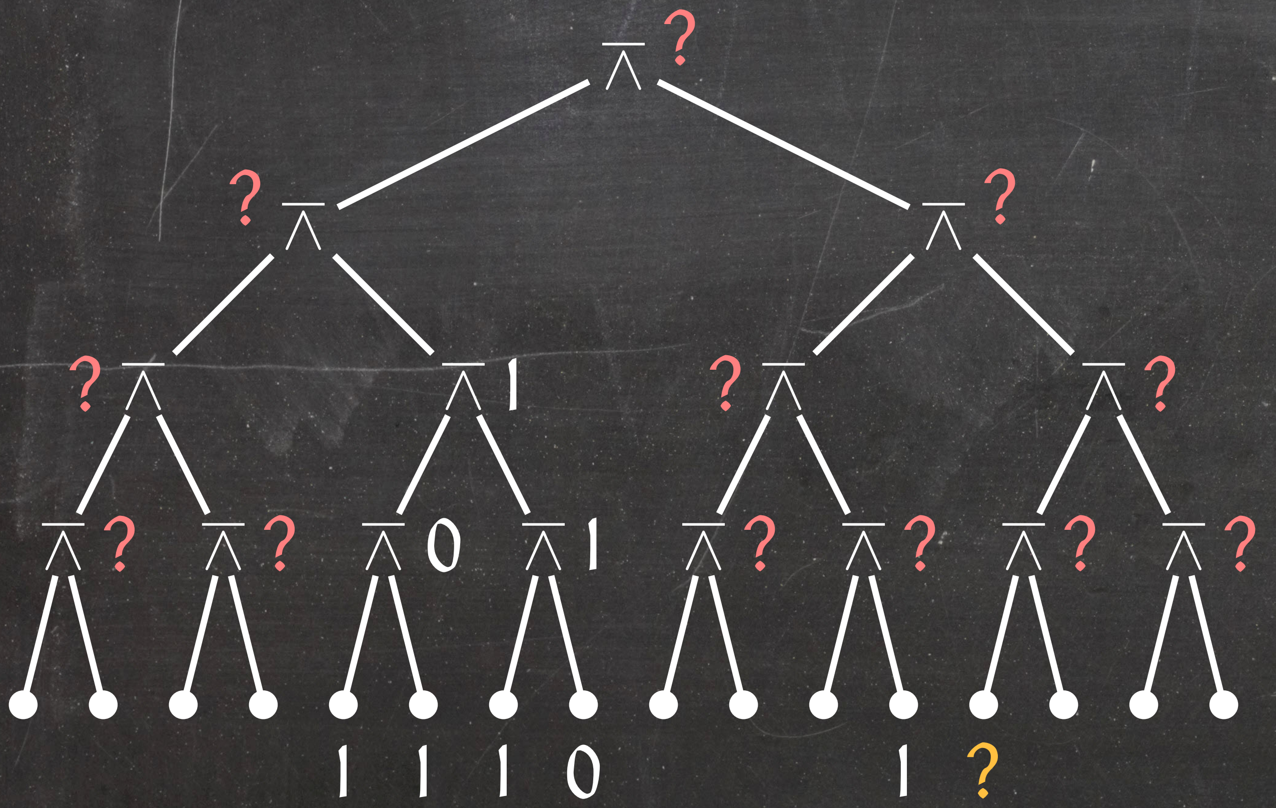
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

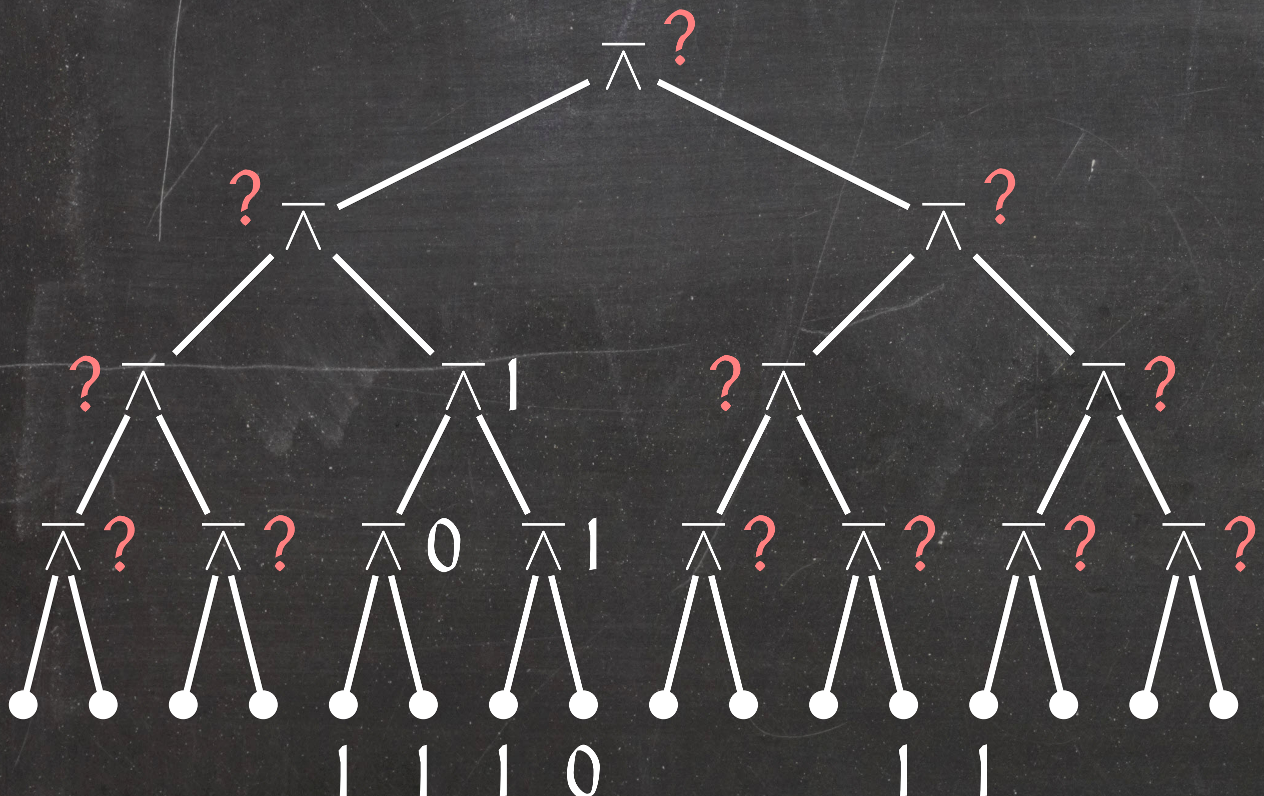
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

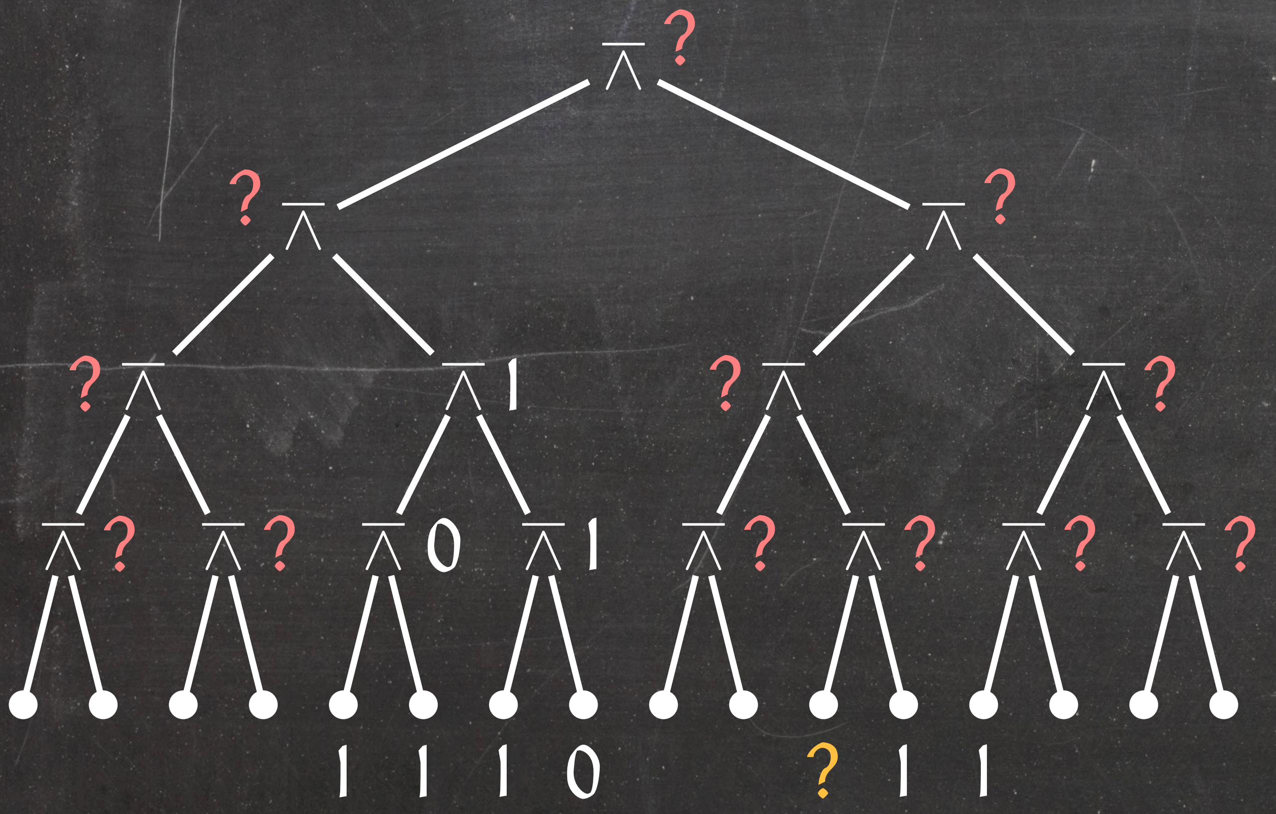
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

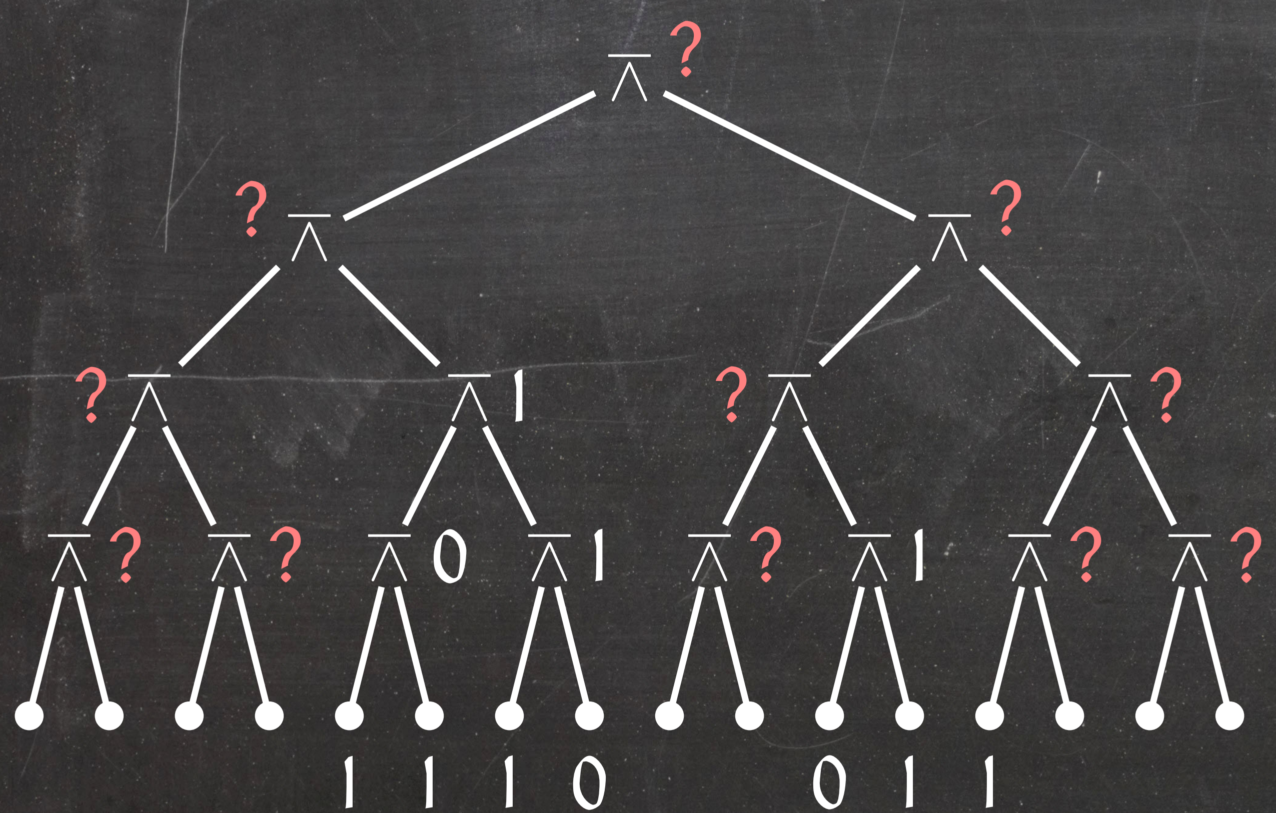
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

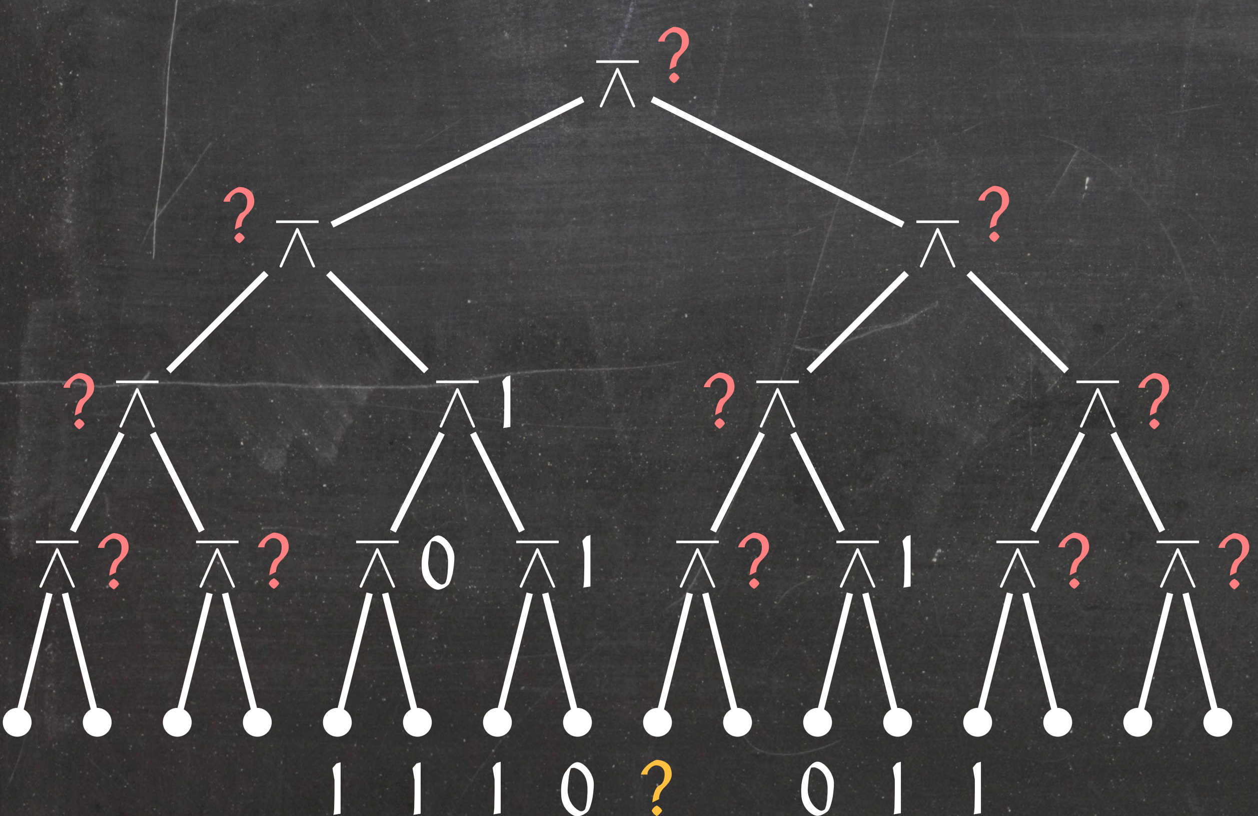
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

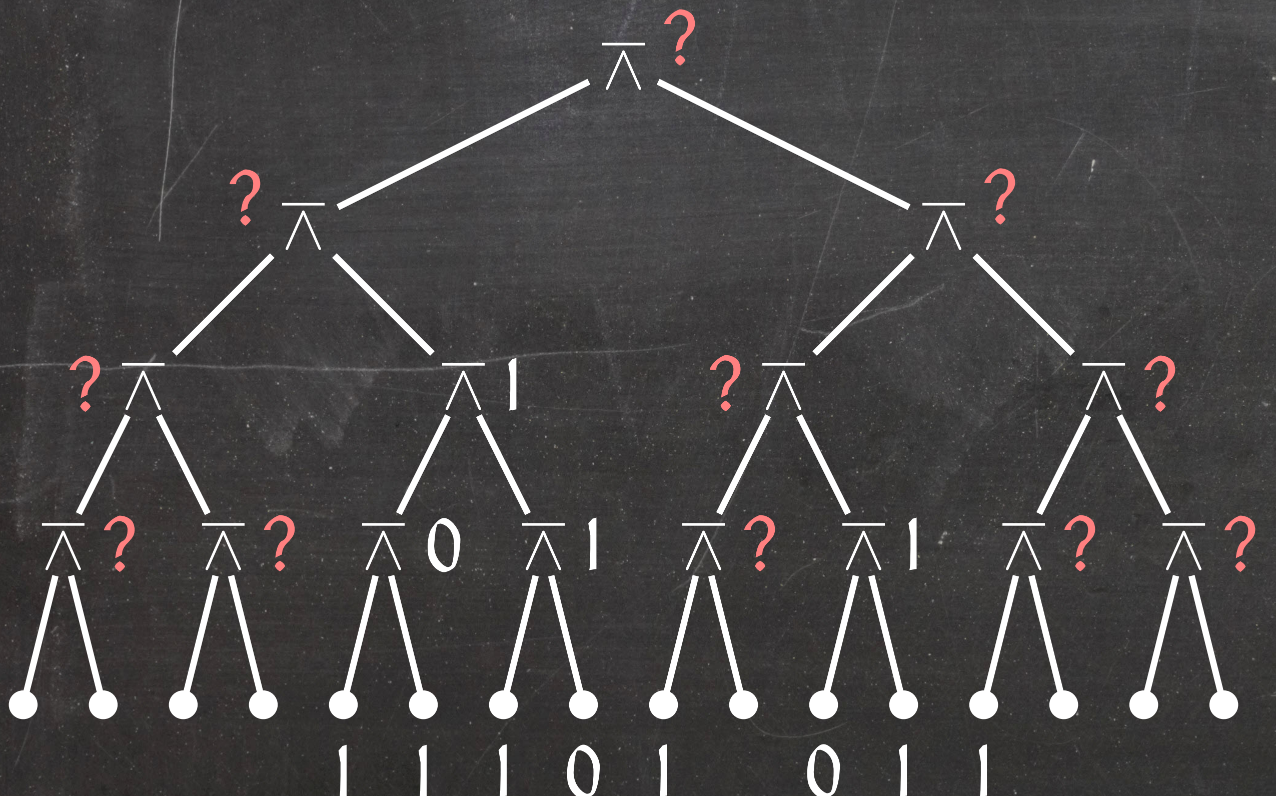
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

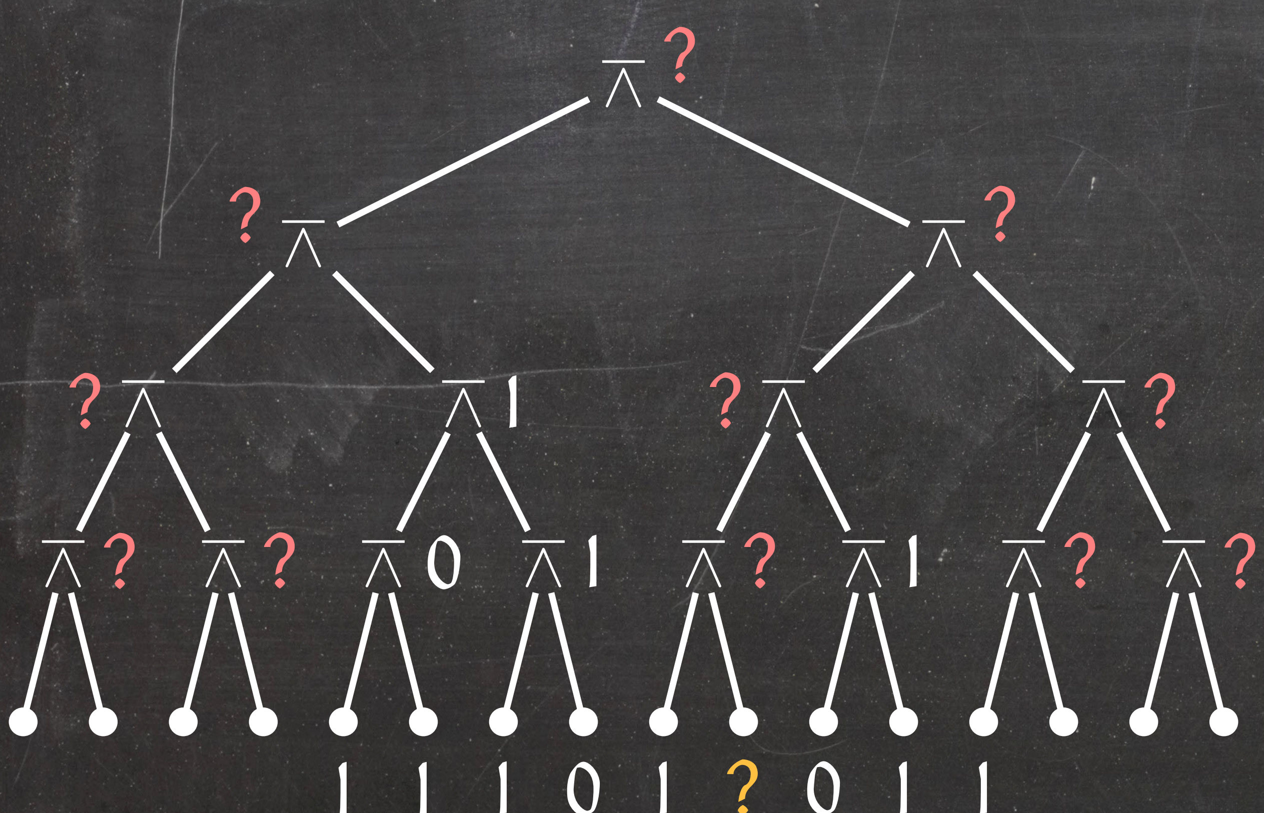
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

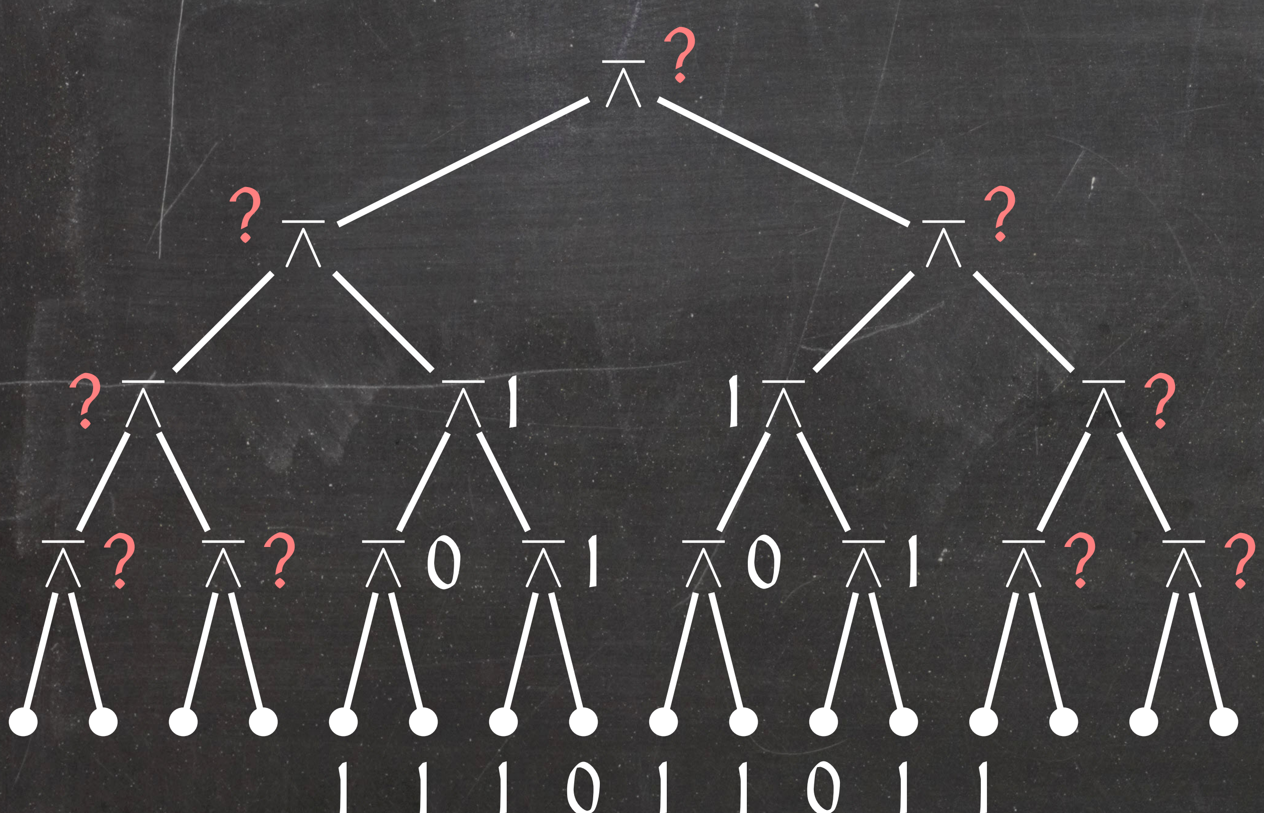
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

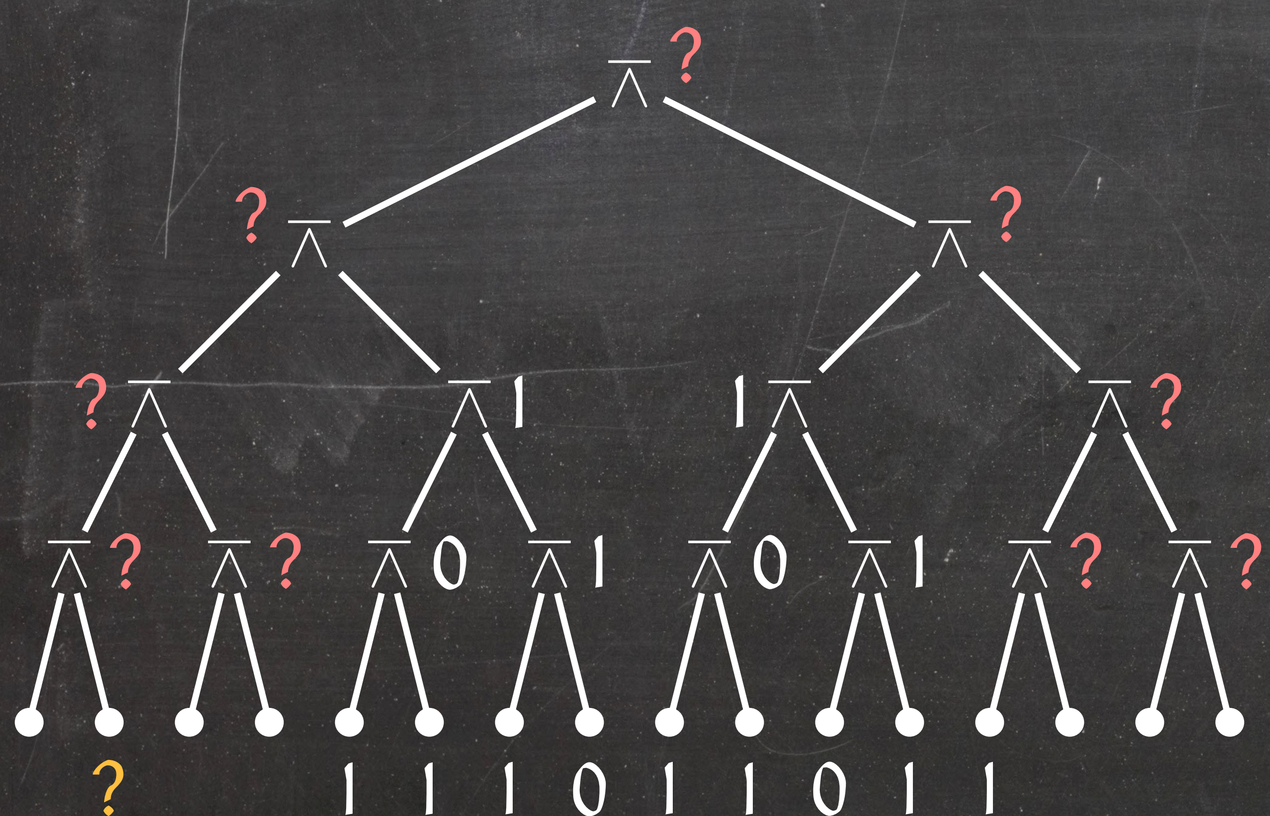
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

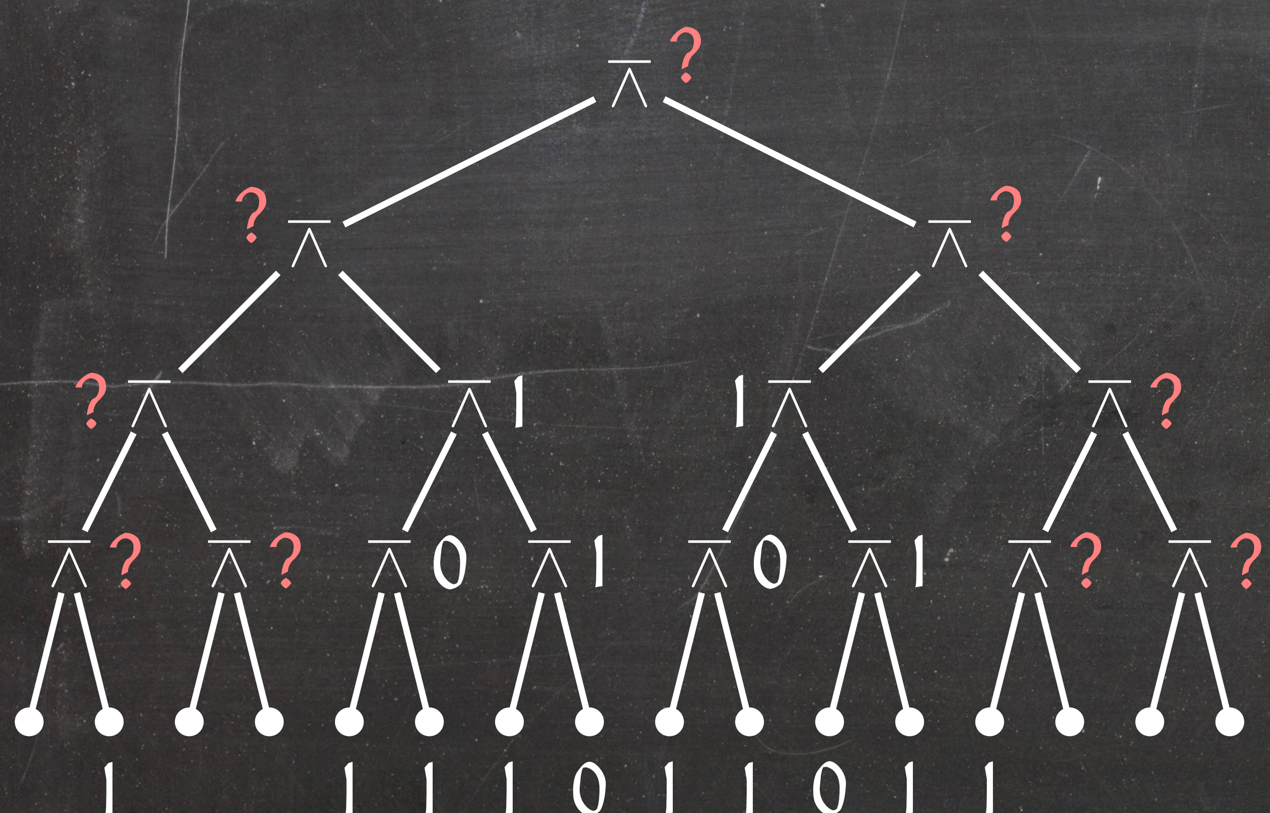
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

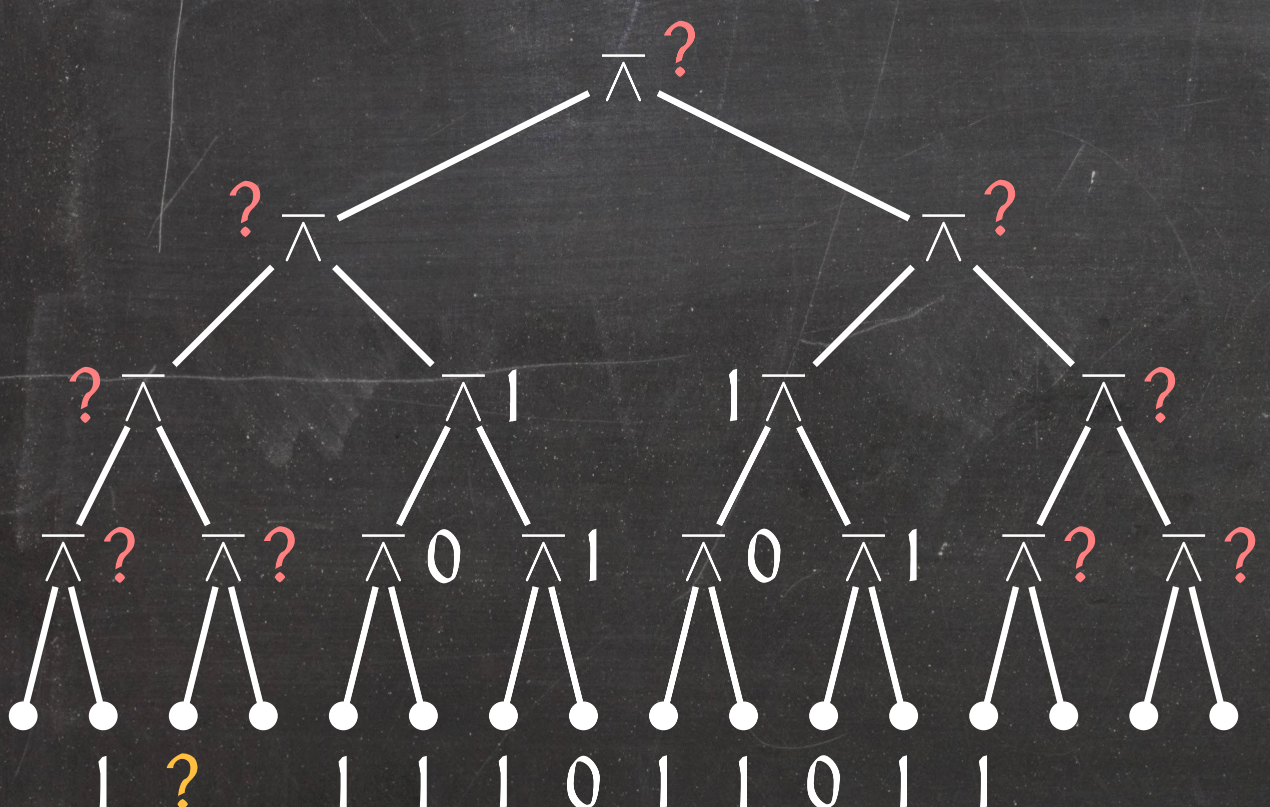
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

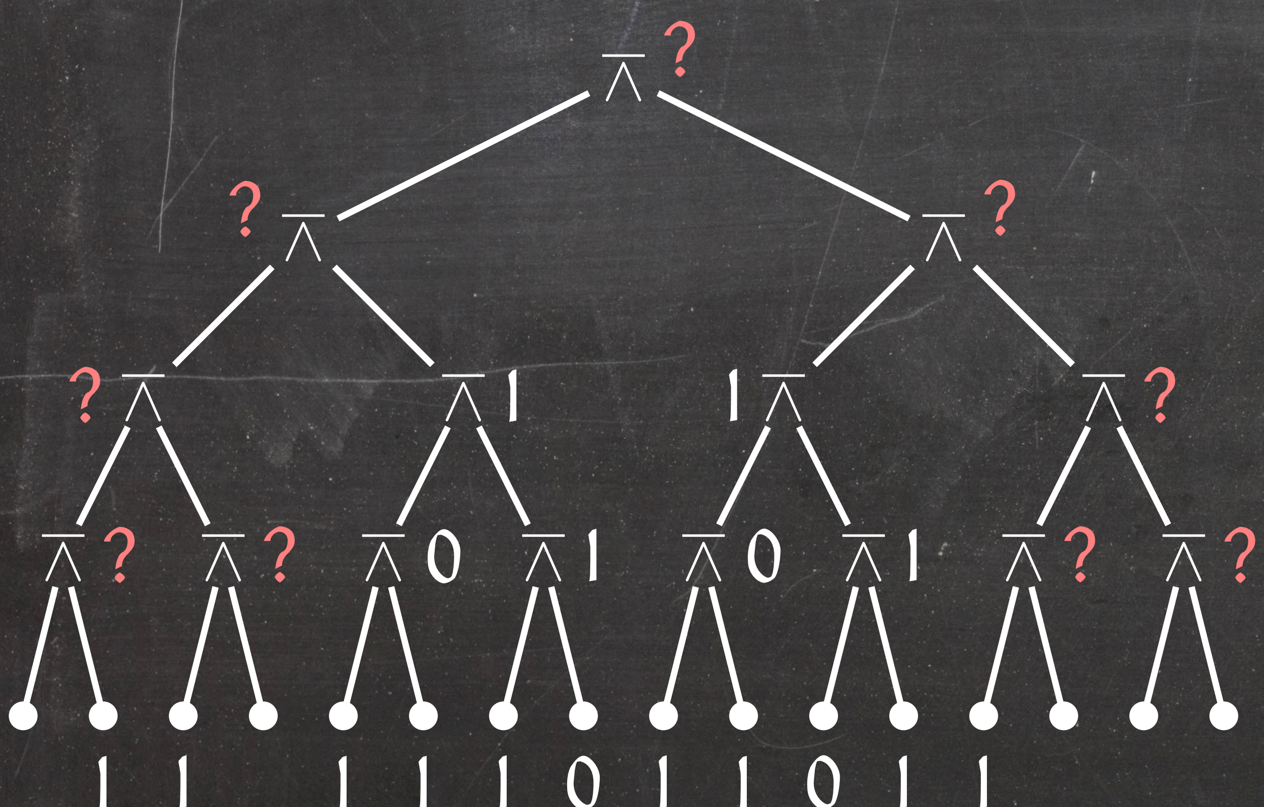
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

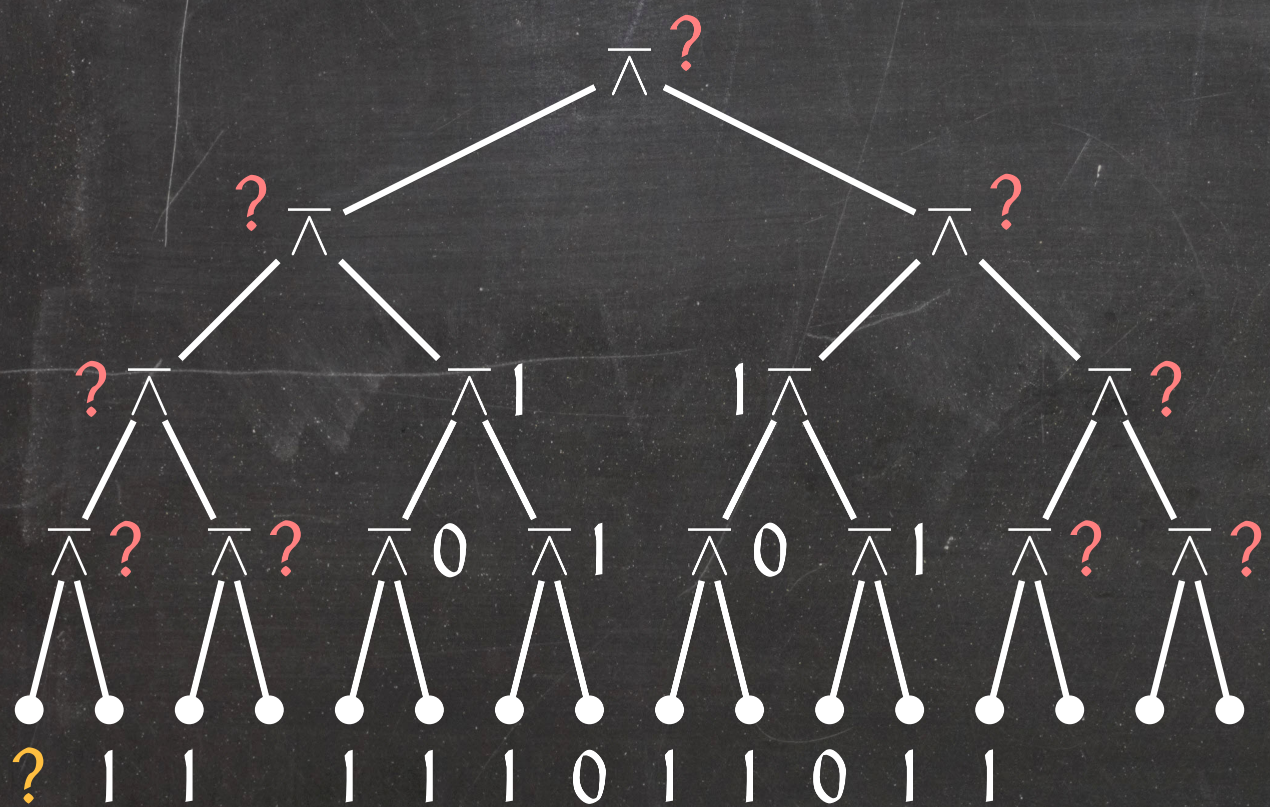
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

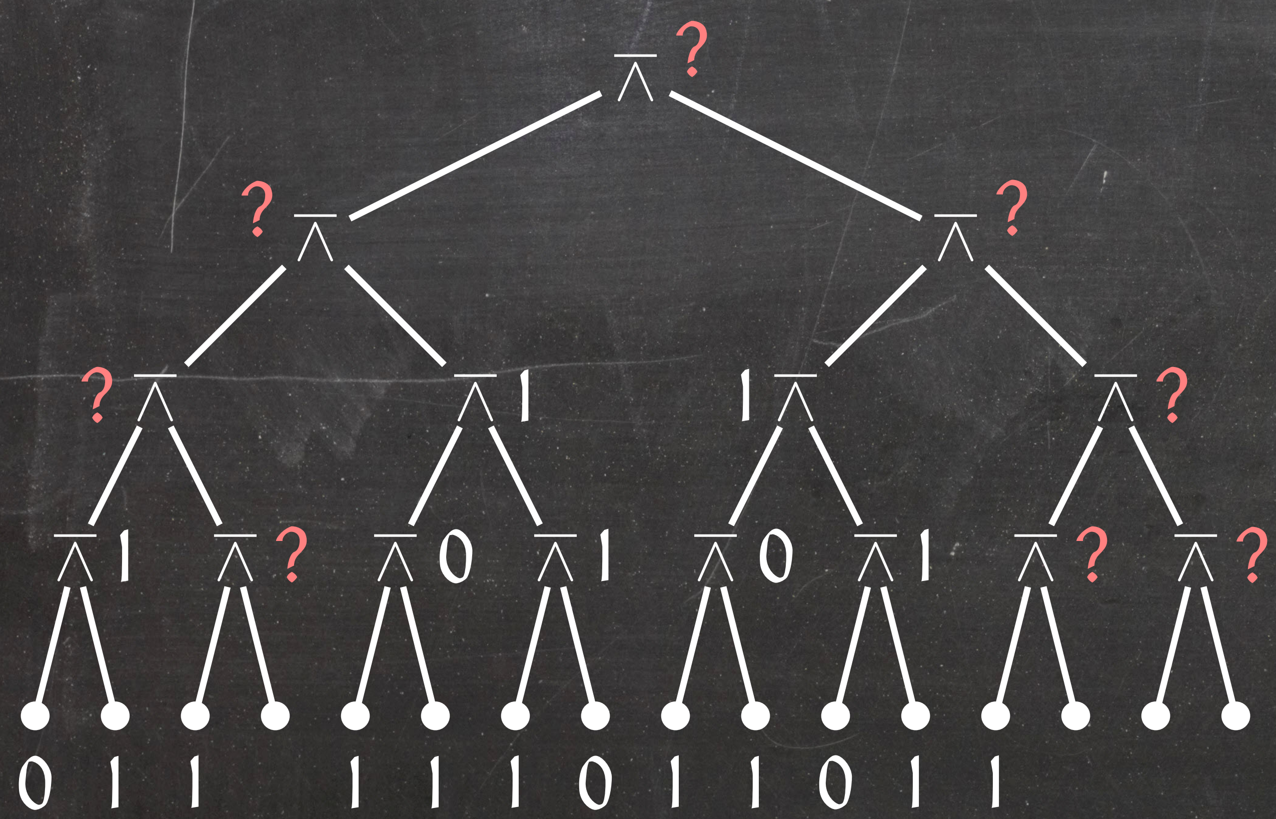
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

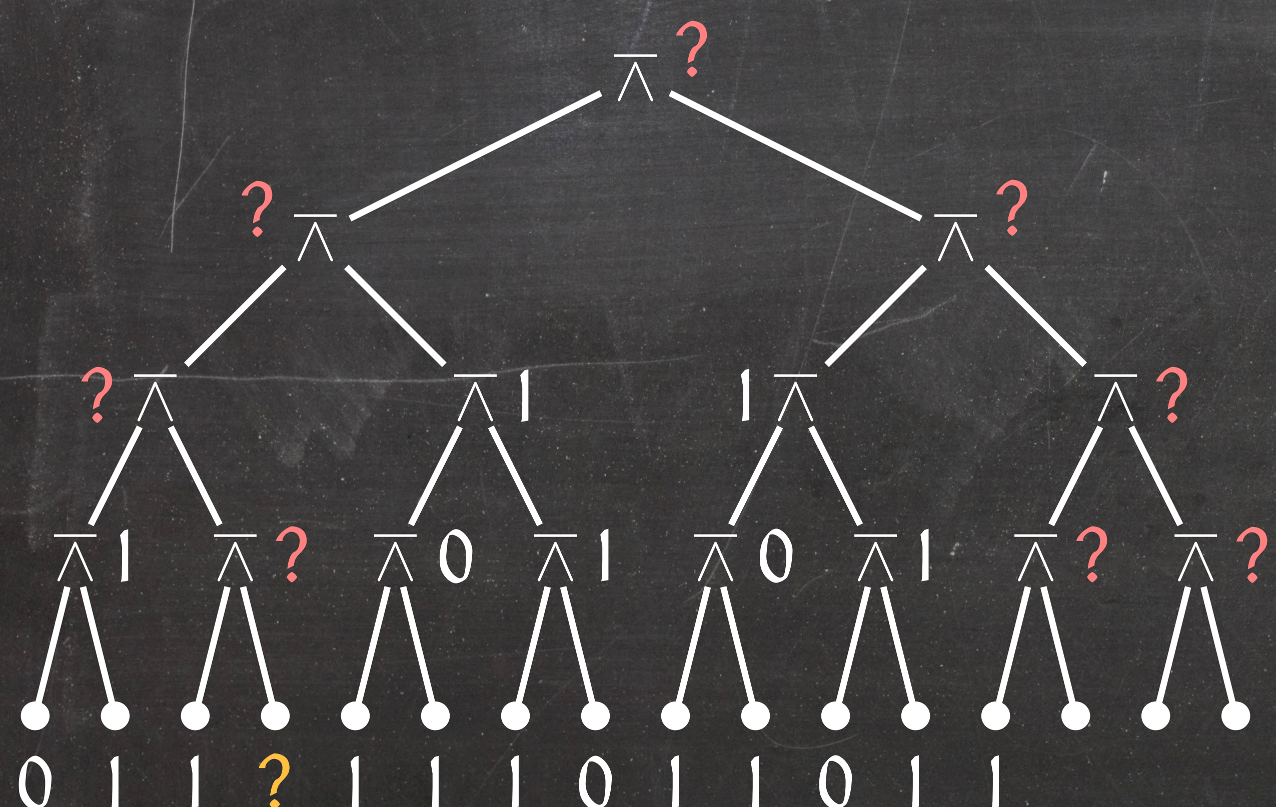
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

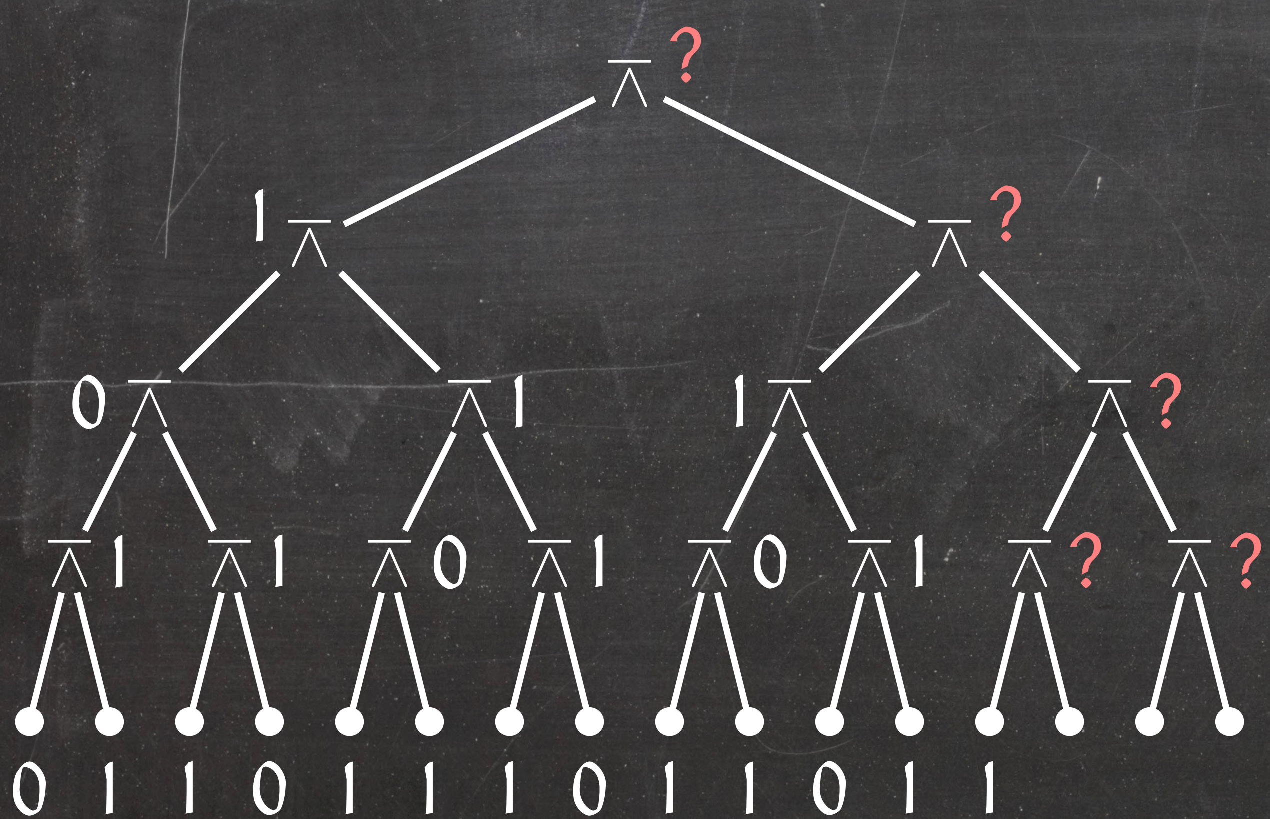
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

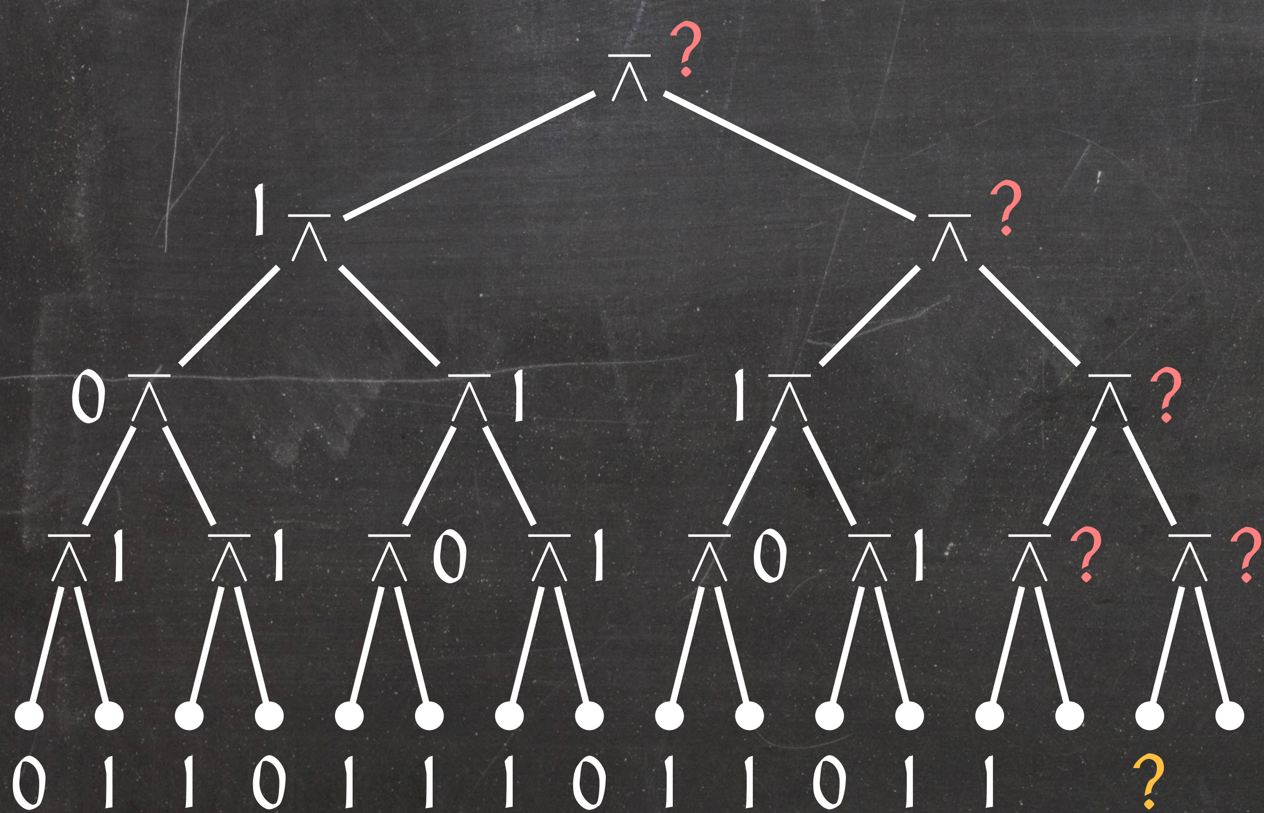
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

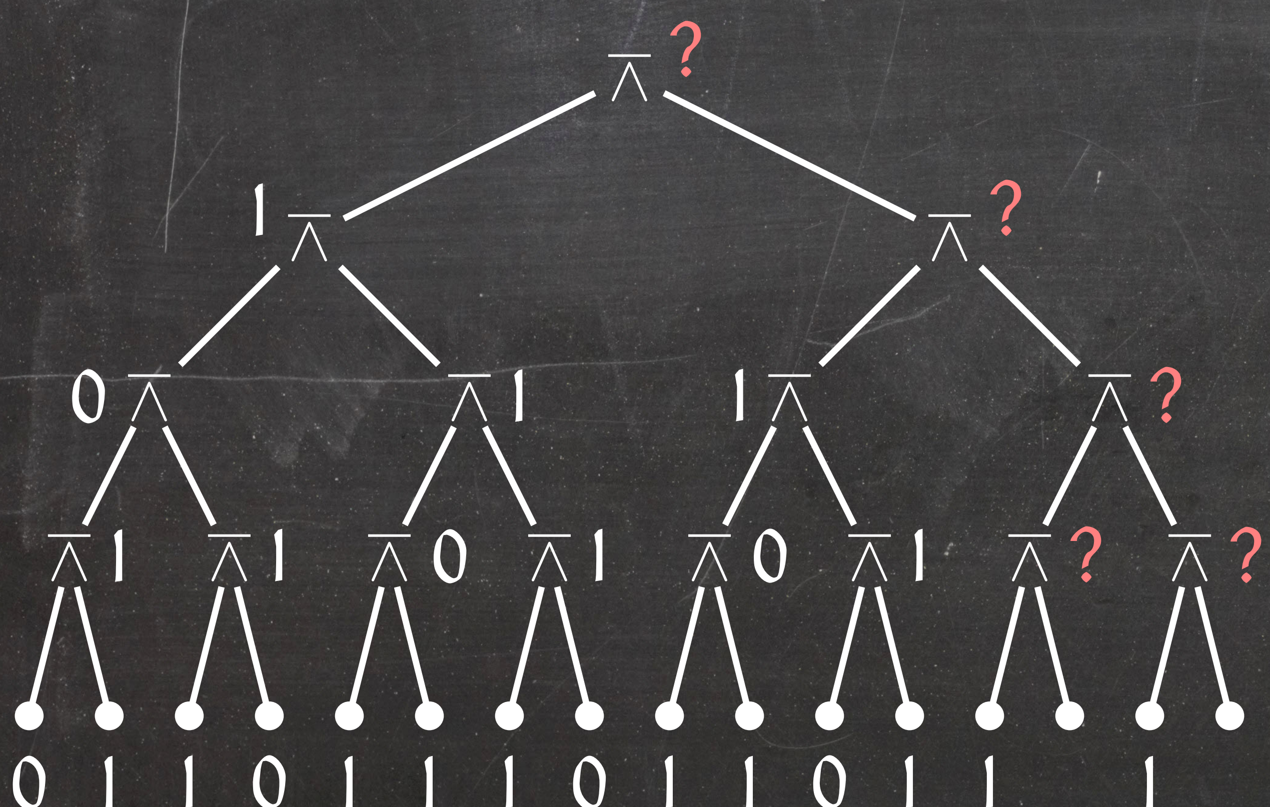
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

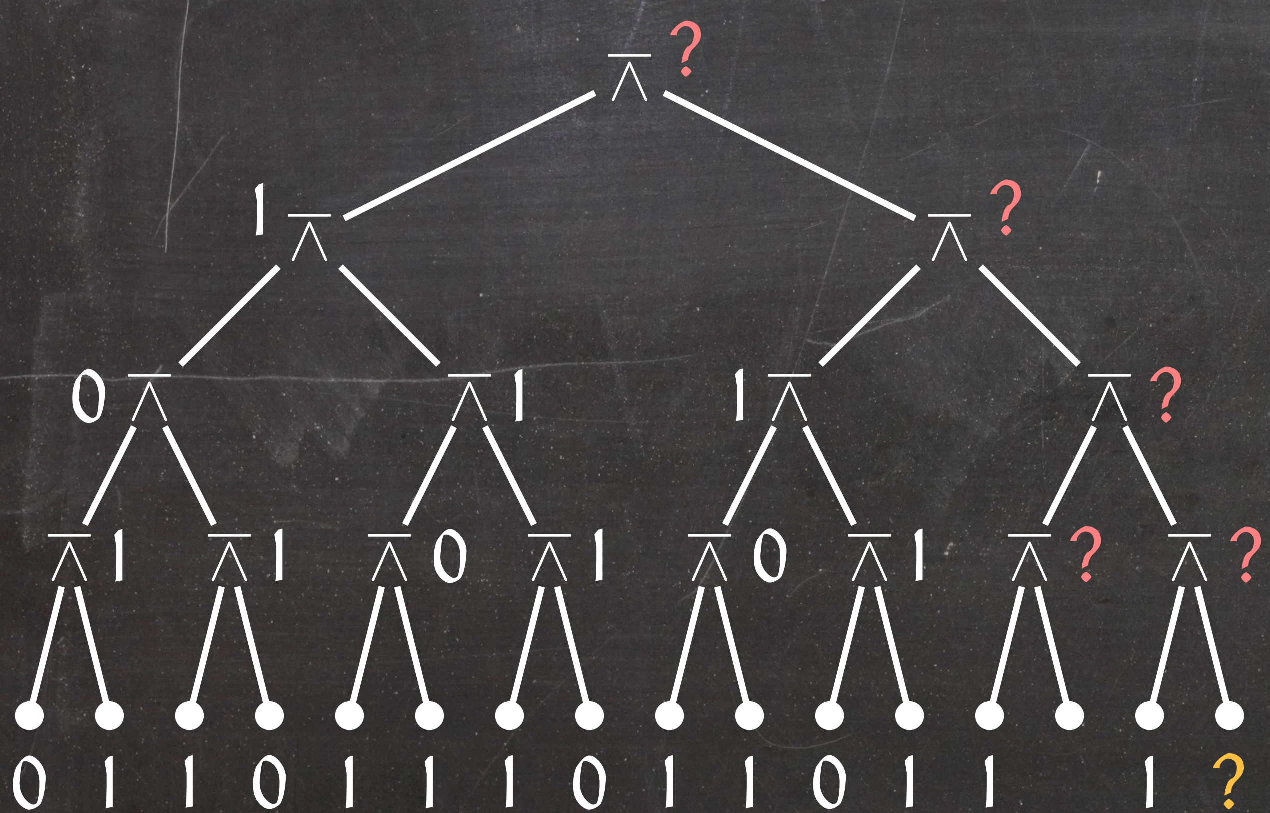
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

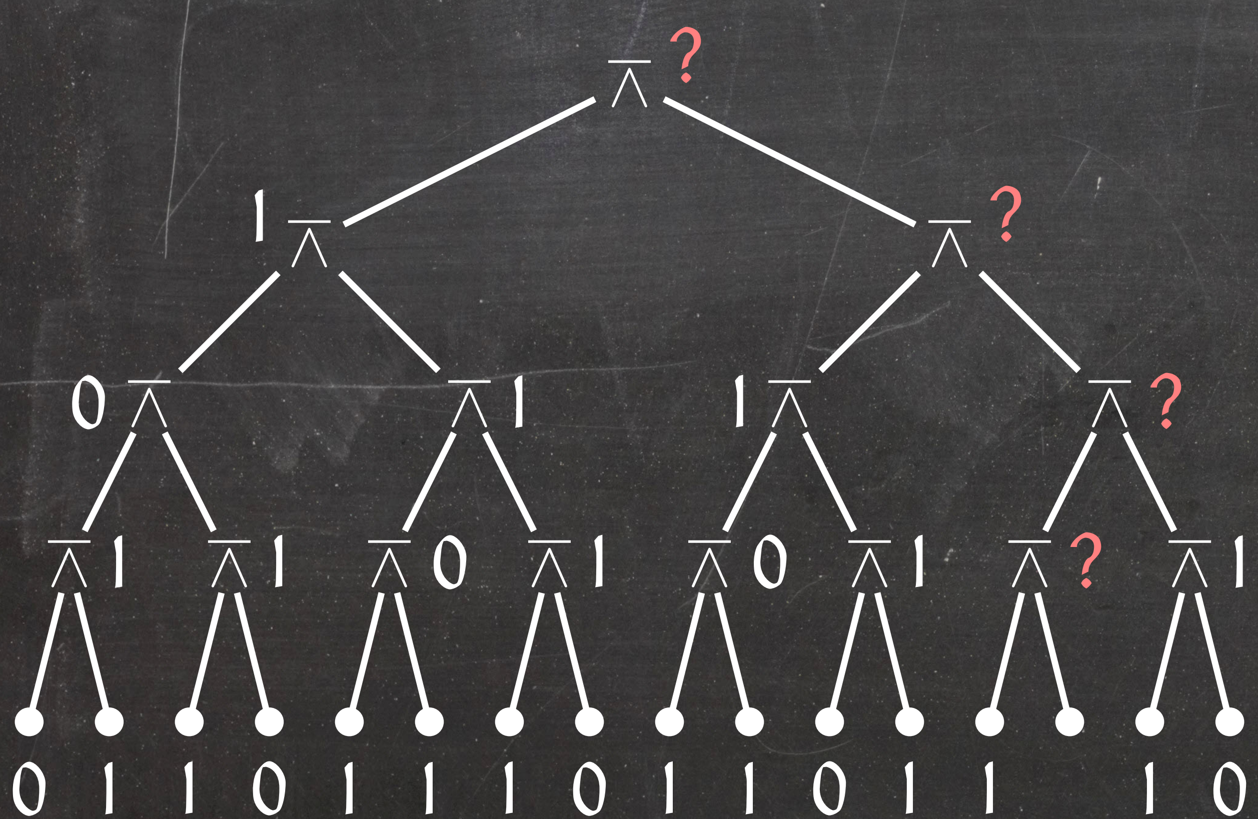
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

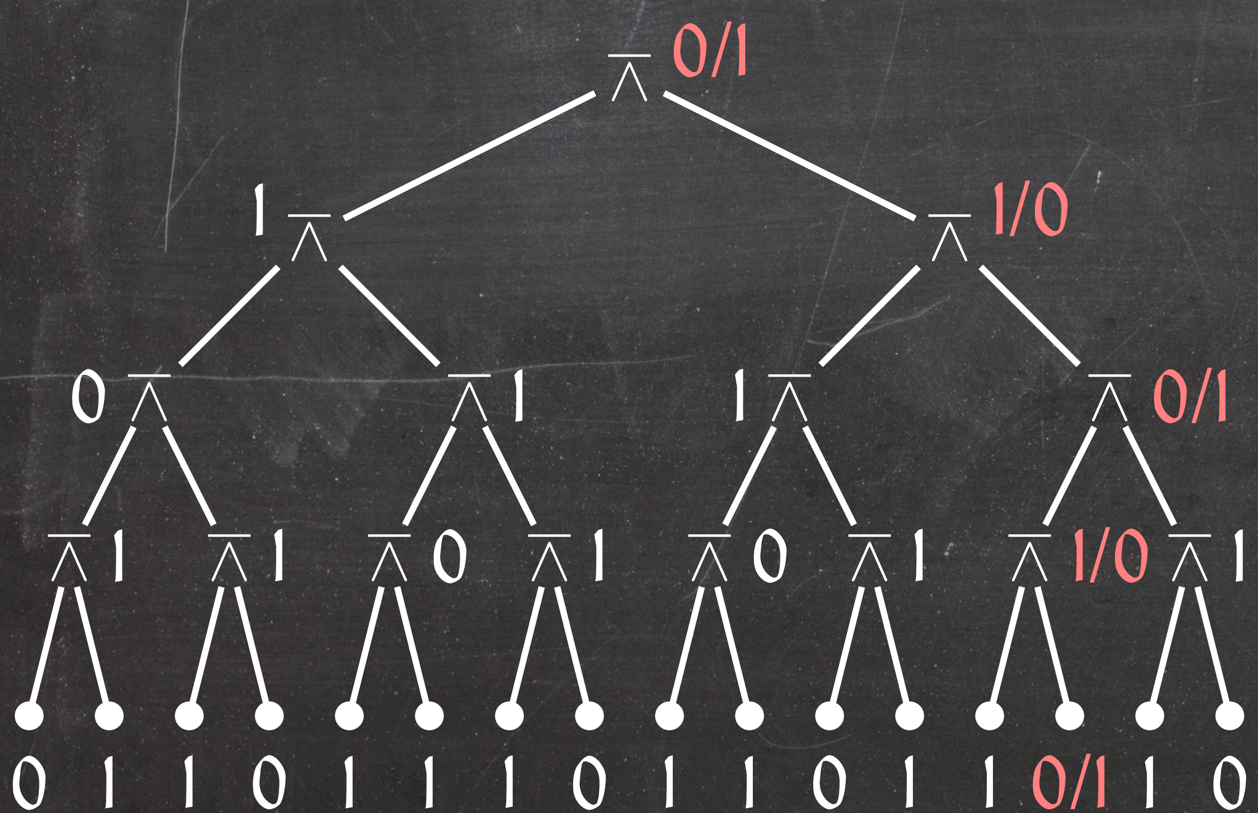
Game Tree Evaluation: A Lower Bound

Observation: Any deterministic algorithm has to inspect every leaf in the worst case and thus takes $\Omega(n)$ time in the worst case.

Adversary argument:

Can be used to construct a worst-case input for any deterministic algorithm, based on how the algorithm behaves.

- Fix every input element the first time the algorithm inspects it.
- Choose this to ensure the algorithm runs as long as possible.



When a leaf is the last unknown leaf in a subtree, we cannot prevent the algorithm from learning the value of the root of the subtree.

Choose the leaf value so the algorithm doesn't learn more than that.

Game Tree Evaluation: Randomized Algorithm

RandomizedGameValue(v)

```
if v is a leaf
    then return its value
coinFlip = RandomNumber(0, 1)
if coinFlip = 1
    then first  = v.leftChild
         second = v.rightChild
    else first  = v.rightChild
         second = v.leftChild
if not f = GameValue(first)
    then return 1
    else return not GameValue(second)
```

Game Tree Evaluation: Randomized Algorithm

RandomizedGameValue(v)

```
if v is a leaf
    then return its value
coinFlip = RandomNumber(0, 1)
if coinFlip = 1
    then first     = v.leftChild
         second = v.rightChild
    else first     = v.rightChild
         second = v.leftChild
if not f = GameValue(first)
    then return 1
    else return not GameValue(second)
```

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0, 1\})$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0, 1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0, 1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

$$E_1[T(n)] \leq \frac{1}{2} \cdot E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot \left(E_1\left[T\left(\frac{n}{2}\right)\right] + E_0\left[T\left(\frac{n}{2}\right)\right]\right) + O(1)$$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0, 1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

$$\begin{aligned} E_1[T(n)] &\le \frac{1}{2} \cdot E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot \left(E_1\left[T\left(\frac{n}{2}\right)\right] + E_0\left[T\left(\frac{n}{2}\right)\right]\right) + O(1) \\ &= E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \end{aligned}$$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0,1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

$$\begin{aligned}E_1[T(n)] &\le \frac{1}{2} \cdot E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot \left(E_1\left[T\left(\frac{n}{2}\right)\right] + E_0\left[T\left(\frac{n}{2}\right)\right]\right) + O(1) \\ &= E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \\ &= 2 \cdot E_1\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)\end{aligned}$$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\quad (i \in \{0, 1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

$$\begin{aligned} E_1[T(n)] &\le \frac{1}{2} \cdot E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot \left(E_1\left[T\left(\frac{n}{2}\right)\right] + E_0\left[T\left(\frac{n}{2}\right)\right]\right) + O(1) \\ &= E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \\ &= 2 \cdot E_1\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \end{aligned}$$

$$E[T(n)] \le \max\left(2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right], E_1[T(n)]\right)$$

Game Tree Evaluation: Randomized Algorithm

Lemma: The expected running time of RandomizedGameValue on any input is in $O(n^{0.754})$.

$E_i[T(n)]$ = expected running time on n leaves if the result is i $\qquad (i \in \{0, 1\})$

$$E_0[T(n)] = 2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1)$$

$$\begin{aligned} E_1[T(n)] &\le \frac{1}{2} \cdot E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot \left(E_1\left[T\left(\frac{n}{2}\right)\right] + E_0\left[T\left(\frac{n}{2}\right)\right]\right) + O(1) \\ &= E_0\left[T\left(\frac{n}{2}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \\ &= 2 \cdot E_1\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_1\left[T\left(\frac{n}{2}\right)\right] + O(1) \end{aligned}$$

$$E[T(n)] \le \max\left(2 \cdot E_1\left[T\left(\frac{n}{2}\right)\right], E_1[T(n)]\right)$$

$$E_1[T(n)] \in O(n^{0.754}) \Rightarrow E[T(n)] \in O(n^{0.754})$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Base case: $1 \le n < 2$.

$T(n) \in O(1) \Rightarrow E_I[T(n)] \le cn^\alpha - d$ for any d and c sufficiently larger than d.

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$E_I[T(n)] \le 2 \cdot E_I\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_I\left[T\left(\frac{n}{2}\right)\right] + a$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$\begin{aligned} E_I[T(n)] &\le 2 \cdot E_I\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_I\left[T\left(\frac{n}{2}\right)\right] + a \\ &\le 2 \cdot \left[c\left(\frac{n}{4}\right)^\alpha - d\right] + \frac{1}{2} \cdot \left[c\left(\frac{n}{2}\right)^\alpha - d\right] + a \end{aligned}$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_l[T(n)] \leq cn^\alpha - d$ for some $c > d > 0$ and all $n \geq 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \leq 0.754$.

Inductive step: $n \geq 2$.

$$
\begin{aligned}
E_l[T(n)] &\leq 2 \cdot E_l\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_l\left[T\left(\frac{n}{2}\right)\right] + a \\
&\leq 2 \cdot \left[c\left(\frac{n}{4}\right)^\alpha - d\right] + \frac{1}{2} \cdot \left[c\left(\frac{n}{2}\right)^\alpha - d\right] + a \\
&= cn^\alpha\left(\frac{2}{4^\alpha} + \frac{1}{2 \cdot 2^\alpha}\right) + a - \frac{5d}{2}
\end{aligned}
$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$\begin{aligned}
E_I[T(n)] &\le 2 \cdot E_I\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_I\left[T\left(\frac{n}{2}\right)\right] + a \\
&\le 2 \cdot \left[c\left(\frac{n}{4}\right)^\alpha - d\right] + \frac{1}{2} \cdot \left[c\left(\frac{n}{2}\right)^\alpha - d\right] + a \\
&= cn^\alpha \left(\frac{2}{4^\alpha} + \frac{1}{2 \cdot 2^\alpha}\right) + a - \frac{5d}{2} \\
&\le cn^\alpha \left(\frac{2}{4^\alpha} + \frac{1}{2 \cdot 2^\alpha}\right) - d \quad \forall d \ge \frac{2}{3}a
\end{aligned}$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_l[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$\begin{aligned}
E_l[T(n)] &\le 2 \cdot E_l\left[T\left(\frac{n}{4}\right)\right] + \frac{1}{2} \cdot E_l\left[T\left(\frac{n}{2}\right)\right] + a \\
&\le 2 \cdot \left[c\left(\frac{n}{4}\right)^\alpha - d\right] + \frac{1}{2} \cdot \left[c\left(\frac{n}{2}\right)^\alpha - d\right] + a \\
&= cn^\alpha\left(\frac{2}{4^\alpha} + \frac{1}{2 \cdot 2^\alpha}\right) + a - \frac{5d}{2} \\
&\le cn^\alpha\left(\frac{2}{4^\alpha} + \frac{1}{2 \cdot 2^\alpha}\right) - d \quad \forall d \ge \frac{2}{3}a \\
&= cn^\alpha\left(\frac{2}{\left(\frac{1+\sqrt{33}}{4}\right)^2} + \frac{1}{2 \cdot \frac{1+\sqrt{33}}{4}}\right) - d
\end{aligned}$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$E_I[T(n)] \le cn^\alpha \left(\frac{2}{\left(\frac{1+\sqrt{33}}{4}\right)^2} + \frac{1}{2 \cdot \frac{1+\sqrt{33}}{4}} \right) - d$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$E_I[T(n)] \le cn^\alpha \left(\frac{2}{\left(\frac{1+\sqrt{33}}{4}\right)^2} + \frac{1}{2 \cdot \frac{1+\sqrt{33}}{4}} \right) - d$$

$$= cn^\alpha \left(\frac{32 + 2 \cdot (1 + \sqrt{33})}{(1 + \sqrt{33})^2} \right) - d$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$E_I[T(n)] \le cn^\alpha \left(\frac{2}{\left(\frac{1+\sqrt{33}}{4}\right)^2} + \frac{1}{2 \cdot \frac{1+\sqrt{33}}{4}} \right) - d$$

$$= cn^\alpha \left(\frac{32 + 2 \cdot (1 + \sqrt{33})}{(1 + \sqrt{33})^2} \right) - d$$

$$= cn^\alpha \left(\frac{34 + 2 \cdot \sqrt{33}}{34 + 2 \cdot \sqrt{33}} \right) - d$$

Game Tree Evaluation: Randomized Algorithm

Claim: $E_I[T(n)] \le cn^\alpha - d$ for some $c > d > 0$ and all $n \ge 1$, where $\alpha = \lg\left(\frac{1+\sqrt{33}}{4}\right) \le 0.754$.

Inductive step: $n \ge 2$.

$$E_I[T(n)] \le cn^\alpha \left(\frac{2}{\left(\frac{1+\sqrt{33}}{4}\right)^2} + \frac{1}{2 \cdot \frac{1+\sqrt{33}}{4}}\right) - d$$

$$= cn^\alpha \left(\frac{32 + 2 \cdot (1+\sqrt{33})}{(1+\sqrt{33})^2}\right) - d$$

$$= cn^\alpha \left(\frac{34 + 2 \cdot \sqrt{33}}{34 + 2 \cdot \sqrt{33}}\right) - d$$

$$= cn^\alpha - d$$

Summary

Algorithms that are fast on average are often easier to design and faster in practice than worst-case efficient algorithms.

In some applications, worst-case guarantees matter!

Average-case analysis provides a valid performance prediction only if the inputs are uniformly distributed.

Randomized algorithms remove this dependence on the input distribution (but rely on a good random number generator).

There are problems where randomized algorithms are provably faster than the best possible deterministic algorithm.