CSI - 3105 Design & Analysis of Algorithms
Course 23

Jean-Lou De Carufel

Fall 2019

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

Can I get another edge from vertex v (please)?

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

Can I get another edge from vertex v (please)?

What is (are) the possible answer(s)?

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

Can I get another edge from vertex v (please)?

What is (are) the possible answer(s)?

- Here is another edge from v.

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

Can I get another edge from vertex v (please)?

What is (are) the possible answer(s)?

- Here is another edge from v.
- There is no more edge from v.

Testing Connectivity

Adjacency lists representation

If we take the adjacency list representation, the running time for testing the connectivity of an undirected graph $G = (V, E)$ is $O(|V| + |E|)$.

In this case, what corresponds to a single operation? What is the question the algorithm can ask to the adversary?

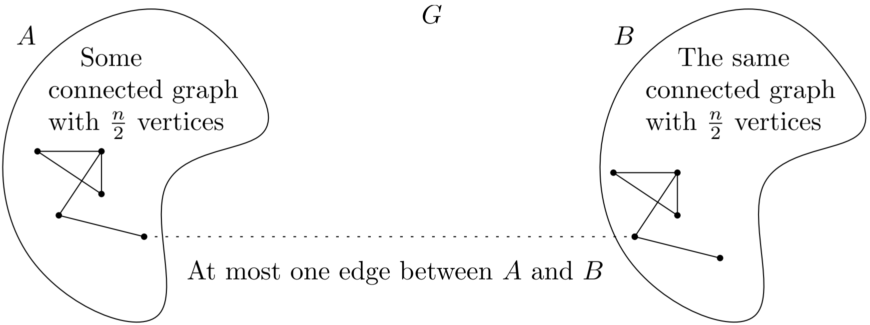
Can I get another edge from vertex v (please)?

What is (are) the possible answer(s)?

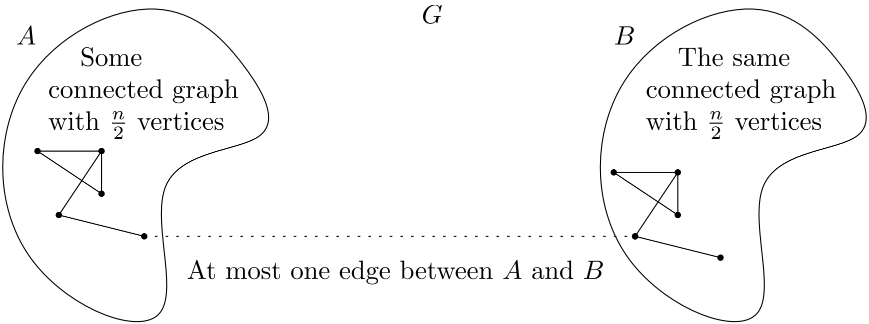
- Here is another edge from v.
- There is no more edge from v.

Remember that in the adjacency lists representation, each edge $\{u, v\}$ is stored twice: once in the list of u and once in the list of v.

Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.

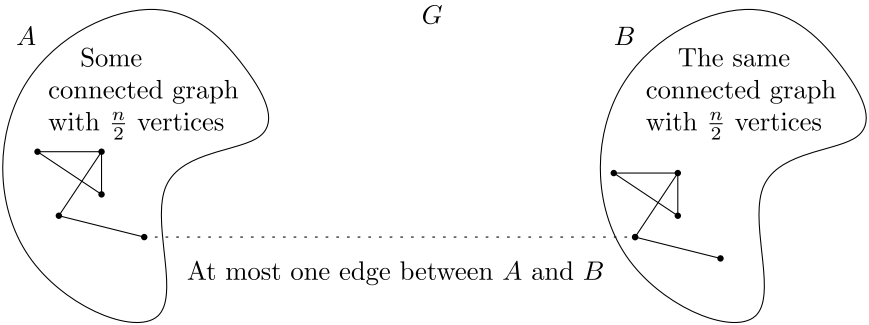


Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.

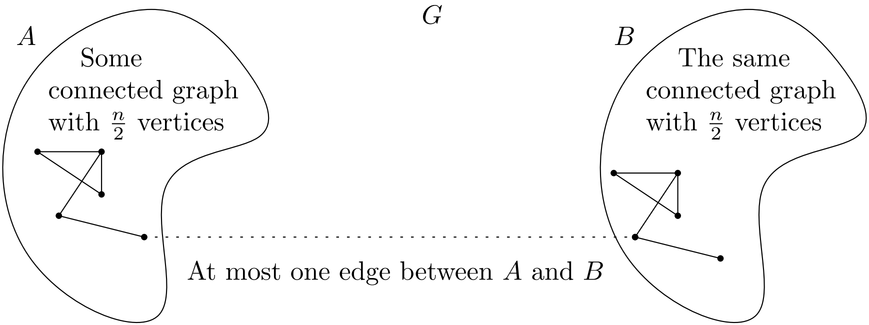


To test whether or not G is connected, we need to ask if $\{u, v\}$ is an edge for all $u \in A$ and all $v \in B$. Otherwise, we might miss the only edge between A and B. So for each vertex $u \in A$, we want to know if there is an edge to B.

Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.

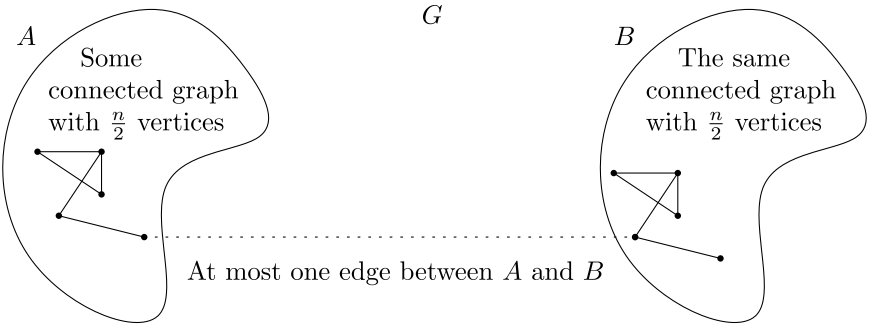


Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.



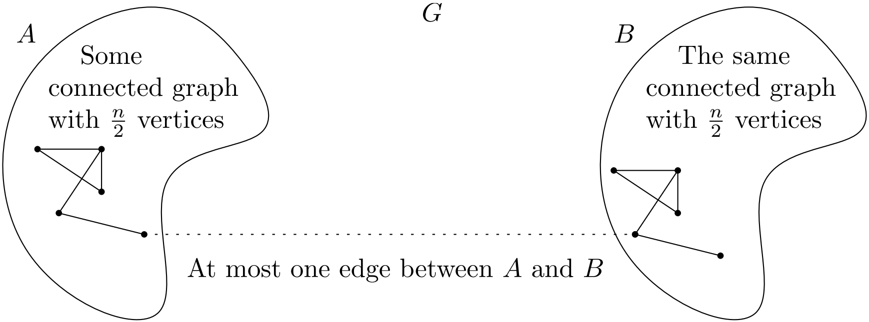
But of course, for each vertex u in A, the adversary will first announce all edges from u that are **within** A before saying whether or not u is connected to B.

Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.

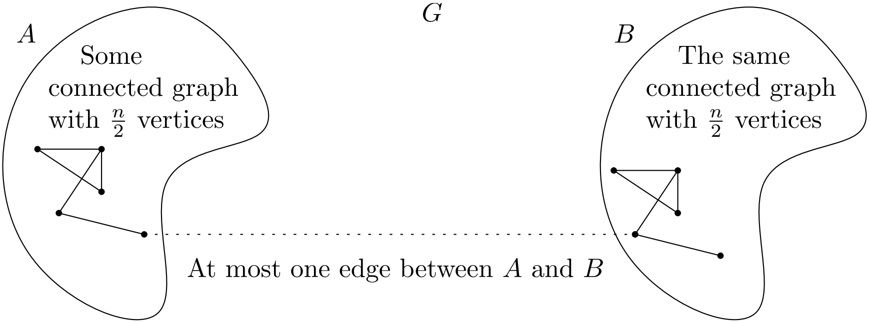


But of course, for each vertex u in A, the adversary will first announce all edges from u that are **within** A before saying whether or not u is connected to B. Also, since each edge is stored twice, the adversary will first announce the edges that are known by the algorithm before giving out a new one. So for each vertex $u \in A$, we need $\deg(u) + 1$ operations to get our answer.

Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.

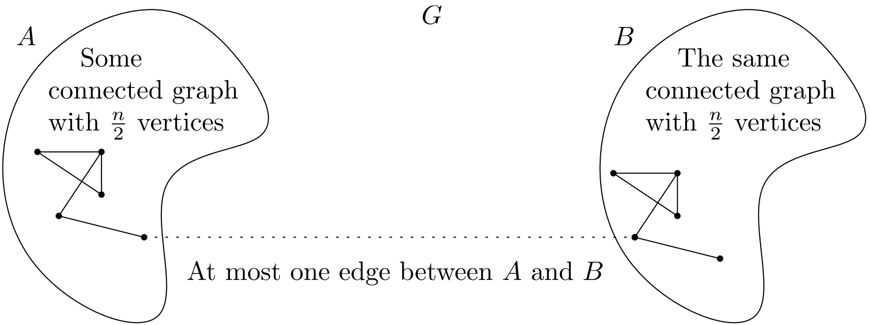


Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.



Moreover, if the algorithm does not even know about all the edges in A, then there is no way the algorithm can be correct. So if the adversary allow $(m' - 1) + \left(\frac{1}{2}n - 1\right)$ questions by the algorithm.

Let $n = |V|$ and $m = |E|$, where $m = 2m' + 1$. The adversary has the following graph in mind.



Moreover, if the algorithm does not even know about all the edges in A, then there is no way the algorithm can be correct. So if the adversary allow $(m' - 1) + \left(\frac{1}{2}n - 1\right)$ questions by the algorithm. So in total, an algorithm needs at least

$$(m' - 1) + \left(\frac{1}{2}n - 1\right) + 1 = \Omega(|V| + |E|) \text{ operations.}$$

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to:

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i,j]$?

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps.

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps. The best known algorithm takes $O(n^{2.38})$ time (Coppersmith & Winograd 1990). Therefore, we need to

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps. The best known algorithm takes $O(n^{2.38})$ time (Coppersmith & Winograd 1990). Therefore, we need to

- Find a faster algorithm,

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps. The best known algorithm takes $O(n^{2.38})$ time (Coppersmith & Winograd 1990). Therefore, we need to

- Find a faster algorithm,
- find a better lower bound argument

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps. The best known algorithm takes $O(n^{2.38})$ time (Coppersmith & Winograd 1990). Therefore, we need to

- Find a faster algorithm, <2.38
- find a better lower bound argument >=2.38
- or both! ,

Matrix Multiplication

Input: Two $n \times n$-matrices A and B.

Output: $C = AB$

Let us say that one operation corresponds to: asking what is $C[i, j]$?

Since C is made of n^2 numbers, we need at least n^2 steps to build the output. So it takes $\Omega(n^2)$ steps. The best known algorithm takes $O(n^{2.38})$ time (Coppersmith & Winograd 1990). Therefore, we need to

- Find a faster algorithm,
- find a better lower bound argument
- or both!

Good luck!!!

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

If we argue that we need to build an output of size n (where one operations corresponds to asking what is $output[i]$?), we get a lower bound of $\Omega(n)$ time.

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

If we argue that we need to build an output of size n (where one operations corresponds to asking what is $output[i]$?), we get a lower bound of $\Omega(n)$ time.

The best known algorithm (Merge sort for instance) takes $O(n \log(n))$ time. Therefore, we need to

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

If we argue that we need to build an output of size n (where one operations corresponds to asking what is $output[i]$?), we get a lower bound of $\Omega(n)$ time.

The best known algorithm (Merge sort for instance) takes $O(n \log(n))$ time. Therefore, we need to

- Find a faster algorithm,

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

If we argue that we need to build an output of size n (where one operations corresponds to asking what is $output[i]$?), we get a lower bound of $\Omega(n)$ time.

The best known algorithm (Merge sort for instance) takes $O(n \log(n))$ time. Therefore, we need to

- Find a faster algorithm,
- find a better lower bound argument

Sorting

Consider the problem of sorting an array $A[1..n]$ of n numbers.

If we argue that we need to know all numbers in A (where one operations corresponds to asking what is $A[i]$?), we get a lower bound of $\Omega(n)$ time.

If we argue that we need to build an output of size n (where one operations corresponds to asking what is $output[i]$?), we get a lower bound of $\Omega(n)$ time.

The best known algorithm (Merge sort for instance) takes $O(n \log(n))$ time. Therefore, we need to

- Find a faster algorithm,
- find a better lower bound argument
- or both?

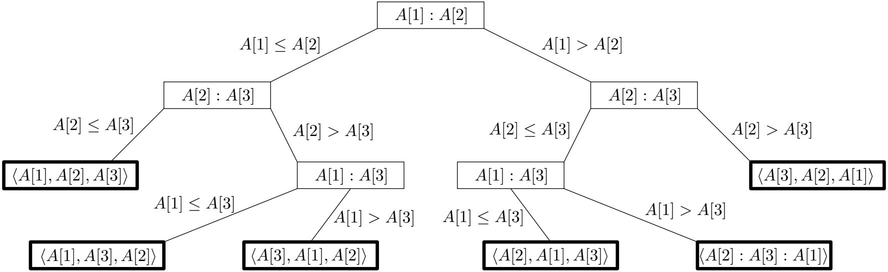
§7.2 Decision Trees and Lower Bounds

Let us focus on problems which can be solved using only comparisons: $<$, $\leq$, $=$, $>$ or $\geq$.

- Sorting an array of numbers.
- Finding an element in an array.
- Decide whether or not all elements in an array are different.
- Find the maximum element in an array.
- etc.

The Sorting Problem

The Sorting Problem



The Decision Tree Model

Definition (Decision Tree)

A *decision tree* is a binary tree $\mathcal{T}$ defined as follows:

- $\mathcal{T}$ has a finite number of nodes,
- the label of each internal node of $\mathcal{T}$ is of the form $A[i] : A[j]$,
- from each internal node of $\mathcal{T}$, there are two outgoing edges:
  - the label of the left outgoing edge is $\leq$
  - the label of the right outgoing edge is $>$.

The Decision Tree Model

Definition (Decision Tree)

A *decision tree* is a binary tree $\mathcal{T}$ defined as follows:

- $\mathcal{T}$ has a finite number of nodes,
- the label of each internal node of $\mathcal{T}$ is of the form $A[i] : A[j]$,
- from each internal node of $\mathcal{T}$, there are two outgoing edges:
  - the label of the left outgoing edge is $\leq$
  - the label of the right outgoing edge is $>$.

To each decision tree $\mathcal{T}$ corresponds an algorithm $\mathcal{A}_{\mathcal{T}}$. Conversely, to each algorithm $\mathcal{A}$ which uses only comparisons ($<$, $\leq$, $=$, $>$ or $\geq$) corresponds a decision tree $\mathcal{T}_{\mathcal{A}}$.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ *time.*

PROOF:

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ *time.*

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$. To sort A, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$. To sort A, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

There are at least $n!$ leaves in $\mathcal{T}$, one for each possible output. If we choose the input carefully, we can reach any of these $n!$ leaves.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ *time.*

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$. To sort A, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

There are at least $n!$ leaves in $\mathcal{T}$, one for each possible output. If we choose the input carefully, we can reach any of these $n!$ leaves. The height of $\mathcal{T}$ is then at least $\log(n!)$.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$. To sort A, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

There are at least $n!$ leaves in $\mathcal{T}$, one for each possible output. If we choose the input carefully, we can reach any of these $n!$ leaves. The height of $\mathcal{T}$ is then at least $\log(n!)$.

We now need to find a lower bound on $\log(n!)$.

Theorem

In the decision tree model, sorting an array of n numbers takes $\Omega(n \log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ solves the sorting problem on an array $A[1..n]$. To sort A, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

There are at least $n!$ leaves in $\mathcal{T}$, one for each possible output. If we choose the input carefully, we can reach any of these $n!$ leaves. The height of $\mathcal{T}$ is then at least $\log(n!)$.

We now need to find a lower bound on $\log(n!)$. Let us use the Stirling formula:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\,\left(\frac{n}{e}\right)^n} = 1.$$

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(n/e)^n} = 1$$

From the limit criterion, we get

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(n/e)^n} = 1$$

From the limit criterion, we get

$$n! = \Omega\left(\sqrt{2\pi n}(n/e)^n\right) \quad ,$$

from which

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(n/e)^n} = 1$$

From the limit criterion, we get

$$n! = \Omega\left(\sqrt{2\pi n}(n/e)^n\right) ,$$

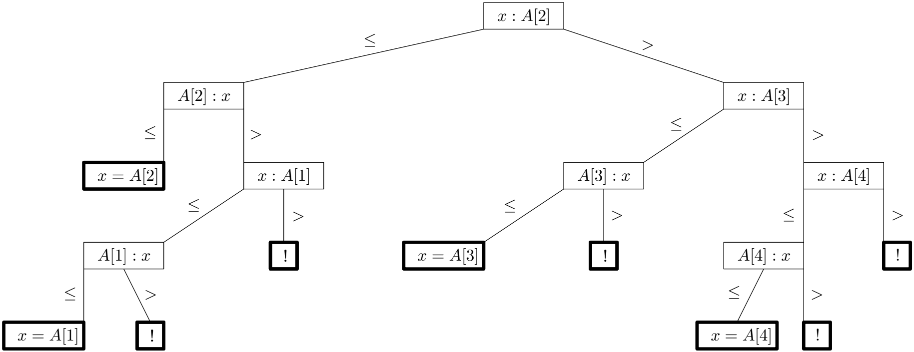
from which

$$\begin{aligned}
\log(n!) &= \Omega\left(\log\left(\sqrt{2\pi n}(n/e)^n\right)\right) \\
&= \Omega\left(\log\left(\sqrt{2\pi}\right) + \log\left(\sqrt{n}\right) + n\log(n) - n\log(e)\right) \\
&= \Omega(n\log(n)) .
\end{aligned}$$

$\square$

Find an Element x in a Sorted Array

Find an Element x in a Sorted Array



Theorem

In the decision tree model, finding the position of an element x in a sorted array $A[1..n]$ (or detecting that x is not in A) takes $\Omega(\log(n))$ time.

PROOF:

Theorem

In the decision tree model, finding the position of an element x in a sorted array $A[1..n]$ (or detecting that x is not in A) takes $\Omega(\log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ finds the position of x in $A[1..n]$ (or say that it is not in A).

Theorem

In the decision tree model, finding the position of an element x in a sorted array $A[1..n]$ (or detecting that x is not in A) takes $\Omega(\log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_\mathcal{T}$ finds the position of x in $A[1..n]$ (or say that it is not in A). To solve this problem, $\mathcal{A}_\mathcal{T}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

Theorem

In the decision tree model, finding the position of an element x in a sorted array $A[1..n]$ (or detecting that x is not in A) takes $\Omega(\log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ finds the position of x in $A[1..n]$ (or say that it is not in A). To solve this problem, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.
There are at least $n+1$ leaves in $\mathcal{T}$. Indeed, there are n possible positions for x in A, and we must also count the possibility of x not being in A. If we choose the input carefully, we can reach any of these $n+1$ leaves.

Theorem

In the decision tree model, finding the position of an element x in a sorted array $A[1..n]$ (or detecting that x is not in A) takes $\Omega(\log(n))$ time.

PROOF: Let $\mathcal{T}$ be a decision tree such that the corresponding algorithm $\mathcal{A}_{\mathcal{T}}$ finds the position of x in $A[1..n]$ (or say that it is not in A). To solve this problem, $\mathcal{A}_{\mathcal{T}}$ must traverse $\mathcal{T}$ from the root to a leaf. Each time the algorithm moves from a node to one of its children, it executes one operation. Therefore, we need to bound the height of $\mathcal{T}$.

There are at least $n+1$ leaves in $\mathcal{T}$. Indeed, there are n possible positions for x in A, and we must also count the possibility of x not being in A. If we choose the input carefully, we can reach any of these $n+1$ leaves. The height of $\mathcal{T}$ is then at least $\log(n+1) = \Omega(\log(n))$. $\square$

In fact, we can show that the number of leaves is always at least $2n$. We can argue that for all decision trees (which solve this problem), for each position i in A, we have

- a leaf for the case $x = A[i]$
- and a leaf for the following case: the only place where x can be is $A[i]$, but it is not there.

But at the end, we still get a lower bound of $\log(2n) = \Omega(\log(n))$.

Find an Element x in an Unsorted Array

What if we try to find a lower bound for the case where the array is not necessarily sorted?

Merging Two Sorted Arrays

Let $A[1..n]$ and $B[1..n]$ be two sorted arrays. We know how to merge A and B into one single sorted array in $O(n)$ time. We need such an algorithm during the merge step of merge sort.

Merging Two Sorted Arrays

Let $A[1..n]$ and $B[1..n]$ be two sorted arrays. We know how to merge A and B into one single sorted array in $O(n)$ time. We need such an algorithm during the merge step of merge sort.
Can we do better?

Merging Two Sorted Arrays

No.

Merging Two Sorted Arrays

No.
First, notice that it is possible to solve this problems using only comparisons ($<$, $\leq$, $=$, $>$ or $\geq$). Hence, we can solve this problem with decision trees.

Merging Two Sorted Arrays

No.
First, notice that it is possible to solve this problems using only comparisons ($<$, $\leq$, $=$, $>$ or $\geq$). Hence, we can solve this problem with decision trees.
Given two arrays $A[1..n]$ and $B[1..n]$, how many different outputs are there?

For instance, if $n = 2$, we have $A[1] < A[2]$ and $B[1] < B[2]$, from which the possible outputs are as follows.

$$
\begin{gathered}
A[1], A[2], B[1], B[2] \\
A[1], B[1], A[2], B[2] \\
A[1], B[1], B[2], A[2] \\
B[1], A[1], A[2], B[2] \\
B[1], A[1], B[2], A[2] \\
B[1], B[2], A[1], A[2]
\end{gathered}
$$

For a general n, how many different outputs are there?

For a general n, how many different outputs are there? We have to fill the $2n$ positions of the output and make sure that the order in A is satisfied: $A[1] < A[2] < ... < A[n]$ as well as the order in B: $B[1] < B[2] < ... < B[n]$.

For a general n, how many different outputs are there? We have to fill the $2n$ positions of the output and make sure that the order in A is satisfied: $A[1] < A[2] < ... < A[n]$ as well as the order in B: $B[1] < B[2] < ... < B[n]$. One way to achieve this is to choose the n positions that will contain the elements of A and then fill the n remaining positions with the elements of B.

For a general n, how many different outputs are there? We have to fill the $2n$ positions of the output and make sure that the order in A is satisfied: $A[1] < A[2] < ... < A[n]$ as well as the order in B: $B[1] < B[2] < ... < B[n]$. One way to achieve this is to choose the n positions that will contain the elements of A and then fill the n remaining positions with the elements of B. How many ways are there to choose n positions among $2n$ positions?

For a general n, how many different outputs are there? We have to fill the $2n$ positions of the output and make sure that the order in A is satisfied: $A[1] < A[2] < ... < A[n]$ as well as the order in B: $B[1] < B[2] < ... < B[n]$. One way to achieve this is to choose the n positions that will contain the elements of A and then fill the n remaining positions with the elements of B. How many ways are there to choose n positions among $2n$ positions? There are $\binom{2n}{n}$ ways.

For a general n, how many different outputs are there? We have to fill the $2n$ positions of the output and make sure that the order in A is satisfied: $A[1] < A[2] < ... < A[n]$ as well as the order in B: $B[1] < B[2] < ... < B[n]$. One way to achieve this is to choose the n positions that will contain the elements of A and then fill the n remaining positions with the elements of B. How many ways are there to choose n positions among $2n$ positions? There are $\binom{2n}{n}$ ways. Therefore, there are at least $\binom{2n}{n}$ leaves in the decision tree. Therefore, the height of the tree is at least $\log\left(\binom{2n}{n}\right)$.

From the Stirling formula, we have

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n .$$

From the Stirling formula, we have

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n .$$

Hence,

$$\begin{aligned}
\binom{2n}{n} &= \frac{(2n)!}{(n!)^2} \\
&\sim \frac{\sqrt{2\pi(2n)}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^2} && \text{Stirling} \\
&= \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} \\
&= \frac{4^n}{\sqrt{\pi n}}
\end{aligned}$$

from which we get

$$\begin{aligned}
\log\left(\binom{2n}{n}\right) &= \Omega\left(\log\left(\frac{4^n}{\sqrt{\pi n}}\right)\right) \\
&= \Omega\left(n\log(4) - \log\left(\sqrt{\pi}\right) - \log(n)\right) \\
&= \Omega(n).
\end{aligned}$$

from which we get

$$\begin{aligned}\log\left(\binom{2n}{n}\right) &= \Omega\left(\log\left(\frac{4^n}{\sqrt{\pi n}}\right)\right)\\ &= \Omega\left(n\log(4) - \log\left(\sqrt{\pi}\right) - \log(n)\right)\\ &= \Omega(n).\end{aligned}$$

Notice that an adversarial argument would be much simpler. Do you see how to proceed?

Conclusion