Pre-lecture brain teaser

We talked a lot about languages representing problems. Consider the problem of adding two numbers. What language class does it belong to?
We talked a lot about languages representing problems. Consider the problem of adding two numbers. What language class does it belong to?
Pre-lecture brain teaser

Let’s say we are adding two unary numbers.

\[ 3 + 4 = 7 \rightarrow 111 + 1111 = 1111111 \]  (1)

Seems like we can make a PDA that considers

\[
\begin{align*}
1, \varepsilon & \rightarrow 1 \\
1, \varepsilon & \rightarrow 1 \\
1, 1 & \rightarrow \varepsilon
\end{align*}
\]

Start \rightarrow q_1 \rightarrow \varepsilon, \varepsilon \rightarrow \$
\rightarrow q_2 \rightarrow +, \varepsilon \rightarrow \varepsilon \rightarrow q_3 \rightarrow =, \varepsilon \rightarrow \varepsilon \rightarrow q_4 \rightarrow \varepsilon, \$ \rightarrow \varepsilon \rightarrow q_5

context-free!
Pre-lecture brain teaser

What if we wanted add **two binary numbers**?

\[ 3 + 4 = 7 \rightarrow 11 + 100 = 111 \]  

At least **context-sensitive** b/c we can build a finite **Turing machine** which takes in the encoding

\[
\begin{array}{cccccc}
\rightarrow & 1 & 1 & + & 1 & 0 & 0 & = & 1 & 1 & 1 & 1 & \leftarrow \\
\end{array}
\]
Pre-lecture brain teaser

What if we wanted add two binary numbers?

\[ 3 + 4 = 7 \rightarrow 11 + 100 = 111 \]  

(3)

Computes value on left hand side
Pre-lecture brain teaser

What if we wanted add two binary numbers?

\[ 3 + 4 = 7 \rightarrow 11 + 100 = 111 \]  \hspace{1cm} (4)

And compares it to the value on the right..

\[ \begin{array}{ccccccc}
\text{\textgreater} & 1 & 1 & + & 1 & 1 & 1 \\
\text{\textless} & 1 & 1 & 1 & = & 1 & 1 & 1 & 1
\end{array} \]

\( q_1 \)
New Course Section: Introductory algorithms
Learning Objectives

At the end of the lecture, you should be able to understand

- the idea of an algorithm and algorithmic problems,
- how to reduce a problem into another,
- the design and analysis of recursive algorithms, and
- some example recursive algorithms for sorting and searching.
Brief intro to the Random Access Machine (RAM) model
• Algorithm solves a specific problem.
• Steps/instructions of an algorithm are simple/primitive and can be executed mechanically.
• Algorithm has a finite description; same description for all instances of the problem
• Algorithm implicitly may have state/memory

A computer is a device that

• implements the primitive instructions
• allows for an automated implementation of the entire algorithm by keeping track of state
Models of Computation vs Computers

• Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
• Computer: an actual physical device that implements a very specific model of computation

In this course: design algorithms in a high-level model of computation.

Question: What model of computation will we use to design algorithms?
Models of Computation vs Computers

- Model of Computation: an idealized mathematical construct that describes the primitive instructions and other details
- Computer: an actual physical device that implements a very specific model of computation

In this course: design algorithms in a high-level model of computation.

Question: What model of computation will we use to design algorithms?

The standard programming model that you are used to in programming languages such as Java/C++. We have already seen the Turing Machine model.
Informal description:

- Basic data type is an integer number.
- Numbers in input fit in a word.
- Arithmetic/comparison operations on words take constant time.
- Arrays allow random access (constant time to access $A[i]$).
- Pointer based data structures via storing addresses in a word.
Example

**Sorting**: input is an array of \( n \) numbers

- input size is \( n \) (ignore the bits in each number),
- comparing two numbers takes \( O(1) \) time,
- random access to array elements,
- addition of indices takes constant time,
- basic arithmetic operations take constant time,
- reading/writing one word from/to memory takes constant time.

We will usually do not allow (or be careful about allowing):

- bitwise operations (and, or, xor, shift, etc).
- floor function.
- limit word size (usually assume unbounded word size).
What is an algorithmic problem?
What is an algorithmic problem?

An algorithmic problem is simply to compute a function \( f : \sum^* \rightarrow \sum^* \) over strings of a finite alphabet.

Algorithm \( A \) solves \( f \) if for all input strings \( w \), \( A \) outputs \( f(w) \).

E.g. Adding two numbers.

\[
\text{num1} + \text{num2} = \text{num3}.
\]

\[
f(\text{num1}, \text{num2}) = \text{num3}
\]
Types of Problems

We will broadly see three types of problems.

• **Decision Problem**: Is the input a **YES** or **NO** input?
  Example: Given graph $G$, nodes $s$, $t$, is there a path from $s$ to $t$ in $G$?
  Example: Given a CFG grammar $G$ and string $w$, is $w \in L(G)$?

• **Search Problem**: Find a **solution** if input is a **YES** input.
  Example: Given graph $G$, nodes $s$, $t$, find an $s$-$t$ path.

• **Optimization Problem**: Find a **best** solution among all solutions for the input.
  Example: Given graph $G$, nodes $s$, $t$, find a shortest $s$-$t$ path.
Given a problem $P$ and an algorithm $A$ for $P$ we want to know:

- Does $A$ correctly solve problem $P$?
- What is the **asymptotic worst-case running time** of $A$?
- What is the **asymptotic worst-case space** used by $A$?

**Asymptotic running-time analysis:** $A$ runs in $O(f(n))$ time if:

“For all $n$ and for all inputs $I$ of size $n$, $A$ on input $I$ terminates after $O(f(n))$ primitive steps.”

$T(n)$ is said to be $O(f(n))$ if

$$T(n) \leq cf(n) \text{ for some } c > 0$$
E.g. \( T(n) = n^2 \)

Check if \( T(n) = O(n^2) \)!

\[ T(n) \leq c \cdot n^2 \]

Take \( c = 1 \) \( \Rightarrow \) \( T(n) \leq n^2 \); \( T(n) = n^2 \) (Given)

Check if \( T(n) = O(n^5) \)!

\[ T(n) \leq c \cdot n^5 \]

\[ n^2 \leq c \cdot n^5 \quad (?) \quad c = 1 \]

\( \Rightarrow \) \( T(n) = O(n^5) \)
Algorithmic Techniques

- Reduction to known problem/algorithm
- Recursion, divide-and-conquer, dynamic programming
- Graph algorithms to use as basic reductions
- Greedy

Some advanced techniques not covered in this class:

- Combinatorial optimization
- Linear and Convex Programming, more generally continuous optimization method
- Advanced data structure
- Randomization
- Many specialized areas
Reductions
Reduction

Reducing problem $A$ to problem $B$:

- Algorithm for $A$ uses algorithm for $B$ as a black box.
Reducing problem $A$ to problem $B$:

- Algorithm for $A$ uses algorithm for $B$ as a black box.

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.
Reduction

Reducing problem $A$ to problem $B$:

- Algorithm for $A$ uses algorithm for $B$ as a black box.

**Q:** How do you hunt a blue elephant?

**A:** With a blue elephant gun.

**Q:** How do you hunt a red elephant?

**A:** Hold its trunk shut until it turns blue, and then shoot it with the blue elephant gun.
Reducing problem $A$ to problem $B$:

- Algorithm for $A$ uses algorithm for $B$ as a black box.

**Q: How do you hunt a blue elephant?**

**A: With a blue elephant gun.**

($) 100 \text{ (for blue elephant gun)}$

**Q: How do you hunt a red elephant?**

**A: Hold its trunk shut until it turns blue, and then shoot it with the blue elephant gun.**

($) 200 \text{ (for blue elephant gun)}$

($) 100 \text{ (for hunting red elephants)}$

**Q: How do you shoot a white elephant?**

**A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.**

($) 500 \text{ (for hunting red elephants)}$
Problem: Given an array $A$ of $n$ integers, are there any duplicates in $A$?
Problem Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```python
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$ do
        for $j = i + 1$ to $n$ do
                return YES
        return NO
```

Running time: $O(n^2)$
UNIQUENESS: Distinct Elements Problem

**Problem** Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```latex
\text{DistinctElements}(A[1..n])
\begin{verbatim}
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            if (A[i] = A[j])
                return YES
         \end{verbatim}
return NO
```

**Running time:** $\mathcal{O}(n^2)$
UNIQUENESS: Distinct Elements Problem

**Problem** Given an array $A$ of $n$ integers, are there any duplicates in $A$?

Naive algorithm:

```plaintext
DistinctElements(A[1..n])
    for $i = 1$ to $n - 1$ do
        for $j = i + 1$ to $n$ do
                return YES
        return NO
```

**Running time:** $O(n^2)$
Reduction to Sorting

DistinctElements\( (A[1..n]) \)

Sort \( A \)

for \( i = 1 \) to \( n - 1 \) do

if \( (A[i] = A[i + 1]) \) then

return \( \text{YES} \)

return \( \text{NO} \)

Running time: \( O(n) \) plus time to sort an array of \( n \) numbers

Important point: algorithm uses sorting as a black box

Advantage of naive algorithm: works for objects that cannot be "sorted". Can also consider hashing but outside scope of current course.

\[
= O(n) + \text{Time taken in sorting} \\
= O(n) + O(n \log n) \\
= O(n \log n)
\]

\[
O(n^2) \geq O(n \log n) \text{ because } \log n \leq n
\]
Reduction to Sorting

**DistinctElements**($A[1..n]$)

Sort $A$

for $i = 1$ to $n - 1$ do
  if ($A[i] = A[i + 1]$) then
    return YES

return NO

Running time: $O(n)$ plus time to sort an array of $n$ numbers

Important point: algorithm uses sorting as a black box
Reduction to Sorting

\[
\text{DistinctElements}(A[1..n])
\]
\[
\text{Sort } A
\]
\[
\text{for } i = 1 \text{ to } n - 1 \text{ do}
\]
\[
\text{if } (A[i] = A[i + 1]) \text{ then return YES}
\]
\[
\text{return NO}
\]

Running time: \( O(n) \) plus time to sort an array of \( n \) numbers

Important point: algorithm uses sorting as a black box

Advantage of naive algorithm: works for objects that cannot be “sorted”. Can also consider hashing but outside scope of current course.
Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

- **Positive direction:** Algorithm for $B$ implies an algorithm for $A$
- **Negative direction:** Suppose there is no “efficient” algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)
Two sides of Reductions

Suppose problem $A$ reduces to problem $B$

- **Positive direction:** Algorithm for $B$ implies an algorithm for $A$
- **Negative direction:** Suppose there is no “efficient” algorithm for $A$ then it implies no efficient algorithm for $B$ (technical condition for reduction time necessary for this)

\[
O(n^2) \quad O(n \log n)
\]

**Example:** Distinct Elements reduces to Sorting in $O(n)$ time

- An $O(n \log n)$ time algorithm for Sorting implies an $O(n \log n)$ time algorithm for Distinct Elements problem.
- If there is no $o(n \log n)$ time algorithm for Distinct Elements problem then there is no $o(n \log n)$ time algorithm for Sorting.
Recursion as self reductions
Recursion

**Reduction:** reduce one problem to another

**Recursion:** a special case of reduction

- reduce problem to a smaller instance of itself
- self-reduction
**Recursion**

**Reduction**: reduce one problem to another

**Recursion**: a special case of reduction

- reduce problem to a smaller instance of itself
- self-reduction

- Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as base cases
Recursion

- Recursion is a very powerful and fundamental technique
- Basis for several other methods
  - Divide and conquer
  - Dynamic programming
  - Enumeration and branch and bound etc
  - Some classes of greedy algorithms
- Makes proof of correctness easy (via induction)
- Recurrences arise in analysis
Move stack of \( n \) disks from peg 1 to peg 2, one disk at a time.

**Rule:** cannot put a larger disk on a smaller disk.

**Question:** what is a strategy and how many moves does it take?
The Tower of Hanoi algorithm; ignore everything but the bottom disk
Recursive Algorithm

$$\text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}) :$$

if \(n > 0\) then

- \(\text{Hanoi}(n-1, \text{src}, \text{tmp}, \text{dest})\)
- Move disk \(n\) from \(\text{src}\) to \(\text{dest}\)
- \(\text{Hanoi}(n-1, \text{tmp}, \text{dest}, \text{src})\)

$$T(n) = T(n-1) + \text{Move disk} + T(n-1)$$
Recursive Algorithm

\[ \text{Hanoi}(n, \text{src}, \text{dest}, \text{tmp}): \]
\[ \textbf{if } (n > 0) \textbf{ then} \]
\[ \text{Hanoi}(n-1, \text{src}, \text{tmp}, \text{dest}) \]
\[ \text{Move disk } n \text{ from } \text{src} \text{ to } \text{dest} \]
\[ \text{Hanoi}(n-1, \text{tmp}, \text{dest}, \text{src}) \]

\( T(n) \): time to move \( n \) disks via recursive strategy
Recursive Algorithm

Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n - 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n - 1, tmp, dest, src)

T(n): time to move n disks via recursive strategy

\[ T(n) = 2T(n - 1) + 1 \quad n > 1 \quad \text{and} \quad T(1) = 1 \]
\[ T(n) = 2T(n-1) + 1 \]
\[ = 2^2 T(n-2) + 2 + 1 \]
\[ = \ldots \]
\[ = 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \]
\[ = \ldots \]
\[ = 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \]
\[ = 2^{n-1} + 2^{n-2} + \ldots + 1 \]
\[ = \frac{(2^n - 1)}{(2 - 1)} = 2^n - 1 \]

\[ T(n) = O(2^n) \]
Merge Sort
Input  Given an array of $n$ elements

Goal  Rearrange them in ascending order
1. **Input**: Array $A[1 \ldots n]$
1. **Input:** Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$
MergeSort

1. **Input:** Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m + 1 \ldots n]$
1. **Input:** Array \(A[1 \ldots n]\)

   \[ A \ L \ G \ O \ R \ I \ T \ H \ M \ S \]

2. Divide into subarrays \(A[1 \ldots m]\) and \(A[m + 1 \ldots n]\), where \(m = \lfloor n/2 \rfloor\)

   \[ A \ L \ G \ O \ R \quad I \ T \ H \ M \ S \]

3. Recursively **MergeSort** \(A[1 \ldots m]\) and \(A[m + 1 \ldots n]\)

   \[ A \ G \ L \ O \ R \quad H \ I \ M \ S \ T \]

4. Merge the sorted arrays

   \[ A \ G \ H \ I \ L \ M \ O \ R \ S \ T \]
1. **Input:** Array $A[1 \ldots n]$

2. Divide into subarrays $A[1 \ldots m]$ and $A[m + 1 \ldots n]$, where $m = \lfloor n/2 \rfloor$

3. Recursively **MergeSort** $A[1 \ldots m]$ and $A[m + 1 \ldots n]$

4. Merge the sorted arrays
Merging Sorted Arrays

- Use a new array $C$ to store the merged array
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order

$$\text{AGLR} \quad \text{HI MST}$$
Merging Sorted Arrays

- Use a new array $C$ to store the merged array
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order

\[
\begin{array}{ccccccc}
A & G & L & O & R & H & I & M & S & T \\
A & G
\end{array}
\]
Merging Sorted Arrays

- Use a new array $C$ to store the merged array.
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order.

$A \ G \ L \ O \ R \ H \ I \ M \ S \ T$
$A \ G \ H$
Merging Sorted Arrays

- Use a new array $C$ to store the merged array
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order

\[
\begin{align*}
A & \quad G \quad L \quad O \quad R \\
A & \quad G \quad H \quad I \\
\end{align*}
\]
Merging Sorted Arrays

- Use a new array $C$ to store the merged array
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order

\[
\text{AGLOR} \quad \text{HIMST}
\]

\[
\text{AGHILMORS}
\]
Merging Sorted Arrays

- Use a new array $C$ to store the merged array
- Scan $A$ and $B$ from left-to-right, storing elements in $C$ in order

$A \ G \ L \ O \ R \quad H \ I \ M \ S \ T$

$A \ G \ H \ I \ L \ M \ O \ R \ S \ T$

- Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical.
Formal Code

\[\text{MergeSort}(A[1..n]):\]
if \( n > 1 \)
\[ m \leftarrow \lfloor n/2 \rfloor \]
\text{MergeSort}(A[1..m])
\text{MergeSort}(A[m+1..n])
\text{Merge}(A[1..n], m)\]

\[\text{Merge}(A[1..n], m):\]
\[ i \leftarrow 1; \ j \leftarrow m + 1 \]
for \( k \leftarrow 1 \) to \( n \)
\[ \text{if } j > n \]
\[ B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \]
\[ \text{else if } i > m \]
\[ B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \]
\[ \text{else if } A[i] < A[j] \]
\[ B[k] \leftarrow A[i]; \ i \leftarrow i + 1 \]
\[ \text{else} \]
\[ B[k] \leftarrow A[j]; \ j \leftarrow j + 1 \]
for \( k \leftarrow 1 \) to \( n \)
\[ A[k] \leftarrow B[k] \]
Running time analysis of merge-sort: Recursion tree method
Recursion tree

MergeSort(A[1..16])
Recursion tree

MergeSort(A[1..16])

MergeSort(A[1..8])

MergeSort(A[9..16])
Recursion tree
Recursion tree
Recursion tree: subproblem sizes

MergeSort(A[1..16]) 16
Recursion tree: subproblem sizes
Recursion tree: subproblem sizes
Recursion tree: subproblem sizes
Recursion tree: subproblem sizes
Recursion tree: Total work?
Running Time

\( T(n) \): time for merge sort to sort an \( n \) element array
Running Time

$T(n)$: time for merge sort to sort an $n$ element array

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$$

What do we want as a solution to the recurrence?

Almost always only an asymptotically tight bound. That is we want to know $f(n)$ such that

- $T(n) = \Theta(f(n))$ - upper bound
- $T(n) = \Omega(f(n))$ - lower bound
Running Time

\( T(n) \): time for merge sort to sort an \( n \) element array

\[
T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn
\]

What do we want as a solution to the recurrence?

Almost always only an asymptotically tight bound. That is we want to know \( f(n) \) such that \( T(n) = \Theta(f(n)) \).

- \( T(n) = O(f(n)) \) - upper bound
- \( T(n) = \Omega(f(n)) \) - lower bound
Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- **Recursion tree method** — imagine the computation as a tree
- **Guess and verify** — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein:
"Everything should be made as simple as possible, but not simpler."
Solving Recurrences: Some Techniques

- Know some basic math: geometric series, logarithms, exponentials, elementary calculus
- Expand the recurrence and spot a pattern and use simple math
- **Recursion tree method** — imagine the computation as a tree
- **Guess and verify** — useful for proving upper and lower bounds even if not tight bounds

**Albert Einstein:** “Everything should be made as simple as possible, but not simpler.”

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!
Recursion Trees: MergeSort: $n$ is a power of $2$

- Unroll the recurrence.

$$T(n) = 2T(n/2) + cn$$
Recursion Trees: MergeSort: \( n \) is a power of 2

- Unroll the recurrence.
  \[ T(n) = 2T(n/2) + cn \]
- Identify a pattern.
Recursion Trees: MergeSort: $n$ is a power of 2

- Unroll the recurrence.
  \[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]

- Identify a pattern. At the $i$th level total work is $cn$. 
Recursion Trees: MergeSort: $n$ is a power of 2

- Unroll the recurrence.
  \[ T(n) = 2T(n/2) + cn \]

- Identify a pattern. At the $i$th level total work is $cn$.

- Sum over all levels.
Recursion Trees: MergeSort: \( n \) is a power of 2

- Unroll the recurrence.
  \[ T(n) = 2T(n/2) + cn \]

- Identify a pattern. At the \( i \)th level total work is \( cn \).

- Sum over all levels. The number of levels is \( \log n \). So total is \( cn \log n = O(n \log n) \).
Recursion Trees

```
n
n/2
n/4
n/4
n/4

n/2
n/4
n/4
n/4
```

`n/4`
Recursion Trees

Work in each node
Recursion Trees

Work in each node
Recursion Trees

\[
\log n \left\{ \begin{array}{c}
\frac{cn}{2} + \frac{cn}{2} \\
\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} \\
\vdots
\end{array} \right.
\]

\[
= cn
\]

\[
= cn
\]

\[
= cn
\]

\[
= cn
\]
Recursion Trees

Recursion Tree:

\[ \log n \left\{ \begin{array}{c}
\frac{cn}{2} + \frac{cn}{2} = cn \\
\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = \frac{cn}{2} \\
\vdots \\
\end{array} \right. \]

\[ = cn \log n = O(n \log \log n) \]

\[ T(n) = O(n \log n)! \]

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
**Question**: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $k$ arrays of size $n/k$ each?
Binary Search

( RIY )
Binary Search in Sorted Arrays

**Input**  Sorted array $A$ of $n$ numbers and number $x$

**Goal**  Is $x$ in $A$?

```
BinarySearch(A[a..b], x):
if (b - a < 0)
    return NO

mid = A[(a + b) / 2]
if (x == mid)
    return YES
if (x < mid)
    return BinarySearch(A[a..mid - 1], x)
else
    return BinarySearch(A[mid + 1..b], x)
```

**Analysis:**
$T(n) = T(n/2) + O(1)$.
$T(n) = O(\log n)$.

**Observation:**
After $k$ steps, size of array left is $n / 2^k$. 
Binary Search in Sorted Arrays

**Input**  Sorted array $A$ of $n$ numbers and number $x$

**Goal**  Is $x$ in $A$?

```python
BinarySearch (A[a..b], x):
    if (b - a < 0) return NO
    mid = A[[a + b]/2]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch (A[a..[(a + b)/2] - 1], x)
    else
        return BinarySearch (A[[1] + a + b]/2, x)
```

**Analysis:**
\[ T(n) = T\left(\frac{n}{2}\right) + O(1) \]
\[ T(n) = O(\log n) \]

**Observation:**
After $k$ steps, size of array left is $n/2^k$.
**Input** Sorted array $A$ of $n$ numbers and number $x$

**Goal** Is $x$ in $A$?

```python
BinarySearch (A[a..b], x):
    if (b - a < 0) return NO
    mid = A[⌈(a + b)/2⌉]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch (A[a..⌈(a + b)/2⌉ - 1], x)
    else
        return BinarySearch (A[⌈(a + b)/2⌉ + 1..b], x)
```

Analysis: $T(n) = T(⌈n/2⌉) + O(1)$. $T(n) = O(\log n)$.

Observation: After $k$ steps, size of array left is $n/2^k$