Algorithmic Insights I - Recursion and Divide and Conquer

K. Subramani

1Lane Department of Computer Science and Electrical Engineering
West Virginia University

February 9, 2015
1 Review of concepts
Outline

1. Review of concepts
2. Algorithmic Insights
Outline

1. Review of concepts
2. Algorithmic Insights
3. Recursion
1. Review of concepts
2. Algorithmic Insights
3. Recursion
4. Divide and Conquer
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. (Matrix multiplication).
3. Polynomial time and tractability.
4. Robustness of \( P \) or not in \( P \). Less emphasis on most efficient algorithms.
Review

Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs)
2. Time and scaling. (Matrix multiplication)
3. Polynomial time and tractability.
4. Robustness of $P$.
5. In $P$ or not in $P$. Less emphasis on most efficient algorithms.
### Main concepts

1. **Instance, Problem, Solutions.**
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling.
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. Matrix multiplication.
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. Matrix multiplication.
3. Polynomial time and tractability.
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. Matrix multiplication.
3. Polynomial time and tractability.
4. Robustness of P.
Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. Matrix multiplication.
3. Polynomial time and tractability.
4. Robustness of $\mathbf{P}$.
5. In $\mathbf{P}$ or not in $\mathbf{P}$. 
### Review

#### Main concepts

1. Instance, Problem, Solutions. (Chess, Eulerian graphs).
2. Time and scaling. Matrix multiplication.
3. Polynomial time and tractability.
4. Robustness of P.
5. In P or not in P. Less emphasis on most efficient algorithms.
<table>
<thead>
<tr>
<th>Review of concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic Insights</td>
</tr>
<tr>
<td>Recursion</td>
</tr>
<tr>
<td>Divide and Conquer</td>
</tr>
</tbody>
</table>

**Main concepts**

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
6. Transformations and reductions.
Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
6. Transformations and reductions.
Main concepts

What makes a problem tractable?
Main concepts

What makes a problem tractable?

1. Recursion.
## Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
5. Iterative approaches (Rewriting).
6. Transformations and reductions.
Recursion

Main Idea

1. Break a large problem into smaller problems having identical form.
2. Continue breaking sub-problems into even smaller sub-problems, until the problems become trivial (Base case).

Algorithmic Insights
Main Idea

1. Break a large problem into smaller problems having identical form.
2. Continue breaking sub-problems into even smaller sub-problems, until the problems become trivial (Base case).
Main Idea

1. Break a large problem into smaller problems having identical form.
Main Idea

1. Break a large problem into smaller problems having identical form.
2. Continue breaking sub-problems into even smaller sub-problems, until the problems become trivial.
Main Idea

1. Break a large problem into smaller problems having identical form.
2. Continue breaking sub-problems into even smaller sub-problems, until the problems become trivial (Base case).
The Array-Max problem

Problem

Given an array of n integers, find the maximum element.

Algorithm

Function $A\text{RAY-M}\text{AX}(A, n)$

- if ($n = 1$) then return ($A[n]$)
- else return (max ($A[n]$, $A\text{RAY-M}\text{AX}(A, n - 1)$)).

Algorithm 4.1: Finding the maximum element in an array

Analysis

$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ T(n - 1) + 1, & \text{otherwise} \end{cases} 
\Rightarrow T(n) = (n - 1)$.
The Array-Max problem

**Problem**

Given an array of n integers, find the maximum element.

**Algorithm**

Function `ARRAY-MAX(A, n)`

- if `n = 1` then return `A[n]`
- else return `max(A[n], ARRAY-MAX(A, n-1))`

**Analysis**

\[ T(n) = \begin{cases} 0, & \text{if } n = 0 \\ T(n-1) + 1, & \text{otherwise} \end{cases} \]

\[ T(n) = n - 1. \]
The Array-Max problem

Problem

*Given an array of \( n \) integers, find the maximum element.*
The Array-Max problem

**Problem**

*Given an array of n integers, find the maximum element.*

**Algorithm**

```plaintext
Function Array-MAX(A, n)
    if (n = 1)
        return (A[n])
    else
        return (max(A[n], Array-MAX(A, n-1)))
    end if
end function
```

**Analysis**

\[ T(n) = \begin{cases} 
0, & \text{if } n = 0 \\
T(n-1) + 1, & \text{otherwise} 
\end{cases} \]

\[ T(n) = n - 1. \]
The Array-Max problem

**Problem**

*Given an array of n integers, find the maximum element.*

**Algorithm**

**Function** $\text{ARRAY-MAX}(A, n)$
### Problem

*Given an array of n integers, find the maximum element.*

### Algorithm

**Function** `ARRAY-MAX(A, n)`

if \( n = 1 \) then
The Array-Max problem

Problem

Given an array of \( n \) integers, find the maximum element.

Algorithm

Function \( \text{ARRAY-MAX}(A, n) \)

if \( n = 1 \) then
    return \( A[n] \)
The Array-Max problem

**Problem**

*Given an array of n integers, find the maximum element.*

**Algorithm**

```plaintext
Function ARRAY-MAX(A, n)
    if (n = 1) then
        return(A[n])
    else
```

Algorithmic Insights

**Computational Complexity**
The Array-Max problem

Problem

*Given an array of n integers, find the maximum element.*

Algorithm

```plaintext
Function ARRAY-MAX(A, n)
    if (n = 1) then
        return(A[n])
    else
        return (max(A[n], ARRAY-MAX(A, n - 1))).
```
The Array-Max problem

Problem

Given an array of n integers, find the maximum element.

Algorithm

Function $\text{ARRAY-MAX}(A, n)$

\[
\begin{align*}
\text{if } (n = 1) \text{ then} & \\
\quad \text{return}(A[n]) & \\
\text{else} & \\
\quad \text{return} \left(\max(A[n], \text{ARRAY-MAX}(A, n - 1))\right) & \\
\end{align*}
\]

Algorithm 4.10: Finding the maximum element in an array
The Array-Max problem

Problem

*Given an array of n integers, find the maximum element.*

Algorithm

```latex
Function ARRAY-MAX(A, n)
    if (n = 1) then
        return(A[n])
    else
        return (max(A[n], ARRAY-MAX(A, n - 1))).
end if
```

**Algorithm 4.11:** Finding the maximum element in an array

Analysis

\[ T(n) = \]
The Array-Max problem

Problem

Given an array of $n$ integers, find the maximum element.

Algorithm

Function `ARRAY-MAX(A, n)`

```
if ($n = 1$) then
    return($A[n]$)
else
    return ($\text{max}(A[n], \text{ARRAY-MAX}(A, n - 1))$).
```

Algorithm 4.12: Finding the maximum element in an array

Analysis

$$T(n) = \begin{cases} 
0, & \text{if } n = 0, \\
\end{cases}$$
The Array-Max problem

Problem

Given an array of \( n \) integers, find the maximum element.

Algorithm

Function `ARRAY-MAX(A, n)`

- if \( n = 1 \) then
  - return \( A[n] \)
- else
  - return \( \max(A[n], ARRAY-MAX(A, n - 1)) \)

*Algorithm 4.13*: Finding the maximum element in an array

Analysis

\[
T(n) = \begin{cases} 
0, & \text{if } n = 0, \\
T(n - 1) + 1, & \text{otherwise}
\end{cases}
\]
The Array-Max problem

Problem

*Given an array of* \( n \) *integers, find the maximum element.*

Algorithm

```plaintext
Function ARRAY-MAX(A, n)
    if \( n = 1 \) then
        return(A[n])
    else
        return(max(A[n], ARRAY-MAX(A, n - 1))).
    end if

Algorithm 4.14: Finding the maximum element in an array
```

Analysis

\[
T(n) = \begin{cases} 
0, & \text{if } n = 0, \\
T(n - 1) + 1, & \text{otherwise} 
\end{cases} \Rightarrow T(n) = (n - 1).
\]
The Array-Search problem

**Problem**

Given an array of n integers, and a key k, return true if any of the array elements is equal to k and false otherwise.
The Array-Search problem

Given an array of n integers, and a key k, return true if any of the array elements is equal to k and false otherwise.
Algorithm

**Algorithm 4.15:** Searching for a key in an array

```
if (n = 1)
    if (A[n] = k)
        return (true)
    else
        return (false)
else
    if (A[n] = k)
        return (true)
    else
        return (ARRAY-SEARCH(A, n-1, k)).
```

Computational Complexity
Function $\text{ARRAY-SEARCH}(A, n, k)$

If

1. $n = 1$
   - If $A[n] = k$
     - Return $true$
   - Else
     - Return $false$

Else

1. If $A[n] = k$
   - Return $true$
   - Else
     - Return $\text{ARRAY-SEARCH}(A, n - 1, k)$

End if

End if

Algorithm 4.16: Searching for a key in an array
Algorithm 4.17: Searching for a key in an array

Function $\text{ARRAY-SEARCH}(A, n, k)$

\[
\begin{align*}
\text{if } \ (n = 1) & \text{ then return } \text{true} \text{ if } A[n] = k \text{ else return false} \\
\text{else } \text{ if } A[n] = k & \text{ then return true else return } \text{ARRAY-SEARCH}(A, n-1, k). 
\end{align*}
\]
**Function** `ARRAY-SEARCH(A, n, k)`

if \((n = 1)\) then
Algorithm

Function `ARRAY-SEARCH(A, n, k)`

if \((n = 1)\) then
  if \((A[n] = k)\) then

```
Algorithm

Function `ARRAY-SEARCH(A, n, k)`

if \((n = 1)\) then
    if \((A[n] = k)\) then
        return (true)
Algorithm

**Function** `ARRAY-SEARCH(A, n, k)`

if $(n = 1)$ then
  if $(A[n] = k)$ then
    return (true)
  else
    return (false)
else

Algorithm

Function $\text{ARRAY-SEARCH}(A, n, k)$

if $(n = 1)$ then
  if $(A[n] = k)$ then
    return(true)
  else
    return(false)
else
  return($\text{ARRAY-SEARCH}(A, n-1, k)$).
Algorithm

Function `ARRAY-SEARCH(A, n, k)`

if \((n = 1)\) then
  if \((A[n] = k)\) then
    return true
  else
    return false
end if
else
  if \((A[n] = k)\) then
    return true
  else
    return `ARRAY-SEARCH(A, n-1, k)`.
end if

Algorithm 4.23: Searching for a key in an array
Algorithm 4.24: Searching for a key in an array

Function ARRAY-SEARCH(A, n, k)
    if (n = 1) then
        if (A[n] = k) then
            return(true)
        else
            return(false)
    end if
else
    return(ARRAY-SEARCH(A, n−1, k)).
**Function** `ARRAY-SEARCH(A, n, k)`

if \((n = 1)\) then
  if \((A[n] = k)\) then
    return(true)
  else
    return(false)
end if
else
  if \((A[n] = k)\) then
    return(true)
  else
    return(false)
  end if
end if
Algorithmic Insights

Computational Complexity

Function `ARRAY-SEARCH(A, n, k)`

if (`n = 1`) then
  if (`A[n] = k`) then
    return `true`
  else
    return `false`
  end if
else
  if (`A[n] = k`) then
    return `true`
  end if
end if
Function $\text{ARRAY-SEARCH}(A, n, k)$

if \((n = 1)\) then

  if \((A[n] = k)\) then
    return(true)
  else
    return(false)
end if

else

  if \((A[n] = k)\) then
    return(true)
  else
    return(ARRAY-SEARCH($A$, n - 1, k)).
  end if
end if

Algorithm 4.27: Searching for a key in an array
Function $\text{ARRAY-SEARCH}(A, n, k)$

if $(n = 1)$ then
    if $(A[n] = k)$ then
        return(true)
    else
        return(false)
    end if
else
    if $(A[n] = k)$ then
        return(true)
    else
        return($\text{ARRAY-SEARCH}(A, n - 1, k)$).
end if
Algorithm 4.29: Searching for a key in an array

Function \textsc{Array-Search}(A, n, k)

\begin{align*}
\text{if } (n = 1) & \text{ then} \\
\text{ if } (A[n] = k) & \text{ then} \\
\quad \text{return}(\text{true}) \\
\text{ else} & \\
\quad \text{return}(\text{false}) \\
\text{ end if} \\
\text{ else} & \\
\text{ if } (A[n] = k) & \text{ then} \\
\quad \text{return}(\text{true}) \\
\text{ else} & \\
\quad \text{return}(\text{Array-Search}(A, n - 1, k)) \\
\text{ end if} \\
\end{align*}
**Algorithm 4.30: Searching for a key in an array**

```pseudo
Function ARRAY-SEARCH(A, n, k)
    if (n = 1) then
        if (A[n] = k) then
            return(true)
        else
            return(false)
        end if
    else
        if (A[n] = k) then
            return(true)
        else
            return(ARRAY-SEARCH(A, n - 1, k)).
        end if
    end if
end if
```
Recursion

\[ T(n) = \begin{cases} 
1, & \text{if } n = 1 \\
T(n - 1) + 1, & \text{otherwise} 
\end{cases} \]

\[ T(n) = n. \]
Analysis

\[ T(n) = \begin{cases} 
1, & \text{if } n = 1 \\
T(n-1) + 1, & \text{otherwise} 
\end{cases} \]

\[ T(n) = n. \]
Analysis

\[ T(n) = \begin{cases} 
1, & \text{if } n = 1, \\
T(n-1) + 1, & \text{otherwise} 
\end{cases} \]
Analysis

\[ T(n) = \begin{cases} 
1, & \text{if } n = 1, \\
T(n-1) + 1, & \text{otherwise} 
\end{cases} \]
Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 1, \\
T(n - 1) + 1, & \text{otherwise}
\end{cases} \Rightarrow T(n) = n.
\]
The Towers of Hanoi problem

You are given three pegs, viz., A, B and C. n disks are stacked on peg A, in decreasing order of size, with the largest disk at the bottom of the stack.

You need to move the disks from peg A to peg B, ensuring that at no time a disk is placed on another disk of smaller size.
The Towers of Hanoi problem

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
</table>

You are given three pegs, viz., A, B and C. `n` disks are stacked on peg A, in decreasing order of size, with the largest disk at the bottom of the stack. You need to move the disks from peg A to peg B, ensuring that at no time a disk is placed on another disk of smaller size.
The Towers of Hanoi problem

Problem

You are given three pegs, viz., A, B and C.
The Towers of Hanoi problem

Problem

You are given three pegs, viz., A, B and C.

n disks are stacked on peg A, in decreasing order of size, with the largest disk at the bottom of the stack.
The Towers of Hanoi problem

Problem

You are given three pegs, viz., A, B and C.

n disks are stacked on peg A, in decreasing order of size, with the largest disk at the bottom of the stack.

You need to move the disks from peg A to peg B, ensuring that at no time a disk is placed on another disk of smaller size.
Algorithm

Review of concepts
Algorithmic Insights
Recursion
Divide and Conquer

Main idea
Break the task into three sub-tasks.
1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).

Analysis
\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
2 \cdot T(n - 1) + 1, & \text{otherwise}
\end{cases}
\Rightarrow T(n) = 2^n - 1
\]
Main idea

- Break the task into three sub-tasks.
- Move the first \((n−1)\) disks from \(A\) to \(C\), using \(B\).
- Move the largest disk from \(A\) to \(B\).
- Move the \((n−1)\) disks from \(C\) to \(B\), using \(A\).

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
2 \cdot T(n−1) + 1, & \text{otherwise}
\end{cases}
\]

\[
T(n) = 2^{n−1} - 1
\]
Main idea

Break the task into three sub-tasks.
Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).
Algorithm

Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).

Analysis
Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
2 \cdot T(n-1) + 1, & \text{otherwise}
\end{cases}
\]
Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
2 \cdot T(n - 1) + 1, & \text{otherwise}
\end{cases}
\]
Algorithm

Main idea

Break the task into three sub-tasks.

1. Move the first \((n - 1)\) disks from \(A\) to \(C\), using \(B\).
2. Move the largest disk from \(A\) to \(B\).
3. Move the \((n - 1)\) disks from \(C\) to \(B\), using \(A\).

Analysis

\[
T(n) = \begin{cases} 
1, & \text{if } n = 0 \\
2 \cdot T(n - 1) + 1, & \text{otherwise}
\end{cases} 
\Rightarrow T(n) = 2^n - 1
\]
Approach

Main Concepts

Some problems can be broken up into:

1. Divide the problem into smaller sub-problems.
2. Conquer the sub-problems either through recursion or through brute-force.
3. Combine the solutions to the sub-problems to get the solution of the original problem.
Some problems can be broken up into independent sub-problems:

1. **Divide** the problem into smaller sub-problems.
2. **Conquer** the sub-problems either through recursion or through brute-force.
3. **Combine** the solutions to the sub-problems to get the solution of the original problem.
Main Concepts

Some problems can be broken up into *independent* sub-problems
Main Concepts

Some problems can be broken up into *independent* sub-problems

1. **Divide** the problem into smaller sub-problems.
Main Concepts

Some problems can be broken up into *independent* sub-problems

1. **Divide** the problem into smaller sub-problems.
2. **Conquer** the sub-problems either through recursion or through brute-force.
### Main Concepts

Some problems can be broken up into *independent* sub-problems:

1. **Divide** the problem into smaller sub-problems.
2. **Conquer** the sub-problems either through recursion or through brute-force.
3. **Combine** the solutions to the sub-problems to get the solution of the original problem.
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1. Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.

Consider a recurrence of the form:

$$ T(n) = \begin{cases} 
  a \cdot T(n/b) + f(n), & \text{if } n > 1 \\
  d, & \text{if } n = 1 
\end{cases} $$

Then,

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c)$.

2. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n \log_b n \cdot \log_b \log_b n)$.

3. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n \log_b n)$.
The Master Theorem

- If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c)$.
- If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n \log_b n \log_b a \cdot \log_b n)$.
- If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n \log_b a)$. 
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1.
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1. Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1. Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.

Consider a recurrence of the form:

$$T(n) = \begin{cases} 
a \cdot T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \\
d, & \text{if } n = 1 \end{cases}$$
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1.

Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.

Consider a recurrence of the form:

$$ T(n) = \begin{cases} 
    a \cdot T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \\
    d, & \text{if } n = 1 
\end{cases} $$

Then,
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1. Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number. Consider a recurrence of the form:

$$T(n) = \begin{cases} 
    a \cdot T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \\
    d, & \text{if } n = 1
\end{cases}$$

Then,

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c)$.
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1. Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.

Consider a recurrence of the form:

$$T(n) = \begin{cases} 
  a \cdot T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \\
  d, & \text{if } n = 1 
\end{cases}$$

Then,

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c)$.
2. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$. 
The Master Theorem

Let $a$ be an integer greater than or equal to 1 and $b$ be a real number greater than 1.
Let $f(n)$ be an increasing function of $n$ and $d$ a nonnegative real number.
Consider a recurrence of the form:

$$T(n) = \begin{cases} 
  a \cdot T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \\
  d, & \text{if } n = 1
\end{cases}$$

Then,

1. If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c)$.
2. If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.
3. If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$. 

The Merge-Sort Algorithm

Algorithm 5.1: MergeSort

Function MERGE-SORT(A, low, high)

if low < high then
    mid = low + high / 2.
    MERGE-SORT(A, low, mid).
    MERGE-SORT(A, mid + 1, high).
    MERGE(A, low, mid, high).
end if
The Merge-Sort Algorithm

Sorting through Merging

Function `MERGE-SORT(A, low, high)`
The Merge-Sort Algorithm

Sorting through Merging

Function `MERGE-SORT(A, low, high)`

if `(low < high)` then
The Merge-Sort Algorithm

Function \textsc{Merge-Sort}(A, \textit{low}, \textit{high})

\begin{align*}
\text{if } (\textit{low} < \textit{high}) \text{ then } \\
\text{mid} &= \frac{\textit{low} + \textit{high}}{2}.
\end{align*}
The Merge-Sort Algorithm

Function MERGE-SORT(A, low, high)

if (low < high) then
    mid = \(\frac{\text{low} + \text{high}}{2}\).
    MERGE-SORT(A, low, mid).

end if
The Merge-Sort Algorithm

**Function** `MERGE-SORT(A, low, high)`

```plaintext
if (low < high) then
    mid = \frac{low + high}{2}.
    `MERGE-SORT(A, low, mid)`.
    `MERGE-SORT(A, mid + 1, high)`.
```

Sorting through Merging
The Merge-Sort Algorithm

Sorting through Merging

Function MERGE-SORT(A, low, high)

if (low < high) then
    mid = \( \frac{low + high}{2} \).
    MERGE-SORT(A, low, mid).
    MERGE-SORT(A, mid + 1, high).
    MERGE(A, low, mid, high).
The Merge-Sort Algorithm

Sorting through Merging

Function \textsc{Merge-Sort}(A, low, high)

\[
\text{if (low < high) then}
\]

\[
\text{mid} = \frac{\text{low} + \text{high}}{2}.
\]

\textsc{Merge-Sort}(A, low, mid).
\textsc{Merge-Sort}(A, mid + 1, high).
\textsc{Merge}(A, low, mid, high).

\textbf{end if}

\textbf{Algorithm 5.8: MergeSort}
Analyzing Time and Space

Analysis

$$T(n) = 2 \cdot T(n^2) + n \in \Theta(n \cdot \log n)$$

$$S(n) \in \Theta(n)$$
Analyzing Time and Space

Analysis

\[ T(n) = 2 \cdot T\left(n^{\frac{1}{2}}\right) + n \in \Theta\left(n \cdot \log n\right) \]
Analyzing Time and Space

Analysis

\[ T(n) = \]

Analysis
Analyzing Time and Space

Analysis

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \]
Analyzing Time and Space

Analysis

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \]
\[ \in \Theta(n \cdot \log n) \]
Analyzing Time and Space

Analysis

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \]
\[ \in \Theta(n \cdot \log n) \]

\[ S(n) \in \Theta(n) \]
The Quick-Sort Algorithm

**Algorithm 5.9: Quicksort**

```
function Quick-Sort(A, low, high)
    if low < high then
        Quick-Sort(A, low, j - 1).
        Quick-Sort(A, j + 1, high).
    end if
```

**Analysis (Space)**

Quick-Sort() uses $O(1)$ extra space. Partitioning can be done in-place.
The Quick-Sort Algorithm

Sorting through Partitioning

Function **QUICK-SORT**(A, *low*, *high*)
The Quick-Sort Algorithm

Sorting through Partitioning

**Function** \texttt{QUICK-SORT}(A, low, high)

\begin{verbatim}
    if (low < high) then
\end{verbatim}
The Quick-Sort Algorithm

Sorting through Partitioning

**Function** QUICK-SORT(A, low, high)

if (low < high) then
  Partition A about A[low].
The Quick-Sort Algorithm

**Function** \texttt{QUICK-SORT}(A, low, high)

\textbf{if} (\texttt{low} < \texttt{high}) \textbf{then}

\hspace{1em} Partition A about A[low].

\hspace{1em} Let \( j \) denote the index of \( A[low] \) after partitioning.
The Quick-Sort Algorithm

Sorting through Partitioning

**Function** QUICK-SORT(A, low, high)

```plaintext
if (low < high) then
    Partition A about A[low].
    Let j denote the index of A[low] after partitioning.
    QUICK-SORT(A, low, j − 1).
```

Algorithm 5.14: Quicksort

**Analysis (Space)**

Quick-Sort() uses $O(1)$ extra space. Partitioning can be done in-place.
The Quick-Sort Algorithm

### Sorting through Partitioning

**Function** `QUICK-SORT(A, low, high)`

```plaintext
if (low < high) then
    Partition A about A[low].
    Let j denote the index of A[low] after partitioning.
    QUICK-SORT(A, low, j − 1).
    QUICK-SORT(A, j + 1, high).
```

**Algorithm 5.15: Quicksort**

**Analysis (Space)**

- Quick-Sort() uses $O(1)$ extra space.
- Partitioning can be done in-place.
### The Quick-Sort Algorithm

#### Sorting through Partitioning

**Function** \( \text{QUICK-SORT}(A, \text{low}, \text{high}) \)

- **if** \((\text{low} < \text{high})\) **then**
  - Partition \(A\) about \(A[\text{low}]\).
  - Let \(j\) denote the index of \(A[\text{low}]\) after partitioning.
  - \(\text{QUICK-SORT}(A, \text{low}, j - 1)\).
  - \(\text{QUICK-SORT}(A, j + 1, \text{high})\).
- **end if**

**Algorithm 5.16:** Quicksort
The Quick-Sort Algorithm

Sorting through Partitioning

**Function** QUICK-SORT(A, low, high)

if \((low < high)\) then
    Partition A about \(A[low]\).
    Let \(j\) denote the index of \(A[low]\) after partitioning.
    QUICK-SORT(A, \(low, j - 1\)).
    QUICK-SORT(A, \(j + 1, high\)).
end if

**Algorithm 5.17:** Quicksort

Analysis (Space)

Quick-Sort() uses \(O(1)\) extra space. Partitioning can be done in-place.
The Quick-Sort Algorithm

Sorting through Partitioning

**Function** \textsc{quick-sort}(A, low, high)

\begin{align*}
\text{if} & \ (low < high) \ \text{then} \\
& \text{Partition } A \text{ about } A[low]. \\
& \text{Let } j \text{ denote the index of } A[low] \text{ after partitioning.} \\
& \textsc{quick-sort}(A, low, j - 1). \\
& \textsc{quick-sort}(A, j + 1, high). \\
\text{end if}
\end{align*}

**Algorithm 5.18:** Quicksort

Analysis (Space)

Quick-Sort() uses \(O(1)\) extra space.
The Quick-Sort Algorithm

Sorting through Partitioning

Function QUICK-SORT(A, low, high)
    if (low < high) then
        Partition A about A[low].
        Let j denote the index of A[low] after partitioning.
        QUICK-SORT(A, low, j - 1).
        QUICK-SORT(A, j + 1, high).
    end if

Algorithm 5.19: Quicksort

Analysis (Space)

Quick-Sort() uses $O(1)$ extra space. Partitioning can be done in-place.
Analysis of running time

Best Case

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + (n - 1) \in \Theta(n \cdot \log n)$$

Worst-case

$$T(n) = T(n-1) + (n - 1) \in \Theta(n^2)$$
Analysis of running time

Best Case

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + (n-1) \in \Theta(n \cdot \log n) \]

Best Case

\[ T(n) = T(n-1) + (n-1) \in \Theta(n^2) \]
Analysis of running time

Best Case

\[ T(n) = \]
Analysis of running time

Best Case

\[ T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n-1) \]
Analysis of running time

Best Case

\[ T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n - 1) \]
\[ \in \Theta(n \cdot \log n) \]
Analysis of running time

**Best Case**

\[
T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n-1)
\]

\(\in\ \Theta(n \cdot \log n)\)

**Worst-case**

Analysis of running time

Best Case

\[ T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n - 1) \]
\[ \in \Theta(n \cdot \log n) \]

Worst-case

\[ T(n) = \]
Analysis of running time

**Best Case**

\[ T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n - 1) \]

\[ \in \Theta(n \cdot \log n) \]

**Worst-case**

\[ T(n) = T(n-1) + (n - 1) \]
Analysis of running time

**Best Case**

\[
T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + (n-1)
\]

\[
\in \Theta(n \cdot \log n)
\]

**Worst-case**

\[
T(n) = T(n-1) + (n-1)
\]

\[
\in \Theta(n^2)
\]
Average case analysis
Average case analysis

Average case

\[ T(n) = \]

\[ (n - 1) + \sum_{r=1}^{n} [T(r-1) + T(n-r)] \]

\[ \approx 2n \cdot \ln n \]
Average case analysis

**Average case**

\[
T(n) = (n - 1) + \frac{1}{n} \cdot \sum_{r=1}^{n} [T(r - 1) + T(n - r)]
\]
Average case analysis

Average case

\[ T(n) = (n - 1) + \frac{1}{n} \sum_{r=1}^{n} [T(r - 1) + T(n - r)] \]

\[ = (n - 1) + \]
Average case analysis

Average case

\[
T(n) = (n - 1) + \frac{1}{n} \sum_{r=1}^{n} [T(r - 1) + T(n - r)] \\
= (n - 1) + \frac{2}{n} \cot \sum_{r=1}^{n} T(r - 1)
\]
Average case analysis

\[
T(n) = (n - 1) + \frac{1}{n} \cdot \sum_{r=1}^{n} [T(r - 1) + T(n - r)]
\]
\[
= (n - 1) + \frac{2}{n} \cot \sum_{r=1}^{n} T(r - 1)
\]
\[
\approx 2 \cdot n \cdot \ln n
\]
Given two n digit integers x and y, compute $x^y \mod p$.

Useful in cryptography and primality checking.

**Approach I**

Function $MOD\cdot EXP(x, y, p)$

- if ($y = 0$) then
  - return (1).
- \( r = 1. \)
- for ($i = 1$ to $y$) do
  - \( r = x \cdot r \mod p. \)
- return ($y$).

**Algorithm 5.20: Modular Exponentiation**
Modular Exponentiation

Problem

Given two n-digit integers x and y, compute $x^y \mod p$.

Useful in cryptography and primality checking.

Approach I

Function MOD-EXP(x, y, p)

if (y = 0) then
  return (1).
end if

r = 1.
for (i = 1 to y) do
  r = x · r \mod p.
end for

return (y).

Algorithm 5.21: Modular Exponentiation
Problem

*Given two n digit integers x and y, compute \(x^y \mod p\).*
Problem

*Given two n digit integers x and y, compute \( x^y \mod p \). Useful in cryptography and primality checking.*
Problem

Given two n digit integers x and y, compute \( x^y \mod p \). Useful in cryptography and primality checking.

Approach I

Program MOD-EXP(x, y, p)

Function MOD-EXP(x, y, p)

if \( y = 0 \) then
    return(1).
endif

r = 1.
for \( i = 1 \) to \( y \) do
    \( r = x \cdot r \mod p \).
end for

return(y).

Algorithm 5.24: Modular Exponentiation
Time Analysis

Assuming \( x \) and \( y \) have \( n \) digits, the number of multiplications is proportional to \( y \), which is exponentially large!
Review of concepts

Recursion

Divide and Conquer

Time Analysis

Analysis

Assuming $x$ and $y$ have $n$ digits, the number of multiplications is proportional to $y^n$, which is exponentially large!
Analysis

Assuming $x$ and $y$ have $n$ digits, the number of multiplications is proportional to $\ldots$
Analysis

Assuming $x$ and $y$ have $n$ digits, the number of multiplications is proportional to $y$, which is exponentially large!
A better approach

Algorithm 5.25: Faster Modular Exponentiation
A better approach

Approach II

**Function** \( \text{Mod-Exp}(x, y, p) \)

```latex
\textbf{Algorithm 5.26: Faster Modular Exponentiation}
```

If \( y = 0 \) then
  \text{return} \((1)\).

Else
  \( t = \text{Mod-Exp}(x, \lfloor y/2 \rfloor, p) \).
  \text{If} \( y \text{ is even} \) then
    \text{return} \((t^2 \mod p)\).
  \text{Else}
    \text{return} \((x \times t^2 \mod p)\).
  \text{End if}
\text{End if}
```
### Approach II

<table>
<thead>
<tr>
<th>Function $\text{MOD-EXP}(x, y, p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $(y = 0)$ then</td>
</tr>
</tbody>
</table>
A better approach

**Approach II**

**Function** `Mod-Exp(x, y, p)`

```plaintext
if (y = 0) then
    return(1).
```
A better approach

Approach II

```
Function MOD-EXP(x, y, p)
    if (y = 0) then
        return(1).
    else
        \textbf{\textit{t}} = MOD-EXP(x, \left\lfloor y/2 \right\rfloor, p).
        \textbf{\textit{t}} = t^{2 \mod p}.
        return(x \cdot t^{2 \mod p}).
    end if
end function
```

Algorithm 5.29: Faster Modular Exponentiation
A better approach

Approach II

Function $\text{MOD-EXP}(x, y, p)$

\[
\begin{align*}
\text{if } (y = 0) & \text{ then} \\
& \text{return}(1).
\end{align*}
\]

else

\[
\begin{align*}
t &= \text{MOD-EXP}(x, \lfloor \frac{y}{2} \rfloor, p).
\end{align*}
\]
Approach II

Function $\text{MOD-EXP}(x, y, p)$

if $(y = 0)$ then
  return(1).
else
  $t = \text{MOD-EXP}(x, \lfloor \frac{y}{2} \rfloor, p)$.
  if $(y$ is even) then
A better approach

Approach II

Function $\text{MOD-EXP}(x, y, p)$

if ($y = 0$) then
  return(1).
else
  $t = \text{MOD-EXP}(x, \lfloor \frac{y}{2} \rfloor, p)$.
  if ($y$ is even) then
    return($t^2 \mod p$).
A better approach

Approach II

Function \text{MOD-EXP}(x, y, p)
    \text{if} (y = 0) \text{ then}
    \text{return}(1).
    \text{else}
    \hspace{1em} t = \text{MOD-EXP}(x, \lfloor \frac{y}{2} \rfloor, p).
    \hspace{1em} \text{if} (y \text{ is even}) \text{ then}
    \hspace{2em} \text{return}(t^2 \mod p).
    \hspace{1em} \text{else}
    \hspace{2em} \text{return}(x \cdot t^2 \mod p).
A better approach

Approach II

Function $\text{MOD-EXP}(x, y, p)$

if $(y = 0)$ then
    return(1).
else
    $t = \text{MOD-EXP}(x, \lfloor \frac{y}{2} \rfloor, p)$.
    if $(y$ is even) then
        return($t^2 \pmod{p}$).
    else
        return($x \cdot t^2 \pmod{p}$).
A better approach

Approach II

Function MOD-EXP(x, y, p)
    if (y = 0) then
        return(1).
    else
        t = MOD-EXP(x, ⌊y/2⌋, p).
        if (y is even) then
            return(t² mod p).
        else
            return(x · t² mod p).
        end if
    end if
A better approach

Approach II

Function `MOD-EXP(x, y, p)`

if \( y = 0 \) then
   return(1).
else
   \( t = MOD-EXP(x, \lfloor \frac{y}{2} \rfloor, p) \).
   if \( y \) is even then
      return\((t^2 \mod p)\).
   else
      return\((x \cdot t^2 \mod p)\).
   end if
end if

Algorithm 5.36: Faster Modular Exponentiation
If $y$ is a power of 2, it is clear that,

$$T(n) = T\left(\frac{n}{2}\right) + 1 = \log_2 n$$

If $n$ is not a power of 2, find all the powers of $x$ up to the largest power of 2 less than $y$.

Then combine these products to get $x^y \mod p$.

The number of multiplications is still $O(\log_2 n)$. 
If $y$ is a power of 2, it is clear that,

$$T(n) = T(n/2) + 1 = \log_2 n$$

If $n$ is not a power of 2, find all the powers of $x$ up to the largest power of 2 less than $y$.

Then combine these products to get $x^y \mod p$.

The number of multiplications is still $O(\log_2 n)$.
If $y$ is a power of 2, it is clear that,
If \( y \) is a power of 2, it is clear that,

\[
T(n) = \log_2 n
\]
If $y$ is a power of 2, it is clear that,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
If $y$ is a power of 2, it is clear that,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$= \log_2 n$$
If $y$ is a power of 2, it is clear that,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$= \log_2 n$$

If $n$ is not a power of 2, find all the powers of $x$ up to the largest power of 2 less than $y$. Then combine these products to get $x^y \mod p$. The number of multiplications is still $O(\log_2 n)$. 

If $y$ is a power of 2, it is clear that,

\[
T(n) = n \left( \frac{n}{2} \right) + 1 = \log_2 n
\]

If $n$ is not a power of 2, find all the powers of $x$ up to the largest power of 2 less than $y$. Then combine these products to get $x^y \mod p$. 
Analysis

If $y$ is a power of 2, it is clear that,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$= \log_2 n$$

If $n$ is not a power of 2, find all the powers of $x$ up to the largest power of 2 less than $y$. Then combine these products to get $x^y \mod p$.

The number of multiplications is still $O(\log_2 n)$.
Matrix multiplication

Problem
Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.

Approach I
1. Compute $C_{ij}$ as the dot product between the $i$th row vector from $A$ ($a_i$) and the $j$th column of $B$ ($b_j$).

Analysis
Computing each product takes $\Theta(n)$ multiplications and $\Theta(n)$ additions. Since there are $n^2$ entries in $C$, it follows that the algorithm takes $\Theta(n^3)$ multiplications and $\Theta(n^3)$ additions.
Matrix multiplication

**Problem**

Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.

**Approach I**

Compute $C_{ij}$ as the dot product between the $i$th row vector from $A$ and the $j$th column of $B$.

**Analysis**

Computing each product takes $\Theta(n)$ multiplications and $\Theta(n)$ additions. Since there are $n^2$ entries in $C$, it follows that the algorithm takes $\Theta(n^3)$ multiplications and $\Theta(n^3)$ additions.
Matrix multiplication

Problem

Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$. 
Matrix multiplication

Problem

*Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.*

Approach I
Matrix multiplication

Problem

*Given two square* $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.

Approach I

1. Compute $C_{ij}$ as the dot product between the $i^{th}$ row vector from $A$ ($a^i$)
**Problem**

*Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.***

**Approach I**

1. Compute $C_{ij}$ as the dot product between the $i^{th}$ row vector from $A$ ($a^i$) and the $j^{th}$ column of $B$ ($b_j$).
Problem

Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.

Approach I

1. Compute $C_{ij}$ as the dot product between the $i^{th}$ row vector from $A$ ($a^i$) and the $j^{th}$ column of $B$ ($b_j$).

Analysis
Matrix multiplication

Problem

*Given two square $n \times n$ matrices $A$ and $B$, compute their product $C = A \cdot B$.*

Approach I

1. Compute $C_{ij}$ as the dot product between the $i^{th}$ row vector from $A$ ($a^i$) and the $j^{th}$ column of $B$ ($b_j$).

Analysis

Computing each product takes $\Theta(n)$ multiplications and $\Theta(n)$ additions.

Since there are $n^2$ entries in $C$, it follows that the algorithm takes $\Theta(n^3)$ multiplications and $\Theta(n^3)$ additions.
A divide and conquer approach

Algorithm 5.37: A Divide and Conquer matrix multiplication algorithm

\[
\text{Function } M_{\text{D and C}}(A, B, n) \\
\text{if } (n = 1) \text{ then return } (A_{11} \cdot B_{11}). \]

Partition \( A \) into 4 square sub-matrices of dimensions \( n^2 \times n^2 \) as shown:

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]

Partition \( B \) into 4 square sub-matrices of dimensions \( n^2 \times n^2 \) as shown:

\[
B = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

end if
A divide and conquer approach

D and C approach

Function $\text{MAT-MULT}(A, B, n)$
A divide and conquer approach

D and C approach

**Function** MAT-MULT(A, B, n)

  if (n = 1) then
A divide and conquer approach

D and C approach

Function MAT-MULT(A, B, n)
if (n = 1) then
    return(A_{11} \cdot B_{11}).
A divide and conquer approach

D and C approach

Function $\text{MAT-MULT}(A, B, n)$

if $(n = 1)$ then
    return $(A_{11} \cdot B_{11})$.
else
A divide and conquer approach

**D and C approach**

**Function** \( \text{MAT-MULT}(A, B; n) \)

if \((n = 1)\) then
    return \((A_{11} \cdot B_{11})\).
else
    Partition \(A\) into 4 square sub-matrices of dimensions \(\frac{n}{2} \times \frac{n}{2}\) as shown:
A divide and conquer approach

**D and C approach**

**Algorithm 5.43:** A Divide and Conquer matrix multiplication algorithm

**Function** \( \text{MAT-MULT}(A, B, n) \)

\[
\text{if } (n = 1) \text{ then} \\
\quad \text{return}(A_{11} \cdot B_{11}). \\
\text{else} \\
\quad \text{Partition } A \text{ into 4 square sub-matrices of dimensions } \frac{n}{2} \times \frac{n}{2} \text{ as shown:}
\]

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]
A divide and conquer approach

D and C approach

Function MAT-MULT(A, B, n)
    if (n = 1) then
        return(A_{11} \cdot B_{11}).
    else
        Partition A into 4 square sub-matrices of dimensions \( \frac{n}{2} \times \frac{n}{2} \) as shown:

        \[
        A = \begin{bmatrix}
        A_{11} & A_{12} \\
        A_{21} & A_{22}
        \end{bmatrix}
        \]

        Partition B into 4 square sub-matrices of dimensions \( \frac{n}{2} \times \frac{n}{2} \) as shown:
### A divide and conquer approach

#### D and C approach

**Function** $\text{MAT-MULT}(A, B, n)$

- **if** $(n = 1)$ **then**
  - return$(A_{11} \cdot B_{11})$.
- **else**
  - Partition $A$ into 4 square sub-matrices of dimensions $\frac{n}{2} \times \frac{n}{2}$ as shown:

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]

  - Partition $B$ into 4 square sub-matrices of dimensions $\frac{n}{2} \times \frac{n}{2}$ as shown:

\[
B = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

**Algorithm 5.45**: A Divide and Conquer matrix multiplication algorithm
Algorithm (contd.)

if \( n > 1 \) then

Let \( C_{11} = \text{MATMULT}(A_{11}, B_{11}, n_2) + \text{MATMULT}(A_{12}, B_{21}, n_2) \).

Let \( C_{12} = \text{MATMULT}(A_{11}, B_{12}, n_2) + \text{MATMULT}(A_{12}, B_{22}, n_2) \).

Let \( C_{21} = \text{MATMULT}(A_{21}, B_{11}, n_2) + \text{MATMULT}(A_{22}, B_{21}, n_2) \).

Let \( C_{22} = \text{MATMULT}(A_{21}, B_{12}, n_2) + \text{MATMULT}(A_{22}, B_{22}, n_2) \).

return \( C = [C_{11} \ C_{12} \ C_{21} \ C_{22}] \).

end if
Algorithm (contd.)

D and C approach (contd.)

if \((n > 1)\) then
Algorithm (contd.)

D and C approach (contd.)

\[
\text{if } (n > 1) \text{ then}
\text{Let } C_{11} = \text{MAT-MULT}(A_{11}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{21}, \frac{n}{2}).
\]
if \( n > 1 \) then
Let \( C_{11} = \text{MAT-MULT}(A_{11}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{21}, \frac{n}{2}) \).
Let \( C_{12} = \text{MAT-MULT}(A_{11}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{22}, \frac{n}{2}) \).
D and C approach (contd.)

\[
\text{if } (n > 1) \text{ then }
\begin{align*}
\text{Let } C_{11} &= \text{MAT-MULT}(A_{11}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{21}, \frac{n}{2}). \\
\text{Let } C_{12} &= \text{MAT-MULT}(A_{11}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{22}, \frac{n}{2}). \\
\text{Let } C_{21} &= \text{MAT-MULT}(A_{21}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{22}, B_{21}, \frac{n}{2}).
\end{align*}
\]

\text{return } C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}
D and C approach (contd.)

\[
\text{if } (n > 1) \text{ then} \\
\text{Let } C_{11} = \text{MAT-MULT}(A_{11}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{21}, \frac{n}{2}). \\
\text{Let } C_{12} = \text{MAT-MULT}(A_{11}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{22}, \frac{n}{2}). \\
\text{Let } C_{21} = \text{MAT-MULT}(A_{21}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{22}, B_{21}, \frac{n}{2}). \\
\text{Let } C_{22} = \text{MAT-MULT}(A_{21}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{22}, B_{22}, \frac{n}{2}).
\]
D and C approach (contd.)

if \( n > 1 \) then

Let \( C_{11} = \text{MAT-MULT}(A_{11}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{21}, \frac{n}{2}) \).

Let \( C_{12} = \text{MAT-MULT}(A_{11}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{12}, B_{22}, \frac{n}{2}) \).

Let \( C_{21} = \text{MAT-MULT}(A_{21}, B_{11}, \frac{n}{2}) + \text{MAT-MULT}(A_{22}, B_{21}, \frac{n}{2}) \).

Let \( C_{22} = \text{MAT-MULT}(A_{21}, B_{12}, \frac{n}{2}) + \text{MAT-MULT}(A_{22}, B_{22}, \frac{n}{2}) \).

return

\[
C = \begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

end if
Analyzing the D and C algorithm

Analysis

\[ T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2) \in \Theta(n^3) \]
Analyzing the D and C algorithm

Analysis

$T(n) = 8T(n/2) + O(n^2)$

$\in \Theta(n^3)$
Analyzing the D and C algorithm

<table>
<thead>
<tr>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ T(n) = ]</td>
</tr>
</tbody>
</table>

\[ T(n) = 8 \cdot T(n/2) + O(n^2) \in \Theta(n^3) \]
Analyzing the D and C algorithm

\[ T(n) = 8 \cdot T \left( \frac{n}{2} \right) \]
Analyzing the D and C algorithm

\[ T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^2) \]
Analyzing the D and C algorithm

\[
T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(n^2)
\]
\[
\in \Theta(n^3)
\]
The Strassen approach

Compute the following matrix products:

- $S_1 = A_{11} \cdot (B_{12} - B_{22})$.
- $S_2 = (A_{11} + A_{12}) \cdot B_{22}$.
- $S_3 = (A_{21} + A_{22}) \cdot B_{11}$.
- $S_4 = A_{22} \cdot (B_{21} - B_{11})$.
- $S_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$.
- $S_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$.
- $S_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{21})$. 

Algorithmic Insights

Computational Complexity
The Strassen approach

Clever sub-matrix multiplication

Compute the following matrix products:

\[ S_1 = A_{11} \cdot (B_{12} - B_{22}) \]

\[ S_2 = (A_{11} + A_{12}) \cdot B_{22} \]

\[ S_3 = (A_{21} + A_{22}) \cdot B_{11} \]

\[ S_4 = A_{22} \cdot (B_{21} - B_{11}) \]

\[ S_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]

\[ S_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]

\[ S_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{21}) \]
Clever sub-matrix multiplication

1. Compute the following matrix products:
The Strassen approach

Clever sub-matrix multiplication

1 Compute the following matrix products:

\[ S_1 = A_{11} \cdot (B_{12} - B_{22}). \]
1. Compute the following matrix products:

\[
S_1 = A_{11} \cdot (B_{12} - B_{22}). \\
S_2 = (A_{11} + A_{12}) \cdot B_{22}.
\]
Clever sub-matrix multiplication

1. Compute the following matrix products:

\[ S_1 = A_{11} \cdot (B_{12} - B_{22}) \cdot \]
\[ S_2 = (A_{11} + A_{12}) \cdot B_{22} \cdot \]
\[ S_3 = (A_{21} + A_{22}) \cdot B_{11} \cdot \]
The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

\[
S_1 = A_{11} \cdot (B_{12} - B_{22}).
\]

\[
S_2 = (A_{11} + A_{12}) \cdot B_{22}.
\]

\[
S_3 = (A_{21} + A_{22}) \cdot B_{11}.
\]

\[
S_4 = A_{22} \cdot (B_{21} - B_{11}).
\]
The Strassen approach

Clever sub-matrix multiplication

Compute the following matrix products:

\[ S_1 = A_{11} \cdot (B_{12} - B_{22}). \]
\[ S_2 = (A_{11} + A_{12}) \cdot B_{22}. \]
\[ S_3 = (A_{21} + A_{22}) \cdot B_{11}. \]
\[ S_4 = A_{22} \cdot (B_{21} - B_{11}). \]
\[ S_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}). \]
Clever sub-matrix multiplication

1. Compute the following matrix products:

\[
\begin{align*}
S_1 & = A_{11} \cdot (B_{12} - B_{22}). \\
S_2 & = (A_{11} + A_{12}) \cdot B_{22}. \\
S_3 & = (A_{21} + A_{22}) \cdot B_{11}. \\
S_4 & = A_{22} \cdot (B_{21} - B_{11}). \\
S_5 & = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}). \\
S_6 & = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}).
\end{align*}
\]
Clever sub-matrix multiplication

1. Compute the following matrix products:

\[
\begin{align*}
S_1 & = A_{11} \cdot (B_{12} - B_{22}). \\
S_2 & = (A_{11} + A_{12}) \cdot B_{22}. \\
S_3 & = (A_{21} + A_{22}) \cdot B_{11}. \\
S_4 & = A_{22} \cdot (B_{21} - B_{11}). \\
S_5 & = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}). \\
S_6 & = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}). \\
S_7 & = (A_{11} - A_{21}) \cdot (B_{11} + B_{21}).
\end{align*}
\]
Review of concepts
Algorithmic Insights
Recursion
Divide and Conquer

Strassen (contd.)

Observe that,
\[ C_{11} = S_4 + S_5 + S_6 - S_2 \]
\[ C_{12} = S_1 + S_2 \]
\[ C_{21} = S_3 + S_4 \]
\[ C_{22} = S_1 - S_3 + S_5 - S_7 \]
Completing the algorithm

Observe that,

\[ C_{11} = S_4 + S_5 + S_6 - S_2 \]

\[ C_{12} = S_1 + S_2 \]

\[ C_{21} = S_3 + S_4 \]

\[ C_{22} = S_1 - S_3 + S_5 - S_7 \]
Complementing the algorithm

1. Observe that,
Completing the algorithm

1. Observe that,

\[ C_{11} = \]
Completing the algorithm

1. Observe that,

\[ C_{11} = S_4 + S_5 + S_6 - S_2 \]
Completing the algorithm

Observe that,

\[ C_{11} = S_4 + S_5 + S_6 - S_2 \]
\[ C_{12} = S_1 + S_2 \]
Completing the algorithm

Observe that,

\[
\begin{align*}
C_{11} &= S_4 + S_5 + S_6 - S_2 \\
C_{12} &= S_1 + S_2 \\
C_{21} &= S_3 + S_4
\end{align*}
\]
Completing the algorithm

Observe that,

\[ C_{11} = S_4 + S_5 + S_6 - S_2 \]
\[ C_{12} = S_1 + S_2 \]
\[ C_{21} = S_3 + S_4 \]
\[ C_{22} = S_1 - S_3 + S_5 - S_7 \]
Analysis of Strassen
Analysis of Strassen

Running Time

\[ T(n) = 7 \cdot T(n^{2}) + O(n^{2}) \in O(n \log_{2}7) \]
Recursion
Divide and Conquer

Analysis of Strassen

Running Time

\[ T(n) = 7 \cdot T(n^2) + O(n^2) \in O(n \log_2 7) \]
Analysis of Strassen

Running Time

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$
Analysis of Strassen

Running Time

\[ T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n^2) \]
\[ \in O(n^{\log_2 7}) \]