Algorithmic Insights I - Recursion and Divide and Conquer

K. Subramani[1]

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

February 9, 2015

Outline

Outline

Outline

Outline

Review

Review

Main concepts

Review

Main concepts

1. Instance, Problem, Solutions.

Review

Main concepts

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

Review

Main concepts

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

Review

Main concepts

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

Review

Main concepts

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

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**.

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**.

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.

Algorithmic Insights

Algorithmic Insights

Main concepts

Algorithmic Insights

Main concepts

What makes a problem tractable?

Algorithmic Insights

Main concepts

What makes a problem tractable?

1. Recursion.

Algorithmic Insights

Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.

Algorithmic Insights

Main concepts

What makes a problem tractable?

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

Algorithmic Insights

Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
4. Dynamic Programming.

Algorithmic Insights

Main concepts

What makes a problem tractable?

1. Recursion.
2. Divide and Conquer.
3. Greedy.
4. Dynamic Programming.
5. Iterative approaches (Rewriting).

Algorithmic Insights

Main concepts

What makes a problem tractable?

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

Recursion

Recursion

Main Idea

Recursion

Main Idea

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

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

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).

The Array-Max problem

The Array-Max problem

Problem

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

The Array-Max problem

Problem

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

Algorithm

Function ARRAY-MAX(**A**, *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**

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($A[n]$)

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

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**, $n - 1$))).

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**, $n - 1$))).
  end if

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

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 (max($A[n]$, ARRAY-MAX(**A**, $n - 1$))).
  end if

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$(\mathbf{A}, n-1)))$.
  end if

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

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

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.*

Algorithm

Algorithm

Algorithm

Algorithm

Algorithm

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

Algorithm

Algorithm

Function ARRAY-SEARCH(**A**, n, k)
  if ($n = 1$) **then**

Algorithm

Algorithm

Function ARRAY-SEARCH(**A**, n, k)
  if ($n = 1$) **then**
    if ($A[n] = k$) **then**

Algorithm

Algorithm

Function ARRAY-SEARCH(**A**, n, k)
  if ($n = 1$) **then**
    if ($A[n] = k$) **then**
      return(**true**)

Algorithm

Algorithm

Function ARRAY-SEARCH(**A**, n, k)
  if ($n = 1$) **then**
    if ($A[n] = k$) **then**
      return(**true**)
    else

Algorithm

Algorithm

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

Algorithm

Algorithm

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

Algorithm

Algorithm

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

Algorithm

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**

Algorithm

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**)

Algorithm

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

Algorithm

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**, $n - 1, k$)).

Algorithm

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

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
  end if

Algorithm 4.30: Searching for a key in an array

Analysis

Analysis

Analysis

$T(n) =$

Analysis

Analysis

$$T(n) = \begin{cases} 1, & \text{if } n = 1, \\ \end{cases}$$

Analysis

Analysis

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

Analysis

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

The Towers of Hanoi problem

Problem

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

Algorithm

Main idea

Algorithm

Main idea

Break the task into three sub-tasks.

Algorithm

Main idea

Break the task into three sub-tasks.

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

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.

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.

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

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

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ \end{cases}$$

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

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

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

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

Approach

Approach

Main Concepts

Approach

Main Concepts

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

Approach

Main Concepts

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

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

Approach

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.

Approach

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.

Divide and Conquer

Divide and Conquer

The Master Theorem

Divide and Conquer

The Master Theorem

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

Divide and Conquer

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.

Divide and Conquer

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:

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

Divide and Conquer

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:

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

Then,

Divide and Conquer

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:

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

Then,

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

Divide and Conquer

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:

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

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 $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

Divide and Conquer

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:

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

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

The Merge-Sort Algorithm

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

Sorting through Merging

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

if ($low < high$) **then**
$mid = \frac{low+high}{2}$.

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*).

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*).

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 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*).
end if

Algorithm 5.8: MergeSort

Analyzing Time and Space

Analyzing Time and Space

Analysis

Analyzing Time and Space

Analysis

$$T(n) \quad =$$

Analyzing Time and Space

Analysis

$$T(n) = 2 \cdot T(\frac{n}{2}) + n$$

Analyzing Time and Space

Analysis

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + n \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Analyzing Time and Space

Analysis

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + n \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

$$S(n) \in \Theta(n)$$

The Quick-Sort Algorithm

The Quick-Sort Algorithm

Sorting through Partitioning

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

The Quick-Sort Algorithm

Sorting through Partitioning

Function QUICK-SORT(**A**, *low*, *high*)
  if (*low* $<$ *high*) **then**

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

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.

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*, $j - 1$).

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*).

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 5.16: 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)

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 5.18: 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

Analysis of running time

Best Case

Analysis of running time

Best Case

$$T(n) \quad =$$

Analysis of running time

Best Case

$$T(n) = 2 \cdot T(\frac{n-1}{2}) + (n-1)$$

Analysis of running time

Best Case

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + (n-1) \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Analysis of running time

Best Case

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + (n-1) \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Worst-case

Analysis of running time

Best Case

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + (n-1) \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Worst-case

$$T(n) =$$

Analysis of running time

Best Case

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + (n-1) \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Worst-case

$$T(n) = T(n-1) + (n-1)$$

Analysis of running time

Best Case

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + (n-1) \\ &\in \Theta(n \cdot \log n) \end{aligned}$$

Worst-case

$$\begin{aligned} T(n) &= T(n-1) + (n-1) \\ &\in \Theta(n^2) \end{aligned}$$

Average case analysis

Average case analysis

Average case

$$T(n) \quad =$$

Average case analysis

Average case

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

Average case analysis

Average case

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

$$= (n-1) +$$

Average case analysis

Average case

$$
\begin{aligned}
T(n) &= (n-1) + \frac{1}{n} \cdot \sum_{r=1}^{n} [T(r-1) + T(n-r)] \\
&= (n-1) + \frac{2}{n} \cot \sum_{r=1}^{n} T(r-1)
\end{aligned}
$$

Average case analysis

Average case

$$
\begin{aligned}
T(n) &= (n-1) + \frac{1}{n} \cdot \sum_{r=1}^{n} [T(r-1) + T(n-r)] \\
&= (n-1) + \frac{2}{n} \cot \sum_{r=1}^{n} T(r-1) \\
&\approx 2 \cdot n \cdot \ln n
\end{aligned}
$$

Modular Exponentiation

Modular Exponentiation

Problem

Modular Exponentiation

Problem

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

Modular Exponentiation

Problem

Given two n digit integers x and y, compute $x^y \bmod p$. *Useful in cryptography and primality checking.*

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**
    $r = x \cdot r \bmod p$.
  end for
  return(y).

Algorithm 5.24: Modular Exponentiation

Time Analysis

Time Analysis

Analysis

Time Analysis

Analysis

Assuming x and y have n digits, the number of multiplications is proportional to

Time Analysis

Analysis

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

A better approach

A better approach

Approach II

Function MOD-EXP(x, y, p)

A better approach

Approach II

Function MOD-EXP(x, y, p)
  if ($y = 0$) **then**

A better approach

Approach II

Function MOD-EXP(x, y, p)
  if ($y = 0$) **then**
    return(1).

A better approach

Approach II

Function MOD-EXP(x, y, p)
  if ($y = 0$) **then**
    return(1).
  else

A better approach

Approach II

Function MOD-EXP(x, y, p)
  if ($y = 0$) **then**
    return(1).
  else
    $t =$MOD-EXP($x, \lfloor \frac{y}{2} \rfloor, p$).

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**

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).

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

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($t^2 \bmod p$).
    else
      return($x \cdot t^2 \bmod p$).

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($t^2 \bmod p$).
    else
      return($x \cdot t^2 \bmod p$).
    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($t^2 \bmod p$).
    else
      return($x \cdot t^2 \bmod p$).
    end if
  end if

Algorithm 5.36: Faster Modular Exponentiation

Time Analysis

Time Analysis

Analysis

Time Analysis

Analysis

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

Time Analysis

Analysis

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

$$T(n) \quad =$$

Time Analysis

Analysis

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

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

Time Analysis

Analysis

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

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= \log_2 n \end{aligned}$$

Time Analysis

Analysis

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

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= \log_2 n \end{aligned}$$

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

Time Analysis

Analysis

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

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= \log_2 n \end{aligned}$$

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 $x^y \bmod p$.

Time Analysis

Analysis

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

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= \log_2 n \end{aligned}$$

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 $x^y \bmod p$.

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

Matrix multiplication

Matrix multiplication

Problem

Matrix multiplication

Problem

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

Matrix multiplication

Problem

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

Approach I

Matrix multiplication

Problem

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

Approach I

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

Matrix multiplication

Problem

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

Approach I

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

Matrix multiplication

Problem

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

Approach I

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

Analysis

Matrix multiplication

Problem

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

Approach I

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

Analysis

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

Since there are n^2 entries in $\mathbf{C}$, it follows that the algorithm takes $\Theta(n^3)$ multiplications and $\Theta(n^3)$ additions.

A divide and conquer approach

A divide and conquer approach

D and C approach

Function 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 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 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

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:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix}$$

A divide and conquer approach

D and C approach

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

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:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix}$$

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

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:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix}$$

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

$$\mathbf{B} = \begin{bmatrix} \mathbf{B_{11}} & \mathbf{B_{12}} \\ \mathbf{B_{21}} & \mathbf{B_{22}} \end{bmatrix}$$

end if

Algorithm 5.45: A Divide and Conquer matrix multiplication algorithm

Algorithm (contd.)

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**
Let $\mathbf{C_{11}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{21}}, \frac{n}{2})$.

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**
Let $\mathbf{C_{11}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{21}}, \frac{n}{2})$.
Let $\mathbf{C_{12}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{12}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{22}}, \frac{n}{2})$.

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**
Let $\mathbf{C_{11}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{21}}, \frac{n}{2})$.
Let $\mathbf{C_{12}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{12}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{22}}, \frac{n}{2})$.
Let $\mathbf{C_{21}} = \text{MAT-MULT}(\mathbf{A_{21}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{22}}, \mathbf{B_{21}}, \frac{n}{2})$.

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**
Let $\mathbf{C_{11}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{21}}, \frac{n}{2})$.
Let $\mathbf{C_{12}} = \text{MAT-MULT}(\mathbf{A_{11}}, \mathbf{B_{12}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{12}}, \mathbf{B_{22}}, \frac{n}{2})$.
Let $\mathbf{C_{21}} = \text{MAT-MULT}(\mathbf{A_{21}}, \mathbf{B_{11}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{22}}, \mathbf{B_{21}}, \frac{n}{2})$.
Let $\mathbf{C_{22}} = \text{MAT-MULT}(\mathbf{A_{21}}, \mathbf{B_{12}}, \frac{n}{2}) + \text{MAT-MULT}(\mathbf{A_{22}}, \mathbf{B_{22}}, \frac{n}{2})$.

Algorithm (contd.)

D and C approach (contd.)

if $(n > 1)$ **then**

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

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

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

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

return

$$\mathbf{C} = \begin{bmatrix} \mathbf{C_{11}} & \mathbf{C_{12}} \\ \mathbf{C_{21}} & \mathbf{C_{22}} \end{bmatrix}$$

end if

Analyzing the D and C algorthm

Analyzing the D and C algorthm

Analysis

Analyzing the D and C algorthm

Analysis

$$T(n) \quad =$$

Analyzing the D and C algorthm

Analysis

$$T(n) = 8 \cdot T(\frac{n}{2})$$

Analyzing the D and C algorithm

Analysis

$$T(n) = 8 \cdot T(\frac{n}{2}) + O(n^2)$$

Analyzing the D and C algorithm

Analysis

$$\begin{aligned} T(n) &= 8 \cdot T(\frac{n}{2}) + O(n^2) \\ &\in \Theta(n^3) \end{aligned}$$

The Strassen approach

The Strassen approach

Clever sub-matrix multiplication

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$\mathbf{S_1} = \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}).$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$\begin{aligned} \mathbf{S_1} &= \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}). \\ \mathbf{S_2} &= (\mathbf{A_{11}} + \mathbf{A_{12}}) \cdot \mathbf{B_{22}}. \end{aligned}$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$\begin{aligned} \mathbf{S_1} &= \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}). \\ \mathbf{S_2} &= (\mathbf{A_{11}} + \mathbf{A_{12}}) \cdot \mathbf{B_{22}}. \\ \mathbf{S_3} &= (\mathbf{A_{21}} + \mathbf{A_{22}}) \cdot \mathbf{B_{11}}. \end{aligned}$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$
\begin{aligned}
\mathbf{S_1} &= \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}). \\
\mathbf{S_2} &= (\mathbf{A_{11}} + \mathbf{A_{12}}) \cdot \mathbf{B_{22}}. \\
\mathbf{S_3} &= (\mathbf{A_{21}} + \mathbf{A_{22}}) \cdot \mathbf{B_{11}}. \\
\mathbf{S_4} &= \mathbf{A_{22}} \cdot (\mathbf{B_{21}} - \mathbf{B_{11}}).
\end{aligned}
$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$
\begin{aligned}
\mathbf{S_1} &= \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}). \\
\mathbf{S_2} &= (\mathbf{A_{11}} + \mathbf{A_{12}}) \cdot \mathbf{B_{22}}. \\
\mathbf{S_3} &= (\mathbf{A_{21}} + \mathbf{A_{22}}) \cdot \mathbf{B_{11}}. \\
\mathbf{S_4} &= \mathbf{A_{22}} \cdot (\mathbf{B_{21}} - \mathbf{B_{11}}). \\
\mathbf{S_5} &= (\mathbf{A_{11}} + \mathbf{A_{22}}) \cdot (\mathbf{B_{11}} + \mathbf{B_{22}}).
\end{aligned}
$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

$$
\begin{aligned}
\mathbf{S_1} &= \mathbf{A_{11}} \cdot (\mathbf{B_{12}} - \mathbf{B_{22}}). \\
\mathbf{S_2} &= (\mathbf{A_{11}} + \mathbf{A_{12}}) \cdot \mathbf{B_{22}}. \\
\mathbf{S_3} &= (\mathbf{A_{21}} + \mathbf{A_{22}}) \cdot \mathbf{B_{11}}. \\
\mathbf{S_4} &= \mathbf{A_{22}} \cdot (\mathbf{B_{21}} - \mathbf{B_{11}}). \\
\mathbf{S_5} &= (\mathbf{A_{11}} + \mathbf{A_{22}}) \cdot (\mathbf{B_{11}} + \mathbf{B_{22}}). \\
\mathbf{S_6} &= (\mathbf{A_{12}} - \mathbf{A_{22}}) \cdot (\mathbf{B_{21}} + \mathbf{B_{22}}).
\end{aligned}
$$

The Strassen approach

Clever sub-matrix multiplication

1. Compute the following matrix products:

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

Strassen (contd.)

Strassen (contd.)

Completing the algorithm

Strassen (contd.)

Completing the algorithm

1. Observe that,

Strassen (contd.)

Completing the algorithm

1. Observe that,

$$\mathbf{C_{11}} =$$

Strassen (contd.)

Completing the algorithm

1. Observe that,

$$C_{11} = S_4 + S_5 + S_6 - S_2$$

Strassen (contd.)

Completing the algorithm

1. Observe that,

$$\begin{aligned} C_{11} &= S_4 + S_5 + S_6 - S_2 \\ C_{12} &= S_1 + S_2 \end{aligned}$$

Strassen (contd.)

Completing the algorithm

1. Observe that,

$$\begin{aligned} C_{11} &= S_4 + S_5 + S_6 - S_2 \\ C_{12} &= S_1 + S_2 \\ C_{21} &= S_3 + S_4 \end{aligned}$$

Strassen (contd.)

Completing the algorithm

1. Observe that,

$$\begin{aligned} C_{11} &= S_4 + S_5 + S_6 - S_2 \\ C_{12} &= S_1 + S_2 \\ C_{21} &= S_3 + S_4 \\ C_{22} &= S_1 - S_3 + S_5 - S_7 \end{aligned}$$

Analysis of Strassen

Analysis of Strassen

Running Time

Analysis of Strassen

Running Time

$$T(n) \quad =$$

Analysis of Strassen

Running Time

$$T(n) = 7 \cdot T(\frac{n}{2}) + O(n^2)$$

Analysis of Strassen

Running Time

$$\begin{aligned} T(n) &= 7 \cdot T(\frac{n}{2}) + O(n^2) \\ &\in O(n^{\log_2 7}) \end{aligned}$$