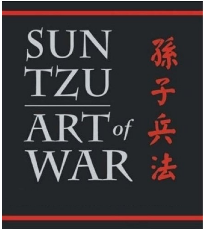

THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Algorithms and Data Structures

Lecture 7 Divide and Conquer

Jiamou Liu
The University of Auckland



故用兵之法，十則圍之，五則攻之，倍則分之， 敵則能戰之，少則能守之，不若則能避之。

``It is the rule in war, if ten times the enemy's strength, surround them; if five times, attack them; if double, be able to divide them; if equal,engage them; if fewer, be able to evade them; if weaker, be able to avoid them.''

---``Chapter III Strategic Attack" 500BC

Algorithm Design Strategy: Divide-and-Conquer

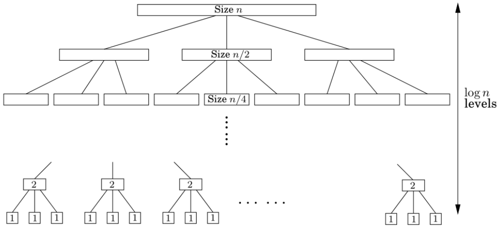
- Karatsuba's algorithm solves integer multiplication in time $O(n^{1.59})$.
- The technique used is called divide-and-conquer

Algorithm Design Strategy: Divide-and-Conquer

- Karatsuba's algorithm solves integer multiplication in time $O(n^{1.59})$.
- The technique used is called divide-and-conquer

Divide-and-Conquer as an algorithm design technique

The divide-and-conquer technique solves a computational problem by dividing it into one or more subprograms of smaller size, conquering each of them by solving them recursively, and then combining their solutions into a solution for the original problem.



Divide-and-Conquer

General Divide-and-Conquer Strategy

`if` $n \leq n_0$ `then`

directly solve problem without dividing

`else`

divide problem into a subproblems of size n/b each

`for` $i \leftarrow 0$ to $a - 1$ `do`

recursively solve the ith subproblem

combine the a solutions into a solution of the original problem

Divide-and-Conquer

General Divide-and-Conquer Strategy

`if` $n \le n_0$ `then`
  directly solve problem without dividing
`else`
  divide problem into a subproblems of size n/b each
  `for` $i \leftarrow 0$ to $a-1$ `do`
    recursively solve the ith subproblem
  combine the a solutions into a solution of the original problem

Running Time Analysis

Come up with a recurrence: $T(n) = aT(n/b) + f(n)$

Example 1: Karatsuba's Algorithm

Karatsuba's algorithm

Given two input numbers x, y: `if` x, y both have length 1 `then`
    directly multiply x, y
`else`
    divide each x and y into two numbers and
    obtain 3 subproblems of size $n/2$ each
    `for` each subprogram `do`
        recursively solve the ith subproblem
    add the 3 solutions

Time complexity: $T(n) = 3T(n/2) + cn$.

Example 2: Binary Search

Problem

Search from a number from a sorted sequence of numbers.

Example 2: Binary Search

Problem

Search from a number from a sorted sequence of numbers.

e.g. Search for the number 64.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	29	43	45	56	58	71	78	83	91	95	99	156	171	222	291
0	29	43	45	56	58	71	78	83	91	95	99	156	171	222	291
0	29	43	45	56	58	71	78	83	91	95	99	156	171	222	291
0	29	43	45	56	58	71									
0	29	43	45	56	58	71									
				56	58	71									
				56	58	71									
						71									
						71									

The number 64 does not belong to the sequence.

Example 2: Binary Search

function BinarySearch(*arr*, *target*, *start*, *end*)
  INPUT: Integer array *arr*, indices *start*, *end*, integer *target*
  OUTPUT: Yes if *target* is in *arr*; No otherwise
  if $start > end$ **then**
    return "the element doesn't belong to the sequence"
  $middle \leftarrow \lceil (start + end)/2 \rceil$
  if $arr[middle] = target$ **then**
    return "the element is found"
  else if $target < arr[middle]$ **then**
    return BinarySearch($arr, target, start, middle - 1$)
  else
    return BinarySearch($arr, target, middle + 1, end$)

Example 2: Binary Search

function BinarySearch(*arr*, *target*, *start*, *end*)
    INPUT: Integer array *arr*, indices *start*, *end*, integer *target*
    OUTPUT: Yes if *target* is in *arr*; No otherwise
    if $start > end$ **then**
        return "the element doesn't belong to the sequence"
    $middle \leftarrow \lceil (start + end)/2 \rceil$
    if $arr[middle] = target$ **then**
        return "the element is found"
    else if $target < arr[middle]$ **then**
        return BinarySearch($arr, target, start, middle - 1$)
    else
        return BinarySearch($arr, target, middle + 1, end$)

Time complexity: $T(n) = T(n/2) + c$

Runtime Analysis

Question. Is there a general scheme to evaluate the time complexity of divide-and-conquer algorithms?

Divide and Conquer

The *running time* $T(n)$ of a Divide-and-Conquer algorithm can normally be specified by

$$T(n) = aT(n/b) + f(n).$$

The problem is entirely mathematical: Solve the above recursion.

Master Theorem

What is the master theorem?

The master theorem provides a direct way to solve recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ are constants and $f(n)$ is a positive function.

Master Theorem

What is the master theorem?

The master theorem provides a direct way to solve recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ are constants and $f(n)$ is a positive function.

Examples

- $T(n) = 3T(n/2) + n^2$
- $T(n) = 16T(n/2) + 3n \log n$
- $T(n) = T(n/2) + 3$
- $T(n) = \sqrt{2}T(n/4) + n^{0.51}$

Intuitive Version

The master theorem allows us to solve the recurrence

$$T(n) = aT(n/b) + f(n)$$

by comparing the function $f(n)$ with $n^{\log_b a}$.
There are three cases:

- Case 1: $f(n)$ is much smaller than $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a}$.
- Case 2: $f(n)$ is the same with $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a} \log n$.
- Case 3: $f(n)$ is much bigger than $n^{\log_b a}$.
  Then $T(n)$ has complexity $f(n)$.

Master theorem

Let $a \geq 1$ and $b > 1$, let $f(n)$ be a positive function, and let $T(n)$ be defined as:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then there are three cases:

1. If $f(n)$ is $O(n^{\log_b a - e})$ for some constant $e > 0$, then

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

2. If $f(n)$ is $\Theta(n^{\log_b a})$, then

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

3. If $f(n)$ is $\Omega(n^{\log_b a + e})$ for some constant $e > 0$, and the regularity condition $af(n/b) \leq rf(n)$ for some $r < 1$ holds, then

$$T(n) = \Theta(f(n)).$$

Note: Most of the functions we see satisfy the regularity condition.

Examples

- $T(n) = 9T(n/3) + n.$

- $T(n) = 4T(n/2) + n^2.$

- $T(n) = 3T(n/3) + n^2.$

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
- $T(n) = 3T(n/3) + n^2$.

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
  $a = 4, b = 2, f(n) = n^2, n^{\log_b a} = n^2$.
  $f(n)$ is $\Theta(n^2)$. Hence we apply case 2.
  $T(n)$ is $\Theta(n^2 \log n)$.
- $T(n) = 3T(n/3) + n^2$.

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3$, $f(n) = n$, $n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
  $a = 4, b = 2$, $f(n) = n^2$, $n^{\log_b a} = n^2$.
  $f(n)$ is $\Theta(n^2)$. Hence we apply case 2.
  $T(n)$ is $\Theta(n^2 \log n)$.
- $T(n) = 3T(n/3) + n^2$.
  $a = 3, b = 3$, $f(n) = n^2$, $n^{\log_b a} = n^{\log_3 3} = n$.
  $f(n)$ is $\Omega(n^e)$ for some e (say $e = 0.5$).
  Hence we apply case 3.
  $T(n)$ is $\Theta(n^2)$.

Note: Master theorem holds without assuming n is a power of b.

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.
- When $f(n)$ is smaller than $n^{\log_b a}$ but is not small enough:
  ($f(n)$ is $O(n^{\log_b a})$ but $f(n)$ is not $O(n^{\log_b a - e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n/\log n$.

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.
- When $f(n)$ is smaller than $n^{\log_b a}$ but is not small enough:
  ($f(n)$ is $O(n^{\log_b a})$ but $f(n)$ is not $O(n^{\log_b a - e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n/\log n$.
- When $f(n)$ is bigger than $n^{\log_b a}$ but is not big enough:
  ($f(n)$ is $\Omega(n^{\log_b a})$ but $f(n)$ is not $\Omega(n^{\log_b a + e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n \log n$.

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

- $n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$
- $f(n) = cn$
- cn is $O(n^{1.59-0.1})$

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

- $n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$
- $f(n) = cn$
- cn is $O(n^{1.59-0.1})$

Case 1: cn is much smaller than $n^{\log_2 3}$.
$\Rightarrow T(n)$ is $\Theta(n^{\log_2 3})$

Example 2: Binary Search

Running time: $T(n) = T(n/2) + c$

Example 2: Binary Search

Running time: $T(n) = T(n/2) + c$

- $n^{\log_2 1} = n^0 = 1$
- $f(n) = c$
- c is $\Theta(1)$

Example 2: Binary Search

Running time: $T(n) = T(n/2) + c$

- $n^{\log_2 1} = n^0 = 1$
- $f(n) = c$
- c is $\Theta(1)$

Case 2: c has the same asymptotic growth as n^0.
$\Rightarrow T(n)$ is $\Theta(\log n)$

Summary

In this lecture, we introduce the algorithm design strategy: **divide and conquer**.

- Divide the problem of size n into one or more subproblems of smaller size
- Conquer each sub-problem by solving them recursively
- Combine their solutions into a solution for the original problem

Examples:

- Karatsuba's algorithm
- Binary search

Analysis of the time complexity of divide and conquer algorithms: $T(n) = aT(n/b) + f(n)$: **Master theorem**: Three cases.


THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND