Lecture 2: Divide and Conquer & Recurrences

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 12, 2023

Overview

- Divide-and-Conquer Paradigm
- Solving Recurrences
- Optional: Maximum Subarray Sum
- Acknowledgements

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.

  Examples:

  1. Sorting: merge sort
  2. Searching: binary search
  3. Matrix Multiplication
  4. Polynomial Multiplication

  many more, (see [CLRS 2009])

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.
- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.
- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.
- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$
  3. **Combine:** solutions $S_1, \ldots, S_a \mapsto$ solution S to instance I

Divide-and-Conquer

- Many problems can be efficiently solved by *dividing* them into *smaller subproblems*, and then *combining* the subproblems to give solution to original problem.
- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$
  3. **Combine:** solutions $S_1, \ldots, S_a \mapsto$ solution S to instance I
- "Recursion for running time:"

$$T(I) = T(I_1) + \cdots + T(I_a) + \text{ time to combine}$$

Example: Merge Sort

- **Input:** array A with n elements
- **Output:** sorted array A

Example: Merge Sort

- **Input:** array A with n elements
- **Output:** sorted array A
- Divide and Conquer algorithm:

sort($A[\alpha, \beta]$):

1. If $\beta - \alpha < 10$, then trivially sort array and return.
2. $B = \text{sort}(A[\alpha, \lfloor(\alpha+\beta)/2\rfloor])$, $C = \text{sort}(A[\lfloor(\alpha+\beta)/2\rfloor + 1, \beta])$
3. return merge(B, C)

Example: Merge Sort

- **Input:** array A with n elements
- **Output:** sorted array A
- Divide and Conquer algorithm:

sort($A[\alpha, \beta]$):

1. If $\beta - \alpha < 10$, then trivially sort array and return.
2. $B = \text{sort}(A[\alpha, \lfloor(\alpha+\beta)/2\rfloor])$, $C = \text{sort}(A[\lfloor(\alpha+\beta)/2\rfloor + 1, \beta])$
3. return merge(B, C)

- Merging algorithm: (input arrays sorted in increasing order)

merge(B, C) :

1. Let $D = []$ be an empty array, and let i, j be two pointers, indexing position on arrays B, C, initialized at 1.
2. Until we are done scanning both B, C:
    - If $B[i] \leq C[j]$, then D.append($B[i]$) and $i \leftarrow i + 1$
    - Else, D.append($C[j]$) and $j \leftarrow j + 1$

Analysis: Merge Sort

- $T_{\text{merge}}(n) = c \cdot n$, since we are doing a linear scan over the input

Analysis: Merge Sort

- $T_{\text{merge}}(n) = c \cdot n$, since we are doing a linear scan over the input
- Letting $T(n)$ be the running time of merge sort on inputs of size n, we see:

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

Analysis: Merge Sort

- $T_{\text{merge}}(n) = c \cdot n$, since we are doing a linear scan over the input
- Letting $T(n)$ be the running time of merge sort on inputs of size n, we see:

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

- Recursion tree (see board): $T(n) = \Theta(n \cdot \log n)$

Analysis: Merge Sort

- $T_{\text{merge}}(n) = c \cdot n$, since we are doing a linear scan over the input
- Letting $T(n)$ be the running time of merge sort on inputs of size n, we see:

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

- Recursion tree (see board): $T(n) = \Theta(n \cdot \log n)$
- Can also "guess and check" the answer

$$\begin{aligned} T(n) &= cn \log n && \text{(guess)} \\ T(n) &= 2 \cdot \left(c \cdot \frac{n}{2} \log(n/2) \right) + cn \\ &= cn(\log n - 1) + cn = cn \log n \end{aligned}$$

- Divide-and-Conquer Paradigm
- Solving Recurrences
- Optional: Maximum Subarray Sum
- Acknowledgements

Recurrences

- Divide-and-conquer leads naturally to the problem of solving recurrences (to get runtime bounds)
- Mergesort recurrence was easy to analyze.

What about in general?

How can we deal with more general recurrences?

Recurrences

- Divide-and-conquer leads naturally to the problem of solving recurrences (to get runtime bounds)
- Mergesort recurrence was easy to analyze.

What about in general?

How can we deal with more general recurrences?

Theorem (Master Theorem (simple))

Given recurrence

$$T(n) = aT(n/b) + \Theta(n^c)$$

with $T(1)$, $a \geq 1, b > 1, c \geq 0$ *(constants), then*

$$T(n) = \begin{cases} \Theta(n^c), & \textit{if } c > \log_b a \\ \Theta(n^c \log n), & \textit{if } c = \log_b a \\ \Theta(n^{\log_b a}), & \textit{if } c < \log_b a \end{cases}$$

Proof of Master Theorem

- *Remark*: it is more important to remember the method than the result
  - Prof. Lau

Proof method works more generally.

Proof of Master Theorem

- *Remark*: it is more important to remember the method than the result

  - Prof. Lau

Proof method works more generally.

- Recursion tree: (see board)
  1. If $c > \log_b a$, then top level dominates

     decreasing geometric sequence, ratio $a/b^c < 1$

Proof of Master Theorem

- *Remark*: it is more important to remember the method than the result
  - Prof. Lau

Proof method works more generally.

- Recursion tree: (see board)
  1. If $c > \log_b a$, then top level dominates
     decreasing geometric sequence, ratio $a/b^c < 1$
  2. If $c = \log_b a$, then every layer same, and $\theta(\log n)$ layers

Proof of Master Theorem

- *Remark*: it is more important to remember the method than the result
  - Prof. Lau

  Proof method works more generally.

- Recursion tree: (see board)
  1. If $c > \log_b a$, then top level dominates
     decreasing geometric sequence, ratio $a/b^c < 1$
  2. If $c = \log_b a$, then every layer same, and $\theta(\log n)$ layers
  3. If $c < \log_b a$, then bottom level dominates
     increasing geometric sequence, ratio $a/b^c > 1$

General Master Theorem

Theorem (Master Theorem)

Given recurrence

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

with $T(1), f(1)$, $a \geq 1, b > 1$ *(constants), then*

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } f(n) = O(n^{\log_b a - \varepsilon}), \text{ for some } \varepsilon > 0 \\ \Theta(n^{\log_b a} \log n), & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b a + \varepsilon}), \text{ for some } \varepsilon > 0 \\ & \text{and if } af(n/b) \leq cf(n) \text{ for some } 0 < c < 1 \end{cases}$$

- Same proof

More recurrences

- Imbalanced trees:
  - $T(n) = T(n/3) + T(2n/3) + c \cdot n$ $\qquad T(n) = \Theta(n \log n)$

More recurrences

- Imbalanced trees:
  - $T(n) = T(n/3) + T(2n/3) + c \cdot n$ $\quad T(n) = \Theta(n \log n)$
  - $T(n) = T(n/2) + 1$ $\quad T(n) = O(\log n)$

More recurrences

- Imbalanced trees:
  - $T(n) = T(n/3) + T(2n/3) + c \cdot n$ $\quad T(n) = \Theta(n \log n)$
  - $T(n) = T(n/2) + 1$ $\quad T(n) = O(\log n)$
  - $T(n) = T(n/2) + n$ $\quad T(n) = O(n)$

More recurrences

- Imbalanced trees:
  - $T(n) = T(n/3) + T(2n/3) + c \cdot n$ $\quad T(n) = \Theta(n \log n)$
  - $T(n) = T(n/2) + 1$ $\quad T(n) = O(\log n)$
  - $T(n) = T(n/2) + n$ $\quad T(n) = O(n)$
  - $T(n) = T(\sqrt{n}) + 1$ $\quad T(n) = O(\log \log n)$

    at level i, subproblem of size $n^{2^{-i}}$

More recurrences

- Imbalanced trees:
  - $T(n) = T(n/3) + T(2n/3) + c \cdot n$ — $T(n) = \Theta(n \log n)$
  - $T(n) = T(n/2) + 1$ — $T(n) = O(\log n)$
  - $T(n) = T(n/2) + n$ — $T(n) = O(n)$
  - $T(n) = T(\sqrt{n}) + 1$ — $T(n) = O(\log \log n)$

    at level i, subproblem of size $n^{2^{-i}}$
- Exponential time recurrences:
  - $T(n) = n \cdot T(n-1) + 1$ — $T(n) = O(n!)$
  - Fibonacci: $T(n) = T(n-1) + T(n-2)$ — $T(n) = O(\phi^n)$

$$\phi = \frac{1 + \sqrt{5}}{2}$$

- Divide-and-Conquer Paradigm
- Solving Recurrences
- Optional: Maximum Subarray Sum
- Acknowledgements

Maximum Subarray Sum

- **Input:** array $A = (a_1, \ldots, a_n)$ where each a_i is an integer
- **Output:** indices $1 \leq i \leq j \leq n$ and s such that

$$s = a_i + \cdots + a_j, \quad \text{and} \quad s = \max_{\alpha \leq \beta} \sum_{k=\alpha}^{\beta} a_k$$

Maximum Subarray Sum

- **Input:** array $A = (a_1, \ldots, a_n)$ where each a_i is an integer
- **Output:** indices $1 \leq i \leq j \leq n$ and s such that

$$s = a_i + \cdots + a_j, \quad \text{and} \quad s = \max_{\alpha \leq \beta} \sum_{k=\alpha}^{\beta} a_k$$

- Divide and conquer approach:
  1. divide array in the middle
  2. largest sum either on left subarray, right subarray, or crossing the middle
  3. recurse on left subarray and on the right subarray
  4. compute max sum that crosses the middle
  5. output the max of items 3 and 4

Maximum Subarray Sum

- **Input:** array $A = (a_1, \ldots, a_n)$ where each a_i is an integer
- **Output:** indices $1 \leq i \leq j \leq n$ and s such that

$$s = a_i + \cdots + a_j, \quad \text{and} \quad s = \max_{\alpha \leq \beta} \sum_{k=\alpha}^{\beta} a_k$$

- Divide and conquer approach:
  1. divide array in the middle
  2. largest sum either on left subarray, right subarray, or crossing the middle
  3. recurse on left subarray and on the right subarray
  4. compute max sum that crosses the middle
  5. output the max of items 3 and 4
- for more details, see [CLRS 2009, Chapter 4.1]

Acknowledgement

- Based on Prof Lau's lecture

  `https://cs.uwaterloo.ca/~lapchi/cs341/notes/L02.pdf`

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)

Introduction to Algorithms, third edition.

MIT Press