ALGORITHMS – COMTEK3,CCT3 & ESD3
Divide and Conquer Algorithms

Ramoni Adeogun
Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

AALBORG UNIVERSITET

Outline

- Divide and conquer algorithms
  - Merge-sort
  - Binary search
  - Algorithm to compute power of a number
  - Matrix multiplication
  - Algorithm for maximum sub-array problem

- Correctness of recursive algorithms
  - Strong mathematical induction
  - Correctness of recursive binary search

- Summary and key takeaways

The Divide-and-Conquer Paradigm

1. **Divide** the problem into one or more disjoint subproblems.

2. **Conquer** the subproblems by solving them recursively.

3. **Combine** the solutions to the subproblems

Merge Sort

1. Divide: Trivial
2. Conquer: Recursively sort 2 subarry
3. Combine: Linear-time merge

$$T(n) = \underbrace{2}_{\#\ subproblems}\, T(\underbrace{n/2}_{subproblem\ size}) + \underbrace{\Theta(n)}_{divide\ and\ combine\ complexity}$$

- Using Master theorem (case 2): $T(n) = \Theta(n \log_2(n))$

Binary Search

- Find an element in a sorted array:
  1. Divide: Check middle element.
  2. Conquer: Recursively search 1 subarray.
  3. Combine: Trivial.
- Example: Find 8

3 5 6 7 8 9 12

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
  if low > high
      return NIL
  mid = ⌊(low + high)/2⌋
  if v == A[mid]
      return mid
  elseif v > A[mid]
      return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
  else return RECURSIVE-BINARY-SEARCH(A, v, low, mid − 1)
```

Recurrence for Binary Search

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

 # subproblems

subproblem size

divide and combine complexity

- Solution using master method:

$$n^{\log_b(a)} = n^{\log_2(1)} = n^0 = 1 \rightarrow Case\ 2\ (k = 0)$$

$$T(n) = \Theta(\log_2(n))\,.$$

Computer power of a number

- Problem: Compute a^n, where $n \in \mathrm{N}$
- Naïve algorithm: $T(n) = \Theta(n)$
- Divide and conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \times a^{n/2}, & \text{if } n \text{ } is \text{ even} \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & \text{if } n \text{ } is \text{ odd} \end{cases}$$

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

Fibonacci numbers

- **Recursive definition**:

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

- The sequence: 0 1 1 2 3 5 ....

- **Naive recursive algorithm**:

$$T(n) = \Omega(\phi^n) \quad \text{(exponential time)}$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Algorithm to compute Fibonacci numbers

- Bottom-up:
  - Compute $F_0, F_1, \dots, F_n$ in order, obtaining each number by summing the two preceding ones
  - Running time: $\Theta(n)$
- Naïve recursive squaring:
  - $F_n = \phi^n / \sqrt{5}$ rounded to the nearest integer
  - Recursive squaring: $\Theta(\log_2(n))$
  - Floating point arithmetic is prone to round-off errors, making this method unreliable.

Recursive squaring

- Theorem:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

- Algorithm: Recursive squaring:

$$T(n) = \Theta(\log_2(n))$$

Base case (n=1):

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$$

Recursive squaring

- Inductive step ($n \geq 2$):

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n}$$

Matrix multiplication

- **Precondition**: $A = [a_{ij}], B = [b_{ij}],\ i, j = 1, 2, \dots, n.$
- **Postcondition**: $C = [c_{ij}] = A \times B, \quad i, j = 1, 2, \dots, n.$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

- **Standard algorithm**:

for $i = 1$ ***to*** n
 for $j = 1$ ***to*** n
  $c_{ij} \leftarrow 0$
  for $k = 1$ ***to*** n
   $c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}$

$$\boldsymbol{T(n) = \Theta(n^3)}$$

Matrix multiplication – Divide and conquer algorithm

- Idea:
  - $n \times n$ matrix = 2 × 2 matrix of $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
$$C = A \times B$$

$$\left.\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{22} + A_{22}B_{22} \end{aligned}\right\}$$

  - 8 recursive multiplications of $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices
  - 4 additions of $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices

Matrix multiplication – Divide and conquer algorithm

```
REC-MAT-MULT(A, B, n)
  let C be a new n × n matrix
  if n == 1
      c11 = a11 · b11
  else partition A, B, and C into n/2 × n/2 submatrices
      C11 = REC-MAT-MULT(A11, B11, n/2) + REC-MAT-MULT(A12, B21, n/2)
      C12 = REC-MAT-MULT(A11, B12, n/2) + REC-MAT-MULT(A12, B22, n/2)
      C21 = REC-MAT-MULT(A21, B11, n/2) + REC-MAT-MULT(A22, B21, n/2)
      C22 = REC-MAT-MULT(A21, B12, n/2) + REC-MAT-MULT(A22, B22, n/2)
  return C
```

Matrix multiplication – Divide and conquer algorithm

- Recurrence for the divide and conquer algorithm for multiplication:

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

 # subproblems

subproblem size

divide and combine complexity

$$n^{\log_b(a)} = n^{\log_2(8)} = n^3 \rightarrow Case\ 1$$

$$T(n) = \Theta(n^3).$$

- Divide and conquer is no better than the standard algorithm

Strassen's Algorithm

- **Idea**:
  - Perform only 7 recursive multiplications of $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrices, rather than 8.
    - Will cost several additions of $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrices, but just a constant number more) can still absorb the constant factor for matrix additions into the $\Theta\left(\frac{n}{2}\right)$ term.

Strassen's Algorithm

- The algorithm:
  - Step 1: As in the recursive method, partition each of the matrices into four $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices. Time: $\Theta(1)$
  - Step 2: Create 10 matrices $S_1, S_2, \dots, S_{10}$. Each is $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ and is the sum or difference of two matrices created in step 1. Time: $\Theta(n^2)$ to create all 10 matrices.
  - Step 3: Recursively compute 7 $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ matrix products $P_1, P_2, \dots, P_7$.
  - Step 4: Compute $\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right)$ submatrices of C by adding and subtracting various combinations of the P_i. Time: $\Theta(n^2)$

Strassens Algorithm

- **Step 2:**

$$S_1 = B_{12} - B_{22} \qquad S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22} \qquad S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22} \qquad S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22} \qquad S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21} \qquad S_{10} = B_{11} + B_{12}$$

Add or subtract $\frac{n}{2} \times \frac{n}{2}$ matrices 10 times → time is $\Theta(n^2)$

- **Step 3**: Create the 7 matrices

$$P_1 = A_{11} \times S_1 = A_{11} \times B_{12} - A_{11} \times B_{22} \,;$$
$$P_2 = S_2 \times B_{22} = A_{11} \times B_{22} + A_{12} \times B_{22}$$
$$P_3 = S_3 \times B_{11} = A_{21} \times B_{11} + A_{22} \times B_{11} \,;$$
$$P_4 = A_{22} \times S_4 = A_{22} \times B_{21} - A_{22} \times B_{11}$$
$$P_5 = S_5 \times S_6 = A_{11} \times B_{11} + A_{11} \times B_{22} + A_{22} \times B_{11} + A_{22} \times B_{22}$$

Strassen’s Algorithm

- **Step 3 (cont’d)**

$$P_6 = S_7 \times S_8 = A_{12} \times B_{21} + A_{12} \times B_{22} - A_{22} \times B_{21} - A_{22} \times B_{22}$$
$$P_7 = S_9 \times S_{10} = A_{11} \times B_{11} + A_{11} \times B_{12} - A_{21} \times B_{11} - A_{21} \times B_{12}$$

- **Step 4:**

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Exercise: Expand the four terms in step 4

Analysis of Strassen's Algorithm

- Recurrence

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2), & if\ n > 1 \end{cases}$$

- Solution (by master method):

$$T(n) = \Theta\left(n^{\log_2(7)}\right)$$

How does this compare with the complexity of the divide-and-conquer algorithm?

Maximum-subarray problem

- **Precondition**: An array $A[1, 2, \ldots, n]$ of numbers.
  - [Assume that some of the numbers are negative, because this problem is trivial when all numbers are nonnegative.]

- **Postcondition**: Indices i and j $(1 \leq i \leq j \leq n)$ such that the sum

$$\sum_{k=i}^{j} A[k]$$

is the largest sum of any nonempty, contiguous subarray of A.

Maximum-subarray problem

- **Example scenario**
  - You have the prices that a stock traded at over a period of n consecutive days.
  - When should you have bought the stock? When should you have sold the stock?
  - Even though it's in retrospect, you can yell at your stockbroker for not recommending these buy and sell dates.

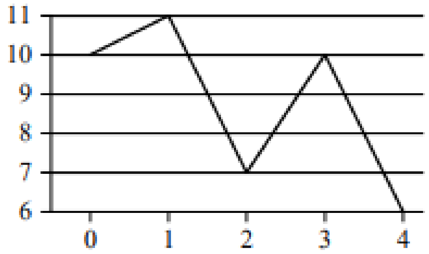
- To convert to a MSP:

$$A[i] = (price\ after\ day\ i) - \big(price\ after\ day\ (i-1)\big).$$

  - Assuming that we start with a price after day 0, i.e., just before day 1.
    - Then the nonempty, contiguous subarray with the greatest sum brackets the days that you should have held the stock
  - If the maximum subarray is $A[i, \dots, j]$ then you should have bought just before day i (i.e., just after day $(i-1)$) and sold just after day j

Maximum-subarray problem – Stock-trading example

- Why do we need to find the maximum subarray? Why not just “buy low, sell high”?
  - Lowest price might occur after the highest price.
  - But wouldn’t the optimal strategy involve buying at the lowest price or selling at the highest price?
    - Not necessarily:



Maximum profit is \$3 per share, from buying after day 2 and selling after day 3. Yet lowest price occurs after day 4 and highest occurs after day 1

Maximum-subarray problem – Divide and Conquer Algorithm

- Use divide-and-conquer to solve in $O(n\log_2(n))$ time.
- **Subproblem**: Find a maximum subarray of $A[low, \ldots, high]$
  - In the initial call, $low = 1, high = n$
- The algorithm:
  - **Divide** the subarray into two subarrays of as equal size as possible. Find the midpoint mid of the subarrays, and consider the subarrays $A[low, \ldots, mid]$ and $A[mid + 1, \ldots, high]$
  - **Conquer** by finding a maximum subarrays of $A[low, \ldots, mid]$ and $A[mid + 1, \ldots, high]$
  - **Combine** by finding a maximum subarray that crosses the midpoint and using the best solution out of the three (the subarray crossing the midpoint and the two solutions found in the conquer step).

Note: Any subarray must either lie entirely on one side of the midpoint or cross the midpoint

Maximum-subarray problem – Divide and Conquer Algorithm

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
  if high == low
      return (low, high, A[low])            // base case: only one element
  else mid = ⌊(low + high)/2⌋
      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
      if left-sum ≥ right-sum and left-sum ≥ cross-sum
          return (left-low, left-high, left-sum)
      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
          return (right-low, right-high, right-sum)
      else return (cross-low, cross-high, cross-sum)
```

Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i . . mid].
  left-sum = −∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 . . j].
  right-sum = −∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

Time: $T(n) = \Theta(n)$

Correctness of Recursive Algorithms

Strong mathematical induction

Problem:

For all integers $n \geq n_0$, prove that property $P(n)$ is true.

Proof by Strong Mathematical Induction:

- **Base Case**: Show that $P(n_0)$ is true. This is usually easy, but it is essential for a correct argument.

- **Inductive Step**: Show that if $P(k)$ is True for all integers $n_0 \leq k \leq n$ then $P(n + 1)$ is true. To do this:
  - choose an arbitrary $n \geq n_0$,
  - show that $P(n+1)$ is true if $P(n), P(n-1), \cdots, P(n_0)$ are true.

- We will apply this method to prove the correctness of recursive algorithms.

Correctness of Recursive Algorithms

1. **Stating correctness:** It is important to state what correctness means to the algorithm carefully
2. **Base case:**
    1. Use the algorithm description to say what gets returned in the base case
    2. Show that this value satisfies the correctness property
3. **Strong Induction step:**
    1. **State the induction hypothesis**: The algorithm is correct on all inputs between the base case and one less than the current case.
    2. Represent the output on the current value in terms of recursive calls
    3. Apply the strong induction hypothesis to replace each recursive call with the correct answer.
    4. Then we need to use algebra and logic to show that this is the same as a correct answer for the current input
    5. Summary of induction argument: Just a reminder that we are finished.

Correctness Proof for Recursive Binary Search

- The algorithm **RECURSIVE_BINARY_SEARCH** returns an index i with $\text{low} \leq i \leq high$ and $A[i] == v$ if such an index exists. Otherwise, it returns NIL.
  - We prove the correctness of the algorithm **RECURSIVE_BINARY_SEARCH** by induction on $n = high - low + 1$.

- **Base case**: If $n = 1$ then $low == high$. Thus, the array contains one element, $A[low]$ and the algorithm returns low if $A[low] == v$ (and NIL otherwise). The algorithm works correctly for $n = 1$.

- **Induction hypothesis**: For inputs with $k = high - low + 1$ the algorithm correctly returns an index i with $\text{low} \leq i \leq high$ and $A[i] == v$ if such an index exists (Otherwise it returns NIL).

Correctness Proof for Recursive Binary Search

- **Induction step:** Assume the induction hypothesis holds for all $1 \leq k < n$.
  - Suppose we have an input $(A, v, low, high)$ with $high - low + 1 = n > 1$ and as in the algorithm define $\mathrm{mid} = \lfloor (low + high)/2 \rfloor$. There are two cases:
    1. $A[mid] < v$: Since the array is sorted, we know that v cannot be in the subarray $A[low, \ldots, mid]$. Thus, if it is in the array $A[low, \ldots, high]$, it must be in the *array* $A[mid + 1, \ldots, high]$. This is exactly what the recursive call in this case checks, if it works correctly. Since $mid \geq low$ holds, we have $high - (mid + 1) + 1 < high - low + 1 = n$, so by the induction hypothesis, the recursive call is indeed correct.
    2. $A[mid] \geq v$: The case is analogous to the first case. If v is in $A[low, \ldots, high]$, it must be in the array $A[low, \ldots, mid]$. This is checked in the recursive call, which works correctly by induction hypothesis since $mid - low + 1 < high - low + 1 = n$
- Thus, in both cases the induction hypothesis again holds. We can conclude that the algorithm is correct.

Summary and Key Take-Aways

- Divide and conquer paradigm is useful for solving many practical algorithmic problems
  - Merge-sort
  - Binary research
  - Matrix multiplication, etc
- We use recurrence relation to represent the time (and space) complexity of divide and conquer algorithms
- Typical implementation of divide and conquer algorithms is recursive.
- To prove the correctness of recursive algorithms, we use strong mathematical induction
  - State correctness
  - Prove the base case
  - State the inductive hypothesis
  - Prove the induction