Algorithms and Datastructures

Lecture 4

Manfred Jaeger

AALBORG UNIVERSITET

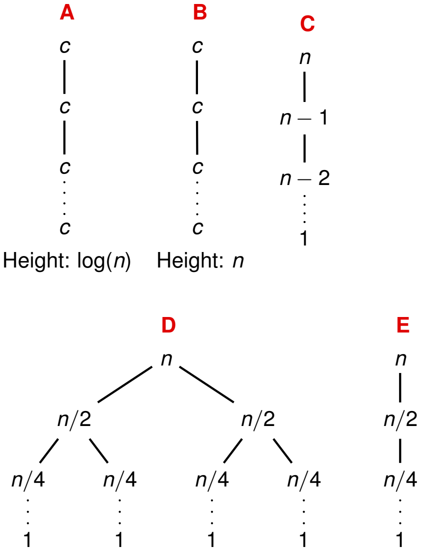
The Master Theorem

input : A sorted array A of integers, indices $l, r : 1 \leq l \leq r \leq A.length$, an integer i
output: An index j with $l \leq j \leq r$ and $A[j] = i$, if such an index exists, *null* otherwise

```
BinarySearch(A, l, r, i)
// Base case
if l == r then
    if A[l] = i then
        return l
    else
        return null
// Recursion
m = ⌊(l + r)/2⌋
if A[m] ≥ i then
    return BinarySearch(A,l,m,i)
else
    return BinarySearch(A,m+1,r,i)
```

What is the recursion tree of binary sort?

```
BinarySearch(A, l, r, i)
// Base case
if l == r then
    if A[l] = i then
        return l
    else
        return null
// Recursion
m = ⌊(l + r)/2⌋
if A[m] ≥ i then
    return BinarySearch(A,l,m,i)
else
    return BinarySearch(A,m+1,r,i)
```



Recursion Tree

$$\begin{array}{c} c \\ | \\ c \\ | \\ c \\ \vdots \\ c \end{array}$$

Depth of tree: $\lg n$

"Guess": $T(n) = O(\lg n)$

Recurrence Equations

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Problem: multiply two $n \times n$ matrices A and B (naive algorithm takes $\Theta(n^3)$ time).

Strassen's approach

Divide: partition A and B into $(n/2) \times (n/2)$ sub-matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

and construct 10 new matrices as sums and differences of the sub-matrices:

$$S_1 = B_{12} - B_{22}, S_2 = A_{11} + A_{12}, \ldots, S_{10} = B_{11} + B_{12}$$

Time: $\Theta(n^2)$

Conquer: compute 7 products of matrices of size $n/2$:

$$P_1 = A_{11} \cdot S_1, \ldots, P_7 = S_9 \cdot S_{10}.$$

Combine: get solution

$$C = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{pmatrix}$$

Time: $\Theta(n^2)$

Strassen's Algorithm

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Binary Search

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Let $a \geq 1, b > 1$ and $u = \log_b a$. Let T be given by the recurrence

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

1. If $f(n) = O(n^{u-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^u)$.
2. If $f(n) = \Theta(n^u)$, then $T(n) = \Theta(n^u \lg n)$.
3. If
   - $f(n) = \Omega(n^{u+\epsilon})$ for some $\epsilon > 0$, and
   - $af(n/b) < cf(n)$ for some c and all sufficiently large n,

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

☞ The same theorem holds when we write the recurrence as

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

Merge Sort

$$T(n) = 2T(n/2) + \Theta(n)$$

- $a = b = 2 \Rightarrow u = 1,$
- $\Rightarrow f(n) = \Theta(n^u)$

$\overset{case2}{\Rightarrow} \quad T(n) = O(n \lg n).$

Merge Sort

$$T(n) = 2T(n/2) + \Theta(n)$$

- $a = b = 2 \Rightarrow u = 1,$
- $\Rightarrow f(n) = \Theta(n^u)$

$\overset{case2}{\Rightarrow} \quad T(n) = O(n \lg n).$

Binary Search

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

- $a = 1, b = 2 \Rightarrow u = 0,$
- $\Rightarrow f(n) = \Theta(n^u)$

$\overset{case2}{\Rightarrow} \quad T(n) = O(\lg n).$

Merge Sort

$$T(n) = 2T(n/2) + \Theta(n)$$

- $a = b = 2 \Rightarrow u = 1,$
- $\Rightarrow f(n) = \Theta(n^u)$

$\overset{case2}{\Rightarrow}$ $T(n) = O(n \lg n).$

Binary Search

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

- $a = 1, b = 2 \Rightarrow u = 0,$
- $\Rightarrow f(n) = \Theta(n^u)$

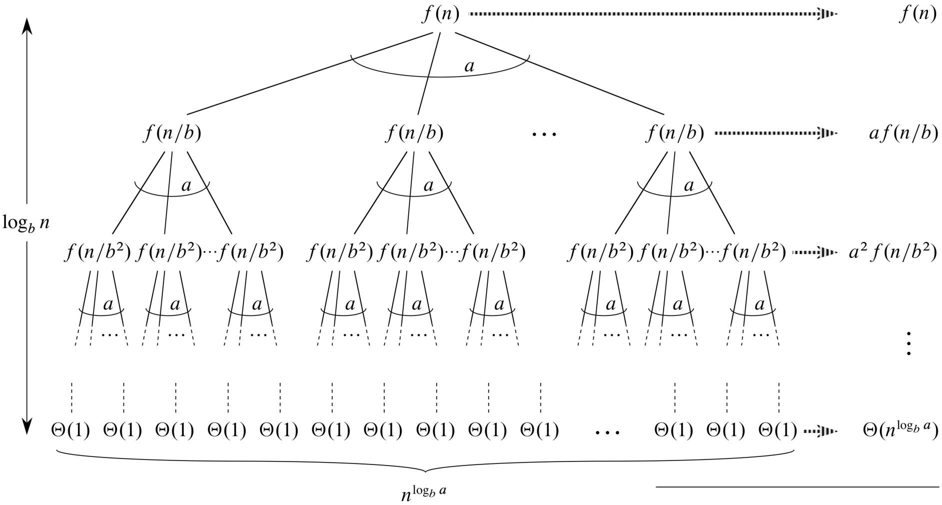
$\overset{case2}{\Rightarrow}$ $T(n) = O(\lg n).$

Strassen's Algorithm

$$T(n) = 7T(n/2) + \Theta(n^2)$$

- $a = 7, b = 2 \Rightarrow u = \log_2 7 \approx 2.8,$
- $\Rightarrow f(n) = O(n^{u-0.5})$

$\overset{case1}{\Rightarrow}$ $T(n) = \Theta(n^u).$



$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

- Height of the recursion tree: $\log_b n$
- Number of leaves: $a^{\log_b n} = n^{\log_b a}$
- Total cost of computation:

$$\Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

☞ The three cases of the Theorem compare the computation cost of the root of the recursion tree with the total cost of the leaves:

Case 1: The "root cost" is small compared to the "leaves cost"

- also the total contribution of intermediate levels does not exceed leaves cost:

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) = O(n^{\log_b a})$$

- Height of the recursion tree: $\log_b n$
- Number of leaves: $a^{\log_b n} = n^{\log_b a}$
- Total cost of computation:

$$\Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

☞The three cases of the Theorem compare the computation cost of the root of the recursion tree with the total cost of the leaves:

Case 2: The "root cost" is about the same as the "leaves cost"

- also all intermediate levels contribute the same amount of computation cost
- $\Rightarrow$ total cost is $\Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n)$

- Height of the recursion tree: $\log_b n$
- Number of leaves: $a^{\log_b n} = n^{\log_b a}$
- Total cost of computation:

$$\Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

☞The three cases of the Theorem compare the computation cost of the root of the recursion tree with the total cost of the leaves:

Case 3: The "root cost" is large compared to the "leaves cost", and the decrease from $f(n)$ to $f(1)$ happens sufficiently fast

- $\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$ is dominated by the root cost ($j = 0$), and is $\Theta(f(n))$.

Consider

$$T(n) = 2T(n/2) + \Theta(n \lg n)$$

Then: $u = \log_b a = 1$, and

- $f(n)$ is not $O(n)$, so cases 1 and 2 do not apply
- For any $\epsilon > 0$:

  $$(n^{1+\epsilon})/(n \lg n) = n^{\epsilon}/\lg n \to \infty,$$

  and therefore: $n \lg n \neq \Omega(n^{1+\epsilon})$.

Proving Correctness

The Problem

For

- a computational problem defined by the set of *possible instances* $\mathcal{I}$ and the *input-output* function $\mathcal{F}$.
- an algorithm accepting inputs from $\mathcal{I}$.

want to prove that for any given input $i \in \mathcal{I}$ the algorithm terminates and returns $\mathcal{F}(i)$.

The Difficulty

- There is no single fixed procedure one can follow to prove correctness (☞ Computability and Complexity course).
- Can require some creativity!
- There are certain standard *reasoning techniques* which can be used to piece together a correctness proof!

Proving correctness is the formal mathematical counterpart of writing bug-free programs!

	Algorithms	Programming
Showing: no errors exist	*Correctness Proof*	*Verification*
Finding errors	*Trying examples*	*Testing, Debugging*

- *Testing* can reveal incorrectness, but never prove correctness (unless input space is finite)
- Correctness proofs and debugging can use similar analysis techniques

☞ Courses *Semantics and Verification (DAT6/SW6)*, *Test and Verification (SW8)*.

General strategy:

- think of *states* of the algorithm defined by
  - the next line being executed
  - the contents of the variables/datastructures manipulated by the algorithm
- show that the steps of the algorithm transform the *initial state* of the program for a given input $i \in \mathcal{I}$, such that
  - the sequence of transformations terminates
  - at the end the return value contains $\mathcal{F}(i)$.

A *state condition* is a precise definitions of a condition a state must satisfy.

Examples:

- The first k elements of the array I are in ascending order
- The algorithm is at line 17, the first k elements of the array I are in ascending order, and the content of the variable *temp* is a non-negative integer

Pre- and Post-conditions

The operations of a certain block of instructions can be characterized by:

- Pre-condition: what is true before the block is executed
- Post-condition: what is true after the block is executed

```
MaxSubArrWEndPoint (I, k)
1 beststart=k
2 bestsum = I[k]
3 currentsum = I[k]
4 for j=k-1 .. 1 do
5     currentsum = currentsum + I[j]
6     if currentsum > bestsum then
7         bestsum = currentsum
8         beststart = j
9 return beststart, bestsum
```

Precondition: before executing line 5, *currentsum* contains $\sum_{i=j+1}^{k} I[i]$; *bestsum* contains the maximum of all sums $\sum_{i=h}^{k} I[i]$ $(h \geq j+1)$.

Postcondition: after executing line 8, *currentsum* contains $\sum_{i=j}^{k} I[i]$; *bestsum* contains the maximum of all sums $\sum_{i=h}^{k} I[i]$ $(h \geq j)$.

```
MaxSubArrWEndPoint (I, k)
1  beststart=k
2  bestsum = I[k]
3  currentsum = I[k]
4  for j=k-1 .. 1 do
5      currentsum = currentsum + I[j]
6      if currentsum > bestsum then
7          bestsum = currentsum
8          beststart = j
9  return beststart, bestsum
```

Precondition: before executing line 5, *currentsum* contains $\sum_{i=j+1}^{k} I[i]$; *bestsum* contains the maximum of all sums $\sum_{i=h}^{k} I[i]$ $(h \geq j+1)$.

Postcondition: after executing line 8, *currentsum* contains $\sum_{i=j}^{k} I[i]$; *bestsum* contains the maximum of all sums $\sum_{i=h}^{k} I[i]$ $(h \geq j)$.

Here: Pre/Post Condition is a two-part specification of a *loop invariant*

```
InsertionSort(I)
1 for j = 2..n do
2     key=I[j]
3     i = j − 1
4     while i > 0 and I[i] > key do
5         I[i + 1] = I[i]
6         i = i − 1
7     I[i + 1] = key
```

Precondition: before executing line 4, the contents of $I[1..j-1]$ are in ascending order.

Postcondition: after executing line 7, $I[1..j]$ contains in ascending order the previous content of $I[1..j-1]$ and *key*.

Not every loop construct needs to be analyzed with loop-invariants (can be overkill).

Loop invariant: state condition involving loop counter variable.

...
$\boxed{\text{/* Precondition loop} \sim \text{Loop Invariant 0 */}}$
for $j = 0..n$ **do**
  $\boxed{\text{/* Precondition iteration } j \sim \text{Loop Invariant } j \text{ */}}$
  ...
  Do something involving j
  ...
  $\boxed{\text{/* Postcondition iteration } j \sim \text{Loop Invariant } j+1 \text{ */}}$

$\boxed{\text{/* Postcondition loop} \sim \text{Loop Invariant } n+1 \text{ */}}$
...

Initialization: Loop invariant 0 holds before loop is started

Maintenance: At iteration j: if loop invariant j is true at the beginning of the iteration, then loop invariant $j+1$ is true at the end of the iteration.

Termination: Translate loop invariant $n+1$ into a suitable postcondition for the complete **for** loop

```
INSERTIONSORT(I)
1 for j = 2..n do
2     key=I[j]
3     i = j − 1
4     while i > 0 and I[i] > key do
5         I[i + 1] = I[i]
6         i = i − 1
7     I[i + 1] = key
```

Loop Invariant j: $I[1..j-1]$ is sorted, and contains the first $j-1$ elements of the original input array.

Initialization ($j = 2$): Before the **for** loop is started, $I[1]$ is sorted and contains the original first element.

Maintenance: If invariant j holds at the beginning of iteration j, then invariant $j+1$ holds at the end. For this use Pre-/Post-condition for lines 4-7.

Termination: Invariant $n+1$ just says that I is now sorted.

Loop invariant: state condition involving a *Progress* indicator P, such that **while** loop terminates when $P = 0$. (P need not be a variable explicitly defined in the algorithm and used in the **while** termination condition).

...
/* Precondition loop $\sim$ Loop Invariant */
while *Boolean condition* **do**
    /* Precondition iteration $\sim$ Loop Invariant, $P = k$ */
    ...
    Do something
    ...
    /* Postcondition iteration $\sim$ Loop Invariant, $P < k$ */

/* Postcondition loop $\sim$ Loop Invariant, $P = 0$ */
...

Initialization: Loop invariant holds before loop is started

Maintenance: If loop invariant is true at the beginning of an iteration, then it is true at the end of the iteration, and the value of P at the end is smaller than the value of P at the beginning.

Termination: The loop terminates exactly when $P = 0$. The loop invariant with $P = 0$ translates into a suitable postcondition for the complete **while** loop.

BubbleSort(I)

```
1 repeat
2     continue = false
3     for i=1 .. I.length-1 do
4         if I[i] > I[i + 1] then
5             swap I[i] and
              I[i + 1]
6             continue = true
7 until continue = false
```

Loop Invariant and Progress Measure: I contains the same elements as the original input array. P: *Transposition Count TC*, i.e., the number of pairs of elements that are in a wrong relative position.

Initialization: Nothing to do.

Maintenance: The contents of I are not changed. TC is reduced by at least 1 in one execution of the **for** loop of lines 3-6.

Termination: $TC = 0$ just says that I is now sorted.

BUBBLESORT(I)

```
1 repeat
2     continue = false
3     for i=1 .. I.length-1 do
4         if I[i] > I[i+1] then
5             swap I[i] and
              I[i+1]
6             continue = true
7 until continue = false
```

Loop Invariant and Progress Measure: I contains the same elements as the original input array. P: the maximal number r, such that $I[n - r + 1..n]$ contains the r largest elements of I in correct order.

Initialization: Nothing to do.

Maintenance: The contents of I are not changed. P is increased by at least 1 in one execution of the **for** loop of lines 3-6: the largest element of $I[1..n - r]$ is brought into position $I[n - r]$.

Termination: $r = n$ just says that I is now sorted.

RECALGO(Input I)

/* Precondition for algorithm */

if *I is a base case* **then**

...

else

...

/* Precondition for recursive call */

RECALGO(I')

/* Postcondition for recursive call */

...

...

/* Postcondition for algorithm */

Pre-/Post-conditions usually directly express correctness of the algorithm.

Proof by induction:

Base case: If the precondition holds and I is a base case, then the postcondition holds at the end of the algorithm.

Induction step:

- The I' in the recursive call is a smaller problem instance than I
- Assuming that the precondition holds for the algorithm, and the recursive call satisfies its pre- and postcondition, then the postcondition holds for the algorithm.

MAXSUBARRDC (*I*)

```
// Base case
if I.length == 1 then
    return 1,1,I[1]
// Divide
m = ⌊I.length/2⌋
LeftI=I[1..m]
RightI=I[m+1 .. I.length]
// Conquer
leftsol=MAXSUBARRDC(LeftI)
rightsol=MAXSUBARRDC(RightI)
// Combine
crosssol=concat(MAXSUBARRWENDPOINT (I, m), MAXSUBARRWSTARTPOINT (I, m+1) )
return best of leftsol, rightsol, crosssol
```

Precondition: *I* is an integer array of length ≥ 1.

Postcondition: return value is the maximum subarray of *I*.

Base Case: Postcondition is satisfied when *I.length* $= 1$.

Induction:

- *LeftI* and *RightI* are strictly smaller than *I*
- Show: preconditions are satisfied for the calls MAXSUBARRDC(*LeftI*), MAXSUBARRDC(*RightI*). Assuming postcondition is true for these calls, show that postcondition is satisfied for the algorithm.

☞The induction step requires separate correctness proofs for the procedures MAXSUBARRWENDPOINT and MAXSUBARRWSTARTPOINT

☞ Correctness proofs are made easier when (complex) algorithms are broken down into smaller "modules" which can be independently analyzed in terms of their pre- and post-conditions.

☞ Same principle helps writing correct programs using object-oriented programming.