Algorithmic paradigms: Divide-and-Conquer

Gianmaria Silvello

Department of Information Engineering
University of Padua
gianmaria.silvello@unipd.it
http://www.dei.unipd.it/~silvello/

Outline

- Divide-and-Conquer paradigm
- Merge-sort and running time
- *Exercises*
    - Recursive algorithms
    - Binary search (Homework)
- Recursive tree method for solving recurrences

Recursive algorithms

- An algorithm is **recursive** if it calls itself one or more times to solve a problem.
- Recursion is useful when a task can be split into similar, but smaller subtasks.
- Let's think about the algorithm for calculating the power *n* of a natural number *x*
- *E.g.* $power(x,n) = power(2,3) = x^n = 2^3$

Algorithms for power(x,n)

```
ITERATIVE-POWER(x,n)
   result = 1
   for i = 1 to n do
      result = result * x
   return result
```

```
RECURSIVE-POWER(x,n)
   if (n==1) then
      return x
   else
      return x * RECURSIVE-POWER(x,n-1)
```

Execution stack

- When an algorithm calls itself (nested call)

1. the current execution is paused
2. the execution context associated with the current algorithm execution is stored in a stack (LIFO structure)
3. the nested call executes
4. After it ends, the previous call is retrieved from the stack and the execution is resumed from where it stopped

Execution stack: Example

```
RECURSIVE-POWER(x,n)
    if (n==1) then
        return x
     else
        return x * RECURSIVE-POWER(x,n-1)
```

`RECURSIVE-POWER(2,3)`

`Execution 1, n≠1` thus we call `RECURSIVE-POWER(2,2)`

STACK
0 context {x:2, n:3}

`Execution 2, n≠1` thus we call `RECURSIVE-POWER(2,1)`

STACK
REMOVE 0 context {x:2, n:2}
1 context {x:2, n:3}

`Execution 3, n==1` thus we `return 2`

STACK
REMOVE 0 context {x:2, n:1}
1 context {x:2, n:2}
2 context {x:2, n:3}

Restore the previous call `RECURSIVE-POWER(2,2)`

subcall `RECURSIVE-POWER(2,1)` that already returned 2

return 2*2 = 4

REMOVE context {x:2, n:2}

Execution stack: Example

```
RECURSIVE-POWER(x,n)
    if (n==1) then
         return x
      else
         return x * RECURSIVE-POWER(x,n-1)
```

STACK

RESTORE 0 context {x:2, n:3}

Restore the previous call `RECURSIVE-POWER(2,3)`

subcall `RECURSIVE-POWER(2,2)` that already returned 4

return 2*4 = 8

`REMOVE context {x:2, n:3}`

Recursion depth in this case was 3

Recursive algorithms

- An algorithm is **recursive** if it calls itself one or more times to solve a problem.

- These algorithms typically follow a **divide-and-conquer strategy**

- **Divide** the problem into smaller sub-problems
- **Conquer** the sub-problems by solving them one at a time
- **Combine** the solutions of the subproblems into the solution for the genera problem

Merge-Sort

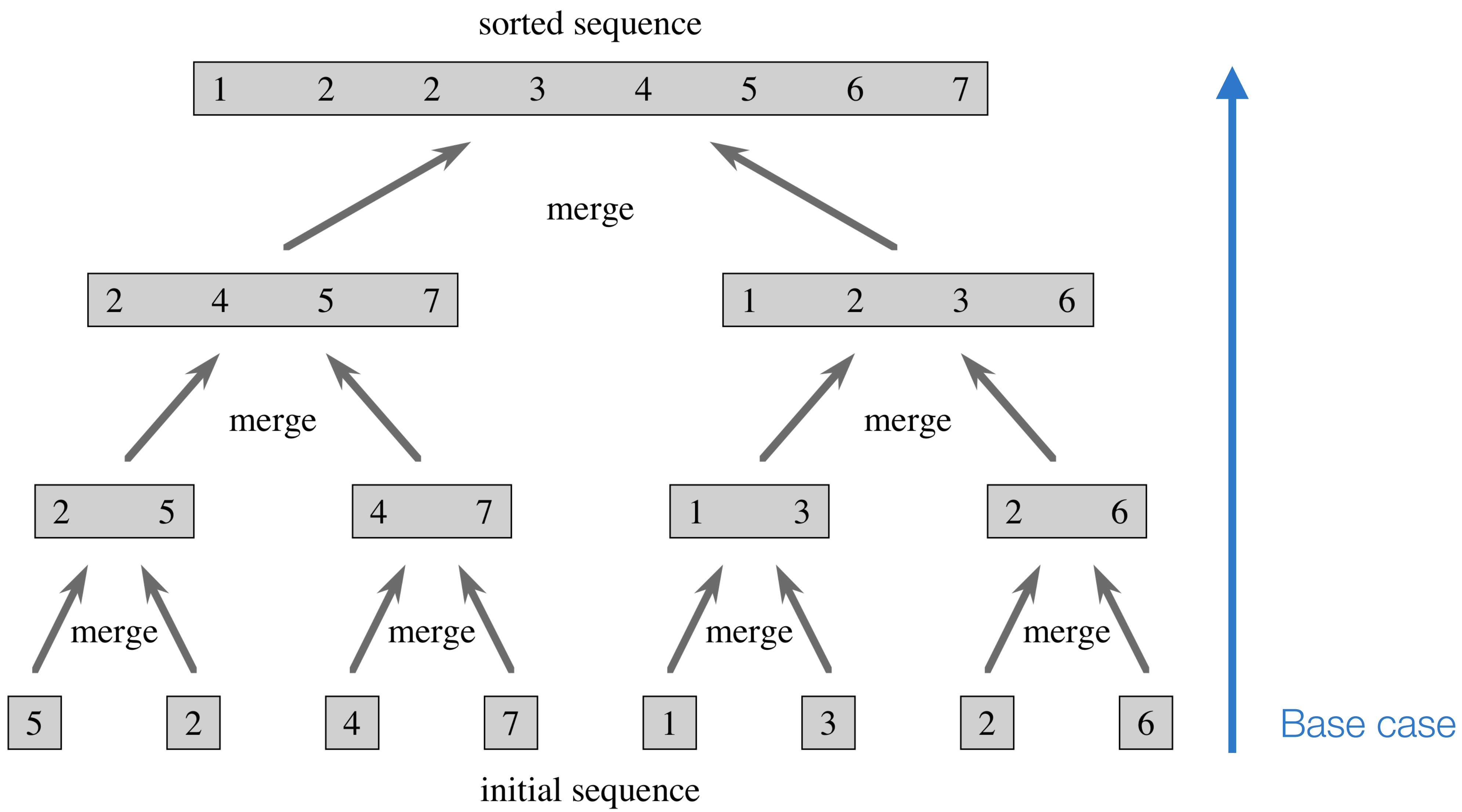
- A classical algorithm employing the divide-and-conquer paradigm is **merge-sort**

1. Assume to have an unordered sequence of *n* elements
2. Divide the sequence in two sequences with length *n/2*
3. Conquer: sort the two sequences using merge-sort recursively
4. Combine: merge the two ordered sequences into the output sequence

- The key passage of this algorithm is the combine step

Merge-sort recursive algorithm

```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```



From CLRS 3rd ed.

Merge procedure

$q = \lfloor \frac{p+r}{2} \rfloor$

Input sequence: < 5, 2, 4, 7, 1, 3, 2, 6 >

$p=1$ $r=8$

MERGE(A, p, q, r)

$n_1 = q - p + 1$ // set the length of the arrays
$n_2 = r - q$ //
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
  $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
  $R[j] = A[q + j]$

→ This is just to copy the elements of A in L and in R

$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$

These are called SENTINELS

$i = 1$
$j = 1$
for $k = p$ **to** r → iterates over the elements to be merged
  if $L[i] \le R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else $A[k] = R[j]$
    $j = j + 1$

At the end of this loop the elements in L and R are sorted A = < 2, 4, 5, 7, 1, 3, 2, 6 >

L R L R

Running time: Θ(n) = Θ(r-p+1)

For instance:

p r

< 2, 5, 4, 7, 1, 3, 2, 6 >

L = < 2, 5 >
R = < 4, 7 >

$i = 1$ $K = 1$
$j = 1$

$L[1] = 2 < R[1] = 4$

$\Rightarrow A[k] = L[i]$

A = < 2, 5, 4, 7, 1, 3, 2, 6

$i = i + 1 = 2$

From CLRS 3rd ed.

Merge procedure

$\text{MERGE}(A, p, q, r)$

$\Theta(n)$ $\Theta(n)$

```
n1 = q − p + 1
n2 = r − q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
      L[i] = A[p + i − 1]
for j = 1 to n2
      R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
      if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
      else A[k] = R[j]
            j = j + 1
```

This loop iterates over all the elements to be merged

$\Theta(n)$

Running time: Θ(n) = Θ(r-p+1)

From CLRS 3rd ed.

Merge-sort recursive algorithm

MERGE-SORT(A, p, r)

if $p < r$ // check for base case
  $q = \lfloor (p+r)/2 \rfloor$ // divide
  MERGE-SORT(A, p, q) // conquer
  MERGE-SORT$(A, q+1, r)$ // conquer
  MERGE(A, p, q, r) // combine

Analysis of the running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{otherwise} \end{cases}$$

This is the general case for any recursive algorithm

From CLRS 3rd ed.

Mergesort: Analysis of the running time

constant running time when $c = 1$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{otherwise} \end{cases}$$

For every recursive call, the input instance is divided into a subproblems $\rightarrow a = 2$

Size of the subproblems $\rightarrow b = 2$

Analysis of the Running Time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & \text{otherwise} \end{cases}$$

divide: we split the array in 2, so $\Theta(1)$

combine: we saw that the merge procedure takes $\Theta(n)$

Mergesort Running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \cancel{\Theta(1)} + \Theta(n) & \text{otherwise} \end{cases}$$

lower order term

How to solve the recursion

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = 2T\left(\frac{n}{2}\right) + \underline{cn}$$

This means that we need linear time to combine n elements

Without loss of generality we consider n = power of 2 $\Rightarrow$ each subproblem IS EXACTLY $\frac{n}{2}$

$$T(n) \rightarrow \begin{tikzcd}[column sep=small] & T(n) \arrow[dl, no head] \arrow[dr, no head] & \\ T\left(\frac{n}{2}\right) & & T\left(\frac{n}{2}\right) \end{tikzcd} \qquad \begin{tikzcd}[column sep=small] & cn \arrow[dl, no head] \arrow[dr, no head] & \\ c\left(\frac{n}{2}\right) & & c\left(\frac{n}{2}\right) \end{tikzcd}$$

How to solve the recurrence