Algorithms and Datastructures

Lecture 3

Manfred Jaeger

AALBORG UNIVERSITET

Divide and Conquer

The 3 steps of D&C

Given a problem instance of size n ...:

Divide: the problem into a number of smaller instances

Conquer: solve the subproblems recursively

Combine: return the solution of the original problem by combining the solutions of the subproblems

D&C vs. Recursion

- all D&C algorithms are recursive
- if one allows “number of smaller instances” = 1, then every recursive algorithm could be seen as D&C
- therefore: “number of smaller instances” ≥ 2

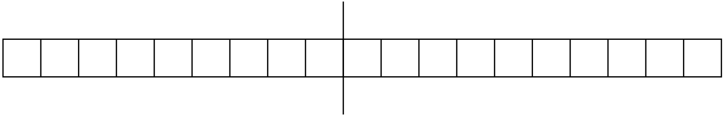
input : An array l of integers
output: An array containing the elements of l in ascending order

<u>MERGESORT(l)</u>

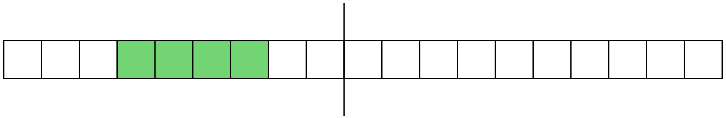
`// Base case`
if $l.length \leq 1$ **then**
    return l
`// Divide`
$Leftl = l[1..\lfloor l.length/2 \rfloor]$
$Rightl = l[\lfloor l.length/2 \rfloor + 1..l.length]$
`// Conquer and Combine`
return MERGE(MERGESORT($Leftl$),MERGESORT($Rightl$))

MERGE(Array l_1, Array l_2) returns a sorted array that contains the elements of the <u>sorted</u> arrays l_1, l_2.

Dividing the problem by splitting the array in the middle:



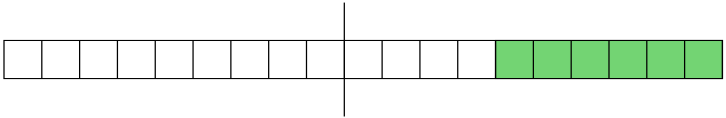
Dividing the problem by splitting the array in the middle:



The solution can either be

- entirely contained in the left sub-array

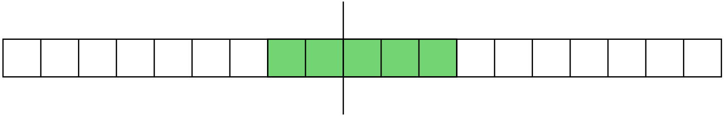
Dividing the problem by splitting the array in the middle:



The solution can either be

- entirely contained in the left sub-array
- entirely contained in the right sub-array

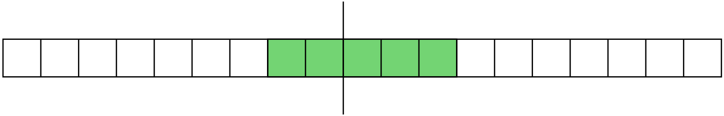
Dividing the problem by splitting the array in the middle:



The solution can either be

- entirely contained in the left sub-array
- entirely contained in the right sub-array
- crossing the mid-point

Dividing the problem by splitting the array in the middle:

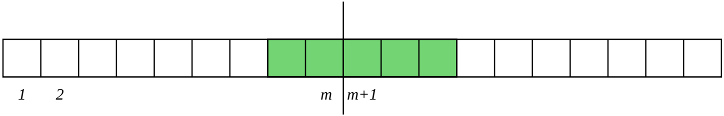


The solution can either be

- entirely contained in the left sub-array
- entirely contained in the right sub-array
- crossing the mid-point

D&C approach:

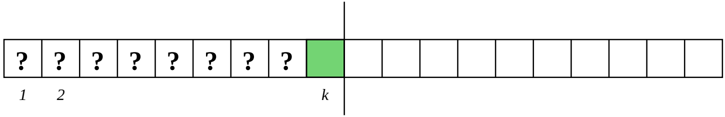
- Divide, and recursively solve left and right sub-problem
- Find best “crossing” solution
- Combine by comparing 3 candidate solutions, and returning best one



The best crossing solution is the concatenation of

- the maximum sub-array among all sub-arrays ending at m
- the maximum sub-array among all sub-arrays starting at $m+1$

☞ The problem reduces to finding the maximum sub-array for a given start- or end-point.



input : An array I of integers, an index $k \leq I.length$
output: The start index of the maximal sub-array of I ending at k, and the sum s of that sub-array

MaxSubArrWEndPoint (I, k)

```
beststart=k
bestsum = I[k]
currentsum = I[k]
for j=k-1 .. 1 do
    currentsum = currentsum + I[j]
    if currentsum > bestsum then
        bestsum = currentsum
        beststart = j
return beststart, bestsum
```

☞ Linear in *I.length*

input : An array I of integers
output: The maximal subarray of I given by its start and end indices, and the sum of that subarray

MAXSUBARRDC (I)

`// Base case`
if $I.length == 1$ **then**
  return *1,1,I[1]*
`// Divide`
$m = \lfloor I.length/2 \rfloor$
LeftI=$I\,[1..m]$
RightI=$I\,[m + 1 .. I.length]$
`// Conquer`
leftsolution=MAXSUBARRDC(*LeftI*)
rightsolution=MAXSUBARRDC(*RightI*)
`// Combine`
crosssolution= *concat*(MAXSUBARRWENDPOINT (I, m), MAXSUBARRWSTARTPOINT (I, $m + 1$))
return *best of* { *leftsolution, rightsolution, crosssolution* }
  `// 'best of' based on comparison of sum-component of solutions`

We also know: maximum sub-array either

- ends at final index *I.length*
- is contained in $I[1 \,..\, I.length - 1]$

```
MAXSUBARRRECURSIVE (I)
// Base case
if I.length == 1 then
    return 1,1,I[1]
// Recursion
Iminusone=I [1 .. I.length − 1]
recsolution=MAXSUBARRRECURSIVE(Iminusone)
rightsolution=MAXSUBARRWENDPOINT (I, I.length)
return best of recsolution, rightsolution
```

Compare worst-case time complexity of

- NaiveMaxSubArr (is $\Theta(n^2)$ according to exercises)
- MaxSubArrDC
- MaxSubArrRec

Which of the following is true?

A $T_{DC} = T_{Rec} = T_{Naive}$
B $T_{DC} = T_{Rec} < T_{Naive}$
C $T_{DC} < T_{Rec} = T_{Naive}$
D $T_{DC} < T_{Rec} < T_{Naive}$

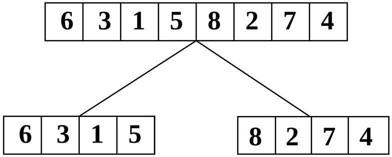
($T_X = T_Y$ means $T_X = \Theta(T_Y)$;
$T_X < T_Y$ means $T_X = O(T_Y)$ and $T_Y \neq O(T_X)$)

Analyzing Recursive Algorithms

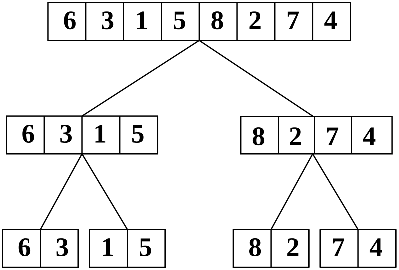
Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:

6	**3**	**1**	**5**	**8**	**2**	**7**	**4**

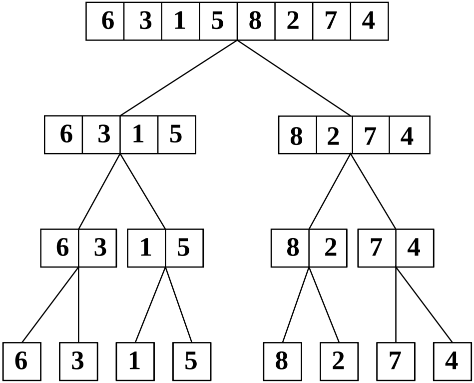
Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:



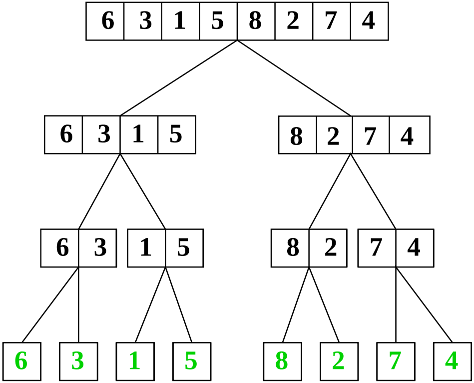
Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:



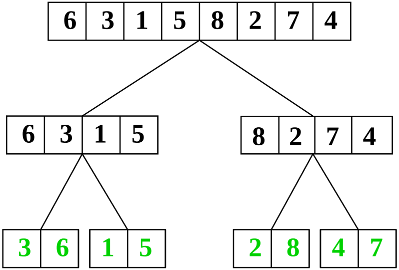
Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:



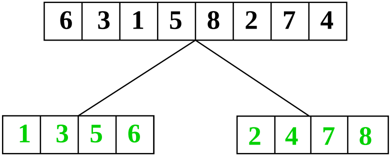
Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:



Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:



Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:

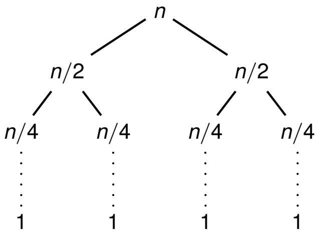


Recursive calls for MERGESORT on input $(6, 3, 1, 5, 8, 2, 7, 4)$:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

- Nodes correspond to recursive calls in an arbitrary run of the algorithm on input of size n
- Nodes represent the time complexity of executing the call, without the cost of the recursion

Recursion tree for MERGESORT (for n a power of 2):



- Total computation time for the nodes at depth k: $O(2^k \cdot \frac{n}{2^k}) = O(n)$
- Number of levels: $\lg n$
- Total computation time: $T(n) = O(n \lg n)$ (This is still a “Guess”, or informal proof!)

The recursion tree for MAXSUBARRDC is identical to the one for MERGESORT

☞ $O(n\,log(n))$ complexity for MAXSUBARRDC.

Recursion Tree for MAXSUBARRRECURSIVE

$$\begin{array}{c} n \\ | \\ n-1 \\ | \\ n-2 \\ \vdots \\ 1 \end{array}$$

- Total computation time:
  $T(n) = O(n + (n-1) + (n-2) + \ldots + 1) = O(n^2)$
- Also: $T(n) = \Omega(n^2)$

Solving Recurrences

The runtime of a recursive algorithm can be described by a *recurrence*.

Recurrence of MERGESORT (and many other D&C algorithms):

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

To **prove** that $T(n) = O(f(n))$:

- "Guess" the correct form of $f(n)$ (e.g. use recursion tree)
- Prove by induction on n that the guessed $f(n)$ satisfies the recurrence equations

The induction proof must show the existence of specific constants c, n_0, so that

$$T(n) \leq c \cdot f(n) \text{ for all } n \geq n_0.$$

This can be done either by

- explicit specification, e.g., $c = 1.7,\ n_0 = 46$.
- implicit specification – c and n_0 are defined as the solutions to a certain collection of equations and conditions.

☞ proof for $T(n) = \Theta(f(n))$ must be done in two parts: $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$, because the constants for the two parts are different.

A D&C recurrence where sub-problems have an “overlap” of size 4:

$$T(n) = 2T(n/2 + 4) + \Theta(n)$$

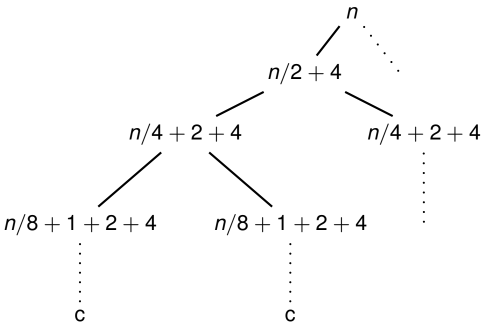
Base case

The base case of the recursion characterized by this recurrence must treat all inputs of size ≤ 8 (otherwise e.g. $T(6) = 2(T(3 + 4)) = 2T(7)$)

More detailed recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 8 \quad \text{(base case)} \\ 2T(n/2 + 4) + \Theta(n) & \text{if } n > 8 \quad \text{(recursion case)} \end{cases}$$

Is this still $O(n \lg n)$?



Total time for level d of the tree (starting with $d=0$):

$$2^d \cdot (n/2^d + 4 \cdot (1 + 1/2 + \ldots + (1/2)^{d-1}) < 2^d \cdot (n/2^d + 8)$$

(using sum formula for geometric series).

Summing over all levels d:

$$\sum_{d=0}^{\lg n - 3} 2^d \cdot (n/2^d + 8) \leq (\lg n - 2)(n + 2^{\lg n - 3} \cdot 8) = (\lg n - 2) \cdot 2n = O(n \lg n)$$

Let $k \geq 2, a \geq 1$:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq \frac{k}{k-1}a \quad \text{(base case)} \\ kT(n/k + a) + \Theta(n) & \text{if } n > \frac{k}{k-1}a \quad \text{(recursion case)} \end{cases}$$

Then $T(n) = O(n \lg n)$.