Chapter 1
Divide and Conquer

Algorithm Theory
WS 2013/14

Fabian Kuhn

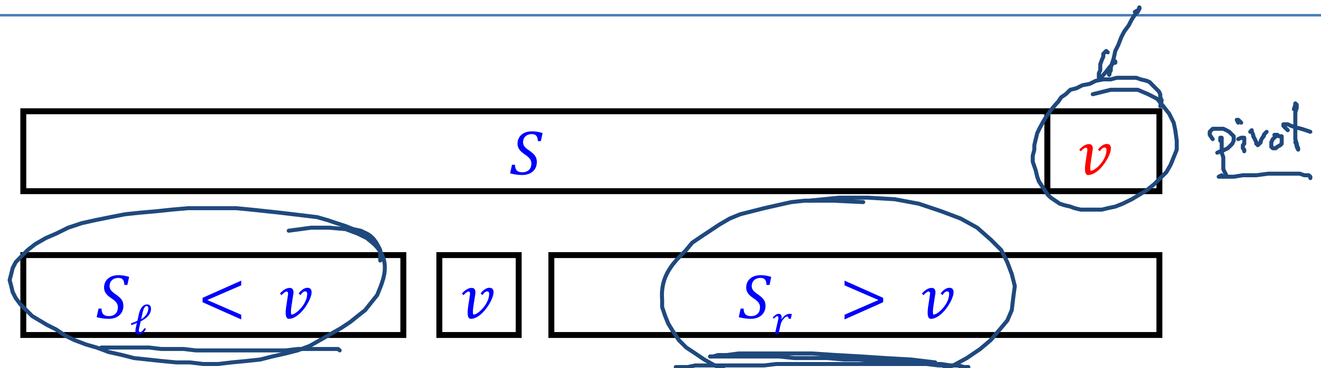
Divide-And-Conquer Principle

- Important algorithm design method

- Examples from Informatik 2:
  - Sorting: Mergesort, Quicksort
  - Binary search can be considered as a divide and conquer algorithm

- Further examples
  - Median
  - Compairing orders
  - Delaunay triangulation / Voronoi diagram
  - Closest pairs
  - Line intersections
  - ~~Integer factorization~~ / FFT
  - ...

Example 1: Quicksort



function Quick (S: sequence): sequence;

{returns the sorted sequence S}

begin

  if $\#S \leq 1$ then **return** S

  else { choose pivot element v in S;

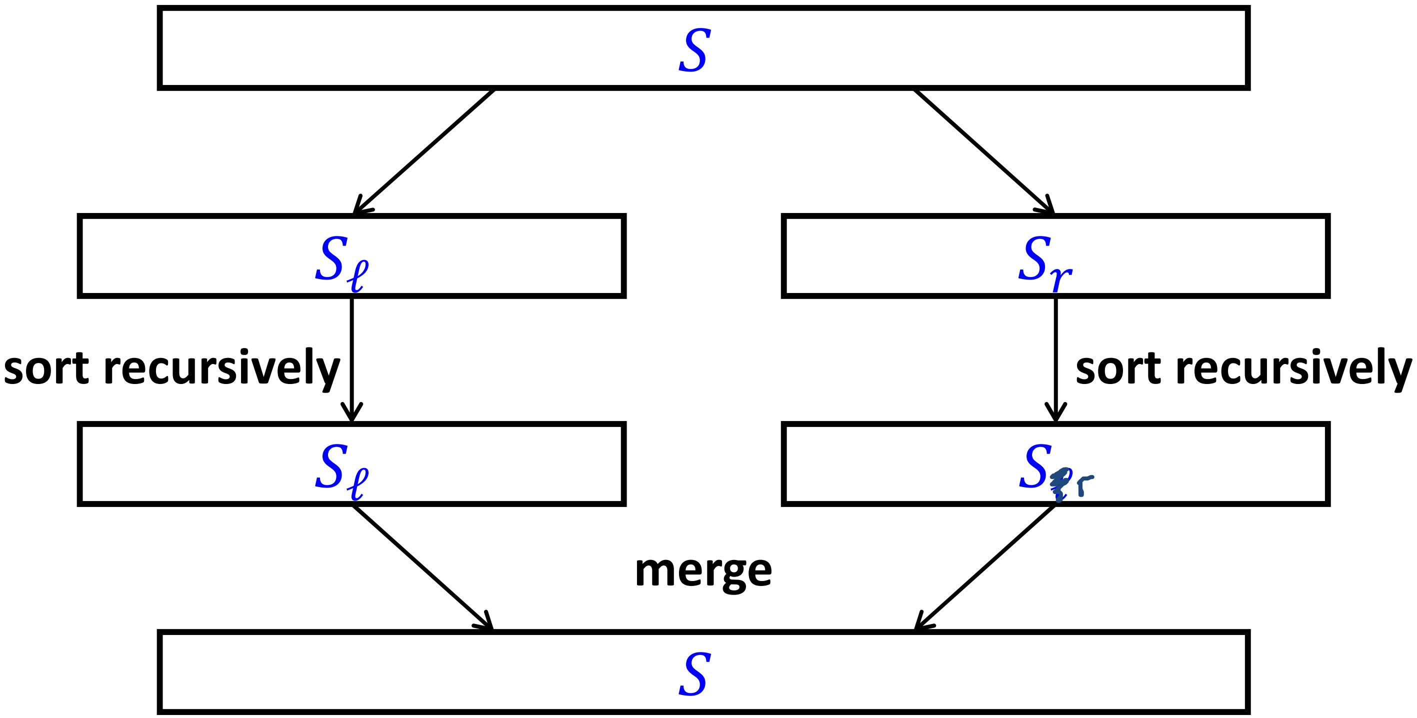
    partition S into S_ℓ with elements $< v$,

    and S_r with elements $> v$

    return $\boxed{\text{Quick}(S_\ell) \mid v \mid \text{Quick}(S_r)}$

end;

Example 2: Mergesort



Formulation of the D&C principle

Divide-and-conquer method for solving a problem instance of size n:

1. Divide

$n \leq c$: Solve the problem directly.

$n > c$: Divide the problem into k subproblems of sizes $n_1, \ldots, n_k < n$ $(k \geq 2)$.

2. Conquer

Solve the k subproblems in the same way (recursively).

3. Combine

Combine the partial solutions to generate a solution for the original instance.

QS	MS
partition	trivial
-	-
trivial	merge

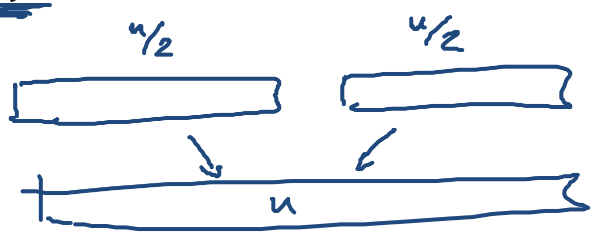
Analysis

Recurrence relation:

- $\boldsymbol{T(n)}$: max. number of steps necessary for solving an instance of size n
- $\boldsymbol{T(n) = \begin{cases} a & \text{if } n \leq c \\ T(n_1) + \cdots + T(n_k) & \text{if } n > c \\ + \text{cost for divide and combine} & \end{cases}}$

Special case: $\boldsymbol{k = 2, n_1 = n_2 = {}^n/_2}$

- cost for divide and combine: $\mathrm{DC}(n)$
- $T(1) = a$
- $T(n) = 2T(n/2) + \mathrm{DC}(n)$



Analysis, Example

Recurrence relation:

$$(*) \qquad T(n) \leq 2 \cdot T(n/2) + cn^2, \qquad T(1) \leq a$$

Guess the solution by repeated substitution:

$$T(n) \leq 2\underbrace{T(n/2)}_{(*)} + cn^2$$

$$\leq 2\left(2 \cdot T(n/4) + c\left(n/2\right)^2\right) + cn^2 = 4\underbrace{T(n/4)}_{(*)} + \left(c + \frac{c}{2}\right)n^2$$

$$\leq 4\left(2T(n/8) + c\left(n/4\right)^2\right) + \left(c + \frac{c}{2}\right)n^2 = 8T(n/8) + \left(c + \frac{c}{2} + \frac{c}{4}\right)n^2$$

$$\vdots$$

$$\leq 2^{\log_2 n} T(1) + 2cn^2 \leq \underbrace{a \cdot n + 2cn^2}_{\text{guess}}$$

Analysis, Example

Recurrence relation:

$$T(n) \leq 2 \cdot T(n/2) + cn^2, \qquad T(1) \leq a$$

Verify by induction:

guess: $T(n) \leq a \cdot n + 2cn^2$

ind. base: $n=1$ $\quad T(1) \leq a + 2c$ ✓

ind. step: $T(n) \leq 2 \cdot T(n/2) + cn^2$

$\overset{\text{I.H.}}{\leq} 2\left(a \cdot \frac{n}{2} + 2c\left(\frac{n}{2}\right)^2\right) + cn^2$

$= a \cdot n + 2cn^2.$ $\square$

Comparing Orders

- Many web systems maintain user preferences / rankings on things like books, movies, restaurants, …

- Collaborative filtering:

  b, a, f, e, d
  2, 1, 4, 5, 3

  – Predict user taste by comparing rankings of different users.
  – If the system finds users with similar tastes, it can make recommendations (e.g., Amazon)

  a, b, d, f, e
  1 2 3 4 5

- Core issue: Compare two rankings
  – Intuitively, two rankings (of movies) are more similar, the more pairs are ordered in the same way
  – Label the first user's movies from 1 to n according to ranking
  – Order labels according to second user's ranking
  – How far is this from the ascending order (of the first user)?

Number of Inversions

Formal problem:

- **Given**: array $A = [a_1, a_2, a_3, \dots, a_n]$ of distinct elements

- **Objective**: Compute number of inversions I

$$I := \left|\{0 \leq i < j \leq n \mid a_i > a_j)\}\right|$$

- **Example**: $A = [\ 4\ ,\ 1\ ,\ 5\ ,\ 2\ ,\ 7\ ,\ 10\ ,\ 6\ ]$

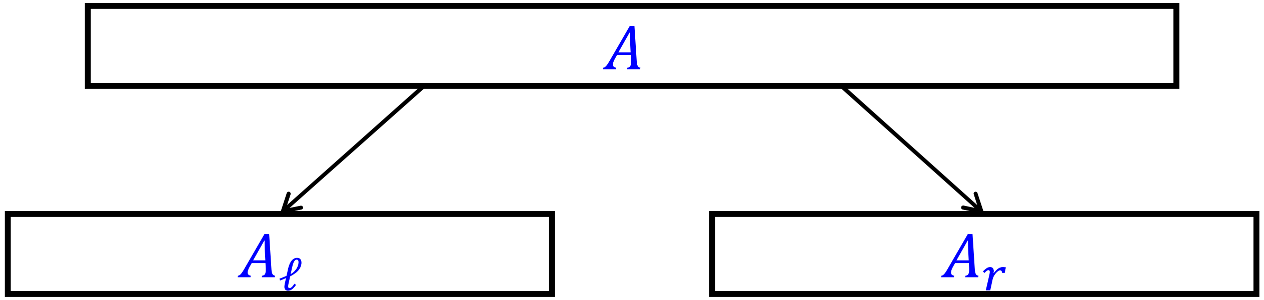
5 inversions

- **Naive solution**:

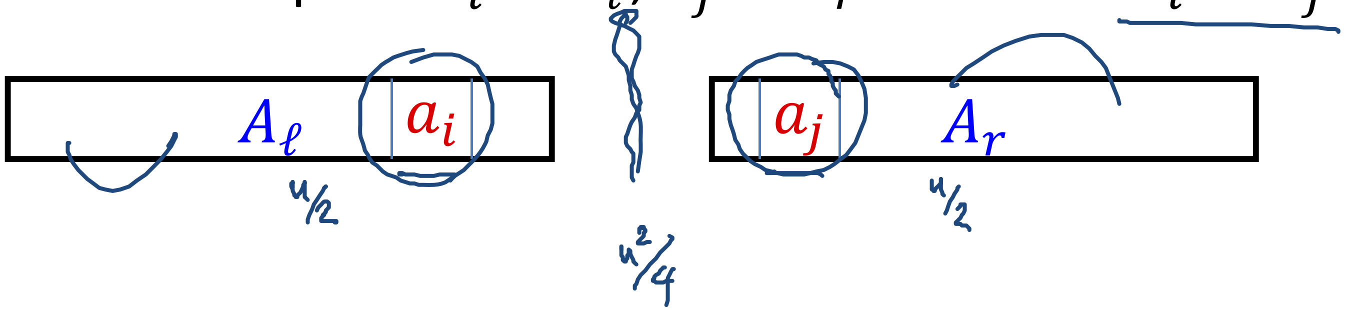
look at all pairs

time: $O(n^2)$

Divide and conquer

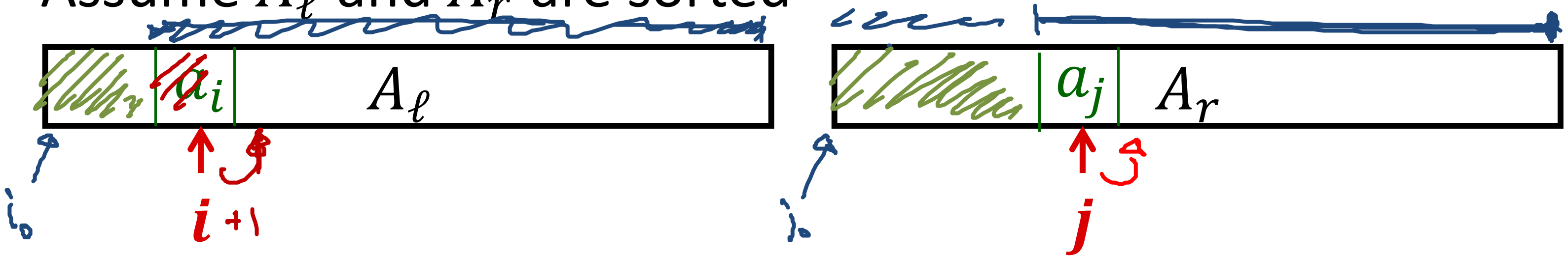


1. Divide array into 2 equal parts A_ℓ and A_r
2. Recursively compute #inversions in A_ℓ and A_r
3. Combine: add #pairs $a_i \in A_\ell$, $a_j \in A_r$ such that $a_i > a_j$



Combine Step

- Assume A_ℓ and A_r are sorted



- Pointers i and j, initially pointing to first elements of A_ℓ and A_r
- If $a_i < a_j$: incr. i
  - a_i is smallest among the remaining elements
  - No inversion of a_i and one of the remaining elements
  - Do not change count
- If $a_i > a_j$: incr. j
  - a_j is smallest among the remaining elements
  - a_j is smaller than all remaining elements in A_ℓ
  - Add number of remaining elements in A_ℓ to count
- Increment point, pointing to smaller element

$O(n)$

Combine Step

- Need sub-sequences in sorted order
- Then, combine step is like merging in merge sort

- **Idea**: Solve sorting and #inversions at the same time!
  1. Partition A into two equal parts A_ℓ and A_r
  2. Recursively compute #inversions and sort A_ℓ and A_r

     count #inv + sort

  3. Merge A_ℓ and A_r to sorted sequence, at the same time, compute number of inversions between elements a_i in A_ℓ and a_j in A_r

$O(n)$