Chapter 1
Divide and Conquer

Algorithm Theory
WS 2016/17

Fabian Kuhn
Divide-And-Conquer Principle

• Important algorithm design method

• Examples from basic alg. & data structures class (Informatik 2):
  • Sorting: Mergesort, Quicksort
  • Binary search

• Further examples
  • Median
  • Comparing orders
  • Convex hull / Delaunay triangulation / Voronoi diagram
  • Closest pairs
  • Line intersections
  • Polynomial multiplication / FFT
  • ...
Example 1: Quicksort

function Quick (S: sequence): sequence;
{returns the sorted sequence S}
begin
if #S ≤ 1 then return S
else { choose pivot element v in S;
partition S into \( S_\ell \) with elements \( \geq v \),
and \( S_\ell \) with elements \( \geq v \)
return Quick(\( S_\ell \)) v Quick(\( S_\ell \))
end;
Example 2: Mergesort

\[ S \]

sort recursively

\[ S_\ell \]

\[ S_\ell \]

merge

\[ S \]

sort recursively

\[ S_r \]

\[ S_\ell \]
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 $n_1, \ldots, n_k < n$ ($k \geq 2$). ($k=1$ possible)

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.
Analysis

Recurrence relation:

- \( T(n) \): max. number of steps necessary for solving an instance of size \( n \)

\[
T(n) = \begin{cases} 
  a & \text{if } n \leq c \\
  T(n_1) + \cdots + T(n_k) + \text{cost for divide and combine} & \text{if } n > c
\end{cases}
\]

Special case: \( k = 2, n_1 = n_2 = \frac{n}{2} \) \( n_1 = \lfloor \frac{n}{2} \rfloor, n_2 = \lceil \frac{n}{2} \rceil \)

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

Mergesort

\[
T(n) = 2T(\frac{n}{2}) + O(n)
\]

\[
T(n) = O(n \log n)
\]
Comparing Orders

• Many web systems maintain user preferences / rankings on things like books, movies, restaurants, ...

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

• 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 $A = [a_1, a_2, a_3, ..., a_n]$ of distinct elements

• **Objective**: Compute number of inversions $I$

\[
I := |\{0 \leq i < j \leq n \mid a_i > a_j\}|
\]

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

• **Naïve solution**: check all the pairs
  running time: $O(n^2)$
Divide and conquer

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

\[\text{count } \# a_i \in A_\ell, a_j \in A_r : a_j < a_i\]
Combine Step

Assume $A_\ell$ and $A_r$ are sorted

Idea:

- Maintain pointers $i$ and $j$ to go through the sorted parts
- While going through the sorted parts, we merge the two parts into one sorted part (like in MergeSort)
- and we count the number of inversions between the parts

Invariant:

- At each point in time, all inversions involving some element left of $i$ (in $A_\ell$) or left of $j$ (in $A_r$) are counted
  - and all others still have to be counted...
Combine Step

Assume \( A_\ell \) and \( A_r \) are sorted

- Pointers \( i \) and \( j \), initially pointing to first elements of \( A_\ell \) and \( A_r \)
- If \( a_i < a_j \):
  - \( a_i \) is smallest among the remaining elements
  - No inversion of \( a_i \) and one of the remaining elements
  - Do not change count
- If \( a_i > a_j \):
  - \( a_j \) is smallest among the remaining elements
  - \( a_j \) is smaller than all remaining elements in \( A_\ell \)
  - Add number of remaining elements in \( A_\ell \) to count
- Increment point, pointing to smaller element
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 $A_\ell$ and $A_r$
2. Recursively compute #inversions and sort $A_\ell$ and $A_r$
   
   and recursively sort $A_\ell$ & $A_r$

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

   combine costs $O(n)$
Combine Step: Example

• Assume $A_\ell$ and $A_r$ are sorted

\[
\begin{array}{cccccccc}
3 & 5 & 8 & 13 & 14 & 18 & 24 & 25 & 30 \\
\hline
6 & 7 & 9 & 19 & 21 & 23 & 28 & 32 & 33
\end{array}
\]

\[
3 \ 5 \ 6 \ 7 \ 8 \ 9 \ 13 \ 14 \ 18
\]

\[0 + 7 + 7 + 6\]
Number of Inversion: Analysis

Recurrence relation:

\[ T(n) \leq 2 \cdot T(n/2) + cn, \quad T(1) \leq a \]

Guess the solution by repeated substitution:

\[
T(n) \leq 2T(n/2) + cn \\
\leq 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\
\leq 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn \\
\vdots \\
\leq 2^k T(n/2^k) + kcn \\
\leq \left( \begin{array}{c}
\log_2 n \quad \text{(set } 2^k = n) \\
\end{array} \right) \\
(n=2^k) \\
\leq n T(1) + cn \log_2 n = n (a + c \log_2 n) \]
Number of Inversions: Analysis

Recurrence relation:

\[ T(n) \leq 2 \cdot T(n/2) + cn, \quad T(1) \leq a \]

Verify by induction:

Guess: \[ T(n) \leq n(a + c \log n) \]

Base: \( n = 1 : T(1) \leq a \)

Step: guess is true for instances of size \( < n \)

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + cn
\]

\[
\leq 2 \cdot \frac{n}{2} (a + c \log \frac{n}{2}) + cn
\]

\[
= n(a + c + c \log \frac{n}{2}) = n(a + c \log n)
\]

\[
\log n - 1
\]
Number of Inversions: Analysis

Recurrence relation:

\[ T(n) \leq 2 \cdot T(n/2) + cn, \quad T(1) \leq a \]

Guess the solution by drawing the recursion tree:
**Geometric divide-and-conquer**

**Closest Pair Problem**: Given a set $S$ of $n$ points, find a pair of points with the **smallest distance**.

**Naïve solution**: check all pairs $O(n^2)$
Divide-and-conquer solution

1. **Divide:** Divide $S$ into two equal sized sets $S_\ell$ und $S_r$.
2. **Conquer:** $d_\ell = \text{mindist}(S_\ell)$, $d_r = \text{mindist}(S_r)$
3. **Combine:** $d_{\ell r} = \min\{d(p_\ell, p_r) \mid p_\ell \in S_\ell, p_r \in S_r\}$
   \[\text{return } \min\{d_\ell, d_r, d_{\ell r}\}\]
Divide-and-conquer solution

1. **Divide:** Divide $S$ into two equal sized sets $S_\ell$ und $S_r$.
2. **Conquer:** $d_\ell = \text{mindist}(S_\ell)$ \quad $d_r = \text{mindist}(S_r)$
3. **Combine:**
   
   $d_{\ell r} = \min\{d(p_\ell, p_r) \mid p_\ell \in S_\ell, p_r \in S_r\}$
   
   return $\min\{d_\ell, d_r, d_{\ell r}\}$  
   \text{Obs: only need } d_{\ell r} \text{ if } d_{\ell r} \leq d$

**Computation of $d_{\ell r}$:**

\[ S \]

\[ d = \min\{d_\ell, d_r\} \]

\[ d_{\ell r} = \min\{d(p_\ell, p_r) \mid p_\ell \in S_\ell, p_r \in S_r\} \]

return $\min\{d_\ell, d_r, d_{\ell r}\}$  
\text{Obs: only need } d_{\ell r} \text{ if } d_{\ell r} \leq d
Combine step

\[ d = \min\{d_\ell, dr\} \]
Combine step

1. Consider only points within distance $\leq d$ of the bisection line, in the order of increasing $y$-coordinates.

2. For each point $p$ consider all points $q$ on the other side which are within $y$-distance less than $d$

3. There are at most 4 such points.
Implementation

• Initially **sort** the points in $S$ in order of increasing $x$-coordinates

• **While** computing closest pair, also **sort** $S$ according to $y$-coord.
  – Partition $S$ into $S_\ell$ and $S_r$, solve and sort sub-problems recursively
  
  – Merge to get sorted $S$ according to $y$-coordinates
    
    \[
    \text{combine + merge in } \Theta(n) \text{ time}
    \]
  – Center points: points within $x$-distance $d = \min\{d_\ell, d_r\}$ of center
  – Go through center points in $S$ in order of incr. $y$-coordinates
Running Time

Recurrence relation:

\[ T(n) = 2 \cdot T(n/2) + c \cdot n, \quad T(1) = a \]

Solution:

- Same as for computing number of number of inversions, merge sort (and many others...)

\[ T(n) = O(n \cdot \log n) \]