VSB TECHNICAL UNIVERSITY OF OSTRAVA
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

Decrease and Conquer

Jiří Dvorský, Ph.D.

Presentation status to date February 24, 2025

Department of Computer Science
VSB – Technical University of Ostrava

Lecture outline

- Decrease and Conquer
  - Sorting by insertion – Insertion Sort
  - Topological sorting
  - Generating combinatorial objects
    - Generating permutations
    - Generating subsets
- Reduction by a constant factor
- Reduction by a variable factor

Decrease and Conquer

Sorting by insertion – Insertion Sort

Decrease and Conquer

Topological sorting

Decrease and Conquer

Generating combinatorial objects

Generating combinatorial objects

- Generating combinations, variations, permutations, subsets is part of various algorithms.
- Typically it involves selecting some alternative, option, setting parameters...
- Examples – Traveling salesman problem, Knapsack problem.
- Mathematics is more interested in counting these objects, while computer science seeks algorithms to generate them.
- The number of these objects grows exponentially or even faster!

Generating permutations

- We will generate permutations of integers $1, 2, \dots, n$.
- More generally, we can generate permutations of elements $\{a_1, a_2, \dots, a_n\}$.
- Using the decrease and conquer strategy:
  1. Generating $n!$ permutations for n elements is reduced to generating $(n - 1)!$ permutations of $n - 1$ elements.
  2. Once we have solved the problem for $n - 1$, we insert element n into all n possible positions in each of the $(n - 1)!$ permutations.
  3. In other words, we have $(n - 1)!$ permutations, and for each of them, we generate n additional ones. Overall, we obtain $n(n - 1)! = n!$ permutations.

Generating permutations – example

permutation of element 1	1	
insertion of 2 into permutation 1 from right to left	1**2**	**2**1
insertion of 3 into permutation 12 from right to left	12**3**	1**3**2
insertion of 3 into permutation 21 from left to right	**3**21	2**3**1

What is evident from the example?

- All permutations are mutually distinct.
- Minimal change between permutations – two consecutive permutations differ by swapping a single pair of elements and even adjacent elements.

Johnson-Trotter algorithm

- Is there a possibility to generate permutations of n elements? Without the need to generate permutations for $n - 1$? Yes, there is.
- We assign an **arrow** (direction) to each of the n elements of the permutation, either to the left or to the right.
- We say that an element k is **mobile** in a given permutation if the neighboring element in the direction of the arrow of element k is smaller than k.

Example

Permutation with arrows

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overrightarrow{1}$$

Elements 3 and **4 are mobile**, elements 2 and **1 are not mobile**.

Johnson-Trotter algorithm

```
   Input  : Natural number $n$
   Output: List of all permutations of numbers $\{1, \dots, n\}$
1  $\pi \leftarrow \overleftarrow{1}\ \overleftarrow{2}\ \dots\ \overleftarrow{n}$;
2  while $\pi$ contains a mobile element do
3      $k \leftarrow$ largest mobile element in $\pi$;
4      swap in $\pi$ the element $k$ with its neighbor in the
         direction of the arrow;
5      change the direction of the arrow for all elements
         greater than $k$;
6      insert the newly created permutation (step 4) $\pi$ into
         the resulting list;
7  end
8  return list of all permutations;
```

Johnson-Trotter algorithm – example

Example of generating permutations for $n = 3$

$$\begin{array}{ccc}
\overleftarrow{1} & \overleftarrow{2} & \overleftarrow{3} \\
\overleftarrow{1} & \overleftarrow{3} & \overleftarrow{2} \\
\overleftarrow{3} & \overleftarrow{1} & \overleftarrow{2} \\
\overrightarrow{3} & \overleftarrow{2} & \overleftarrow{1} \\
\overleftarrow{2} & \overrightarrow{3} & \overleftarrow{1} \\
\overleftarrow{2} & \overleftarrow{1} & \overrightarrow{3}
\end{array}$$

We say that an element k is **mobile** in a given permutation if the neighboring element in the direction of the arrow of element k is smaller than k.

Johnson-Trotter algorithm – example, $n = 4$

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}$

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}$

$\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}$

$\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$

$\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$

$\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}$

$\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}$

$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}$

$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}$

$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}$

$\overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}$

$\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$

$\overrightarrow{4}\ \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

$\overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$

$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$

$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}$

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}$

$\overleftarrow{2}\ \overleftarrow{4}\ \overrightarrow{3}\ \overleftarrow{1}$

$\overleftarrow{4}\ \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$

$\overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}\ \overrightarrow{3}$

$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}$

Johnson-Trotter algorithm

- One of the most efficient algorithms for generating permutations.
- The time complexity of the algorithm is $\Theta(n!)$.
- The ”fearsome” complexity of the algorithm, however, is not caused by the algorithm itself, which works very quickly. It is caused by the enormous number of permutations that must be generated...

Thanks for your attention