## Algorithms

- **Instructors:**
  - Vladimir Kolmogorov
  - Krzysztof Pietrzak

- **Teaching assistant:**
  - Pepa Tkadlec

- Final grade = 50% homeworks + 50% exam

- HW1: available today, due next Monday before the lecture

## Topics

- **Design techniques**
  1. Divide-and-Conquer (VK)
  2. Prune-and-Search (VK)
  3. Dynamic programming (KP)
  4. Greedy algorithms (KP)

- **Search trees, priority queues**
  5. Binary search trees (KP)
  6. Amortized analysis (KP)
  7. Heaps, heapsort (VK)

- **Graph algorithms (VK)**
  8. Graph search
  9. Shortest paths

- **String algorithms**
  10. Knuth-Morris-Pratt alg. (KP)
  11. Data structures for strings (VK)

- **“Hard” problems**
  12. Concluding lecture (KP)

### The sorting problem

- **Input:** sequence of items
  - e.g. integers, words in a dictionary, ...
  - stored in an array
  
  \[
  \begin{array}{cccccccc}
  5 & 4 & 3 & 7 & 2 & 9 & 1 & 8 & 1 \\
  \end{array}
  \]

- **Output:** sorted sequence
  
  \[
  \begin{array}{cccccccc}
  1 & 1 & 2 & 3 & 4 & 5 & 7 & 8 & 9 \\
  \end{array}
  \]

### Complexity of Bubble Sort

- **Claim:** after \( i \) iterations the \( i \) largest numbers are in the correct positions (in the end of the array)

- **Worst-case complexity:** \( O(n^2) \)
  - \( \text{achieved on some inputs} \)

- **Best-case complexity:** \( O(n) \)

### Bubble Sort algorithm

```plaintext
\[
\begin{align*}
\text{do } & \quad \text{success} = \text{true} \\
\text{for } i = 1..n-1 & \quad \text{do } \\
& \quad \text{if } A[i] > A[i+1] \quad \text{then} \\
& \quad \quad \text{exchange } A[i] \text{ and } A[i+1]; \\
& \quad \quad \text{success} = \text{false}; \\
\text{end } & \quad \text{while } (\text{success} == \text{false})
\end{align*}
\]
```

\[
\begin{array}{cccccccc}
5 & 4 & 3 & 7 & 2 & 9 & 1 & 8 & 1 \\
\end{array} \rightarrow \begin{array}{cccccccc}
2 & 3 & 4 & 5 & 7 & 1 & 8 & 9 \\
\end{array}
\]

### Big O notation

- **Algorithm’s runtime on an input of size \( n \) is at most \( c \cdot f(n) \) for some constant \( c > 0 \), if \( n \) is sufficiently large (i.e. if \( n > N \) for some constant \( N \))**

- **Sorting \( n \) numbers:**
  - Merge sort: \( O(n \log n) \)
  - Bubble sort, insertion sort: \( O(n^2) \)

- **Quering \( i \)-th element of array \( A[1..n] \): \( O(1) \)**

- \( O(n \log n) \) alg. typically faster than \( O(n^2) \) alg. for sufficiently large inputs
Big O notation

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow O(f(n)) = O(g(n)) \]

- \( O(an^2 + bn + c) = O(n^2) \) \((a > 0)\)
- \( O(an \log n + bn + c) = O(n \log n) \)

Divide-and-conquer

- Divide the problem into several subproblems
- Solve each subproblem recursively
- Combine solutions

Applications:
- Sorting (QuickSort, MergeSort)
- Fast Fourier Transform (FFT)
- Matrix multiplication (Strassen’s algorithm)
- ...

void QuickSort(int left, int right)

\[
\text{if } left < right \text{ then }
\{
\text{i = Split(left, right);}
\text{QuickSort(left, i - 1);}
\text{QuickSort(i + 1, right);}
\}
\]

int Split(int left, int right)

- Can be implemented in-place
  - no extra memory allocated
- Idea:
  - make sure that elements \( x < 5 \) come before elements \( y > 5 \)
Running time

- Running time: sum of all lengths

Running time: worst case

- Already sorted sequence: $T(n) = n + T(n-1)$
  $$n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2}$$
- Quadratic running time: $T(n) = O(n^2)$
  - i.e. $T(n) \leq \text{const} \cdot n^2$ for some const

Running time: best case

- Master theorem (see e.g. wikipedia):
  - recipe for solving such recurrences
- Not covered in this course

```c
int Split(int left, int right)
pivot = A[left]; i = left; j = right + 1;
while TRUE do {
dot j -- while i < j and A[j] \geq pivot;
dot i ++ while i < j and A[i] \leq pivot;
if i < j then exchange A[i] and A[j];
else {
exchange A[left] and A[i];
return i;
}
}```
Running time: best case

\[
T(n) = n + 2 \cdot T \left( \frac{n-1}{2} \right)
\]

\[
n = 2^k - 1
\]

\[
T(n) = n - (2^0 - 1) + n - (2^1 - 1) + n - (2^2 - 1) + \ldots + n - (2^{k-1} - 1)
\]

\[
= kn - (2^k - 1) + k
\]

\[
T(n) = (n+1) \cdot \log_2(n+1) - n
\]

For general \( n \):

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

\[
T(n) \leq \text{const} \cdot n \log n
\]

Running time

- QuickSort: runtime depends on input data
  - worst case: \( O(n^2) \) (e.g. if already sorted)
  - best case: \( O(n \log n) \)
- Randomized QuickSort: randomly select pivot

```c
int rSplit(int left, int right)
{
    p = Random(left, right);
    exchange A[left] and A[p];
    return Split(left, right);
}
```

Randomized QuickSort - Average complexity

- \( T(n) \): expected runtime for \( n \) elements
- Assumption: all elements are distinct

\[
T(n) = n + \frac{1}{n} \sum_{m=0}^{n-1} (T(m) + T(n-m-1))
\]

Randomized QuickSort - Average complexity

- \( T(n) \): expected runtime for \( n \) elements
- Assumption: all elements are distinct

\[
T(n) = n + \frac{1}{n} \sum_{i=0}^{n-1} T(i)
\]

\[
(n-1)T(n-1) = (n-1)^2 + 2 \cdot \sum_{i=0}^{n-2} T(i)
\]

\[
nT(n) - (n-1)T(n-1) = n^2 - (n-1)^2 + 2T(n-1)
\]
Randomized QuickSort - Average complexity

\[ \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \]

\[ U(n) = \frac{T(n)}{n+1} \]

\[ U(n) = U(n-1) + \frac{2n-1}{n(n+1)} \]

\[ = \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \]

\[ = 2 \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)} \]

Randomized QuickSort - Average complexity

\[ \frac{T(n)}{n+1} = \frac{2}{n+1} \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)} \]

\[ \sum_{i=1}^{n} \left( \frac{1}{i} - \frac{1}{i+1} \right) = 1 - \frac{1}{n+1} \]

Randomized QuickSort - Average complexity

- Complexity: \( T(n) = O(n \log n) \)

\[ T(n) < 2 \cdot (n+1) \cdot \log(n+1) \]

- Approximately \( \frac{2}{\log_2 e} \approx 1.386... \) slower than the best case

Randomized QuickSort - Average complexity

\[ \text{Stack & extra space} \]

- Worst-case stack size: \( O(n) \)
- \texttt{QuickSort} is \textit{tail-recursive}
  - as the last step calls itself
  - naive implementation (with stack) inefficient
Removing tail recursion

void QuickSort(int left, int right)
while left < right do {
  i = Split(left, right);
  QuickSort(left, i - 1);
  left = i + 1;
}

- Worst-case stack size: still \(O(n)\) ...

Removing tail recursion for larger side

void QuickSort(int left, int right)
while left < right do {
  i = Split(left, right);
  if i - left < right - i then
    QuickSort(left, i - 1); left = i + 1;
  else
    QuickSort(i + 1, right); right = i - 1;
  endif
}

- Worst-case stack size: \(O(\log n)\)

Summary

- Deterministic QuickSort:
  - Worst-case: \(O(n^2)\)
  - Average over data instances (random permutations): \(O(n \log n)\)
- Randomized QuickSort:
  - Worst-case: \(O(n^2)\)
  - Average: \(O(n \log n)\) [assuming distinct elements]
- Extra space (worst-case): \(O(\log n)\)
  - Use recursive call only for the smaller side
- One of the fastest sorting algorithms in practice

- Techniques:
  - Divide-and-conquer
  - Randomization