In the previous episodes

- **O-notation**

- **Data structure implementations:**
  Lists, Queues, Heaps, Trees, Maps, Graphs

- **Algorithms:**
  Sorting, Searching, Dynamic programming, Graph Algorithms, …
A typical algorithmic problem goes as follows:

Given a list, produce a list, which has the same elements in sorted order.

**SORTING**

- Given a list,
- produce a list,
- which has the same elements in sorted order.

Given a graph, produce a tree, which is spanning and has minimal weight.

**MST**
Suppose you have an algorithmic problem at hand. What are the ways of approaching it?
Generic algorithmic techniques

- Exhaustive search
Generic algorithmic techniques

- Exhaustive search
Generic algorithmic techniques

- Exhaustive search
Generic algorithmic techniques

- Iterative improvement (directed search)
Generic algorithmic techniques

- Iterative improvement (directed search)
  - Start with a random state
  - If it is not optimal:
    - (e.g. there are two nearby positions in wrong order)
    - Improve
  - Repeat until solution optimal
Generic algorithmic techniques

- Iterative improvement (directed search)

- Start with a random state
- If it is not optimal:
  - (e.g. there are two nearby positions in wrong order)
  - Improve
- Repeat until solution optimal
Generic algorithmic techniques

- Greedy algorithm
Generic algorithmic techniques

- Greedy algorithm
  - Find the letter with the smallest value among the unsorted elements and use it to extend solution
  - Repeat recursively
Generic algorithmic techniques

- **Greedy algorithm**
  - Find the letter with the smallest value among the unsorted elements and use it to extend solution
  - Repeat recursively
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing $f(n)$ to $f(n-1)$)

- Multiple recurrence (reducing $f(n)$ to $[f(n-1), f(n-2), \ldots]$)
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

BDCA → BDCA → BDAC → ABCD
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

BDCA → BD → BD → ABCD

Merge sort
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

```
BDCA → BD
  CA → BD
    AC → ABCD
```

```
BDCA → BA
  DC → AB
    CD → ABCD
```
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

Quicksort
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing $f(n)$ to $f(n-1)$)
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing $f(n)$ to $f(n-1)$)

BDCA $\rightarrow$ B $\rightarrow$ DCA $\rightarrow$ B $\rightarrow$ ACD $\rightarrow$ ABCD
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing $f(n)$ to $f(n-1)$)

```
BDCA  →  B  DCA  →  B  ACD  →  ABCD
```

Insertion sort
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing $f(n)$ to $f(n-1)$)

- Multiple recurrence (reducing $f(n)$ to $[f(n-1), f(n-2)$)
Generic algorithmic techniques

- Reducing to simpler tasks
  - “Divide and conquer”

- Single recurrence (reducing \( f(n) \) to \( f(n-1) \))

- Multiple recurrence (reducing \( f(n) \) to \([f(n-1), f(n-2), \ldots]\))

Diagram:

- BDCA
- BDC
- DCA
- BCD
- ACD
- ABCD

(bad) sort
Generic algorithmic techniques

- Exhaustive search
- Iterative improvement (directed search)
- Greedy algorithm
- Reducing to simpler tasks
  - Divide and conquer
  - Single recurrence
  - Multiple recurrence (Dynamic programming)
Generic algorithmic techniques

- Exhaustive search
- Iterative improvement
- Greedy algorithm
- Reducing to simpler tasks
  - Divide and conquer
  - Single recurrence
  - Multiple recurrence (Dynamic programming)

- \( O(n!) \)
- \( O(kn) \)
- \( O(n \lceil \log n \rceil) \)
- \( O(n \log n) \)
- \( O(n \log n) \)
- \( O(n^2) \)
Generic algorithmic techniques

- Complementary to the abovementioned techniques, there are helpful “tricks”:
  - Time / CPU trade-off (parallel/distributed computation)
  - Time / space trade-off (precomputation, recomputation)
  - Time / quality trade-off (approximation, heuristics)
  - “Time / probability trade-off” (randomization)
Randomized Algorithms

Konstantin Tretyakov (kt@ut.ee)

MTAT.03.238 Advanced Algorithmics
April 25, 2012
Worst-case scenarios

- If you are unlucky, then the complexity of
  - Quicksort is …
  - Hashmap is …
Worst-case scenarios

- If you are unlucky, then the complexity of
  - Quicksort is $O(n^2)$

- Hashmap is $O(n^2)$ for $n$ insertions

For $n = 10000$, this makes a 10 000 difference!

1 second becomes 3 hours!

- But why would you get unlucky?
  - Can’t we assume that inputs are random?
Inputs are NOT random!

Example:

- Bro is an IDS/packet filter
- Bro sniffs IP packets and stores them in a hash map.
Inputs are malicious

- **Example:**
  - Bro is an IDS/packet filter
  - Bro sniffs IP packets and stores them in a hash map.
  - It is easy to generate many packets that will hash to the same bucket.
  - \(O(n^2)\) performance!

<table>
<thead>
<tr>
<th></th>
<th>Attack</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>44.50 min</td>
<td>.86 min</td>
</tr>
<tr>
<td>Hash table time</td>
<td>43.78 min</td>
<td>.02 min</td>
</tr>
</tbody>
</table>

Table 2: Total CPU time and CPU time spent in hash table code during an offline processing run of 64k attack and 64k random SYN packets.

<table>
<thead>
<tr>
<th>Packet rate</th>
<th>Packets sent</th>
<th>Drop rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>16kb/s</td>
<td>192k</td>
<td>31%</td>
</tr>
<tr>
<td>16kb/s (clever)</td>
<td>128k</td>
<td>71%</td>
</tr>
<tr>
<td>64kb/s</td>
<td>320k</td>
<td>75%</td>
</tr>
<tr>
<td>160kb/s</td>
<td>320k</td>
<td>78%</td>
</tr>
</tbody>
</table>

Table 3: Overall drop rates for the different attack scenarios.

Inputs are malicious

Example:

<table>
<thead>
<tr>
<th>File version</th>
<th>Perl 5.6.1 program</th>
<th>Perl 5.8.0 program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl 5.6.1</td>
<td>6506 seconds</td>
<td>&lt;2 seconds</td>
</tr>
<tr>
<td>Perl 5.8.0</td>
<td>&lt;2 seconds</td>
<td>6838 seconds</td>
</tr>
</tbody>
</table>

Table 1: CPU time inserting 90k short attack strings into two versions of Perl.
Avoiding the worst case

- The algorithm needs to have a “secret key” – something the adversary does not know about!

- If we do not have any information at all about this key, we call it “randomness”.

25.04.2012
Algorithmics
Avoiding the worst case

Instead of considering a single algorithm, consider a **family of algorithms**.

- HashMap₁
- HashMap₂
- HashMap₃
- HashMap₄

...
Avoiding the worst case

- Instead of considering a single algorithm, consider a family of algorithms.

Each algorithm has its weak points:

- HashMap_1: Hard inputs: 43, 21, 1
- HashMap_2: Hard inputs: 10, 15, 1
- HashMap_3: Hard inputs: 22, 33, 6
- HashMap_4: Hard inputs: 26, 43, 2
- HashMap_2^{32}: Hard inputs: 39, 2, 74
Avoiding the worst case

- Instead of considering a single algorithm, consider a **family of algorithms**.

  - HashMap\(_1\)
  - HashMap\(_2\)
  - HashMap\(_3\)
  - HashMap\(_4\)

  But we’ll pick one at random, and the adversary **won’t know** which one!

- HashMap\(_{2^{32}}\)
  - Hard inputs: 39, 2, 74
Avoiding the worst case

- Instead of considering a single algorithm, consider a **family of algorithms**.

- **Universal hashing**
  - HashMap₁
  - Hard inputs: 43, 21, 1
  - HashMap₂
  - Hard inputs: 10, 15, 1
  - HashMap₃
  - Hard inputs: 22, 33, 6
  - HashMap₄
  - Hard inputs: 26, 43, 2
  - HashMap₂³²
  - Hard inputs: 39, 2, 74
A hash function family is called "\( \varepsilon \)-almost-universal" if

- For any two inputs \( x \) and \( y \)
- And a randomly picked \( h \)

The probability of a collision
\[
P(h(x) = h(y))
\]
Is no greater than \( \varepsilon \).

Ideally, \( \varepsilon = 1/m \) where \( m \) is the size of the domain of \( h \),
then we say simply "universal".
Example

```c
int8_t hash(int32_t x) {
    return (unsigned) x >> (32-8)
}
```

- Is this a hash function?
- Is it \( \varepsilon \)-universal?
Avoiding the worst case

Instead of considering a single algorithm, consider a family of algorithms.

Example

```c
int8_t hash(int32_t x) {
    return (unsigned) x >> (32-8)
}
```

• How to make it universal?
Avoiding the worst case

Example

```c
int8_t hash(int32_t key, int32_t x) {
    return (unsigned) k*x >> (32-8)
}
```

.. is $2 / 2^M$ - almost-universal

A Reliable Randomized Algorithm for the Closest-Pair Problem. Dietzfelbinger et al. (1997).
Avoiding the worst case

Example

```cpp
int128 hash(string x) {
    return md5(x)
}
```
Avoiding the worst case

Example

```cpp
int128 hash(string key, string x) {
    return md5(key + x)
}
```

No simple proof, but as good as universal.
Avoiding the worst case

- Instead of considering a single algorithm, consider a family of algorithms.

```
 HashMap₁
 HashMap₂
 HashMap₃
 HashMap₄

 Hard inputs: 43, 21, 1
 Hard inputs: 10, 15, 1
 Hard inputs: 22, 33, 6
 Hard inputs: 26, 43, 2

 ... Universal hashing

 HashMap₂³₂

 Hard inputs: 39, 2, 74
```
Avoiding the worst case

Instead of considering a single algorithm, consider a **family of algorithms**.

- QuickSort\(_1\)(A)
- QuickSort\(_2\)(A)
- QuickSort\(_3\)(A)
- QuickSort\(_4\)(A)

Hard inputs:
- QuickSort\(_1\)(A): 43, 21, 1
- QuickSort\(_2\)(A): 10, 15, 1
- QuickSort\(_3\)(A): 22, 33, 6
- QuickSort\(_4\)(A): 26, 43, 2

... Randomized QuickSort

- QuickSort\(_{232}\)(A)
  - Hard inputs: 39, 2, 74
Breaking symmetry

- Instead of considering a single algorithm, consider a **family of algorithms**.
Instead of considering a single algorithm, consider a **family of algorithms**.

... each algorithm in the family will find the solution eventually.
Instead of considering a single algorithm, consider a **family of algorithms**.

... each algorithm in the family will find the solution eventually,

... but their timing varies from “very quick” to “infinite”. The *expected time* is good, though.
Las Vegas Algorithms

- This approach is called “Las Vegas algorithm”

- Do we need Las-Vegas algorithms at all (i.e. do they give us additional power)?
Do we need Las-Vegas?

- The set of problems which have a polynomial Las-Vegas solution is called ZPP (Zero error Probabilistic Polynomial)

Obviously, $P \subseteq ZPP$. 
Do we need Las-Vegas?

- The set of problems which have a polynomial Las-Vegas solution is called ZPP (Zero error Probabilistic Polynomial).

Obviously, $P \subseteq ZPP$.
It is **not known** whether $P=ZPP$. 
The **real** randomized algorithms

- are those which can produce **wrong** answers.

Again, a family of functions.

We know that most produce the correct answer (but we don’t know which ones)
The **real** randomized algorithms

- Because often *sampling from a probability distribution* around the correct answer is much easier than producing the answer.

  - \(\text{Solve}_1(A)\) → Yes
  - \(\text{Solve}_2(A)\) → Yes
  - \(\text{Solve}_3(A)\) → Yes
  - \(\text{Solve}_4(A)\) → No

  ...  

Again, a family of functions.

We know that most produce the correct answer (but we don’t know which ones)

- \(\text{Solve}_{232}(A)\) → Yes
Example

- Given a property, that holds for at least 50% of nodes, find a node with that property (e.g. find a leaf in a binary tree)
Example

- Given a property, that holds for at least 50% of nodes, find a node with that property (e.g. find a leaf in a binary tree)

```javascript
function find(objects, randomness) {
    return (random object)
}
```
Example

- If we can check the output correctness (i.e. whether the picked node is indeed a leaf), we can amplify:

  - One run: fails with probability $0.50$
  - Two runs: fails with probability $0.25$
  - 64 runs: fails with probability $2^{-64}$
Monte-Carlo Algorithms

- Monte-Carlo algorithms have **nondeterministic** output, but deterministic run time.

- Are Monte-Carlo and Las-Vegas equivalent?
Monte-Carlo Algorithms

- Monte-Carlo algorithms have **nondeterministic** output, but deterministic run time.

- Are Monte-Carlo and Las-Vegas equivalent?

- If we can verify answers, then yes. Otherwise – no.
Monte-Carlo Algorithms

- Do we need Monte-Carlo algorithms?
Monte-Carlo Algorithms

- Do we need Monte-Carlo algorithms?

- The set of Monte-Carlo solvable problems is called BPP (Bounded error Probability Polynomial-time)

  \[ P \subseteq ZPP \subseteq BPP, \text{ but it is not known whether } ZPP = BPP. \]
Monte-Carlo Algorithms

- Two main tasks where Monte-Carlo algorithms are heavily applied:
  - Approximation
  - Verification
Monte-Carlo Algorithms

- Two main areas where Monte-Carlo algorithms are heavily applied:
  - Statistics, Data mining & AI
  - Cryptography

25.04.2012
Algorithmics
Monte-Carlo Examples

- Polynomial identity testing
  - Problem: Are two given multivariate polynomials equal?
Monte-Carlo Examples

- Polynomial identity testing
  - Problem: Are two given multivariate polynomials equal?
  - Solution: Test equality at a given number of points

- No known polynomial-time exact algorithm!
Monte-Carlo Examples

- Remote file comparison
  - Problem: How to check that a local file is equal to a remote one?
Remote file comparison

- Problem: How to check that a local file is equal to a remote one?
- Solution: Verify that hashes are equal.
Monte-Carlo Examples

- Matrix multiplication verification
  - Problem: verify that $A = BC$
Monte-Carlo Examples

- **Matrix multiplication verification**
  - Problem: verify that $A = BC$
  - Solution: sample random vectors $v$ and check that $Av = B(Cv)$
Monte-Carlo Examples

- Primality testing
  - Problem: verify that $p$ is prime
Monte-Carlo Examples

- Primality testing
  - Problem: verify that $p$ is prime
  - Solution* [Fermat test]:
    check that $a^p = a \mod p$
Monte-Carlo Examples

- Primality testing
  - Problem: verify that $p$ is prime
  - Solution* [Fermat test]: check that $a^p \equiv a \mod p$

PRIMES is in P

Manindra Agrawal Neeraj Kayal
Nitin Saxena* 2002, 2005
Monte-Carlo Examples

- Max 3-SAT
  - Given a set of 3-CNf formula, find a variable assignment that satisfies the largest number of clauses

\[
\begin{align*}
x_1 & \lor \overline{x}_2 \lor x_4 \\
x_2 & \lor x_5 \lor x_6 \\
\overline{x}_2 & \lor x_5 \lor \overline{x}_8 \\
x_4 & \lor \overline{x}_8 \lor x_9 \\
x_5 & \lor x_6 \lor \overline{x}_7 \\
\overline{x}_5 & \lor x_7 \lor \overline{x}_8 \\
\overline{x}_6 & \lor \overline{x}_7 \lor x_9 \\
\end{align*}
\]
Monte-Carlo Examples

Solution:

Pick a variable assignment at random!

Let \( Z_i = \begin{cases} 
1 & \text{if clause } i \text{ is satisfied} \\
0 & \text{otherwise.} 
\end{cases} \)

The total number of satisfied clauses is then

\[
Z = \sum_i Z_i
\]

And the expected number is:

\[
E[Z] = E \left[ \sum_i Z_i \right] = \sum_i E[Z_i] = \sum_i \frac{7}{8} = \frac{7}{8} n
\]

Surprisingly, it is close to the best possible algorithm
Summary: Randomized algorithms

- ZPP
  - Avoid adversaries
  - Break symmetries

- BPP
  - Approximations
  - Verification

P ⊂ ZPP ⊂ BPP