Algorithm Design: Nondeterministic and Randomized Algorithms

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2019/2020

Plan

Problem solved by an algorithm

An algorithm A solves a problem P if:

- for any instance p of P, there is an initial configuration $\langle A, \sigma_p \rangle$ such that σ_p includes data structures describing p;
- the execution starting from the initial configuration $\langle A, \sigma_p \rangle$ ends into a final configuration $\langle \cdot, \sigma' \rangle$, write $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$; and
- σ' includes data structures that describes $P(p)$.

Formally:

P is specified by $(pre, post)$

A rsolves $P \equiv (\forall \sigma)$ with $\sigma \models pre$ $(\exists \sigma')$ s.t. $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$ and $\sigma' \models post$

The execution time for an instance: the deterministic case

Deterministic algorithm: for any reachable configuration $\langle A_i, \sigma_i \rangle$ there is at most a successor configuration, i.e., at most a $\langle A', \sigma' \rangle$ such that $\langle A_i, \sigma_i \rangle \Rightarrow \langle A', \sigma' \rangle$.

All algorithms we considered up to now are deterministic! Later we shall consider nondeterministic algorithms as well.

Let P be a problem and A a deterministic algorithm that solves P.

For each $p \in P$ there is an unique execution path E_p.

The time for computing $P(p)$ is

$$time_d(A, p) = time_d(E_p)$$

where $d \in \{log, unif, lin\}$.

The size of an instance

The dimension of a state σ is

$$size_d(\sigma) = \sum_{\mathtt{x} \mapsto v \in \sigma} size_d(v)$$

The dimension of a configuration is

$$size_d(\langle A, \sigma\rangle) = size_d(\sigma)$$

where $d \in \{log, unif, lin\}$.

Let P be a problem, $p \in P$, and A a deterministic algorithm that solves P.

The size of p is the the size of its intial configuration:

$$size_d(p) = size_d(\langle A, \sigma_p\rangle) \quad (= size(\sigma_p))$$

where $d \in \{log, unif, , lin\}$.

The worst case time complexity

Let P be a problem and A a determinstic algorithm that solves P and fix $d \in \{log, unif, lin\}$.

Group the instances p of P into equivalences classes: p and p' are in the same equivalence class iff $size(p) = size(p')$.

A natural numaber n can be seen as the equivalence class of instances p of size n ($size_d(p) = n$).

The worst case time complexity:

$$T_{A,d}(n) = \max\{time_d(A, p) \mid p \in P, size_d(p) = n\}$$

Experiments with Alk interpreter1/3

```
isPrime1(x) {
  if (x < 2) return false;
  for (i= 2; i <= x/2; ++i)
    if (x % i == 0) return false;
  return true;
}
```

Experiments with Alk interpreter 2/3

A worst case:

```
print(isPrime1(2147483647));

$ time alki.sh -a isPrime.alk
^C
real 41m3.065s
user 40m45.849s
sys 0m15.643s
```

A more favorable case:

```
print(isPrime1(2147483647*457241)); //457241^2=209069332081

$ time alki.sh -a isPrime.alk
false

real 0m3.480s
user 0m5.921s
sys 0m0.236s
```

Experiments with Alk interpreter 3/3

A minor change could bring major improvements:

```
isPrime2(x) {
  if (x < 2) return false;
  for (i= 2; i*i <= x; ++i)
    if (x % i == 0) return false;
  return true;
}

print(isPrime2(2147483647));

$ time alki.sh -a isPrime.alk
true

real 0m1.659s
user 0m3.475s
sys 0m0.163s
```

Plan

Motivație

- abstraction of the state space is useful in analysis
- non-deterministic algorithms bring an additional abstraction level, which combines state abstraction with procedural abstraction
- ignores details how some data structures are created
- useful in complexity analysis
- preliminary notion for randomized algorithms

Non-deterministic Algorithms, Intuitively

- for some configurations there are more than one way to continue the execution
- consequently, for the same input the algorithm may have many executions with different results
- angelic version: the algorithm "guesses" the execution that leads to the correct result
- execution time: the time of the execution that leads to the correct result

Extending the Language

`choose` x `in` S;

– returns an element from S, arbitrarily chosen

– execution time (uniform): $O(1)$

`choose` x `in` S `s.t.` B;

– returns an element from S that satisfies B

– equivalent to

`choose` x `in` S;

`if` $(\neg B(x)$ ends with failure;

– execution time: $T(B)$

Demo with the new statements

```
choose x1 in { 1 .. 5 };
```

```
$ alki.sh -a choose.alk

    x1 |-> 2

Note that the executed algorithm is nondeterministic.
$ alki.sh -a choose.alk

    x1 |-> 4

Note that the executed algorithm is nondeterministic.
```

Demo with the new statements

```
odd(x) {
  return x % 2 == 1;
}
choose x1 in { 1 .. 5 } s.t. odd(x1);
choose x2 in { 1 .. 5 } s.t. odd(x2);

$ alki.sh -a choosest.alk

    x1 |-> 3
    x2 |-> 1

Note that the executed algorithm is nondeterministic.
$ alki.sh -a choosest.alk

    x1 |-> 5
    x2 |-> 5

Note that the executed algorithm is nondeterministic.
```

Demo with the new statements

```
odd(x) {
  return x % 2 == 1;
}

L = emptyList;
for (i = 0; i < 8; i = i+2)
  L.pushBack(i);
choose x in L s.t. odd(x);
```

```
$ alki.sh -a failure.alk

Error at line 8: Choose can't find any suitable value.
```

Problem Solved by an Non-deterministic Algorithm

- a non-deterministic algorithm A has many executions for the same input
- so, what means that A solves a problem P?
- we say that A solves P if $\forall x \in P$
  $\exists$ an execution that
  is terminating and
  whose final configuration includes $P(x)$

Example: N Queens Problem

Input: a chessboard $n \times n$.
Output: place n chess queens so that no two queens attack each other.

```
attacked(i, j, b) {
  attack = false;
  for (k = 0;  k < i; ++k)
     if ((b[k] == j) || ((b[k]-j) == (k-i)) || ((b[k]-j) == (i-k)))
       attack = true;
  return(attack);
}

nqueens (n) {
   for (i = 0; i < n; ++i) {
     choose j in { 0 .. n-1 } s.t. ! (attacked(i, j, b));
     b[i] = j;
    }
}
```

Execution time: $O(n^2)$

Details on the blackboard.

Example: N Queens Problem, demo

```
n = 4;
b = [-1 | i from [0..n-1]];
```

```
$ alki.sh -a nqueens.alk

Error at line 19: Choose can't find any suitable value. i.e. failure

$ alki.sh -a nqueens.alk
b |-> [2, 0, 3, 1]
i |-> 3
n |-> 4

Note that the executed algorithm is nondeterministic.
```

Example: Subset Sum Problem (SSP)

/*
Input: A set S of integers, M a positive integer.
Output: A subset of $S' \subseteq S$ s.t. $\sum_{x \in S'} x = M$, if any.

*/

```
PM = 0;
/* choose a maximal size for the subset */
choose k in {1 .. S.size()};
/* try to choose at most k-1 elements */
for(i = 0; i < k-1; ++i) {
  choose x in S s.t. PM + x <= M;
  S = S \ singletonSet(x);
  PM = PM + x;
}
/* try to choose the k-th element, if needed */
if (PM != M)
  choose x in S s.t. PM + x == M;
```

TExecution time: $O(n)$, where $n = S.size()$ (we assumed $T(\texttt{S} \setminus \texttt{singletonSet(x)}) = O(1)$).

Details on blackboard.

Example: Subset Sum Problem (SSP), demo

ssd.in

```
S |-> {1, 3, 4, 7, 9} M |-> 14
```

Execution:

```
$ alki.sh -a ssd.alk -i ssd.in
Error at line 18: Choose can't find any suitable value.  failure

$ alki.sh ssd.alk  -i ssd.in
S |-> 3, 7     success
x |-> 1
i |-> 3
k |-> 4
M |-> 14
PM |-> 14
```

Note that the executed algorithm is nondeterministic.

Reduction to Deterministic Algorithms

Let $\sim$ be an equivalence between states. For instance, $\sigma \sim \sigma'$ iff σ and σ' encode the same instance p of a problem P or both encode the answer $P(p)$.

Definition

We say that an algorithm A is equivalent to an algorithm B (w.r.t. $\sim$) iff:

1. $\langle A, \sigma_1 \rangle \Rightarrow^* \langle \cdot, \sigma_1' \rangle$ and $\sigma_1 \sim \sigma_2$ implies the existence of σ_2' s.t. $\langle B, \sigma_2 \rangle \Rightarrow^* \langle \cdot, \sigma_2' \rangle$ și $\sigma_1' \sim \sigma_2'$, and
2. reciprocally, $\langle B, \sigma_2 \rangle \Rightarrow^* \langle \cdot, \sigma_2' \rangle$ și $\sigma_1 \sim \sigma_2$ the existence of σ_1' s.t. $\langle A, \sigma_1 \rangle \Rightarrow^* \langle \cdot, \sigma_1' \rangle$ și $\sigma_1' \sim \sigma_2'$.

Theorem

For any non-deterministic algorithm A there is an equivalent deterministic algorithm B, which has the worst case execution time $T_B(n) = O(2^{T_A(n)})$.

Plan

The Approach

- there a two main steps:
  - first "guesses" a certain structure S
  - then checks if S satisfies the property requested by the question
  - if yes the the execution finishes with `success`, otherwise it finishes with `failure`;
- Extending the language:

  `success;` – signals the successful termination of an execution

  `failure;` – signals the termination of a failing execution

Example

SAT

Instance: A set of n propositional variables and a propositional formula F in conjunctive normal form.

Question: Is F satisfiable?

```
// guess
for (i = 0; i < n; ++i) {
  choose z in {false, true};
  x[i] = z;
}
// check
if (f(x)) success;
else faulure;
```

Example of SAT instance

```
n = 4;

f(x) {
  return (x[0] || x[1]) &&
         (!x[0] || x[3] || x[2]) &&
         (x[2] || !x[3]) &&
         (!x[1] || !x[2] || x[3]);
}
```

Demo

```
$ alki.sh -a sat.alk
failure
x |-> [false, false, false, true]
i |-> 4
z |-> true
n |-> 4

Note that the executed algorithm is nondeterministic.

$ alki.sh -a sat.alk
failure
x |-> [true, true, true, false]
i |-> 4
z |-> false
n |-> 4

Note that the executed algorithm is nondeterministic.
```

Demo

```
$ alki.sh -a sat.alk
failure
x |-> [false, true, true, false]
i |-> 4
z |-> false
n |-> 4

Note that the executed algorithm is nondeterministic.

$ alki.sh -a sat.alk
success
x |-> [true, true, true, true]
i |-> 4
z |-> true
n |-> 4

Note that the executed algorithm is nondeterministic.
```

Example: Subset sum Problem (SSP3)

```
/*
Instance: A set S of intyegers, M a positive integer.
Question: Is there a subset S' ⊆ S s.t. ∑_{x∈S'} x = M?

*/

PM = 0;
choose k in 1 .. S.size();
for(i = 0; i < k; ++i) {
  choose x in S;
  S = S \ singletonSet(x);
  PM = PM + x;
}
if (PM == M) print("success");
else print("failure");
```

Demo (SSD3)

```
$ alki.sh -a ssd3.alk -i ssd.in
failure
S |-> 1, 3, 7
x |-> 4
i |-> 2
k |-> 2
M |-> 14
PM |-> 13
Note that the executed algorithm is nondeterministic.

$ alki.sh -a ssd3.alk -i ssd.in
success
S |-> 1, 9
x |-> 7
i |-> 3
k |-> 3
M |-> 14
PM |-> 14
Note that the executed algorithm is nondeterministic.
```

Plan

Plan

Definition

Definition

A random variable is a function X defined over a set of possible outcomes Ω of a random phenomenon.

Example (only discrete variables)

1. $D2$ (two dice):
   - random phenomenon: rolling two dice
   - $D2$ returns the pair representing the numbers on the two dice
2. $SD2$ (the sum of two dice):
   - random phenomenon: rolling two dice
   - $SD2$ returns the sum of numbers on the two dice
3. CB ("chocolate bar"):
   - random phenomenon: randomly choose a number i in the set $\{1, 2, \ldots, n\}$, $n > 1$
   - CB returns $\max(\frac{i}{n}, \frac{n-i}{n})$, $i = 1, \ldots, n-1$

Probability Distribution

$X : \Omega \to V$

$X(\Omega) = x_0, x_1, x_2, \ldots$

$p_i = \mathrm{Prob}(X = x_i)$ $(= \mathrm{Prob}(\omega \in \Omega \mid X(\omega) = x_i))$[1]

$SD2$:
$SD2(\Omega) = \{2, 3, 4, \ldots, 12\}$
$\mathrm{Prob}(SD2 = 2) = \frac{1}{36}$, $\mathrm{Prob}(SD2 = 3) = \frac{2}{36}$, $\mathrm{Prob}(SD2 = 4) = \frac{3}{36}$, $\ldots$

x_i	2	3	4	5	$\cdots$
p_i	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{4}{36}$	$\frac{4}{36}$	$\cdots$

[1] The exact terminology for $\mathrm{Prob}(X = x_i)$ is "probability mass function". Here we use the more general term of probability distribution (the way the total probability of 1 is distributed over all various possible outcomes).

Extending the Language: "Function" `random()`

`random(n)` - returns an integer $x \in \{0, 1, \ldots, n-1\}$ uniformly chosen (i.e. with the probability $\frac{1}{n}$)

Execution time: $O(1)$

test-random.alk:

```
n = random(5);
print(n);
```

Running test-random.alk

```
$alki.sh -a test-random.alk
4
n |-> 4

The probability for this execution is: 0.2000000000
```

Experiment with random()

Executing the algorithm

```
a = [0, 0, 0, 0];
for (i = 0; i < 10000; ++i) {
  j = random(4);
  a[j] = a[j] + 1;
}
```

we get the following final state:

```
$ alki.sh -a random.alk
a |-> [2508, 2537, 2486, 2469]
i |-> 10000
j |-> 0

The probability for this execution is: 0E-10
```

The probabilities experimentally computed (e.g. $\frac{2508}{10000}$) are close to the theoretical ones; in fact are approximations of them.

Random Variables as Algorithms

```
D2()
{
  d[0] = random(6) + 1;
  d[1] = random(6) + 1;
  return d;
}

SD2()
{
  d1 = random(6) + 1;
  d2 = random(6) + 1;
  return d1 + d2;
}

CB(n)
{
  i = random(n-1) + 1;
  s1 = float(i) / float(n);
  s2 = float(n - i) / float(n);
  if (s1 > s2 ) return s1;
  return s2;
}
```

Random Variable: the Expected Value

Consider only discrete random variable X, whose values are real numbers $x_1, x_2, \ldots$.

$p_i = Pr(X = x_i)$ - probability as X to have the value x_i

Expected Value of X: $E(X) = \sum_i x_i \cdot p_i$

Properties:
$E(X + Y) = E(X) + E(Y)$
$E(X \cdot Y) = E(X) \cdot E(Y)$
(X și Y independente)

Expected value of CB

1. n odd:
   – possible values for CB are $\left\{ \frac{k}{n} \,\middle|\, k = n-1, n-2, \ldots, \frac{n+1}{2} \right\}$, each of them with the probability $\frac{2}{n-1}$
   $E(CB) = \sum_{k=\frac{n+1}{2}}^{n-1} \frac{k}{n}\frac{2}{n-1} = \frac{3n-1}{4n} < \frac{3}{4}$
2. n even:
   – possible values for CB are $\left\{ \frac{k}{n} \,\middle|\, k = n-1, n-2, \ldots, \frac{n}{2}+1 \right\}$, each of them with the probability $\frac{2}{n-1}$, și $\frac{1}{2}$ cu probabilitatea $\frac{1}{n-1}$
   $E(CB) = \sum_{k=\frac{n}{2}+1}^{n-1} \frac{k}{n}\frac{2}{n-1} + \frac{1}{2(n-1)} = \frac{3n-4}{4n-4} < \frac{3}{4}$

Obs. $\frac{3n-4}{4n-4} = \frac{3(n-1)-1}{4(n-1)}$, which implies that the expected values for n and $n-1$ are the same if n is even.

Conclusion: $E(CB) < \frac{3}{4}$

Approximating the Expected Value by SuccessiveExecutions

```
sum = 0.0;
for (j = 0; j < 100; ++j)
  sum = sum + CB(31);

m = sum / float(31);
```

Values obtained using three runs: 0.7341935483870967741943, 0.7416129032258064516137, 0.7606451612903225806456.

Randomized Algorithms: Definition

There are two approaches:

1. Monte Carlo Algorithms
   – may produce incorrect results with some small probability, but whose execution time is deterministic
   – if runned multiple times with independent random choices each time , the failure probability can be made arbitrarily small, at the cost of the running time .
2. Las Vegas Algorithms
   – never produce incorrect results, but whose execution time may vary from one run to another
   – random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input

Plan

Motivation 1/2

Naive algorithm for primality testing:

```
isPrime2(x) {
  if (x < 2) return false;
  for (i = 2; i*i <= x; ++i)
    if (x % i == 0) return false;
  return true;
}
```

Motivation 2/2

A first execution:

```
print(isPrime2(2147483647));

$ time alki.sh -a isPrime.alk
true

real 0m1.465s
user 0m3.385s
sys 0m0.155s
```

A second execution:

```
print(isPrime2(2305843009213693951));

$ time alki.sh -a isPrime.alk
^C
real 50m14.407s
user 49m18.198s
sys 0m20.615s
```

Composite numbers: problem domain 1/3

Legendre Symbol:

$$(a/p) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p}, \\ 1 & \text{if } a \not\equiv 0 \pmod{p} \text{ and there is } x \text{ s.t.} a \equiv x^2 \pmod{p}, \\ -1 & \text{if } a \not\equiv 0 \pmod{p} \text{ and there is NO such an } x. \end{cases}$$

where p is prime;

Jacobi Symbol :

$$(a|n) = (a/p_1)^{\alpha_1}(a/p_2)^{\alpha_2} \cdots (a/p_k)^{\alpha_k},$$

where n is a positive integer and $p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$ its prime factorization.

Composite numbers: problem domain 2/3

1. If $n > 3$ and prime, the $(a|n) = (a/n)$, therefore often is used the same notation $\frac{a}{n}$.
2. If $a \equiv b \pmod{n}$ then $(a|n) = (b|n)$.
3. $(a|n) = \begin{cases} 0 & \text{if } \gcd(a,n) \neq 1, \\ \pm 1 & \text{if } \gcd(a,n) = 1. \end{cases}$
4. $(ab|n) = (a|n)(b|n), \quad \text{so } \left(a^2|n\right) = 1 \text{ or } 0$
5. $(a|mn) = (a|m)(a|n), \quad \text{so } \left(a|n^2\right) = 1 \text{ or } 0$

Composite numbers: problem domain 3/3

5. If m and n are coprime, then
$(m|n)(n|m) = (-1)^{\frac{m-1}{2}\cdot\frac{n-1}{2}} =$
$\begin{cases} 1 & \text{dacă } n \equiv 1 \pmod 4 \text{ or } m \equiv 1 \pmod 4, \\ -1 & \text{dacă } n \equiv m \equiv 3 \pmod 4 \end{cases}$
6. $(-1|n) = (-1)^{\frac{n-1}{2}} = \begin{cases} 1 & \text{dacă } n \equiv 1 \pmod 4, \\ -1 & \text{dacă } n \equiv 3 \pmod 4, \end{cases}$
7. $(2|n) = (-1)^{\frac{n^2-1}{8}} = \begin{cases} 1 & \text{dacă } n \equiv 1, 7 \pmod 8, \\ -1 & \text{dacă } n \equiv 3, 5 \pmod 8. \end{cases}$

Other Properties (inferred)

$$(a \cdot 2|n) = (a|n) \cdot (2|n) = \begin{cases} -(a|n) & \text{if } n \equiv 3, 5 \pmod 8 \\ (n|a) & \text{otherwise} \end{cases}$$

$$(a|n) = \begin{cases} -(n|a) & \text{if } a \neq 0 \neq n \text{ și } n \equiv a \equiv 3 \pmod 4 \\ (n|a) & \text{otherwise} \end{cases}$$

From the problem domain to the algorithm

```
jacobi(a, n)
{
  j = 1;
  while (a != 0) {
    while (a % 2 == 0) {  // a is even
      a = a / 2;
      if (n % 8 == 3 || n % 8 == 5) j = 0-j;
    }
    swap(a, n);
    if (a % 4 == 3 && n % 4 == 3) j = 0-j;
    a = a % n;
  }
  if (n == 1) return j;
  else return 0;
}
```

Solovay-Strassen Algorithm: descriptive

Input: a odd positive integer n,
Output: "composite" if n is composite, "maybe prime" otherwise

1. randomly choose a in $[2, n-1]$
2. $x = (a|n)$
3. if $x == 0$ or $a^{(n-1)/2} \not\equiv x \pmod{n}$ then returns "composite"
4. otherwise returns "maybe prime"

Solovay-Strassen Algorithm in Alk

```
isComp(n)
{
  a = random(n-3) + 2;
  if (gcd(a, n) != 1) return "composite";
  x = jacobi(a, n);
  if (x < 0) x = x + n;
  if (x != power(a, (n-1)/2, n)) return "composite";
  return "maybe prime";
}
```

Solovay-Strassen Algorithm: demo

```
m1[0] = 2147483647*457241;
m1[1] = isComp(m1[0]);

m2[0] = 2147483647;
m2[1] = isComp(m2[0]);
```

Rezultat:

```
$ time alki.sh -a compos.alk
m1 |-> [981917570237927, composite]
m2 |-> [2147483647, may be prime]

The probability for this execution is: 0E-10

real 0m0.570s
user 0m1.085s
sys 0m0.101s
```

Solovay-Strassen Algorithm as a Prime Test with a Certain Probability

Input: a positive odd integer n,
a positive integer k representing the accuracy
Output: "composite" if n is composite, "probably prime" otherwise

```
isProbPrime(n, k) {
  while (k > 0 && isComp(n) != "composite")
    --k;
  if (k == 0) return "probably prime";
  return "composite";
}
```

Failure probability is 2^{-k}.
(A proof can be found in Richard M. Karp. An introduction to randomized algorithms. Discrete Applied Mathematics 34 (1991) 165-201.)

Solovay-Strassen Algorithm as a Prime Test: demo

```
m4[0] = 2305843009213693951;
m4[1] = isProbPrime(m4[0], 100);
```

Result:

```
$ time alki.sh -a compos.alk
m4 |-> [2305843009213693951, probably prime]

real 0m1.192s
user 0m3.043s
sys 0m0.161s
```

The test with the naive algorithm:

```
print(isPrime2(2305843009213693951));
$ time alki.sh -a isPrime.alk
^C
real 50m14.407s
user 49m18.198s
sys 0m20.615s
```

Solovay-Strassen Algorithm as a Prime Test: demo

```
m3[0] = 170141183460469231731687303715884105727;
m3[1] = isProbPrime(m3[0], 100);
```

Result:

```
$ time alki.sh -a compos.alk
m3 |-> [170141183460469231731687303715884105727, probably prime]

real 0m1.851s
user 0m4.856s
sys 0m0.262s
```

The test with the naive algorithm: an optimistic estimation for terminating is the end of the semester ...

An Efficient Algorithm for the Power Function Modulo

From the problem domain:

$$a^n \pmod p = \begin{cases} 1 & \text{dacă } n = 0, \\ a \pmod p & \text{dacă } n = 1, \\ (a^{\frac{n}{2}} \pmod p) * (a^{\frac{n}{2}} \pmod p) \pmod p & \text{dacă } n \% 2 == 0, \\ (a * a^{n-1}) \pmod p & \text{dacă } n \% 2 == 1, \end{cases}$$

to the algorithm:

```
power(a, n, p) {
  x = 1;
  while (n > 0)
    if (n % 2 == 0) {
      a = (a * a) % p ;
      n = n / 2;
    }
    else {
      x = (a * x) % p;
      n = n - 1;
    }
  return x;
}
```

Exercises

1. Find the execution time for `power(a, n, p)`.
2. Find the execution time for `isComp(n)`.
3. Find the execution time for `isProbPrime(n, k)`.

Plan

1. Recap
2. Non-deterministic Algorithms, Generally
3. Non-deterministic Algorithms for Decision Problems
4. **Randomized Algorithms**
   - Random Variable
   - Example of Monte Carlo Algorithm: Primality Test
   - **Las Vegas Algorithms**
   - Example of Las Vegas Algorithm: k-median

The Expected time for the Randomized Algorithms 1/2

– never produce incorrect results, but whose execution time may vary from one run to another
– random choices made within the algorithm are used to establish an expected running time for the algorithm that is, essentially, independent of the input

Notations:

$prob_{A,x}(C)$ = the probability that the algorithm A to execute C for the input x

$time_{A,x}(C)$ = the time that A to execute C for the input x (a bit different from the deterministic case)

The Expected time for the Randomized Algorithms 2/2

the expected time of A for the input x is
$exp\text{-}time(A, x) = E[time_{A,x}] = \sum_C prob_{A,x}(C) \cdot time_{A,x}(C)$.

$time_{A,x}$ is a random variable.

the expected time of A for the worst case is
$exp\text{-}time(A, n) = max\{exp\text{-}time(A, x) \mid size(x) = n\}$

If A is understood from the context, we write only $exp\text{-}time(n)$ ($exp\text{-}time(x)$) instead of $exp\text{-}time(A, n)$ (resp. $exp\text{-}time(A, x)$).

Plan

k-median: the problem

Definition

Let S be a list with n elements from a totally ordered set. The k-median is the k-th element from the sorted list S.

Assume that S is represented by an array.

Consider the next problem:

Input	an array $(a[i] \mid 0 \leq i < n)$ and a number $k \in \{0, 1, \ldots, n-1\}$,
Output	k-median

k-median: description of the algorihtm

Select a pivote x in $a[0..n-1]$.
Partition the array a around x: the elements of the array are permuted such that $a[j] = x$ and

$$(\forall i)(i < j \implies a[i] \leq x) \wedge (i > j \implies a[i] \geq x)$$

which is equivalent to:

$$(\forall i)(i < j \implies a[i] \leq a[j]) \wedge (i > j \implies a[i] \geq a[j])$$

1. $j = k \implies$ the problem is solved
2. $j < k \implies$ search k in `a[j+1..n-1]`
3. $j > k \implies$ search k in `a[0..j-1]`

Partitioning Lomuto

```
partition(out a, p, q)
{
  pivot = a[q];
  i = p - 1;
  for (j = p; j < q; ++j)
    if (a[j] < pivot) {
        i = i + 1;
        swap(a, i , j);
    }
  swap(a, i+1, q);
  return i + 1;
}

swap(out a, i, j) {
  if (i != j) {
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
}
```

Analysis of the Lomuto Partitioning

Correctness:
The for invariant:
$i < j \wedge (\forall \ell)(p \leq \ell \leq i \implies a[\ell] \leq pivot) \wedge (i < \ell < j \implies a[i] > pivot)$
After for:
the invariant and $j = q$, which implies $a[i+1] > pivot$
After the last `swap`:
$a[i+1] = pivot$ și
$(\forall \ell)(p \leq \ell \leq i \implies a[\ell] \leq pivot) \wedge (i < \ell < q \implies a[i] > pivot)$.

The number of comparisons: $q - p$

Randomized Partioning

The pivot is randomly chosen from $a[p..q]$:

```
randPartition(out a, p, q) {
  if (p < q) {
    i = p + random(q - p);
    swap(a, i, q);
    return partition(a, p, q);
  }
}
```

k-median: Las Vegas Algorithm

```
randSelectRec(out a, p, q, k)
{
  j = randPartition(a, p, q);
  if (j == k) return a[j];
  if (j < k) return randSelectRec(a, j+1, q, k);
  return randSelectRec(a, p, j-1, k);
}

randSelect(out a, k)
{
  return randSelectRec(a, 0, a.size()-1, k);
}
```

randSelect: analysis 1/3

exp-time(n, k) - tthe expected time to find the k-median in an array of length n

exp-time$(n) = \max_k$ *exp-time*(n, k)

Since we are interested in the worst case analysis, we assume that the recursive call chooses always the longest subarray.

Recall that $E(CB) < \frac{3}{4}$

It follows that the expected length of the longest subarray is at most $\frac{3}{4}n$.

randSelect: analysis 2/3

Lemă

The expected length of the array after i call is at most $\left(\frac{3}{4}\right)^i n$.

Proof

L_i the random variable that returns the length of the array after i calls.
P_j the random variable that returns the fraction of the elements preserved at the level j
X_j the random variable that returns the length of the longest subarray at the level j

We have: $L_i = n\prod_{j=1}^{i} P_j$, $P_j = \dfrac{X_j}{n}$, $E(P_j) = E\left(\dfrac{X_j}{n}\right) = \dfrac{E(X_j)}{n} \leq \dfrac{\frac{3}{4}n}{n} = \dfrac{3}{4}$,

$P_1, \ldots, P_n$ are independent,

$E(L_i) = E(n\prod_{j=1}^{i} P_j) = n\prod_{j=1}^{i} E(P_j) \leq \left(\dfrac{3}{4}\right)^i n$

Now the lemma is proved.

randSelect: analysis 3/3

At the levell i, the number of operations is liniar, let say $\leq aX_i + b$ (recal that X_i is the length of the longest subarray).
Let $r \leq n$ be the number of recursive calls. The the expected time is:

$$\begin{aligned}
\textit{exp-time}(n) &= E\left(\sum_{i=1}^{r}(aX_i + b)\right) \\
&= \sum_{i=1}^{r} E(aX_i + b) \\
&\leq \sum_{i=1}^{n}(aE(X_i) + b) \\
&\leq \sum_{i=1}^{n}\left(a\left(\frac{3}{4}\right)^i n + b\right) \\
&= an\sum_{i=1}^{n}\left(\frac{3}{4}\right)^i + bn \\
&\leq 3an + bn \\
&= O(n)
\end{aligned}$$