Randomized Algorithms

CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material based on Material based on *Algorithm Design* by Jon Kleinberg and Éva Tardos (Chapter 13) and *An Introduction to the Analysis of Algorithms* by Michael Soltys (Chapter 6)

Randomized Algorithms

- It is very interesting that we can design procedures which, when confronted with a profusion of choices, instead of laboriously examining all the possible answers to those choices, they ip a coin to decide which way to go, and still "tend to" obtain the right output.
- Obviously we save time when we resort to randomness, but what is very surprising is that the output of such procedures can be meaningful.
- That is, there are problems that computationally appear very difficult to solve, but when allowed the use of randomness it is possible to design procedures that solve those hard problems in a satisfactory manner: the output of the procedure is correct with a small probability of error.
- Thus, many experts believe that the definition of "feasibly computable" ought to be computable in polynomial time with randomness, rather than just in polynomial time.

Diffie-Hellman Schem

- First proposed by James Ellis in 1970, but classified until 1997.
- Reinvented by W. Diffie and M. Hellman in 1976.
- It is based on the *Discrete Logarithm Problem*.

Definition (Discrete Logarithm Problem)

Find k such that

$$n = g^k \mod p$$

for a given natural numbers n, g and a prime number p.

- The Discrete Logarithm Problem is infeasible for big p.

Algorithm (Diffie-Hellman Protocol)

- *Shared Knowledge*: p *and* g, *where* $g \neq 0, 1, p-1$.
- *Each user chooses a private key* k_a *and computes a public key*
$$K_a = g^{k_a} \mod p.$$
- *If A and B want to communicate, they encipher the other's public key using they own public key using the formulas:*
$$S_{A,B} = K_B^{k_A} \mod p \text{ (used by } A\text{), and}$$
$$S_{B,A} = K_A^{k_B} \mod p \text{ (used by } B\text{).}$$
- *The protocol is based on the following theorem:*

Theorem

$$S_{A,B} = S_{B,A}$$

- *The key* $S_{A,B} = S_{B,A}$ *is used for communication between A and B.*

RSA Protocol

- Invented by R. **R**ivest, A. **S**hamir and L. **A**dleman in 1978.
- It is based on the properties of the totient function $\Phi(n)$.

Definition

A number k is relatively prime to a number n if k has no factors in common with n.

Definition

The totient function $\Phi(n)$ is the number of positive integers less than n and *relatively prime* to n.

Example

- $\Phi(10) = 4$, as $1, 3, 7, 9$ are relatively prime to 10.
- $\Phi(21) = 12$, as $1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20$ are relatively prime to 21.

Theorem

If p and q are two distinct primes, then $\Phi(pq) = (p-1)(q-1)$.

Algorithm (RSA Protocol)

- *Choose two large prime numbers p and q.*
- *Compute $n = pq$. Then $\Phi(n) = (p-1)(q-1)$.*
- *Choose $e < n$ such that e is* relatively prime *to $\Phi(n)$.*
- *Compute d such that $ed \mod \Phi(n) = 1$.*
- *PUBLIC KEY: (e, n)*
- *PRIVITE KEY: d*
- *ENCIPHER: $c = m^e \mod n$ (uses PUBLIC KEY (e, n))*
- *DECIPHER: $m = c^d \mod n$ (uses PRIVATE KEY d)*

- Actual RSA primes p and q should be at least 512 bits long, giving a modulus, i.e. $n = pq$, of at least 1024 bits.

Prime Numbers

- All modern cryptography algorithms require **large random primes** (in practice about 1,000 bit long); a single prime Diffie-Hellman, a pair of primes for RSA, and similarly for other protocols. How to go about it?
- **How to get large random primes quickly?**
- One way to determine whether a number p is prime, is to try all possible numbers $n < \sqrt{p}$, and check if any are divisors.
- Obviously, this brute force procedure has exponential time complexity in the length of $\sqrt{p}$, and so it has a prohibitive time cost.
- Although a polytime (deterministic) algorithm for primality is known from 2004, the randomized algorithm for primality testing that we will present is simpler and more efficient, and therefore still used in practice.

Natural Numbers

- Define $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- For all $x, y \in \mathbb{N}$ we say the x *divides* y, and write $x|y$ if $y = qx$ for some $q \in \mathbb{N}$.
- If $x|y$ we say that x is *divisor* (also *factor*) of y.
- For all $x, y \in \mathbb{N}$, the *greatest common divider* of x and y, $gcd(x, y)$ is defined as
$$gcd(x, y) = \max\{q \mid q|x \wedge q|y\}.$$
For example $gcd(63, 147) = 7$, as $63 = 3 \cdot 3 \cdot 7$ and $147 = 3 \cdot 7 \cdot 7$.
- Let $m \geq 1$. We say that a and b are *congruent* (or *equal*) *modulo* m, and write
$$a \equiv b \mod m.$$

Proposition

If $m \geq 1$, then $a \cdot b \equiv 1 \mod m$ for some b if and only if

$$gcd(a, m) = 1.$$

Fermat's Little Theorem and Fermat Test

- Let $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$ - the set of integers *modulo* m.
- Let $\mathbb{Z}_m^* = \{a \in \mathbb{Z}_m \mid gcd(a, m) = 1\} \subseteq \mathbb{Z}_m$.

Theorem (Fermat's Little Theorem)

Let p be a prime number and $gcd(a, p) = 1$. Then

$$a^{p-1} \equiv 1 \mod p.$$

- We say that p passes the Fermat test at a iff $a^{p-1} \equiv 1 \mod p$.
- All primes pass the Fermat test for all $a \in \mathbb{Z}_p \setminus \{0\}$.
- Unfortunately, there are also composite numbers n that pass the Fermat test for all $a \in \mathbb{Z}_n$.
- They are called *Carmichael numbers*, for example, $561 = 3 \cdot 11 \cdot 17$, $1005 = 5 \cdot 11 \cdot 17$, $1720 = 7 \cdot 13 \cdot 19$, etc.

- We say that p passes the Fermat test at a iff $a^{p-1} \equiv 1 \mod p$.

Lemma

1. *If p is a composite non-Carmichael number, then it passes* **at most half** *of the tests in $\mathbb{Z}_p^*$.*
2. *If $gcd(a, p) \neq 1$ then $a^{p-1} \not\equiv 1 \mod p$.*

- A number is pseudoprime if it is either prime or Carmichael number.

Algorithm for pseudoprimes:

- On input p, check whether $a^{p-1} \equiv 1 \mod p$ for some random $a \in \mathbb{Z}_p \setminus \{0\}$.
- If p fails this test (i.e., $a^{p-1} \not\equiv 1 \mod p$), then p is composite for sure (test $gcd(a, p) \neq 1$ first).
- If p passes the test, then p is **probably** pseudoprime.
- From the above lemma (2), it follows that the probability of error in the case of passing the test is $\leq \frac{1}{2}$.

Rabin-Miller algorithm

- The Rabin-Miller algorithm extends the pseudoprimeness test to deal with Carmichael numbers.

Rabin-Miller Algorithm

1: If $n = 2$, accept; if n is even and $n > 2$, reject.
2: Choose at random a positive a in $\mathbb{Z}_n$.
3: **if** $a^{(n-1)} \not\equiv 1 \pmod{n}$ **then**
4: reject
5: **else**
6: Find s, h such that s is odd and $n - 1 = s2^h$
7: Compute the sequence $a^{s \cdot 2^0}, a^{s \cdot 2^1}, a^{s \cdot 2^2}, \ldots, a^{s \cdot 2^h} \pmod{n}$
8: **if** all elements in the sequence are 1 **then**
9: accept
10: **else if** the last element different from 1 is -1 **then**
11: accept
12: **else**
13: reject
14: **end if**
15: **end if**

1: If $n = 2$, accept; if n is even and $n > 2$, reject.
2: Choose at random a positive a in $\mathbb{Z}_n$.
3: **if** $a^{(n-1)} \not\equiv 1 \pmod{n}$ **then**
4: reject
5: **else**
6: Find s, h such that s is odd and $n - 1 = s2^h$
7: Compute the sequence $a^{s\cdot 2^0}, a^{s\cdot 2^1}, a^{s\cdot 2^2}, \ldots, a^{s\cdot 2^h} \pmod{n}$
8: **if** all elements in the sequence are 1 **then**
9: accept
10: **else if** the last element different from 1 is -1 **then**
11: accept
12: **else**
13: reject
14: **end if**
15: **end if**

- Note that this is a polytime (randomized) algorithm: computing powers (mod n) can be done efficiently with *repeated squaring*.
- If $n - 1 = c_r \ldots c_1 c_0 \mod b$, then compute:

$$a_0 = a, a_1 = a_0^2, a_2 = a_1^2, \ldots, a_r = a_{r-1}^2 \mod n.$$

- Hence $a^{n-1} = a_0^{c_0} a_1^{c_1} \cdots a_r^{c_r} \mod n$.
- Thus obtaining the powers in lines 6 and 7 is not a problem.

Rabin-Miller Algorithm

Theorem

If n is a prime then the Rabin-Miller algorithm accepts it; if n is composite, then the algorithm rejects it with probability $\geq \frac{1}{2}$.

- Note that by running the algorithm k times on independently chosen a, we can make sure that it rejects a composite with probability $\geq 1 - \frac{1}{2^k}$ (it will always accept a prime with probability 1).
- Thus, for $k = 100$ the probability of error, i.e., of a *false positive*, is negligible.

Generating Primes

- Let $\pi(x)$ be the *prime-counting function* that gives the number of primes less than or equal to x, for any real number x.
- For example, $\pi(10.3) = 4$ because there are four prime numbers (2, 3, 5 and 7) less than or equal to 10.1.
- Let π_n denotes the n^{th} prime number, for example $p_4 = 7$.

Theorem (Prime Number Theorem)

$$\lim_{x \to \infty} \frac{\pi(x)}{x/\log x} = 1$$

- Using asymptotic notation this result can be restated as

$$\pi(x) \sim \frac{x}{\log x} \text{ or } p_n \sim n \log n.$$

- This means that there are $2^n/n$ primes among n-bit integers, roughly 1 in n, and these primes are fairly uniformly distributed.
- **So we pick an integer at random, in a given range, and apply the Rabin-Miller algorithm to it.**

Basic Probability

- A (finite) probability space (S, p) consists of a (finite) set S and a function $p : S \to \mathit{Reals}$ satisfying:
  - $0 \leq p(x) \leq 1$ for all $x \in S$,
  - $\sum_{x \in S} p(x) = 1$.
- An *event* A is a subset of S and its probability is given by:

$$p(A) = \sum_{x \in A} p(x).$$

Facts

1. $p(\emptyset) = 0,\ p(S) = 1$
2. $A \cap B = \emptyset \implies p(A \cup B) = p(A) + p(B)$
3. $p(A \cup B) = p(A) + p(B) - p(A \cup B)$
4. $p(\bigcup_{j \in I} A_i) \leq \sum_{i \in I} p(A_i)$

Basic Probability

- A and B are *mutually exclusive* if $p(A \cap B) = \emptyset$.
- A and B are *independent* if $p(A \cap B) = p(A)p(B)$.
- *Conditional probability*: A and B are events and $p(B) > 0$
  The *probability of A given B* is defined as:

$$p(A|B) = \frac{p(A \cap B)}{p(B)}$$

Random Variables

- A function $X : S \to \mathit{Reals}$ is called a *random variable*.
- The *expectation* (*expected value*) of X is:

$$E(X) = \sum_{x \in S} p(x)X(x)$$

- For $x \in \mathit{Reals}$,

$$p(\{x \in S \mid X(x) = r\}) = p(X^{-1}(r))$$

is often written as $p(X = r)$ and interpreted as "the probability that $X = r$".

Fact

$$E(X) = \sum_{r \in X(S)} r \cdot p(X = r)$$

Fact

Expected value is linear *i.e.*

1. $E(X+Y) = E(X) + E(Y)$
2. $E(cX) = cE(X)$
3. $E(\sum_i c_i X_i) = \sum_i c_i E(X_i)$

- Waiting for first success in independent trial:
$$P(X = j) = (1-p)^{j-1}p$$
- $E(X) = \sum_{j=1}^{\infty} jP(X=j) = \sum_{j=1}^{\infty} j(1-p)^{j-1}p =$
$$\frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^j = \frac{p}{1-p}\cdot\frac{1-p}{p^2} = \frac{1}{p}$$
since $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$.
- Expected number of trials to first success is $\frac{1}{p}$.

Finding the Median

- Let $S = \{a_a, a_2, \ldots, a_m\}$ be a set of numbers.
- The **median** of S is equal to the k^{th} largest element in S, where:

$$k = \begin{cases} \frac{n+1}{2} & n \text{ is odd} \\ \frac{n}{2} & n \text{ is even} \end{cases}$$

  Assume for simplicity that all elements of S are different.
- Obvious solution: Sort first
  Complexity: $O(n \log n)$
- Cab we do better?
- We will provide a randomized algorithm that can do it in $O(n)$ time.

Generic Algorithm Based on Splitters

Select(S,k):
Choose a splitter $a_i \in S$
For each element a_j of S
    Put a_j in S^- if $a_j < a_i$
    Put a_j in S^+ if $a_j > a_i$
Endfor
If $|S^-| = k-1$ then
    The splitter a_i was in fact the desired answer
Else if $|S^-| \geq k$ then
    The k^{th} largest element lies in S^-
    Recursively call $Select(S^-, k)$
Else suppose $|S^-| = \ell < k-1$
    The k^{th} largest element lies in S^+
    Recursively call $Select(S^+, k-1-\ell)$
Endif

```
Select(S,k):
Choose a splitter a_i ∈ S
For each element a_j of S                    } Θ(n)
    Put a_j in S^- if a_j < a_i
    Put a_j in S^+ if a_j > a_i
Endfor
If |S^-| = k − 1 then
    The splitter a_i was in fact the desired answer
Else if |S^-| ≥ k then
    The k^th largest element lies in S^-
    Recursively call Select(S^-, k)
Else suppose |S^-| = ℓ < k − 1
    The k^th largest element lies in S^+
    Recursively call Select(S^+, k − 1 − ℓ)
Endif
```

- The algorithm is always called recursively on a strictly smaller set, so it must terminate.
- $|S| = 1 \implies k = 1$.

Generic Algorithm Based on Splitters

Conclusion

Regardless of how the splitter is chosen, the algorithm returns the k^{th} largest element of S.

- CHOOSING A GOOD SPLITTER
- LUCKY CASE: We always choose the median as the splitter:

$$T(n) \leq T(n/2) + dn$$

Master theorem: $T(n) = aT(n/b) = f(n)$,
$a = 1, b = 2, \log_2 1 = 0$
$f(n) = f \cdot n = \Omega(n^{0+\varepsilon})$ for any $0 < \varepsilon \leq 1$,
$a \cdot f(n/b) = f(n/2) = \frac{1}{2}dn \leq \frac{1}{2}f(n) = cf(c)$,
where $c = \frac{1}{2} < 1$.
Thus, by (3) of Master Theorem: $T(n) = \Theta(f(n)) = \Theta(n)$.

- WEIRD, as we just compute median!

"WELL-CENTERED" CASE

- We are able to choose splitter a_i such that always at least $\varepsilon \cdot n$ elements both larger and smaller than a_i for any fixed constant $\varepsilon > 0$.

$$T(n) \leq T((1-\varepsilon)n) + dn$$

$$T(n) = dn + (1-\varepsilon)dn + (1-\varepsilon)^2 dn + \ldots =$$
$$\underbrace{[1 + (1-\varepsilon) + (1-\varepsilon)^2 + \ldots]}_{\frac{1}{\varepsilon} \text{ since } \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}, |x|<1} dn = \frac{1}{\varepsilon} dn = \Theta(n)$$

- "Lucky case" is when $\varepsilon = \frac{1}{2}$:
- Very "OF CENTER" splitter:

$$T(n) \leq T(n-1) + dn$$

$$T(n) = dn + d(n-1) + d(n-2) + \ldots = \frac{dn(n+1)}{2} = \Theta(n^2).$$

- Choose a splitter $a_i \in S$ uniformly at random.
- Intuitively random case should be close to well centered case.

Random Case Analysis

- The algorithm is in *phase j* if the set under consideration has a size s such that
$$n(\tfrac{3}{4})^{j+1} < s \leq n(\tfrac{3}{4})^{j}$$
- For a set A, a element $a \in A$ is *central*, if
$$\{x \in A \mid x < a\}| > \tfrac{1}{4}|A|, \text{ and}$$
$$\{x \in A \mid x < a\}| > \tfrac{1}{4}|A|$$
- Splitter is *central* $\Longrightarrow$ at least $\frac{1}{4}$ of elements is thrown away, i.e. the set shrinks by $\frac{3}{4}$ factor.
- Probability of being central is $\frac{1}{2}$.
- Expected number of trials to find a central element is 2.
- Expected number of iterations in phase j, for any j is 2.

- X - random variable modeling the number of steps taken by the algorithm:

$$X = X_0 + X_1 + X_2 + \ldots$$

where X_j is the expected number of steps in phase j.

- In phase j, the set has size at most $n(\frac{3}{4})^j$, so the number of steps for one iteration is at most $c \cdot n \cdot (\frac{3}{4})^j$, some constant c.
- Expected number of iterations in phase j is 2, so

$$E(X_j) \leq 2cn(\tfrac{3}{4})^j.$$

- Hence

$$E(X) = \sum_j E(X_j) \leq \sum_j 2cn(\frac{3}{4})^j = 2cn \sum_j (\frac{3}{4})^j <$$

$$2cn \sum_{j=0}^{\infty} (\frac{3}{4})^j = 2cn \frac{1}{1 - \frac{3}{4}} = 8c \cdot n$$

as $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ for $|x| < 1$.

- **The expected time of** $Select(n, k)$ **is** $O(n)$**.**

Random Case Analysis

- **The expected time of $Select(n, k)$ is $O(n)$.**
- How to implement randomness?
- If the distribution is uniform, probability of value x on any position of the list of n elements is $\frac{1}{n}$.
- **Solution:**
  For example: Choose the last (first) element on the list as a splitter.

Quick Sort (C. A. R. Hoare 1962)

```
Quicksort(S):
If |S| ≤ 3 then
   Sort S
   Output the sorted list
Else
   Choose a splitter a_i ∈ S uniformly at random
   For each element a_j of S

      Put a_j in S^- if a_j < a_i
      Put a_j in S^+ if a_j > a_i
   Endfor
   Recursively call Quicksort(S^-) and Quicksort(S^+)
   Output the sorted set S^-, then a_i, then the sorted set S^+
Endif
```

- For this algorithm, splitter is called *pivot*.

Quick Sort: Analysis

- Lucky Case: Splitters are medians. Then
$$T(n) \leq 2T(n/2) + cn$$
By Master Theorem (case 2), $O(n \log n)$.
- colorblue Worst Case: Splitters split 1 and $n-1$
$$T(n) \leq T(n-1) + cn = O(n^2)$$
- Intuition tells us that random case is close to lucky case.
- We assume uniform distribution.
- We assume that all elements are different. this assumption can be dropped, but then proofs need to be modified.

Quick Sort: intuition

- Consider the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced.

$$T(n) = T(9n/10) + T(n/10) + cn$$

- It can be proved, either by expansion, or by analyzing recursion tree that:

$$T(n) = T(9n/10) + T(n/10) + cn = O(n \log n)$$

- Even a 99-to-1 split yields an $O(n \log n)$ running time (big constant).

Quick Sort: Average Case Complexity

- Average case complexity for Quick Sort is considered difficult and in most textbooks, including Kleinberg-Tardos, is not provided.
- The proof in this lecture notes is probably the easier known in literature, it uses a trick that often makes proofs easier; it generalizes a discrete case to continuous case and replaces sums with integrals.

Quick Sort: Average Case Complexity

- Suppose $|S^-| = j - 1$ and $|S^+| = n - j$, i.e.
$$T(n) \leq T(j-1) + T(n-j) + cn.$$
- Since j is random with $\frac{1}{n}$ probability, we have
$$T_{avg}(n) \leq \frac{1}{n} \sum_{j=1}^{n} (T_{avg}(j-1) + T_{avg}(n-j)) + cn =$$
$$cn + \frac{2}{n} \sum_{j=0}^{n-1} T_{avg}(j),\ n \geq 2$$
- Assume $T_{avg}(0) \leq b$, $T_{avg}(1) \leq b$ and define $d = 2(b + c)$.
- We will show **by induction** that, for $n \geq 2$
$$T_{avg}(n) \leq dn \log_e n = O(n \log n),$$
where $e = 2.71838...$

Quick Sort: Average Case Complexity

- Base: $n = 2$
  $T_{avg}(2) \leq 2c + \frac{2}{2}(T(0) + T(1)) \leq 2c + b + b = 2(b + c) =$
  $d \leq d \cdot \underbrace{2 \log_e 2}_{\geq 1}$
- We will use the 2^{nd} induction scheme.
- Induction Assumption: for $1 \leq n$

$$T_{avg}(m) \leq dn \log_e n.$$

- We need to show that

$$T_{avg}(n+1) \leq d(n+1) \log_e (n+1).$$

- Recall that $\int x \ln x \, dx = x^2(\frac{\ln x}{2} - \frac{1}{4})$.
- Let $m = n + 1$. Recall $T_{avg}(m) = cn + \frac{2}{m}\sum_{j=0}^{m-1} T_{avg}(j)$.

$$T_{avg}(m) \leq cm + \frac{2}{m}(\underbrace{T(0)}_{\leq b} + \underbrace{T(1)}_{\leq b}) + \frac{2}{m}\sum_{j=2}^{m-1} \underbrace{T_{avg}(j)}_{\leq dj \log_e j} \leq$$

$$cm + \frac{4b}{m} + \frac{2d}{m}\sum_{j=2}^{m-1} j \log_e j \leq cm + \frac{4b}{m} + \frac{2d}{m}\int_0^m x \log_e x \, dx =$$

$$cm + \frac{4b}{m} + \frac{2d}{m}\left[\frac{m^2 \log_e m}{2} - \frac{m^2}{4}\right] = cm + \frac{4b}{m} + dm \log_e m - \frac{dm}{2} =$$

$$cm + \frac{4b}{m} - \frac{2bm}{2} - \frac{2cm}{2} + dm \log_e m =$$

$$dm \log_e m + \underbrace{\left(\frac{4b}{m} - bm\right)}_{\leq 0 \text{ if } m \geq 2} \leq dm \log_e m$$

- Hence: $T_{avg}(n) = O(n \log_e n) = O(n \log n)$.

Quick Sort

- **Quick Sort** is considered the fastest sorting when input is random. WHY?
- Because it allows a very efficient implementation which is superfast for half of cases!
- Less abstract formulation:

Input: unsorted array $A[1..n]$
Output: sorted array $A[1..n]$

We want to use **only one** array.

- **QUICK SORT Algorithm:**

QuickSort($A[1..n]$)
  $v \leftarrow PIVOT(A[1..n])$ $\leftarrow$ it chooses the splitter
  $k \leftarrow PARTITION(v, A[1..n])$ $\leftarrow$ it maintains S^- and S^+ in A.
QuickSort($A[1..k-1]$)
QuickSort($A[k..n]$)

Pivot

How to implement: $v \leftarrow PIVOT(A[1..n])$?

- $v \leftarrow Random(A[1..n])$ $\leftarrow$ not feasible in full
- $v \leftarrow A[n]$ $\leftarrow$ as good as random if input has uniform distribution
- $v \leftarrow$ largest from the first two different values of $A[i]$, starting from $A[1]$ $\leftarrow$ original choice of C. A. R. Hoare, slightly better if distribution not entirely uniform
- $v \leftarrow$ a formula that depends on input distribution, if known.

Pivot

- Consider the case $v = A[n]$ (*simplest*)
- We may now write:

QuickSort(A, p, r)
IF $p < r$ THEN
  $q \leftarrow PARTITION(A, p, r)$ $\leftarrow$ must guarantee $p = r$ at some point

  QuickSort$(A, p, q-1)$
  QuickSort$(A, q+1, n)$
ENDIF

- It is use as QuickSort$(A, 1, n)$ for $A[1.n]$.

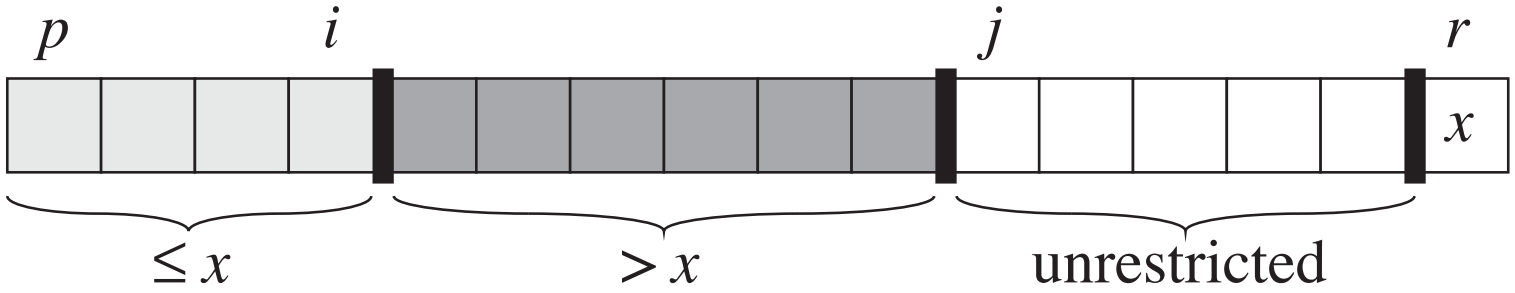
Partition

- **Partition is tricky and important.**

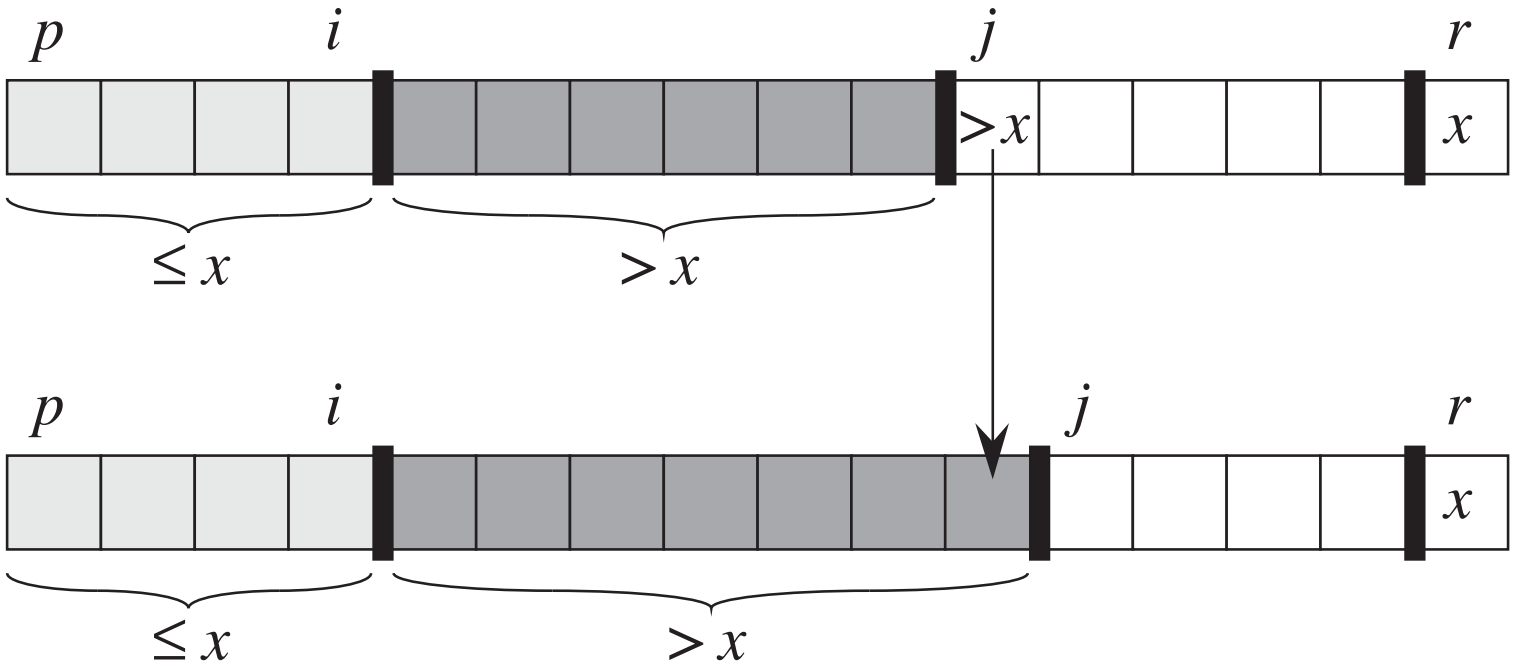
$PARTITION(A, p, r)$
$x \leftarrow A[r]$
$i \leftarrow p - 1$
FOR $j = p$ TO $r - 1$ DO
  IF $A[j] \leq x$ THEN
    BEGIN
      $i \leftarrow i + 1$
      $swap(A[i], A[j])$
    END
  ENDIF
ENDFOR
$swap(A[i + 1], [A[r])$
$return(i + 1)$

nothing if $A[j] > x$.
It happens in about $\frac{1}{2}$ cases!
Very efficient

- The procedure *Partition* maintains 4 regions on a subarray $A[p..r]$

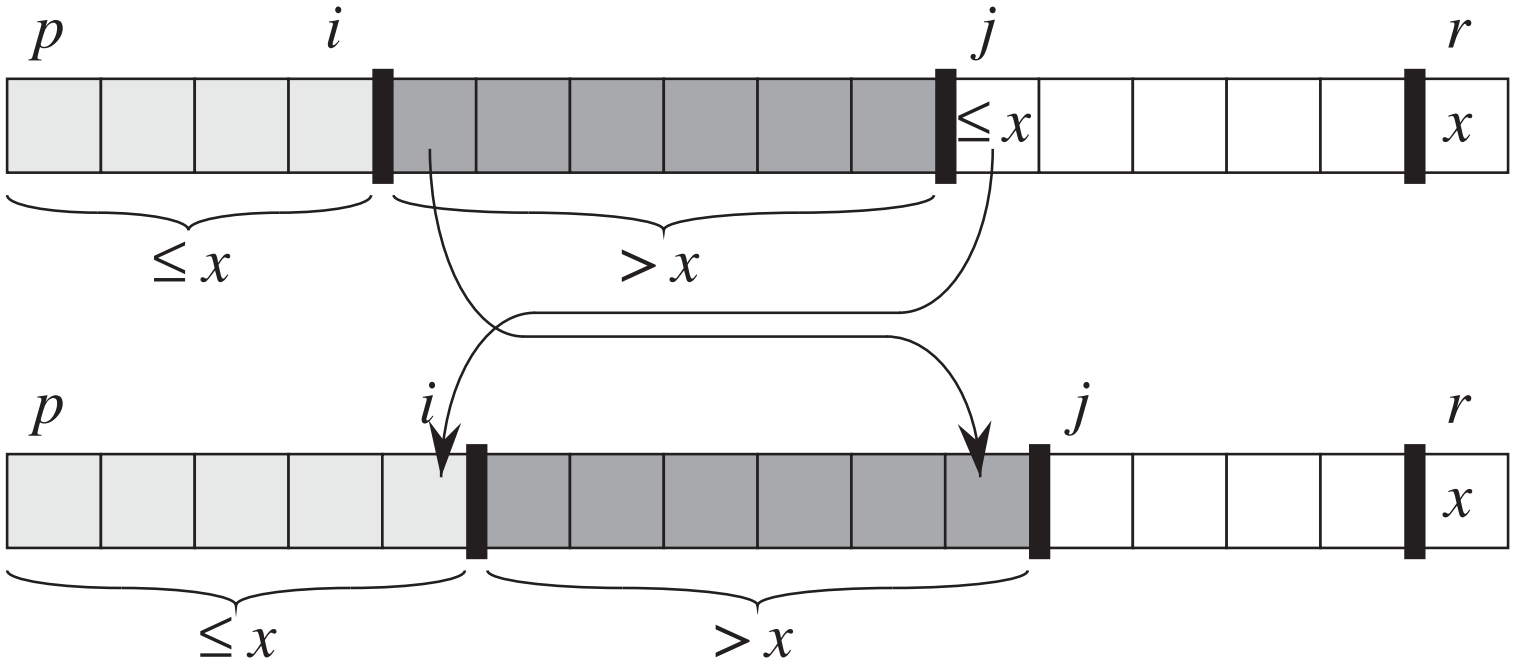


- Swapping when $A[j] > x$:



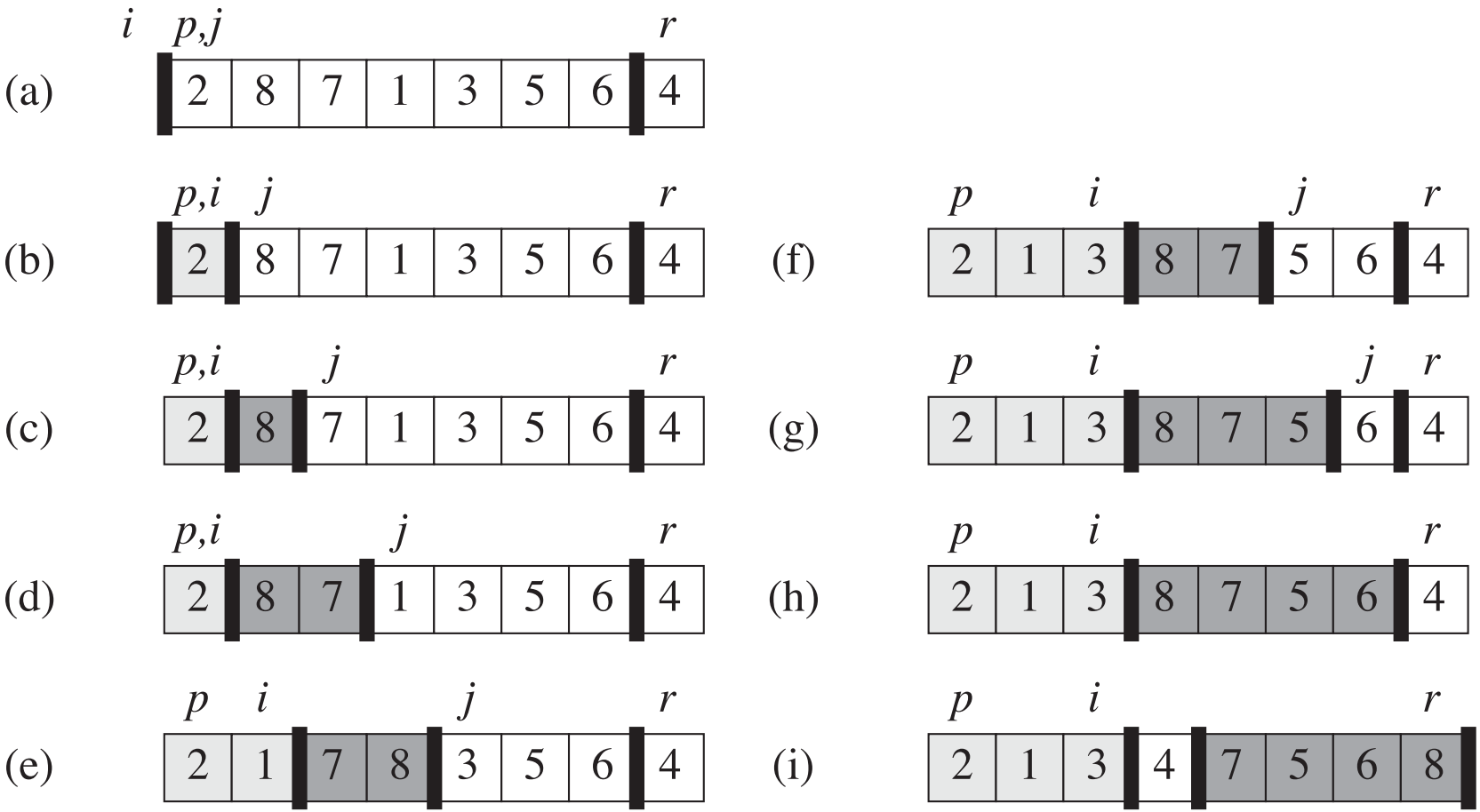
- **The only action is $j \leftarrow j + 1$!**

- Swapping when $A[j] \leq x$:



- i is incremented
- $A[i]$ and $A[j]$ are swapped
- j is incremented

Example


(a)
i p,j r
2 8 7 1 3 5 6 4
(b)
p,i j r
2 8 7 1 3 5 6 4
(c)
p,i j r
2 8 7 1 3 5 6 4
(d)
p,i j r
2 8 7 1 3 5 6 4
(e)
p i j r
2 1 7 8 3 5 6 4
(f)
p i j r
2 1 3 8 7 5 6 4
(g)
p i j r
2 1 3 8 7 5 6 4
(h)
p i r
2 1 3 8 7 5 6 4
(i)
p i r
2 1 3 4 7 5 6 8

Randomized Caching

- Another description of a class of Marking Algorithms.

```
Each memory item can be either marked or unmarked
At the beginning of the phase, all items are unmarked
On a request to item s:
   Mark s
   If s is in the cache, then evict nothing
   Else s is not in the cache:
      If all items currently in the cache are marked then
         Declare the phase over
         Processing of s is deferred to start of next phase
      Else evict an unmarked item from the cache
      Endif
   Endif
```

- These is no precise description of eviction procedure.
- *LRU* and *FWF* are special cases of the above algorithm.

Randomized Caching

```
Each memory item can be either marked or unmarked
At the beginning of the phase, all items are unmarked
On a request to item s:
   Mark s
   If s is in the cache, then evict nothing
   Else s is not in the cache:
      If all items currently in the cache are marked then
         Declare the phase over
         Processing of s is deferred to start of next phase

         Else evict an unmarked item chosen uniformly at random
              from the cache

      Endif
   Endif
```

- Some random number generator is needed!

Theorem

The random marking algorithm RALG is $O(\log k)$-competitive against OPT (i.e. LFD), where k is the size of its cache.

Randomized Approximation Algorithm for MAX 3-SAT

- **We will show how randomization may help solving NP-complete problems, as 3-SAT.**

3-SAT Problem

Literal. A boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form. A propositional formula Φ that is the conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT. Given CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

3-SAT: Given a set of clauses $\{C_1, \ldots, C_k\}$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, does there exist a satisfying truth assignment?

MAX 3-SAT

- Given the set of input clauses $\{C_1, \ldots, C_k\}$, each of length 3, over a set of variables $X = \{x_1, \ldots, x_n\}$, **find a truth assignment that satisfies as many clauses as possible.**
- We will the Maximum 3-Satisfiability Problem (or **MAX 3-SAT** for short).
- This is an NP-hard optimization problem, since it is NP-complete to decide whether the maximum number of simultaneously satisfiable clauses is equal to k.

Randomized Approximation Algorithm for MAX 3-SAT

- Suppose we set each variable $\{x_1, \ldots, x_n\}$ independently to 0 or 1 with probability $\frac{1}{2}$ each.
- Let Z denote the random variable equal to the number of satisfied clauses.
- Thus $Z = Z_1 + Z_2 + \ldots + Z_k$, where $Z_i = 1$ if the clause C_i is satisfies and $Z_i = 0$ otherwise.
- Now $E(Z_i)$ is equal to the probability that C_i is satisfied.
- In order for C_i not to be satisfied, each of its three variables must be assigned the value that fails to make it true; since the variables are set independently, the probability of this is $(\frac{1}{2})^3 = \frac{1}{8}$.
- Thus clause C_i is satisfied with probability $1 - \frac{1}{8} = \frac{7}{8}$, and so $E(Z_i) = \frac{7}{8}$.
- Hence: $E(Z) = E(Z_1) + E(Z_2) + \ldots + E(Z_k) = \frac{7}{8}k$.

Randomized Approximation Algorithm for MAX 3-SAT

Proposition

Consider a 3-SAT formula, where each clause has three different variables. The expected number of clauses satisfied by a random assignment is within an approximation factor $\frac{7}{8}$ of optimal.

Proof.

Since $E(Z) = E(Z_1) + E(Z_2) + \ldots + E(Z_k) = \frac{7}{8}k$. □

Theorem

For every instance of 3-SAT, there is a truth assignment that satisfies at least a $\frac{7}{8}$ fraction of all clauses.

Proof.

For every instance of 3-SAT, a random truth assignment satisfies a $\frac{7}{8}$ fraction of all clauses in expectation; so, in particular, there must exist a truth assignment that satisfies a number of clauses that is at least as large as this expectation. □

Waiting to Find a Good Assignment

- Suppose we are **not** satisfied with a "one-shot" algorithm that produces a single assignment with a large number of satisfied clauses in expectation.
- Rather, we would like a randomized algorithm whose expected running time is polynomial and that is guaranteed to output a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.
- A simple way to do this is to generate random truth assignments until one of them satisfies at least $\frac{7}{8}k$ clauses. We know that such an assignment exists, but how long will it take until we find one by random trials?
- If we can show that the probability a random assignment satisfies at least $\frac{7}{8}k$ clauses is at least p, then the expected number of trials performed by the algorithm is $\frac{1}{p}$ (see page 18).
- In particular, we would like to show that this quantity p is at least as large as an inverse polynomial in n and k.

Waiting to Find a Good Assignment

- If we can show that the probability a random assignment satisfies at least $\frac{7}{8}k$ clauses is at least p, then the expected number of trials performed by the algorithm is $\frac{1}{p}$ (see page 18).
- In particular, we would like to show that this quantity p is at least as large as an inverse polynomial in n and k.
- For $j = 0, 1, 2, \ldots, k$, let p_j denote the probability that a random assignment satisfies exactly j clauses.
- Hence: $E(X) = \sum_{j=0}^{k} j p_i = \frac{7}{8}k.$
- Now we have:

$$\frac{7}{8}k = E(X) = \sum_{j=0}^{k} j p_i = \sum_{j<\frac{7}{8}k} j p_i + \sum_{j\geq\frac{7}{8}k} j p_i.$$

- and $\sum_{j<\frac{7}{8}k} p_j = 1 - p.$

Waiting to Find a Good Assignment

- Thus: $\sum_{j<\frac{7}{8}k} p_j = 1-p$, $\sum_{j\geq\frac{7}{8}k} p_j = p$ and

$$\frac{7}{8}k = \sum_{j=0}^{k} jp_i = \sum_{j<\frac{7}{8}k} jp_i + \sum_{j\geq\frac{7}{8}k} jp_i$$

- Let k' denote the largest natural number that is strictly smaller than $\frac{7}{8}k$.
- Now we have:

$$\frac{7}{8}k \leq \sum_{j<\frac{7}{8}k} k'p_i + \sum_{j\geq\frac{7}{8}k} kp_i = k'(1-p) + kp \leq k' + kp$$

- Hence $kp \geq \frac{7}{8}k - k'$, and, since k' is a natural number strictly smaller that $\frac{7}{8}$ another natural number, so $\frac{7}{8}k - k' \geq \frac{1}{8}$, i.e.

$$kp \geq \frac{7}{8}k - k' \geq \frac{1}{8}.$$

- Which means:

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8k} \quad \text{i.e.} \quad \frac{1}{p} \leq 8k.$$

Waiting to Find a Good Assignment

- We have $\frac{1}{p} \leq 8k$.
- The expected number of trials needed to find the satisfying assignment we want is at most $8k$.

Proposition

There is a randomized algorithm with polynomial expected running time that is guaranteed to produce a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.