Lecture 1: Introduction to randomized algorithms

A randomized algorithm is an algorithm whose working not only depends on the input but also on certain random choices made by the algorithm.

Assumption: We have a random number generator $\text{Random}(a, b)$ that generates for two integers $a, b$ with $a < b$ an integer $r$ with $a \leq r \leq b$ uniformly at random. In other words, $\Pr[r = i] = 1/(b - a + 1)$ for all $a \leq i \leq b$. We assume that $\text{Random}(a, b)$ runs in $O(1)$ time (even though it would perhaps be more fair if we could only get a single random bit in $O(1)$ time, and in fact we only have pseudo-random number generators).

1.1 Some probability-theory trivia

Running example: flip a coin twice and write down the outcomes (heads or tails).

• Sample space = elementary events = possible outcomes of experiment = \{HH, HT, TH, TT\}.

• Events = subsets of sample space. For example \{HH, HT, TH\} (at least one heads).

• Each event has a certain probability. Uniform probability distribution: all elementary events have equal probability: $\Pr[HH] = \Pr[HT] = \Pr[TH] = \Pr[TT] = 1/4$.

• Random variable: assign a number to each elementary event. For example random variable $X$ defined as the number of heads: $X(HH) = 2$, $X(HT) = X(TH) = 1$, $X(TT) = 0$.

• Expected value of (discrete) random variable.

$$E[X] = \sum x \cdot \Pr[X = x],$$

with the sum taken over all possible values of $X$. For the example above:

$$E[X] = (1/4) \cdot 2 + (1/4) \cdot 1 + (1/4) \cdot 1 + (1/4) \cdot 0 = 1.$$

• Define some other random variable $Y$, e.g. $Y(HH) = Y(TT) = 1$ and $Y(TH) = Y(HT) = 0$. We have $E[Y] = 1/2$. Now consider the expectation of $X + Y$:

$$E[X + Y] = \Pr[HH] \cdot (X(HH) + Y(HH)) + \cdots + \Pr[TT] \cdot (X(TT) + Y(TT))$$

$$= (1/4) \cdot (2 + 1) + (1/4) \cdot (1 + 0) + (1/4) \cdot (1 + 0) + (1/4) \cdot (0 + 1) = 3/2.$$

Thus, $E[X + Y] = E[X] + E[Y]$. This is always the case, because of linearity of expectation:

For any two random variables $X, Y$ we have $E[X + Y] = E[X] + E[Y]$.

• Do we also have $E[X \cdot Y] = E[X] \cdot E[Y]$? In general, no. (Check example above: $E[X^2] = 3/2 \neq 1 = E[X]^2$.) It is true if the variables are independent.

• Let $X$ be a random variable. Then the Markov Inequality gives a bound on the probability that the actual value of $X$ is $t$ times larger than its expected value.
Lemma 1.1 (Markov inequality) Let $X$ be a non-negative random variable, and $\mu = \mathbb{E}[X]$ be its expectation. Then for any $t > 0$ we have $\Pr[X > t \cdot \mu] \leq 1/t$.

• Bernoulli trial: experiment with two possible outcomes: success or fail. If the probability of success is $p$, then the expected number of trials before a successful experiment is $1/p$.

If we consider experiments with two possible outcomes, 0 or 1, but the success probability is different for each experiment, then these are called Poisson trials. The following result is often useful to obtain high-probability bounds for randomized algorithms:

Lemma 1.2 (Tail estimates for Poisson trials) Suppose we do $n$ Poisson trials. Let $X_i$ denote the outcome of the $i$-th trial and let $p_i = \Pr[X_i = 1]$, where $0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ be the expected number of successful experiments. Then

$$\Pr[X > (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu.$$ 

Thus the probability of deviating from the expected value decreases exponentially. For example, for $\delta = 2$ we have $e^\delta/(1 + \delta)^{1+\delta} < 1/2$, so we get

$$\Pr[X > 3\mu] \leq \left(\frac{1}{2}\right)^\mu.$$ 

1.2 Randomized approximate median

Let $S$ be a set of $n$ numbers. Assume for simplicity that all numbers are distinct. The rank of a number $x \in S$ is 1 plus the number of elements in $S$ that are smaller than $x$:

$$\text{rank}(x) = 1 + |\{y \in S : y < x\}|.$$ 

Thus the smallest element has rank 1 and the largest element has rank $n$ (under the assumption that all elements are distinct). A median of $S$ is a number of rank $[(n + 1)/2]$ or $\lceil (n + 1)/2 \rceil$. Later we will look at the problem of finding a median in $S$---in fact, we will look at the more general problem of finding an element of a given rank. In many applications, however, we do not need the exact median; a number close to the median is good enough.

A $\delta$-approximate median is an element of rank $k$ with $[(\frac{1}{2} - \delta)(n+1)] \leq k \leq [(\frac{1}{2} + \delta)(n+1)]$, for some given constant $0 \leq \delta \leq 1/2$. The following algorithm tries to find a $\delta$-approximate median in a set of number given in an array $A[1..n]$.

Algorithm ApproxMedian1($\delta$, $A$)
1. $\triangleright A[1..n]$ is array of $n$ distinct numbers.
2. $r \leftarrow \text{Random}(1,n)$
3. $x^* \leftarrow A[r]$; $k \leftarrow 1$
4. for $i \leftarrow 1$ to $n$
5. do if $A[i] < x^*$ then $k \leftarrow k + 1$
6. if $[(\frac{1}{2} - \delta)(n+1)] \leq k \leq [(\frac{1}{2} + \delta)(n+1)]$
7. then return $x^*$
8. else return “error”
ApproxMedian1 clearly runs in $O(n)$ time. Unfortunately, it does not always correctly solve the problem: when it reports an element $x^*$ this element is indeed a $\delta$-approximate median, but it may also report “error”. When does the algorithm succeed? This happens when the random element that is picked is one of the $\lceil(1/2 + \delta)(n+1)\rceil - \lfloor(1/2 - \delta)(n+1)\rfloor + 1$ possible $\delta$-approximate medians. Since the index $r$ is chosen uniformly at random in line 2, this happens with probability

$$\frac{\lceil(1/2 + \delta)(n+1)\rceil - \lfloor(1/2 - \delta)(n+1)\rfloor + 1}{n} \approx 2\delta.$$  

For example, for $\delta = 1/4$—thus we are looking for an element of rank between $n/4$ and $3n/4$—the success probability is $1/2$. If we are looking for an element that is closer to the median, say $\delta = 1/10$, then things get worse. However, there’s an easy way to improve the success rate of the algorithm: we choose a threshold value $c$, and repeat ApproxMedian1 until we have found a $\delta$-approximate median or until we have tried $c$ times.

**Algorithm ApproxMedian2($\delta$, $A$)**

1. $j \leftarrow 1$
2. repeat $\text{result} \leftarrow \text{ApproxMedian1}(A, \delta)$; $j \leftarrow j + 1$
3. until ($\text{result} \neq \text{“error”}$) or ($j = c + 1$)
4. return $\text{result}$

The worst-case running time of this algorithm is $O(cn)$. What is the probability that it fails to report a $\delta$-approximate median? For this to happen, all $c$ trials must fail, which happens with probability (roughly) $(1 - 2\delta)^c$. So if we want to find a $(1/4)$-approximate median, then by setting $c = 10$ we obtain a linear-time algorithm whose success rate is roughly 99.9%. Pretty good. Even if we are more demanding and want to have a $(1/10)$-approximate median, then with $c = 10$ we still obtain a success rate of more than 89.2%.

Of course if we insist on finding a $\delta$-approximate median, then we can just keep on trying until finally ApproxMedian1 is successful:

**Algorithm ApproxMedian3($\delta$, $A$)**

1. repeat $\text{result} \leftarrow \text{ApproxMedian1}(A, \delta)$
2. until $\text{result} \neq \text{“error”}$
3. return $\text{result}$

This algorithm always reports a $\delta$-approximate median—or perhaps we should say: it never produces an incorrect answer or “error”—but its running time may vary. If we are lucky then the first run of ApproxMedian1 already produces a $\delta$-approximate median. If we are unlucky, however, it may take many many trials before we finally have success. Thus we cannot bound the worst-case running time as a function of $n$. We can say something about the expected running time, however. Indeed, as we have seen above, a single run of ApproxMedian1($A, \delta$) is successful with probability (roughly) $2\delta$. Hence, the expected number of trials until we have success is $1/(2\delta)$. We can now bound the expected running time of ApproxMedian3 as follows.

$$\text{E[ running time of ApproxMedian3 ]} = \text{E[ (number of calls to ApproxMedian1) \cdot O(n) ]}$$

$$= O(n) \cdot \text{E[ number of calls to ApproxMedian1 ]}$$

$$= O(n) \cdot (1/2\delta)$$

$$= O(n/\delta)$$
Thus the algorithm runs in $O(n/\delta)$ expected time. The expectation in the running time has nothing to do with the specific input: the expected running time is $O(n/\delta)$ for any input. In other words, we are not assuming anything about the input distribution (in particular, we do not assume that the elements are stored in $A$ in random order): the randomization is under full control of the algorithm. In effect, the algorithm uses randomization to ensure that the expected running time is $O(n/\delta)$, no matter in which order the numbers are stored in $A$. Thus it is actually more precise to speak of the worst-case expected running time: for different inputs the expected running time may be different—this is not the case in the ApproxMedian3, by the way—and we are interested in the maximum expected running time. As is commonly done, when we will talk about “expected running time” in the sequel we actually mean “worst-case expected running time”.

Monte Carlo algorithms and Las Vegas algorithms. Note that randomization shows up in different ways in the algorithms we have seen. In ApproxMedian1 the random choices made by the algorithm influenced the correctness of the algorithm, but the running time was independent of the random choices. Such an algorithm is called a Monte Carlo algorithm. In ApproxMedian3 on the other hand, the random choices made by the algorithm influenced the running time, but the algorithm always produces a correct solution. Such an algorithm is called a Las Vegas algorithm. ApproxMedian2 is mixture: the random choices influence both the running time and the correctness. Sometimes this is also called a Monte Carlo algorithm.

1.3 The hiring problem

Suppose you are doing a very important project, for which you need the best assistant you can get. You contact an employment agency that promises to send you some candidates, one per day, for interviewing. Since you really want to have the best assistant available, you decide on the following strategy: whenever you interview a candidate and the candidate turns out to be better than your current assistant, you fire your current assistant and hire the new candidate. If the total number of candidates is $n$, this leads to the following algorithm.

**Algorithm** Hire-Assistant

1. $Current\text{Assistant} \leftarrow \text{nil}$
2. for $i \leftarrow 1$ to $n$
3. do Interview candidate $i$
4. if candidate $i$ is better than $Current\text{Assistant}$
5. then $Current\text{Assistant} \leftarrow$ candidate $i$

Suppose you have to pay a fee of $f$ euro whenever you hire a new candidate (that is, when $Current\text{Assistant}$ changes). In the worst case, every candidate is better than all previous ones, and you end up paying $f \cdot n$ euro to the employment agency. Of course you would like to pay less. One way is to proceed as follows. First, you ask to employment agency to send you the complete list of $n$ candidates. Now you have to decide on an order to interview the candidates. The list itself does not give you any information on the quality of the candidates, only their names, and in fact you do not quite trust the employment agency: maybe they have put the names in such an order that interviewing them in the given order (or in the reverse order) would maximize their profit. Therefore you proceed as follows.
Algorithm Hire-Assistant-Randomized
1. Compute a random permutation of the candidates.
2. \( CurrentAssistant \leftarrow \text{nil} \)
3. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \)
4. \textbf{do} Interview candidate \( i \) in the random permutation
5. \textbf{if} candidate \( i \) is better than \( CurrentAssistant \)
6. \textbf{then} \( CurrentAssistant \leftarrow \text{candidate } i \)

What is the expected cost of this algorithm, that is, the expected total fee you have to pay to the employment agency?

\[
E[\text{cost}] = E[\sum_{i=1}^{n} \text{(cost to be paid for } i\text{-th candidate)}] = \sum_{i=1}^{n} E[\text{(cost to be paid for } i\text{-th candidate)}] \quad \text{(by linearity of expectation)}
\]

You have to pay \( f \) euro when \( i\)-th candidate is better than candidates \( 1, \ldots, i-1 \), otherwise you pay nothing. Consider the event "\( i\)-th candidate is better than candidates \( 1, \ldots, i-1 \)”, and introduce indicator random variable \( X_i \) for it:

\[
X_i = \begin{cases} 
1 & \text{if } i\text{-th candidate is better than candidates } 1, \ldots, i-1 \\
0 & \text{otherwise}
\end{cases}
\]

(An indicator random variable for an event is a variable that is 1 if the event takes place and 0 otherwise.) Now the fee you have to pay for the \( i\)-th candidate is \( X_i \cdot f \) euro, so we can write

\[
E[\text{(fee to be paid for } i\text{-th candidate)}] = E[X_i \cdot f] = f \cdot E[X_i]
\]

Let \( C_i \) denote the first \( i \) candidates in the random order. \( C_i \) itself is also in random order, and so the \( i\)-th candidate is the best candidate in \( C_i \) with probability \( 1/i \). Hence,

\[
E[X_i] = \Pr[X_i = 1] = 1/i,
\]

and we get

\[
E[\text{total cost}] = \sum_{i=1}^{n} E[\text{(fee to be paid for } i\text{-th candidate)}] = \sum_{i=1}^{n} f \cdot E[X_i] = f \cdot \sum_{i=1}^{n} (1/i) \approx f \ln n
\]

The algorithm above needs to compute a random permutation on the candidates. Here is an algorithm for computing a random permutation on a set of elements stored in an array \( A \):

Algorithm RandomPermutation(\( A \))
1. \( \triangleright \) Compute random permutation of array \( A[1..n] \)
2. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n-1 \)
3. \textbf{do} \( r \leftarrow \text{Random}(i, n) \)
4. \text{Exchange } A[i] \text{ and } A[r] \)

Note: The algorithm does not work if we replace \( \text{Random}(i, n) \) in line 3 by \( \text{Random}(1, n) \).