CSE 632 (Fall 2019): Analysis of Algorithms II : Randomized Algorithms

Lecture 1-2 (8/28/2019,8/30/2019): Introduction

Lecturer: Shi Li *Scriber: Shi Li*

1 Introduction

- Almost all algorithms learned in CSE431/531: Analysis of Algorithms I are "deterministic": if we run the algorithm twice on the same input, the courses of the two executions and their outputs will be exactly the same.
- In this course, we shall study "randomized" algorithms, which have "random" behaviors.
- Why do we use randomized algorithms?
  1. We can get simpler algorithms by using randomness. For example, randomized quicksort, randomized 7/8-approximation for 3-SAT are much simpler than their deterministic counterparts, and there is a simple random algorithm for 2-SAT.
  2. We can get faster algorithms by using randomness. Examples for these algorithms include randomized probability identity testing, Freivald's matrix multiplication verification algorithm, and algorithms using sampling and fingerprinting.
  3. Some randomized algorithms are mathematically beautiful. The "distribution view" may give nice properties about the considered problem (e.g, Nash equilibrium).
  4. Sometimes, we can use probabilistic method to argue about the existence of an object.
- Price of using randomness. However, the benefits come with some prices.
  1. There is some chance that the algorithm is incorrect.
  2. There is some chance that the algorithm takes a long time to terminate.

2 Two examples of randomized algorithms

2.1 Example 1: Freivald's algorithm for verifying matrix multiplication

- Problem: Given three matrices $A, B, C \in \mathbb{Z}^{n\times n}$, we need to check if $C = AB$.
- Straightforward algorithm: compute $C' = AB$ and check if $C' = C$.
- Running time of the algorithm depends on the algorithm we choose for matrix multiplication.
  - Naive algorithm: $O(n^3)$
  - Strassen's algorithm: $O(n^{2.81})$
  - Best algorithm for matrix multiplication: $O(n^{2.3729})$.

  To make the algorithm faster and faster, we need to make it more and more complicated.
- Instead, Freivald's algorithm is a randomized algorithm that runs in $O(n^2)$ time.
- Algorithm $\mathcal{A}$: Freivald's Algorithm with one experiment

  1: randomly choose a vector $r \in \{0,1\}^n$
  2: check if $ABr = Cr$
- What is the running time of the algorithm? It depends on how we compute ABr
  - If compute using the order $(AB)r$, then we have to multiply A and B and the algorithm is not better than the deterministic version.
  - We can compute using the order $A(Br)$ and the running time is $O(n^2)$.

  So the running time of $\mathcal{A}$ is $O(n^2)$.
- What about the correctness of the algorithm?
  - If $AB = C$, then it will always output "yes", which is desired.

– If $AB \neq C$, we need to the output to be "no", but $\mathcal{A}$ may output "yes" since it might happen that $ABr = Cr$. However, we show that the probability of $ABr = Cr$ is small.

Lemma 1. *If $AB \neq C$, then $\mathcal{A}$ outputs "no" with probability at least $1/2$.*

Proof. Let $D = C - AB \neq 0$. $Cr = ABr$ is equivalent to $Dr = 0$. So, we need to compute $\Pr[Dr \neq 0]$. Since $D \neq 0$, there exists some $i, j \in [n]$ such that $D_{j,i} \neq 0$. We shall only focus on the i-th row D_i of D.

$$D_i r = \sum_{j'=1}^{n} D_{i,j'} r_{j'} =: X + Y,$$

where we define $X = \sum_{j' \in [n], j' \neq j} D_{i,j'} r_{j'}$ and $Y = D_{i,j} r_j$. Then

$$\begin{aligned}
&\Pr[D_i r \neq 0] = \Pr[Y \neq -X] = \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \Pr[Y \neq -x | X = x] \\
&= \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \Pr[D_{i,j} r_j \neq -x | X = x] \\
&\geq \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \frac{1}{2} = \frac{1}{2}.
\end{aligned}$$

The inequality in the third line used the fact that $D_{i,j} \neq 0$ and at most one value of r_j from $\{0, 1\}$ can make $D_{i,j} r_j = -x$.

So, we have $\Pr[Dr \neq 0] \geq \Pr[D_i r \neq 0] \geq \frac{1}{2}$. □

The probabilities of the algorithm outputting "Yes" and "No" under the cases $AB = C$ and $AB \neq C$ are as follows:

	Yes	No
$AB = C$	1	0
$AB \neq C$	$\leq 1/2$	$\geq 1/2$

The constant $1/2$ in the lemma may be unsatisfactory. But we can boost the probability by running $\mathcal{A}$ k times. The new algorithm outputs "Yes" if and only all the k results are "Yes". For the new algorithm, we have the following probabilities:

	Yes	No
$AB = C$	1	0
$AB \neq C$	$\leq 1/2^k$	$\geq 1 - 1/2^k$

We say $1/2^k$ is the error probability of the algorithm. If we let $k = 10$, then the error probability of the algorithm will be $1/2^{10} \leq 0.001$.

We say the Frievald's algorithm is a *Monta Carlo* algorithm :

Definition 2. *A Monta Carlo algorithm is a randomized algorithm whose output may be incorrect with some probability.*

For a "Yes/No" output Monta Carlo algorithm, we say the algorithm has one-sided error if it makes error only if the correct output is "Yes" (or "No"). Otherwise, we say the algorithm has two-sided error.

2.2 Randomized quick-sort and selection

Given n integers $A[1..n]$ (for simplicity, we assume they are all distinct), two problems of interest are:

- the sorting problem: it asks for sorting the n integers in increasing order.
- the selection problem: we are given an integer $i \in [n]$, and need to output the i-th smallest number in A.

- We have many $O(n \log n)$-time deterministic sorting algorithms : merge-sort, heap-sort and quick-sort.
- The selection problem can be solved using any sorting algorithm.

We shall consider quick-sort algorithm.

- It is more practical.
- It leads to an algorithm for the selection problem with $O(n)$-running time.

There are two versions of quick-sort algorithm. The deterministic quick-sort algorithm has $O(n \log n)$-running time in the worst case but it needs to use the complicated divide-and-conquer median-finder algorithm, which is impractical. The randomized quick-sort algorithm, on the other hand, only has expected running time of $O(n \log n)$ but is easy to implement.

Algorithm 1 deterministic(randomized)-quick-sort(A)

1: **if** $|A| \leq 1$ **then return** A
2: $x \leftarrow$ median of A (a random number in A)
3: $B \leftarrow$ array of integers in A that are smaller than x
4: $C \leftarrow$ array of integers in A that are bigger than x
5: **return** deterministic(randomized)-quick-sort(B) concatenating $[x]$ concatenating deterministic(randomized)-quick-sort(C)

Algorithm 2 deterministic(randomized)-selection(A, i)

1: **if** $|A| \leq 1$ **then return** $A[1]$
2: $x \leftarrow$ median of A (a random number in A)
3: $B \leftarrow$ array of integers in A that are smaller than x
4: $C \leftarrow$ array of integers in A that are bigger than x
5: **if** $i \leq |B|$ **then**
6: **return** deterministic(randomized)-selection(B, i)
7: **else if** $i = |B| + 1$ **then**
8: **return** x
9: **else**
10: **return** deterministic(randomized)-selection($C, i - |B| - 1$)

2.3 Analysis of Expected Running Time of Randomized Selection

In this lecture we only analyze the expected running time of the randomized selection algorithm. For every $j = 0, 1, 2, \cdots$,, let X_j denote the size of A in the j-th recursion of randomized-selection (notice that each recursion shall call at most one recursive call). For simplicity, the root-recursion is the 0-th recursion. When the j-the recursion does not exist, we define $X_j = 0$. For every $j \geq 0$, and integer n', we have

$$\begin{aligned}
\mathbb{E}[X_{j+1}|X_j = n'] &\leq \frac{1}{n'} \sum_{k=1}^{n'} \max\{k-1, n'-k\} \\
&\leq \frac{1}{n'} \left(\int_{k=0}^{n'/2} (n'-k) \mathsf{d}k + \int_{k=n'/2}^{n'} k \mathsf{d}k \right) \\
&= \frac{1}{n'} \left(\left(n'k - \frac{k^2}{2} \right) \Big|_0^{n'/2} + \frac{k^2}{2} \Big|_{n'/2}^{n'} \right) = \frac{1}{n'} \left(\frac{n'^2}{2} - \frac{n'^2}{8} + \frac{n'^2}{2} - \frac{n'^2}{8} \right) = \frac{3n'}{4}.
\end{aligned}$$

For the inequality in the first line, we let k be the rank of the random number we selected in A. (An integer in A has rank k if it is the k-th smallest integer in A). So, k is evenly distributed in $[n']$. So there are $k-1$ integers in A that are smaller than x and $n'-k$ integers in A that are bigger than x. In the next iteration, A will have have size at most $\max\{k-1, n'-k\}$. So the inequality in the first line follows. The equality in the second line is by some simple observations.

Thus, we have $\mathbb{E}[X_{j+1}] \leq \frac{3}{4}\mathbb{E}[X_j]$. Applying the inequality for every $j \geq 0$ and using the fact that $X_0 = n$ always holds, we have $\mathbb{E}[X_j] \leq \left(\frac{3}{4}\right)^j n$. One simple observation is that the running time of the randomized quick sort is at most $O(1) \times \sum_{j=0}^{\infty} X_j$. By linearity of expectation, we have

$$\begin{aligned}\mathbb{E}[\text{running time of randomized selection}] &\leq \mathbb{E}\left[O(1)\sum_{j=0}^{\infty} X_j\right] \leq O(1)\sum_{j=0}^{\infty}\mathbb{E}[X_j] \\ &\leq O(1)\sum_{j=0}^{\infty}\left(\frac{3}{4}\right)^j n = O(1)\cdot 4n = O(n).\end{aligned}$$

The randomized selection algorithm we gave is a Las-Vegas algorithm, which is a randomized algorithm that always outputs a correct solution but has randomized running time. It may happen that a Las-Vegas algorithm can take a long time; often we bound its expected running time.

Table 1: Comparisons between Monta Carlo and Las Vegas Algorithms.

	correctness	running time
Monta Carlo	may be wrong	usually has good worst-case running time
Las Vegas	always correct	may take a long time and usually only has good "expected running time"

The properties about the running time of Monta Carlo and Las Vegas algorithms in the table are not required by their definitions, but rather, are what we expect when we call an algorithm a Monta Carlo or Las Vegas algorithm.

2.4 A Relationship between Monta Carlo and Las Vegas Algorithms

Lemma 3. *Given a Las Vegas algorithm $\mathcal{A}$ with expected running time at most $T(n)$, we can design a Monta Carlo algorithm $\mathcal{A}'$ with worst-case running time $O(T(n))$ and error at most* 0.99.

The 0.99 in the above lemma can be changed to any constant smaller than 1. To show the lemma, in our algorithm $\mathcal{A}'$, we run $\mathcal{A}$ for $100T(n)$ units time. If $\mathcal{A}$ terminated, we output what $\mathcal{A}$ outputs; otherwise, we output anything (a garbage). So, the algorithm takes $O(T(n))$ time (the O notation hides the constant 100 and a constant required for simulating A'). By Markov Inequality that we shall learn in the future, we have $\Pr[\mathcal{A}$ runs for more than $100T(n)$ units time$] \leq 1/100$. So the probability that $\mathcal{A}'$ outputs what $\mathcal{A}$ outputs is at least 0.99. Since $\mathcal{A}$ is always correct, $\mathcal{A}'$ is correct with probability at least 0.99.