CSCE-658 Randomized Algorithms

Lecture #9, March 1, 2016

Lecturer: Professor Jianer Chen

9 Analysis of algorithm expected complexity

We have seen that the expectation of a random variable sometimes may not provide much useful information for what one might be interested in. On the other hand, if we have more information on the value distribution of a random variable, such as its upper bound or lower bound, then we will have a better control on its values, in terms of its expectation. There are a number of results addressing this issue. We will start with the simpliest (probably also the most famous) one. We will show how this is used in the analysis of "expected complexity" of algoritms.

9.1 Markov Inequality

Recall our discussion in the previous section on the expected size of a random cut for an undirected graph. We observed that an upper bound on the values of the random variable will limit the probability for the random variable to have values much smaller than the expectation. Similarly, if we can bound the values of a random variable from below, we will be able to limit the probability for the random variable to have values much larger than the expectation. This is given by the famous *Markov Inequality*, which assumes a lower bound 0 for the values of the random variable.

Theorem 9.1 (Markov Inequality) *Let X be a random variable that takes only non-negative values. Then for all $t > 0$, we have* $\Pr[X \geq t] \leq \mathbf{E}[X]/t$.

PROOF. First note that the events "$X \geq t$" and "$X < t$" form a partition of the sample space. Therefore, by Lemma 8.3, we have

$$\mathbf{E}[X] = \Pr[X \geq t] \cdot \mathbf{E}[X \mid X \geq t] + \Pr[X < t] \cdot \mathbf{E}[X \mid X < t].$$

Since $\mathbf{E}[X \mid X \geq t] \geq t$, $\Pr[X < t] \geq 0$, and $\mathbf{E}[X \mid X < t] \geq 0$ (the last inequality uses the assumption that X is non-negative), from the above formula, we have

$$\mathbf{E}[X] \geq \Pr[X \geq t] \cdot t.$$

Dividing both sides by the positive number t gives Markov Inequality. $\square$

If we let $t = r \cdot \mathbf{E}[X]$ in Theorem 9.1, we will get the following result, which is also called Markov Inequality.

Corollary 9.2 (Markov Inequality) *Let X be a random variable that takes only non-negative values. Then for all $r > 0$, we have* $\Pr[X \geq r \cdot \mathbf{E}[X]] \leq 1/r$.

Corollary 9.2 is probably more intuitive: the probability for the random variable X to have a value at least r times its expected value $\mathbf{E}[X]$ is not larger than $1/r$.

Markov Inequality (either Theorem 9.1 or Corollary 9.2) does not seem to address our concern on the randomized algorithm for the MAX-CUT problem in subsection 8.3: where we wanted to limit the probability for the cut size from below so that it would not go too much smaller than the expectation. On the other hand, Markov Inequality limits the probability for the random variable values from above so that it would not be too much larger than the expectation. Indeed, the MAX-CUT problem is a *maximization problem* in which we are most interested in preventing the solution values from going too small. On the other hand, Markov Inequality seems to be more directly applicable to *minimization problems* where we are interested in preventing the solution values from going too large.

In particular, Markov Inequality is very helpful when we study the complexity of an algorithm (either deterministic or randomized), which is something we surely want to minimize, assuming that we can represent the complexity as a random variable X and derive the expectation $\mathbf{E}[X]$. Since the complexity of an algorithm can never be negative, the condition on the random variable X in Markov Inequality is satisfied automatically. Moreover, by letting the parameter r be a properly chosen (large) constant, Corollary 9.2 shows that we can bound the complexity of the algorithm by the same asymptotic order as its expected value $\mathbf{E}[X]$ with a hight probability, i.e., $\Pr[X < r \cdot \mathbf{E}[X]] \geq 1 - 1/r$.

We will give two examples to illustrate this technique.

9.2 Expected complexity of an algorithm

Sorting is a fundamental task in computer science, which, on a given set of elements (without loss of generality, we can assume that these elements are real numbers), outputs the elements in a nondecreasing order. There have been many sorting algorithms. One of the most famous sorting algorithms is QuickSort, whose general framework looks as follows (to simplify our discussion, we will assume that all elements given in the input set are distinct):

1. pick a "pivot" element x in the input;
2. split the input into two sets S_1 and S_2, such that all elements in S_1 are smaller than x and all elements in S_2 are larger than x;
3. recursively sort S_1 and S_2;
4. output the concatenation of S_1, x, and S_2.

One advantage of QuickSort is that step 2 of the algorithm that splits the input set can be implemented "in place," i.e., it can be implemented using almost no extra storage space. For example, suppose that the input set is given as an array $A[1..n]$, and suppose that we want to split the elements in the subarray $A[l..u]$ using the element $A[k]$, $l \leq k \leq u$, then we can use the following subroutine:

```
Algorithm 14 Split($A, l, u, k$)

1. swap $A[l]$ and $A[k]$;
2. $i = l + 1$;  $j = u$;
3. while $i \leq j$ do {if $A[l] > A[i]$ then $i = i + 1$ else swap $A[i]$ and $A[j]$; $j = j - 1$}
4. swap $A[l]$ and $A[j]$;  return $(j)$.
```

To see the correctness of the algorithm, observe that we keep the following conditions in the **while** loop in step 3: all elements before $A[i]$ (but not including $A[i]$) are not larger than $A[l]$ and all elements after $A[j]$ (but not including $A[j]$) are larger than $A[l]$.

A well-known fact is that QuickSort does poorly in the worst case if we pre-specify the way of picking the pivot in the algorithm. For example, if we always pick the first element in the subarray $A[l..u]$ as the pivot, then in the cases when the subarray is already sorted, QuickSort takes time $\Omega(h^2)$, where $h = u - l + 1$ is the size of the subarray.

Thus, a natural approach to overcome the worst-time performance is to let the algorithm to pick a pivot *randomly*, which suggests the following randomized algorithm:

```
Algorithm 15 R-QuickSort($A, l, u$)
Input: subarray $A[l..u]$;
Output: subarray $A[l..u]$ with the elements sorted in non-decreasing order

1. if $l \geq u$ then return;
2. uniformly pick a random integer $k$ in $[l..u]$; \\ pick $A[k]$ as the pivot
3. $h = $ Split$(A, l, u, k)$;
4. recursively call R-QuickSort$(A, l, h - 1)$ and R-QuickSort$(A, h + 1, u)$.
```

Of course, to sort the entire array $A[1..n]$, we simply call R-QuickSort$(A, 1, n)$.

It may be helpful at this point to discuss what are the outcomes of our sample space. Let $A[1..n]$ be an array that is the input to the algorithm R-QuickSort$(A, 1, n)$. Each possible execution of R-QuickSort$(A, 1, n)$ can be represented by a binary tree (by the way, here we allow a node in a binary tree to have only one child), as follows. The root of the tree is the pivot picked in Split$(A, 1, n)$ (i.e., the element $A[h]$ after step 3 in R-QuickSort$(A, 1, n)$), whose left subtree is the tree corresponding to the recursive call R-QuickSort$(A, 1, h-1)$ (thus, the root of the left subtree, i.e., the left child of $A[h]$, is the pivot picked in Split$(A, 1, h-1)$), and whose right subtree is the tree corresponding to the recursive call R-QuickSort$(A, h+1, n)$ (thus, the root of the right subtree, i.e., the right child of $A[h]$, is the pivot picked in Split$(A, h+1, n)$). Since after step 3 in the algorithm R-QuickSort$(A, 1, n)$, all elements in $A[1..h-1]$ are smaller than $A[h]$ and all elements in $A[h+1..n]$ are larger than $A[h]$, the above definition of the binary tree actually gives a *binary search tree*.[1] In conclusion, each outcome of our sample space is a binary search tree of the elements in the input $A[1..n]$.

We analyze the expected running time of the algorithm R-QuickSort$(A, 1, n)$. Each execution of R-QuickSort consists of some "local steps" (i.e., steps 1-3), plus two recursive calls to R-QuickSort. Thus, the execution of the algorithm R-QuickSort$(A, 1, n)$ is a collection of the (recursive) executions of R-QuickSort. As a result, the running time of the algorithm R-QuickSort$(A, 1, n)$ is the sum of the "local complexities" (i.e., steps 1-3) of these executions. Since the complexity of the local steps of each execution is dominated by step 3 (i.e., the call to Split that splits the input elements), and the complexity of each execution of the subroutine Split is proportional to the number of comparisons of input elements (i.e., step 3 in Split that compares $A[l]$ and $A[i]$), we conclude that the complexity of the algorithm R-QuickSort$(A, 1, n)$ is proportional to the total number of element comparisons carried out in all these recursive executions. Thus, in order to derive an asymptotic order for the running time, we only need to concentrate on counting the number of element comparisons in the execution of the algorithm R-QuickSort$(A, 1, n)$.

Let $\{a_1, a_2, \ldots, a_n\}$ be the elements in $A[1..n]$, sorted in non-decreasing order: $a_1 < a_2 < \cdots < a_n$. For any two indices i and j, $1 \le i < j \le n$, let X_{ij} be the random variable that is equal to 1 if elements a_i and a_j are compared in the algorithm R-QuickSort$(A, 1, n)$ and 0 otherwise. By the above discussion, the running time of the algorithm R-QuickSort$(A, 1, n)$ is of the same order of $X = \sum_{i \ne j} X_{ij}$. Thus, the expected running time of the algorithm is $\mathbf{E}[X]$, which, by Linearity of Expectation, is given by

$$\mathbf{E}[X] = \sum_{i \ne j} \mathbf{E}[X_{ij}] = \sum_{i \ne j} \left(\sum_{\omega \in \Omega} \Pr[\omega] \cdot X_{ij}(\omega) \right) = \sum_{i \ne j} \Pr[X_{ij} = 1]. \tag{22}$$

here we have used "$X_{ij} = 1$" to represent an event. Since when two elements are compared, one of them must be a pivot, if $X_{ij} = 1$ then one of a_i and a_j must be a pivot (see step 3 in algorithm Split). Moreover, if *any* a_h between a_i and a_j is the pivot, $i < h < j$, then after the splitting using the pivot a_h, the elements a_i and a_j are placed in two different subarrays that will be processed by two different recursive executions of the algorithm, so they will never be compared. Summarizing this, we get

"$X_{ij} = 1$" if and only if the first pivot in $\{a_i, a_{i+1}, \ldots, a_j\}$ is either a_i or a_j.

Thus, the event "$X_{ij} = 1$" can be described verbally as "when the first element in $\{a_i, a_{i+1}, \ldots, a_j\}$ is picked as a pivot, this pivot is either a_i or a_j."

Before any element in $\{a_i, a_{i+1}, \ldots, a_j\}$ is used as a pivot, all elements in $\{a_i, a_{i+1}, \ldots, a_j\}$ remain in the same subarray $A[l..u]$ (but not necessarily consecutive). Define $\mathcal{E}_{ij}$ to be the event that the execution of R-QuickSort(A, l, u) is the first that uses an element in $\{a_i, a_{i+1}, \ldots, a_j\}$ as its pivot, and let E_{ij} be the event that the execution of R-QuickSort(A, l, u) uses the element a_i or a_j as its pivot. Then the event "$X_{ij} = 1$" is the event E_{ij} under the condition $\mathcal{E}_{ij}$. Thus,

$$\Pr[X_{ij} = 1] = \Pr[E_{ij} \mid \mathcal{E}_{ij}] = \Pr[E_{ij} \cap \mathcal{E}_{ij}] / \Pr[\mathcal{E}_{ij}].$$

[1]A *binary search tree* for a set S of n elements is an n-node binary tree T for which there is a one-to-one mapping between the nodes of T and the elements in S such that the element in S mapped to a node v in T is larger than all elements in S mapped to the left subtree of v, and smaller than all elements in S mapped to the right subtree of v.

Let $h = u - l + 1$ be the size of the subarray $A[l..u]$, then it is easy to verify: $\Pr[E_{ij} \cap \mathcal{E}_{ij}] = 2/h$ and $\Pr[\mathcal{E}_{ij}] = (j - i + 1)/h$. Therefore, $\Pr[X_{ij} = 1] = 2/(j - i + 1)$. bringing this to (22), we get

$$\mathbf{E}[X] = \sum_{i \neq j} \Pr[X_{ij} = 1] = \sum_{i \neq j} \frac{2}{j - i + 1} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{n} \sum_{h=2}^{n-i+1} \frac{1}{h} \leq 2 \sum_{i=1}^{n} H_n = 2nH_n,$$

where $H_n = \sum_{h=1}^{n} (1/h) \leq \ln n + 1$ is the n-th *Harmonic number* (note that $\ln n < \log n$), which is another important number used heavily in combinatorial and probabilistic analysis.

Theorem 9.3 *The expected running time of the algorithm* R-QuickSort *is* $O(n \log n)$.

We show how to apply Markov Inequality to the algorithm R-QuickSort. The random variable X is the number of element comparisons in the algorithm R-QuickSort, which is clearly a non-negative number. Thus, Corollary 9.2 is applicable. Since $\mathbf{E}[X] \leq 2nH_n$, we have

$$\Pr[X \geq 10nH_n)] \leq \Pr[X \geq 5 \cdot \mathbf{E}[X]] \leq \frac{1}{5} = 20\%,$$

where the second inequality used Markov Inequality (Corollary 9.2). Therefore, with a high probability (at least 80%) the algorithm R-QuickSort makes less than $10nH_n$ element comparisons. This explains why QuickSort runs fast in most cases in practice, although its worst-case time complexity is $\Omega(n^2)$.

We remark that Theorem 9.3 holds true for *every* input. Thus, given any input array $A[1..n]$, we can expect that R-QuickSort sorts the array in time $O(n \log n)$ in *most cases*. Here the sample space we use consists of all executions of the algorithm on the given input, and the expected time is taken over all these executions. There is also another version of the probabilistic analysis on *deterministic* QuickSort algorithms, where the sample space consists of all possible inputs in a pre-assumed distribution, and the expected running time is derived over all these inputs (see [20]).

We give an example for this kind of probabilistic analysis, which derives the expected running time of a deterministic algorithm, while the sample space is taken over all possible inputs.

Image that we are watching a stream of data (e.g., data from stock market), and we have a (complicated) reaction strategy that is developed based on the minimum value of the data and we want to keep the strategy updated as promptly as possible. We may use the following (deterministic) algorithm:

```
1.   min = $+\infty$;
2.   **loop**
2.1      read the next value $v$;
2.2      **if** min $> v$ **then** {min $= v$;  update the strategy;}
```

Since the reaction strategy is complicated, updating it can be expensive. Thus, an immediate question for this simple algorithm is: how often do we have to update the strategy?

The worst case is terrible: if the input values are sorted in decreasing order, then we have to update the strategy for every value we read from the stream! If we are more interested in the stochastic performance of the algorithm, we might want to ask a question like this:

What is the *expected* number of strategy updates when we totally read n values?

Since we are only concerned with the orderings of input values, not the values themselves, we can assume that our sample space consists of the $n!$ permutations of the numbers $\{1, 2, \ldots, n\}$, each with a probability $1/n!$. Thus, let the input stream be $\{a_1, a_2, \ldots, a_n\}$, which is a permutation of $\{1, 2, \ldots, n\}$. Define a random variable X_h that is equal to 1 if an update is needed after reading a_h, and 0 otherwise. Then $X = X_1 + X_2 + \cdots + X_n$ is the total number of updates we need when we read the stream. It is easy to see that "$X_h = 1$" if and only if a_h is the smallest among the first h elements $\{a_1, a_2, \ldots, a_h\}$ in the stream. Therefore, to compute $\mathbf{E}[X_h] = \Pr[X_h = 1]$, we need to count the total number of permutations of $\{1, 2, \ldots, n\}$ in which the h-th element is the smallest among the first h elements.

The $n!$ permutations of $\{1, 2, \ldots, n\}$ can be divided into $\binom{n}{h}$ groups. Each group $\mathcal{G}_S$ is associated with a subset S of h elements in $\{1, 2, \ldots, n\}$ such that the first h elements in each permutation in

$\mathcal{G}_S$ are the h elements in S. Thus, the group $\mathcal{G}_S$ contains exactly $h!(n-h)!$ permutations, in which $(h-1)!(n-h)!$ of the permutations have the smallest element in S in the h-th position. In conclusion, the total number of permutations of $\{1, 2, \ldots, n\}$ in which the h-th element is the smallest among the first h elements is equal to $\binom{n}{h} \cdot (h-1)!(n-h)!$.

Since our sample space is the set of all $n!$ permutations of $\{1, 2, \ldots, n\}$ and since all permutations are assumed to have the same probability, we derive

$$\mathbf{E}[X_h] = \Pr[X_h = 1] = \frac{\binom{n}{h} \cdot (h-1)!(n-h)!}{n!} = \frac{1}{h}.$$

This, by Linearity of Expectation, gives

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{h=1}^{n} X_h\right] = \sum_{h=1}^{n} \mathbf{E}[X_h] = \sum_{h=1}^{n} \frac{1}{h} = H_n \leq \ln n + 1.$$

Here again we see the Harmonic number H_n.

This shows that the simple algorithm given above probably works fine in practice, and relatively, only very few strategy updates are needed. For example, if n is one trillion (10^9), then the number of strategy updates is around $\ln 10^9 < 21$. Using Markov Inequality (Corollary 9.2), we also conclude that the probability that more than 105 strategy updates are performed is less than 20%.

9.3 Las Vegas and Monte Carlo

There is a conceptual difference between the randomized algorithm R-QuickSort (Algorithm 15) and the randomized algorithms we have seen in previous sections, such as Karger's algorithm for Min-Cut (Algorithm 3) and the Divide-and-Conquer algorithm for Max-Path (Algorithm 13). One type of algorithms such as R-QuickSort always return a correct solution (e.g., for R-QuickSort, the sorted sequence of the input elements) where the random variation from one execution to another is its running time, for which we study its expectation. This kind of algorithms are called *Las Vegas algorithms*. On the other hand, algorithms such as Karger's algorithm always guarantee a specific running time (in the worst case) while the random variation is the algorithm correctness (i.e., certain random executions of the algorithms may lead to incorrect conclusions) where we study its success probability. This kind of algorithms are called *Monte Carlo algorithms*.

Of course, we are most interested in algorithms whose running time (or expected running time) is bounded by a polynomial of the input size. For decision problems (see Section 4.2), we have seen the complexity classes defined based on polynomial-time Monte Carlo algorithms, such as RP (one-side error Monte Carlo algorithms) and BPP (two-side error Monte Carlo algorithms). We also have the following complexity class defined based on polynomial-time Las Vegas algorithms:

Definition 9.1 (Zero-Error Class ZPP) A decision problem Q is in the class *ZPP* if it can be solved by a Las Vegas algorithm of expected running time bounded by a polynomial of the input size.

Thus, if a (decision) problem Q is in ZPP, then there is a randomized algorithm A that on every instance x of Q always returns a correct conclusion (yes/no), with its running time being a random variable whose expectation is bounded by a fixed polynomial $p(|x|)$ of $|x|$.

We have the following simple relationship between ZPP and RP.

Lemma 9.4 *ZPP $\subseteq$ RP.*

PROOF. Let Q be a decision problem in ZPP. Let A_{zpp} be a Las Vegas algorithm that solves Q with running time $t(n)$. By the assumption of the lemma, $\mathbf{E}[t(n)] \leq p(n)$. Consider the following algorithm:

```
Algorithm $A_{\text{rp}}(x)$
1. execute the algorithm $A_{\text{zpp}}$ on $x$ for at most $2p(|x|)$ steps;
2. if the algorithm $A_{\text{zpp}}(x)$ stops within $2p(|x|)$ steps with a conclusion
   then return the conclusion of $A_{\text{zpp}}(x)$
   else return "no".
```

The running time of the algorithm $A_{\text{rp}}(x)$ is bounded by $O(p(n))$ on any input of size n. Thus, it is a Monte Carlo algorithm. For any instance x of size n, by Markov Inequality (Corollary 9.2),

$$\Pr[t(n) \geq 2p(n)] \leq \Pr[t(n) \geq 2 \cdot \mathbf{E}[t(n)]] \leq \frac{1}{2}.$$

That is, the probability that the algorithm $A_{\text{zpp}}(x)$ runs $2p(n)$ or more steps is bounded by 1/2. Thus, with a probability at least 1/2, the algorithm $A_{\text{zpp}}(x)$ on a yes-instance x of Q will run in at most $2p(n)$ steps, so step 1 of the algorithm $A_{\text{rp}}(x)$ will return a "yes", which will make the algorithm $A_{\text{rp}}(x)$ to return "yes" on x (so $A_{\text{zpp}}(x)$ returns "yes" with a probability at least 1/2). On the other hand, for a no-instance y, either the algorithm $A_{\text{zpp}}(y)$ reaches a "no" decision in step 1 of the algorithm $A_{\text{rp}}(y)$ that will make the algorithm $A_{\text{rp}}(y)$ to return "no", or the algorithm $A_{\text{zpp}}(y)$ does not reach a decision in step 1, in this case, the algorithm $A_{\text{rp}}(y)$ also returns "no" in step 2. Thus, on the no-instance y, the algorithm $A_{\text{rp}}(y)$ always (i.e., with a probability 1) returns "no". The algorithm A_{rp} shows that the problem Q is in RP. Since Q is an arbitrary problem in ZPP, this concludes that ZPP $\subseteq$ RP. $\square$

Thus, the class ZPP seems "weaker" than the one-side error class RP. Since the class RP is a subclass of the two-side error class BPP, we get the following inclusion relations:

$$\text{ZPP} \subseteq \text{RP} \subseteq \text{BPP}.$$

Recall that it is unlikely that an NP-complete problem can be solved by a BPP-algorithm (Theorem 4.4). The above relation shows that it is also very unlikely that an NP-complete problem is solvable by a Las Vegas algorithm whose expected running time is bounded by a polynomial.

For students who are more interested in complexity theory, the following theorem gives a more precise characterization of the class ZPP.

Theorem 9.5 *ZPP = RP $\cap$ co-RP.*

PROOF. By definition, a problem S is in co-RP if its complement $\overline{S}$ is in RP ($\overline{S}$ is defined to be the decision problem such that for all x, x is a yes-instance of $\overline{S}$ if and only if x is a no-instance of S). Note that the complement of the complement of a decision problem S is the problem S itself.

Let Q be a problem in ZPP. Lemma 9.4 claims $Q \in$ RP. Note that the complement $\overline{Q}$ of Q is also in ZPP — simply reversing the yes/no decisions in the ZPP-algorithm for Q gives a ZPP-algorithm for $\overline{Q}$. Thus, by Lemma 9.4 again, $\overline{Q}$ is in RP so Q is in co-RP. This proves ZPP $\subseteq$ RP $\cap$ co-RP.

For the other direction, let Q be a decision problem in RP $\cap$ co-RP. Then there are two Monte Carlo algorithms A_{rp} and A_{corp}, such that

- For a yes-instance x of Q, the algorithm A_{rp} returns a "yes" with a probability $\geq 1/2$,
- For a no-instance y of Q, the algorithm A_{corp} returns a "yes" with a probability $\geq 1/2$.

Without loss of generality, we assume that both algorithms A_{rp} and A_{corp} have their running time bounded by the same polynomial $p(n)$. We construct a new algorithm A_{zpp} that works as follows:

```
$A_{\text{zpp}}(x)$
  **loop**
  1. **if** $A_{\text{rp}}(x)$ = yes **then** return "yes"; stop;
  2. **if** $A_{\text{corp}}(x)$ = yes **then** return "no"; stop.
```

In case neither of the algorithms $A_{\text{rp}}(x)$ and $A_{\text{corp}}(x)$ in steps 1-2 returns "yes", the algorithm $A_{\text{zpp}}(x)$ continues for the execution of the next loop. Note that the algorithm A_{zpp} has zero error: by the assumption, both algorithms A_{rp} and A_{corp} are RP-algorithms. Thus, if $A_{\text{rp}}(x)$ returns "yes" in any execution then x must be a yes-instance of Q, and if $A_{\text{corp}}(x)$ returns "yes" in any execution then x must be a yes-instance of $\overline{Q}$, i.e., a no-instance of Q. To study the expected running time of the algorithm A_{zpp}, we assume that each execution of steps 1-2 on an input of size n takes exactly $2p(n)$ steps (we can simply modify the algorithms to let each of them stop in exactly $p(n)$ steps). Therefore, if the algorithm $A_{\text{ZPP}}(x)$ stops after h executions of steps 1-2, then the algorithm $A_{\text{ZPP}}(x)$ runs in exactly $T(n) = 2hp(n)$ steps, where $n = |x|$.

Consider a yes-instance x of the problem Q, and suppose that the algorithm A_{ZPP} stops after h executions of steps 1-2. Then for the first $h-1$ executions, the algorithm $A_{\text{rp}}(x)$ in step 1 returns "no". Since $A_{\text{rp}}(x)$ is an RP-algorithm for Q and since these $h-1$ executions of $A_{\text{rp}}(x)$ are independent, we conclude that this happends with a probability not larger than $1/2^{h-1}$. This gives the following formula for computing the expectation of the running time $T(n)$ of A_{ZPP} (using Definition in (20)):

$$\mathbf{E}[T(n)] = \sum_{T(n)=2h\cdot p(n)} (2h \cdot p(n)) \cdot \Pr[T(n) = 2h \cdot p(n)] \leq 2p(n) \sum_{h\geq 1} \frac{h}{2^{h-1}} \leq 8p(n),$$

where we have used the formula $\sum_{h=1}^{\infty} h/2^h = 2$ (please verify this by yourself). Therefore, on the yes-instance x of the problem Q, the expected running time of the algorithm A_{ZPP} is $O(p(n))$.

Similarly, for a no-instance y of the problem Q, since the algorithm A_{corp} is an RP-algorithm for $\overline{Q}$, the probability that the algorithm A_{ZPP} stops after h executions of steps 1-2, i.e., the first $h-1$ independent executions of the algorithm A_{corp} on y all return "no" is bounded by $1/2^{h-1}$. From this we can derive that the expected running time of the algorithm A_{ZPP} is $O(p(n))$.

This shows that the algorithm A_{ZPP} has its expected running time bounded by $O(p(n))$, where p is a polynomial. Since A_{ZPP} has zero error, A_{ZPP} is a ZPP-algorithm for the problem Q. Since Q is an arbitrary problem in RP $\cap$ co-RP, we conclude that RP $\cap$ co-RP $\subseteq$ ZPP. This completes the proof for the equality RP $\cap$ co-RP = ZPP $\square$

We remark that the precise relationships between the complexity classes ZPP, RP, and BPP are all unknown, which are all significant open problems in the research of complexity theory. In particular, it is unknown whether ZPP = RP and/or RP = BPP.

In the discussions starting from now, we will extend our notations of the types of randomized algorithms and of randomized complexity classes to search or optimization problems, whenever there is no serious confusion. Therefore, a ZPP-algorithm for a search or optimization problem is an algorithm that always returns a correct solution to the input instance, with its expected running time bounded by a polynomial of the input size. For instance, we can say that R-QuickSort is a ZPP-algorithm for the Sorting problem. Similarly, an RP-algorithm for a search or optimization problem Q is an algorithm that runs in polynomial time in the worst case, and returns a correct solution to the input instance with a probability at least 1/2. The algorithm Karger (Algorithm 3 in Section 1) is an example of an RP-algorithm. We can also similarly define a BPP-algorithm if the algorithm runs in polynomial time (in the worst case), and with a high probability (e.g., $\geq 3/4$), either returns a correct solution to the input instance or reports correctly that no meaningful solution exists for the input instance.