Computer Science 351

Application: The Analysis of Algorithms

Instructor: Wayne Eberly

Department of Computer Science
University of Calgary

Lecture #22

Learning Goals

- Learn about discrete probability theory can be applied to carry out an ***average case analysis*** of a deterministic algorithm, and to analyze various kinds of ***randomized algorithms***.
- Learn about various kinds of randomized algorithms for decision problems *that are allowed to fail* in various ways.

Example: Linear Search

Consider a ***linear search algorithm*** (given as pseudocode):

```
integer search (integer[] A, integer key) {
1.  integer n := A.length
2.  integer i := 0
3.  while (i < n) {
4.    if (A[i] == key) {
5.      return i
      }
6.    i := i + 1
    }
7.  throw a NoSuchElementException
}
```

Example: Linear Search

- To simplify our analysis, suppose we count the number of numbered steps that are carried out when this algorithm is executed.
    - If $0 \leq i \leq n-1$ and the first copy of `key` is in position i then $3i+5$ steps are executed.
    - If there is no copy of `key` stored in the array, then $3n+3$ steps are executed.

Worst-Case Analysis

- If an algorithm is deterministic — like the “linear search” algorithm here — then the number of steps that is used, on any given input, is a constant that only depends on that input.
- We often want to measure, or bound, the number of steps used as a function of the “size” of the input.
- For this problem, let us *define* the “size” of the input to be the length, n, of the array `A` that is part of the input.
- The ***worst-case running time*** of an algorithm is the *maximum* number of steps that is used by algorithm when it is executed on an input with a given size.

Worst-Case Analysis

- It follows from the above that — if "running time" and "size" are defined as shown here — then the ***worst-case running time*** of this linear search algorithm is the maximum of

$$\max_{0 \le i \le n-1} (3i + 5) \quad \text{and} \quad 3n + 3,$$

that is, $3n + 3$.

Average-Case Analysis

- Sometime, the “worst-case running time” seems too pessimistic because you almost never have an execution of the algorithm that uses the number of steps given as its “worst-case running time” — and you are interested in (and satisfied by) knowing how many steps are used, most of the time, instead.
- An ***average-case analysis*** might be more helpful here.

Average-Case Analysis

In order to perform an average-case analysis for a given input size n, consider an ***experiment*** in which you are executing your algorithm on an input with size n.

- The ***sample space*** Ω should generally include *sets* of inputs with size n, corresponding to different kinds of executions of the algorithm.
- For the "linear search" algorithm we might use a sample space

  $$\Omega = \{s_0, s_1, s_2, \dots, s_{n-1} u\}$$

  such that
  - for $0 \le i \le n-1$, s_i includes all inputs $\mathtt{A}$ (with length n) and $\mathtt{key}$ such that $\mathtt{A}[j] \ne \mathtt{key}$ for $0 \le j \le i-1$, and $\mathtt{A}[i] = \mathtt{key}$.
  - u includes all inputs $\mathtt{A}$ (with length n) and $\mathtt{key}$ such that $\mathtt{A}[i] \ne \mathtt{key}$, for every integer i such that $0 \le i \le -1$.

Average-Case Analysis

For $\mu \in \Omega$ let $T(\mu)$ be the number of steps used by an execution of the "linear search" algorithm when it is executed on an input from set μ.

- If $0 \leq i \leq n-1$ and $\mu = s_i$ then the loop is executed $i+1$ times (with the `key` found during the last of these execution) — so that $T(s_i) = 3i + 5$.
- If $\mu = u$ then the loop is executed n times, without the `key` being found, and there is one more step after that — $T(u) = 3n + 3$.

Thus the number of steps used, or "running time", has been expressed as a ***random variable*** $T : \Omega \to \mathbb{N}$.

Average-Case Analysis

In order to complete an average-case analysis we need to know — or, more generally, make an ***assumption*** — about how likely each outcome in Ω is.

- This is modelled by a ***probability distribution*** $\mathrm{P} : \Omega \to \mathbb{R}$.
- The "expected running time" (or results of this "average-case analysis") is the ***expected value*** $\mathrm{E}[T]$ of the random variable T with respect to the probability distribution P.
- This can depend, quite heavily, on the distribution P being used — and can be "technically correct" but also ***misleading*** or ***irrelevant*** if the assumptions about likelihoods are not correct.

Average-Case Analysis

Example: Suppose that $\mathsf{P} : \Omega \to \mathbb{R}$ is the ***uniform distribution***, so that

$$\mathsf{P}(s_i) = \mathsf{P}(u) = \tfrac{1}{|\Omega|} = \tfrac{1}{n+1}.$$

Then

$$\begin{aligned}
\mathsf{E}[T] &= \sum_{\mu \in \Omega} T(\mu) \times \mathsf{P}(\mu) \\
&= \sum_{i=0}^{n-1} T(s_i) \times \mathsf{P}(s_i) + T(u) \times \mathsf{P}(u) \\
&= \sum_{i=0}^{n-1} (3i+5) \times \tfrac{1}{n+1} + (3n+3) \times \tfrac{1}{n+1}
\end{aligned}$$

Average-Case Analysis

$$
\begin{aligned}
&= \frac{3}{n+1} \times \sum_{i=0}^{n-i} i + \frac{5}{n+1} \sum_{i=0}^{n-1} 1 + 3 \\
&= \frac{3}{n+1} \times \frac{n(n-1)}{2} + \frac{5}{n+1} \times n + 3 \\
&= \frac{3n^2 - 3n + 10n + 6n + 6}{2(n+1)} \\
&= \frac{3n^2 + 13n + 6}{2(n+1)} \\
&= \tfrac{3}{2} n + 5 - \tfrac{2}{n+1}.
\end{aligned}
$$

Average-Case Analysis

Example: Suppose, instead, that successful searches are extremely likely and, furthermore, the `key` is almost always near the beginning of the array. In particular, suppose that $\mathrm{P} : \Omega \to \mathbb{R}$ such that

$$\mathrm{P}(s_i) = 2^{-i-1} \quad \text{for } 0 \leq i \leq n-1$$

and $\mathrm{P}(u) = 2^{-n}$. Then

$$\begin{aligned} \mathrm{E}[T] &= \sum_{\mu \in \Omega} T(\mu) \times \mathrm{P}(\mu) \\ &= \sum_{i=0}^{n-1} T(s_i) \times \mathrm{P}(s_i) + T(u) \times \mathrm{P}(u) \end{aligned}$$

Average-Case Analysis

$$
\begin{aligned}
&= \sum_{i=0}^{n-1} 2^{-i-1}(3i+5) + 2^{-n}(3n+6) \\
&= 3 \times \sum_{i=0}^{n-1} \frac{i}{2^{i+1}} + 5 \times \sum_{i=0}^{n-1} \frac{1}{2^{i+1}} + 3 \times \frac{n}{2^n} + \frac{6}{2^n} \\
&\leq 3 \times \sum_{i\geq 0} \frac{i}{2^{i+1}} + 5 \times \sum_{i\geq 0} \frac{1}{2^{i+1}} + 3 \times \frac{n}{2^n} + \frac{6}{2^n} \\
&\leq 3 \times 2 + 5 \times 1 + 3 \times 1 + 3 \\
&= 17.
\end{aligned}
$$

Conclusion: Assumptions made about the likelihood of outcomes can significantly effect the results about expected running times of algorithms that one can obtain.

Average-Case Analysis

Where Might You See Average-Case Analysis?

- In CPSC 331 a deterministic version of a `QuickSort` algorithm might be given, using assumptions about whether entries in the input array distinct, and the relative orderings of the entries in the input array.
- This course might include average-case analyses involving ***hash tables*** and ***randomly constructed*** binary search trees.

Randomized Algorithms

Suppose, now, that a ***Boolean*** array $\mathtt{A}$, with length n, is given as input.

- We are trying to find some integer i such that $\mathtt{A}[i] = \mathtt{true}$ — but the algorithm is ***allowed to give up***: It is acceptable to report failure (by throwing an exception) even if `true` is an entry somewhere in the array, but we do not find it.
- Consider the ***randomized algorithm*** — which can choose an integer randomly from a given finite set (whose size might depend on the algorithm's input) that is shown on the following slide.

Randomized Algorithms

```
integer rSearch (boolean[] A) {
   integer n := A.length
   integer i := 0
   while (i < n) {
     Choose j uniformly from the set {0, 1, 2, ..., n − 1}
     — independently from any previous selections.
     if (A[j]) {
       return j
     }
     i := i + 1
   }
   throw a NoSuchElementException
}
```

Randomized Algorithms

- This is an example of a ***randomized algorithm***: It uses a random number generator during its execution so that neither the output it returns, nor the number of steps it uses to generate this output is fixed, even when the input is.
- If $\texttt{A[j]} = \texttt{false}$ for every integer $\texttt{j}$ such that $0 \leq \texttt{j} \leq n-1$ then — even though choices of the values for $\texttt{j}$ might be different for different executions of the algorithm — the test at line 5 can never pass, so that there will always be n executions of the body of the loop, and the step at line 8 will always be reached and executed.
- This can be used to argue that there will always be exactly $4n+3$ executions of numbered steps, in this case.

Randomized Algorithms

- Suppose `A` is an array such that `A[j]` $=$ `true` for every integer `j` such that $0 \leq \texttt{j} \leq n-1$, instead. This time, the test at line 5 must pass during the first execution of the loop.
- The output that is returned might be any integer `j` such that $0 \leq \texttt{j} \leq n-1$ — and each is returned with probability $\frac{1}{n}$. However, the number of steps executed is fixed: Since there is always exactly one execution of the loop body, each execution of the algorithm would include an execution of exactly six numbered steps.

Randomized Algorithms

- Now consider a more general — and somewhat more complicated — case: The sets

  $$S_{\mathtt{A}} = \{j \in \mathbb{N} \mid 0 \leq j \leq n-1 \text{ and } \mathtt{A}[j] = \mathtt{true}\}$$

  and

  $$F_{\mathtt{A}} = \{j \in \mathbb{N} \mid 0 \leq j \leq n-1 \text{ and } \mathtt{A}[j] = \mathtt{false}\}$$

  are both non-empty. Let $k = |S_{\mathtt{A}}|$, so that $1 \leq k \leq n-1$ and $|F_{\mathtt{A}}| = n - k$.
- Now neither the output, nor the number of steps used is fixed — even though the input is.

Randomized Algorithms

Consider, now, the ***experiment*** of executing the `rSearch` algorithm on an input including the array `A` as described above.

- The ***sample space*** would include enough information, about the random choices made, so that the execution of the algorithm could be studied.
- In particular, the sample space could be set to be

$$\Omega = \Omega_0 \cup \Omega_1 \cup \cdots \cup \Omega_{n-1} \cup \Omega_n$$

where $\Omega_0, \Omega_1, \ldots, \Omega_{n-1}, \Omega_n$ are as follows.

Randomized Algorithms

Ω_0 includes executions such that an index j, such that `A`[j] is `true`, is found immediately.

- This would include each sequence $\langle j_1 \rangle$ of values, with length one, where $j_1 \in S_{\mathtt{A}}$ — so that $|\Omega_0| = |S_{\mathtt{A}}| = k$.
- Each outcome in Ω_0 would have probability $\frac{1}{n}$, so that (for the probability distribution $\mathrm{P} : \Omega \to \mathbb{R}$ now being defined) $\mathrm{P}(\Omega_0) = \frac{k}{n}$.
- For each outcome $\mu \in \Omega_0$, the number of steps used in the execution of the algorithm, when the random values used are given by μ, is 6 — so that $T(\mu) = 6$, when $T : \Omega \to \mathbb{N}$ is the number of steps used.

Randomized Algorithms

For $1 \leq i \leq n-1$, Ω_i includes executions such that an index j is found, such that `A`$[j]$ is true, during the $i+1^{\text{st}}$ execution of the body of the loop.

- This would include each sequence

$$\langle j_1, j_2, \ldots, j_{i+1} \rangle$$

with length $i+1$, where $j_1, j_2, \ldots, j_i \in F_{\mathtt{A}}$ and $j_{i+1} \in S_{\mathtt{A}}$ — so that $|\Omega_i| = |F_{\mathtt{A}}|^k \cdot |S_{\mathtt{A}}| = (n-k)^i \cdot k$.

Randomized Algorithms

- Each outcome in Ω_i would have probability $n^{-(i+1)}$, so that (for the probability distribution being defined),

$$\mathsf{P}(\Omega_i) = \left(1 - \frac{k}{n}\right)^i \cdot \frac{k}{n}.$$

- For each outcome $\mu \in \Omega_i$, the number of steps used during the corresponding execution of the algorithm would be $T(\mu) = 4i + 6$.

Randomized Algorithms

Ω_n includes executions where no index j, such that $\mathtt{A}[j] = \mathtt{j} = \mathtt{true}$, is ever found at all.

- This would include each sequence

  $$\langle j_1, j_2, \dots, j_n \rangle$$

  with length n such that $j_1, j_2, \dots, j_n \in F_{\mathtt{A}}$ — so that $|\Omega_n| = |F_{\mathtt{A}}|^n = (n-k)^n$.

Randomized Algorithms

- Each outcome in Ω_n would have probability n^{-n}, so that (for the probability distribution being defined),

$$\mathrm{P}(\Omega_n) = \left(1 - \frac{k}{n}\right)^n.$$

- For each outcome $\mu \in \Omega_n$, the number of steps used during the corresponding execution of the outcome would be $T(\mu) = 4n + 3$.

Randomized Algorithms

- With a bit of work, you can verify that a probability distribution

$$\mathsf{P} : \Omega \to \mathbb{R}$$

has been defined.

- The number of steps used by the execution of the algorithm on the input $\mathtt{A}$ has been expressed as a ***random variable***

$$T : \Omega \to \mathbb{N}.$$

Randomized Algorithms

- The ***expected value*** of this random variable can be seen to be

$$
\begin{aligned}
\mathrm{E}[T] &= \sum_{\mu \in \Omega} T(\mu) \cdot \mathrm{P}(\mu) \\
&= \sum_{i=0}^{n-1} \mathrm{P}(\Omega_i) \cdot (4i+6) \;+\; \mathrm{P}(\Omega_n) \cdot (4n+3) \\
&= \sum_{i=0}^{n-1} \left(1 - \frac{k}{n}\right)^i \cdot \frac{k}{n} \cdot (4i+6) + \left(1 - \frac{k}{n}\right)^n \cdot (4n+3).
\end{aligned}
$$

Randomized Algorithms

This is certainly complicated, and not very helpful!

If time permits, the lecture presentation will include a discussion of how to deal with results like this one.

Randomized Algorithms

Definition:

- The ***expected running time*** of a randomized algorithm's execution, on a given input, is the expected value of the random variable, representing its running time, when modelled as suggested above (so that the "sample space" models the random values that are generated as an execution of the algorithm proceeds).
- The ***worst-case expected running time*** of a randomized algorithm is a function of the *size* of an input, just as the "worst-case running time" of a deterministic algorithm is: It is, essentially, the "maximum" of the expected running times of the algorithm's executions on inputs of the given size.

Randomized Algorithms

Where You Might See This Kind of Analysis?

- In CPSC 331 a *randomized* version of a `QuickSort` algorithm is also given. The process described here is used to bound the expected number of steps used by this algorithm to sort an array of length n. This is then used to bound the "worst-case expected running time" of this randomized algorithm.
- This application, from CPSC 331, is a little different than the above example because *the randomized algorithm, being considered, never gives up*.

Decision Problems

A ***decision problem*** is a computational problem that answers a "Yes-or-No" question — so that the algorithm's output is always either `true` (corresponding to the answer "Yes") or `false` (corresponding to the answer "No").

Example: Consider *another* version of "Searching in an Integer Array":

- As with the previous "Linear Search" example, the inputs are an integer array `A` and an integer `key`.
- For this problem, we are asking whether the `key` is stored in the array — so that the desired output is `true` if there an integer i such that $0 \leq i < \texttt{A.length}$ and $\texttt{A}[i] = \texttt{key}$ — and the desired output is `false`, otherwise.

Las Vegas Algorithms

A ***Las Vegas*** algorithm is a randomized algorithm that can never return an incorrect answer — so that, when an execution ends, the output provided (either `true` or `false`) is correct.

- The number of steps executed by this algorithm, when it is run on a given input, is a ***random variable*** over a sample space defined using the "random" choices made during the algorithms's executions — just as it is for other randomized algorithms.

Example: This is *Not* a Las Vegas Algorithm

Consider a modified version, `rSearch2`, of the randomized algorithm from the previous example — which searches for a key in an integer array instead of looking for a copy of "`true`".

- Rather than checking whether `A`[*i*] is `true` for a sequence of randomly chosen indices *i* it checks whether `A`[*i*] is equal to the input `key`.
- Rather than throwing an `NoSuchElementException` if the desired value is not found at the end, the algorithm returns `false`.

Pseudocode for this algorithm is on the following slide.

Example: This is *Not* a Las Vegas Algorithm

```
boolean rSearch2 (integer[] A, integer key) {
1.  integer n := A.length
2.  integer i := 0
3.  while (i < n) {
4.    Choose j uniformly from the set {0, 1, 2, ..., n − 1}
      — independently from any previous selections.
5.    if (A[j] == key) {
6.      return true
      }
7.    i := i + 1
    }
8.  return false
}
```

Example: This is *Not* a Las Vegas Algorithm

This is not a Las Vegas algorithm!

- To see why, notice that if the `key` is stored in the array — at some position ℓ such that $0 \leq \ell \leq \texttt{A.length} - 1$ — but it is not stored anywhere else, and the value ℓ is never chosen as the value for `j` at line 4 when this step is executed, then the step at line 8 will eventually be reached.
- In this case the incorrect output `false` will be returned, when a ***Las Vegas algorithm*** would be required to return the correct output, `true`, instead.

Example: This *is* a Las Vegas Algorithm

Suppose we make one more change to the above algorithm:

- Instead of returning "`false`" if attempts to locate the `key` and the step at line 8 is reached, a minor variant of the ***deterministic*** "Linear Search" algorithm, considered at the beginning of this lecture (which returns the desired output — `true`, or `false`) is used to solve the problem.
- Then, since the variant of the "Linear Search" algorithm is correct, the resulting randomized algorithm's output would also be correct — so that it *would* be a ***Las Vegas*** algorithm.

Pseudocode for the resulting algorithm, "`rSearch3`", is on the following slide.

Example: This *is* a Las Vegas Algorithm

```
boolean rSearch3 (integer[] A, integer key) {
  integer n := A.length
  integer i := 0
  while (i < n) {
    Choose j uniformly from the set {0, 1, 2, ..., n − 1}
    — independently from any previous selections.
    if (A[j] == key) {
      return true
    }
    i := i + 1
  }
  return dSearch(A, key)
}
```

This *is* a Las Vegas Algorithm

The variant of the "Linear Search" algorithm, called as a subroutine by the randomized algorithm, is as follows.

```
integer dSearch (integer[] A, integer key) {
1.  integer n := A.length
2.  integer i := 0
3.  while (i < n) {
4.    if (A[i] == key) {
5.      return true
      }
6.    i := i + 1
    }
7.  return false
}
```

Example: This *is* a Las Vegas Algorithm

- The analysis of the ***expected running time*** of the above algorithms, on a given input, as well as the ***worst-case running times*** of these algorithms, will be considered in the lecture presentation.

Monte Carlo Algorithms

A ***Monte Carlo*** algorithm is a randomized algorithm that can, sometimes, return an incorrect answer — but that does so with small probability.

- The algorithm only returns `true`, when executed on a given input, if this is the correct answer for that input. That is, if the algorithm is executed on an input where the answer that *should* be returned is `false`, then the algorithm is guaranteed to return `false`.
- If the algorithm is executed on an input where the answer that *should* be returned is `true`, then the probably that the algorithm *does* return `false` is at least $\frac{1}{2}$.
- *Another Way to Think about This:* This algorithm can generate ***false negatives*** but it only does so with small probability — and it cannot return ***false positives***, at all.

Monte Carlo Algorithms

Consider the randomized algorithm `rSearch2`, which was given above as an example of a randomized algorithm that is ***not*** a Las Vegas algorithm.

- As noted above, this algorithm only returns `true` when the input `key` is stored in the input array `A` — so that there are no ***false positives***.
- As shown in a supplemental document for this lecture it can be established that if the input `key` is stored in the input array `A` — so that `true` should be returned — then the probability that `false` is returned instead (so that there has been a ***false positive***) is less than $\frac{1}{2}$.
- This is, therefore, an example of a ***Monte Carlo algorithm***.

Two-Sided Error

- While they are not discussed as often, randomized algorithms that allow ***false positives*** with small probability, but never allow ***false negatives***, are sometimes of interest.
- Randomized algorithms that allow *both* “false positives” *and* “false negatives” are of interest too — but only if the likelihood of a mistake is significantly reduced!
- In particular, randomized algorithms that allow both false negatives and false positives *but where the probability of an incorrect result is never more than $\frac{1}{4}$, for any input,* are also of some interest.
- This will be considered, further, in the tutorial exercise for this topic.

What *Really* Happens?

As repeatedly stated: The results obtained, using probability theory, are only relevant if the assumptions being made — which are used to define probability distributions — are satisfied.

- The results are “technically correct” but ***irrelevant***, otherwise.
- Thus the results of average-case analyses of deterministic algorithms might not be relevant, because the distribution of inputs might be different than assumed.
- Results concerning randomized algorithms are even more problematic because ***modern programming languages, in widespread use, do not really provide random number generators!***

What *Really* Happens?

- Instead, these use ***deterministic*** processes that might take a small amount of information (like the time of day, machine load, or a user-supplied value) as a "seed" and use this to produce a sequence of values whose properties "approximate" those of a sequence of randomly generated values, in some sense.
- ***Results observed in practice*** typically agree with the results that analyses supplied using probability theory, assuming the use of truly "random" sequences.
- A further discussion of this is beyond the scope of this course — but students who are interested in this can investigate ***pseudorandom number generators*** if they wish to learn more.