Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
November 10th, 2024

3.1: Symbol Tables

Exercise 1. Write a client that creates a symbol table mapping letter grades to numerical scores, as in the table below, then reads from standard input a list of letter grades and computes and prints the GPA (the average of the numbers corresponding to the grades).

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

Solution. See `com.segarciat.algs4.ch3.sec1.ex01.GPA`

Exercise 2. Develop a symbol-table implementation `ArrayST` that uses an (unordered) array as the underlying data structure to implement our basic symbol-table API.

Solution. See `com.segarciat.algs4.ch3.sec1.ex02.ArrayST`.

Exercise 3. Develop a symbol-table implementation of `OrderedSequentialSearchST` that uses an ordered linked list as the underlying data structure to implement our ordered symbol-table API.

Solution. See `com.segarciat.algs4.ch3.sec1.ex03.OrderedSequentialSearchST`.

Exercise 4. Develop `Time` and `Event` ADTs that allow processing of data as in the example illustrated on page 367.

Solution. See `com.segarciat.algs4.ch3.sec1.ex04.Time`. The class is immutable, and it it implements `Comparable<Time>`, so that it has a natural order. I was unclear about what an `Event` ADT would include, so I did not provide an implementation for this ADT.

Exercise 5. Implement `size()`, `delete()`, and `keys()` for `SequentialSearchST`.

Solution. See `com.segarciat.algs4.ch3.sec1.ex05.SequentialSearchST`.

Exercise 6. Give the number of calls to `put()` and `get()` issued by `FrequencyCounter`, as a function of the number W of words and the number D of distinct words in the input.

Solution. The following assumes that the minimum length accepted for a word is 1,.

During the first phase, the program builds the symbol tables by processing all W words. For each word, there is a call to `contains()`, for a total of W such calls. Since a call to `put()` is made regardless of the result, there are W calls to `put()` during this phase. Each result of `false` from the call to `contains()` corresponds to a distinct word, so there are D such outcomes. Thus, there are $W - D$ direct calls to `get()` in the branch of the `if-else`, where `get()` is used to retrieve the count of a previously-seen word. Note also that each call to `contains()` leads to a call to `get()`, accounting for W more calls.

In the second phase, one addition call to `put()` is made, which enters the empty string, so that there are now $D+1$ keys in the symbol table. In the loop, 2 calls to `get()` are made in each iteration, for a total of $2(D+1)$ calls. A final call to `get()` is made after the loop.

Thus, if f is the number of calls made to `put()`, and g is the number of calls made to `get()`, then

$$\begin{aligned} f(W,D) &= W+1 \\ g(W,D) &= W+(W-D)+2(D+1)+1 \\ &= 2W+D+3 \end{aligned}$$

Exercise 7. What is the average number of distinct keys that `FrequencyCounter` will find among N random nonnegative integers less than 1,000, for $N = 10, 10^2, 10^3, 10^4, 10^5$, and 10^6?

Solution. Consider the random experiment of picking N integers at random, where each integer is between 0 and 999, and is chosen independently of the other. Then each outcome is an N-tuple, where each component is an integer between 1 and 1,000. Let X be a random variable that counts the number of distinct keys in an N-tuple. Then X is a discrete random variables, whose values range from 1 through $\min\{N, 1000\}$.

Consider the number of outcomes with $X = k$ distinct integers. If k is not an integer, or $k > \min\{N, 1000\}$, or $k \leq 0$, then there are 0 such outcomes. Otherwise, we can count in two steps:

(i) Choose k distinct integers. There are $\binom{1000}{k}$ ways of doing this, for $1 \leq k \leq 1000$, and 0 for $k > 1000$.

(ii) Having chosen the k distinct keys, there are $\binom{N}{k} \cdot k!$ possible positions for them.

(iii) To ensure k distinct integers, each of the remaining integers must be one of the k integers we have seen before. Since there are $N-k$ positions to fill, and k integers to choose from, there are k^{N-k} ways to do this (we repeat an experiment of choosing among k values a total of $N-k$ times).

By the multiplication principle of counting, we find that there are $\binom{1000}{k} \cdot \binom{N}{k} \cdot k^{N-K}$ ways to choose k distinct integers when choosing a total of N integers. There are a total of 1000^N outcomes. Since every outcome is equally likely, the probability of an outcome having k distinct keys is therefore equal to:

$$P(\{X=k\}) = \frac{\binom{1000}{k} \cdot \binom{N}{k} \cdot k! \cdot k^{N-k}}{1000^N}, \quad 1 \leq k \leq \min\{1000, N\}, \quad N \text{ fixed.}$$

Unfortunately, writing a closed form for this is difficult, and so is evaluating it as-is, and it's hard to verify its correctness. My goal was then to compute the expectation as:

$$E[X] = \sum_{k=1}^{\min\{N,1000\}} P(\{X=k\})$$

I found an alternative approach in this Stack Overflow answer by qwr. The idea is to let $A = \{0, \ldots, 999\}$, the set of numbers that we choose from (sample) at random, and the let X_j be an indicator random variable. That is, if we sample N elements from A, then $X_j = 1$ if j is in the sample, and 0 otherwise, where $j \in A$.

The idea is that we have a collection of random variables $X_0, \ldots, X_{999}$, and since we sample at random. Then, by defining $X = \sum_{j=0}^{999} X_j$, we obtain a random variable X that gives the number of distinct keys in the sample. Then, we use the fact that the expectation is linear (regardless of independence), meaning:

$$E[X] = E\left[\sum_{j=0}^{999} X_j\right] = \sum_{j=0}^{999} E[X_j]$$

Therefore, this reduces the problem of finding the expectation of X (the average number of distinct values in a sample) to finding the expectation of the indicator random variables. The latter turns out to be simple, and user `dwr` argues as follows. Consider the number of ways to choose a sample of N items without j in it. There are $1000-1$ other numbers, and N numbers to choose (with replacement), so this amounts to $(1000-1)^N$ choices. Meanwhile, there's 1000^N ways to choose N integers from the sample.

Thus,

$$\mathrm{P}(\{X_j = 0\}) = \frac{999^N}{1000^N}$$

which means that

$$\mathrm{P}(\{X_j = 1\}) = 1 - \left(\frac{999}{1000}\right)^N$$

Now the expectation of X_j is simple:

$$E[X_j] = 0 \cdot P(\{X_j = 0\}) + 1 \cdot P(\{X_j = 1\}) = 1 - \left(\frac{999}{1000}\right)^N$$

Hence, by linearity of expectation:

$$E[X] = 1000 \cdot \left(1 - \left(\frac{999}{1000}\right)^N\right)$$

Now, plugging in the different N values:

N	10	10^2	10^3	10^4	10^5	10^6
Average	9.96	95.21	632.30	999.95	1000.00	1000.00

See `com.segarciat.algs4.ch3.sec1.ex07.AverageDistinct`, which verifies these results.

Exercise 8. What is the most frequency used word of ten letters or more in *Tale of Two Cities*?

Solution. It is `monseigneur`, which I found out by running the program directly. See `com.segarciat.algs4.ch3.sec1.ex08`.

Exercise 9. Add code to `FrequencyCounter` to keep track of the *last* call to `put()`. Print the last word inserted and the number of words that were processed in the input stream prior to this intersection. Run your program for `tale.txt` with length cutoffs 1, 8, and 10.

Solution. See `com.segarciat.algs4.ch3.sec1.ex09.FrequencyCounter`.

Exercise 12. Modify `BinarySearchST` to maintain one array of `Item` objects that conntain keys and values, rather than two parallel arrays. Add a constructor that takes an array of `Item` values as argument and uses mergesort to sort the array.

Solution. See `com.segarciat.algs4.ch3.sec1.ex12.BinarySearchST`.

Exercise 13. Which of the symbol-table implementations in this section would you use for an application that does 10^3 `put()` operations and 10^6 `get()` operations, randomly intermixed? Justify your answer.

Solution. I would use a `BinarySearchST` because the relative number of search operations is far greater. It's true that they are intermixed, which means the cost of an insert (`put()`) is linear in the current length off the symbol-table. However, if we were to use a `SequentialSearchST`, both operations are have a linear time complexity on the average. Meanwhile, logarithmic performance of the search operation for `BinarySearchST` gives us an edge since we have such a high proportion of searches.

Exercise 14. Which of the symbol-table implementations would you use for an application that does 10^6 `put()` operations and 10^3 `get()` operations, randomly intermixed? Justify your answer.

Solution. Although inserts are slow for `BinarySearchST`, I would still opt for them in this scenario because `SequentialSearchST` perform poorly on large symbol-tables. For example, to find out that a key is not in the linked list, we must search the entire linked list.

Exercise 16. Implement the `delete()` method for `BinarySearchST`.

Solution. See `com.segarciat.algs4.ch3.sec1.ex16.BinarySearchST`.

Exercise 17. Implement the `floor()` method for `BinarySearchST`.

Solution. See `com.segarciat.algs4.ch3.sec1.ex17.BinarySearchST`.

Exercise 18. Prove that the `rank()` method in `BinarySearchST` is correct.

Solution.

Proof. Suppose that `key` is not `null`, and that the array we are searching for `key` is ordered with all keys distinct.

If after computing `mid`, the index of the middle of key, we find that the array at that index matches `key`, then the algorithms stops. Since the array is ordered an all keys are unique, any item at an index larger than `mid` cannot be less than `key`, and all keys at an index less than `mid` compare less than `key`. Therefore, the number of keys in the array that are less than the key at index `mid` is precisely `mid`, because arrays are 0-indexed.

Suppose that after comparison we find that `cmp < 0`. This means that `keys[mid]` is larger than `key`. In reducing the search space to `lo..mid-1`, the value of `lo` (which is its known rank so far) does not change, consistent with the fact that all keys at an index higher than or equal to `mid` are strictly larger than `key`. That is, they cannot contribute ot the rank.

Suppose that after comparison we find that `cmp > 0`. That is, `key` compares larger than `keys[mid]`. By the transitivity of the total ordering of `compareTo()`, this means `key` is larger than the keys in the index range `0..mid`, which constitutes a total of `mid + 1` keys. This is consistent with the update of `lo` to `mid + 1` in this range.

After every iteration, we maintain the invariant that `key` is greater than all of the keys in the index range `0..lo-1`, a total of `lo` keys. The search eventually ends because the length of the interval reduces by half each time, and when the length is 0, it reduces by 1, so that eventually the condition `lo <= hi` becomes `false`. With the invariant maintained after each iteration, we are certain that `lo` indeed represents the number of keys less than `key`, its rank. □

Exercise 19. Modify `FrequencyCounter` to print all of the values having the highest frequency of occurrence, not just one of them. *Hint*: Use a `Queue`.

Exercise 20. Complete the proof of **Proposition B** (show that it holds for all values of n). *Hint*: Start by showing that $C(n)$ is monotonic: $C(n) \leq C(n+1)$ for all $n \geq 0$.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms.* 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.