Easy calculations show that

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + c\left[\frac{3}{n+1} - \frac{1}{n}\right].$$

Unfolding the recurrence yields the expression

$$T(n) = 2c(n+1)H_n - 3cn,$$

where H_n is the n-th harmonic number. But $\ln(n+1) \leqslant H_n \leqslant \ln n + 1$, so $T(n) = \Theta(n \log n)$. This implies that although randomization fails to provide any guarantee about the goodness of each partitioning task, the expected running time of the randomized quick-sort algorithm is the best that a comparison-based sorting algorithm can achieve. Moreover, this guarantee pertains not just to some or most inputs (like random) but to all inputs (including those that force worst-case behavior of the deterministic quick-sort algorithm).

25.2.4 Randomized data structures

Like algorithms, we may think of data structures that are randomized. For example, think about inserting n elements in an (initially empty) binary search tree. If the input elements are inserted in a sorted sequence, the tree achieves a height of $n-1$. There are (many) other sequences that lead to this worst-case height of the tree. One possibility to avoid this problem is the use height-balanced trees (like AVL trees). The resulting implementation (for insertion) becomes somewhat involved.

We may think about a significantly simpler randomized strategy. For the time being, assume that all the n elements to be inserted in the tree are available to us from the beginning. (Exercise 4.194 handles an incremental construction.) We first make a random permutation of the input elements, and insert the elements in that permuted order. The random permutation may yield a sequence of insertions, that leads the tree to gain the worst-case height, but the probability of such an event is rather low. Indeed, using an analysis similar to the case of randomized quick sort, we can show that the expected height of the tree becomes $\Theta(\log n)$ (solve Exercise 6.83).

In this example, we may call the binary search tree randomized. Although this view can perhaps be upheld, one may argue that in this case we have actually randomized the insertion procedure, rather than the data structure. So let me present a better example.

Consider a hash table of size s with chaining, storing n elements. Assume that, for each input, the hash function produces an element from the set $\{0, 1, 2, \ldots, s-1\}$ with probability $1/s$. In that case, the expected length of each list is $\Theta(n/s)$ which is $\Theta(1)$ if $s = \Theta(n)$ (solve Exercise 6.84). The insertion of the elements may be carried out in an incremental way, that is, unlike binary search trees, it is not necessary to know all the elements beforehand and randomly permute them.

The above data structures are randomized in the Las Vegas sense, that is, they are expected to exhibit good performance irrespective of the choice of the input. When a subsequent search is carried out, we obtain the correct result with certainty.

It is also possible to conceive of data structures that are randomized in the Monte Carlo sense. This means that insertion and subsequent searching are necessarily efficient, but the output of a search request may be incorrect with some (small) probability of error. In what follows, I explain such a data structure known as the *Bloom filter* (designed by Burton H. Bloom in 1970). Bloom filters are often used as compressed storage of data.

A bloom filer consists of a bit array A of size s. Initially, all the bits of A are set to 0. We use k random and independent hash functions $h_0, h_1, \ldots, h_{k-1}$ each producing an integer output between 0 and $s-1$ (both inclusive).

A bloom filter allows only insertion and searching. In order to insert an element x in the bloom filter, we first compute the k hash values $y_i = h_i(x)$ for $i = 0, 1, \ldots, k-1$. We then set the bits in A at positions $y_0, y_1, \ldots, y_{k-1}$.

In order to search for an element u in the filter, we compute the k hash values $v_i = h_i(u)$ for $i = 0, 1, \ldots, k-1$. If any of the bits at positions $v_0, v_1, \ldots, v_{k-1}$ in A is 0, we output *No* (that is, u is not present). If all these positions in A hold the bit 1, we output *Yes* (u is present).

When this search algorithm outputs *No*, it is definitely correct. However, an output *Yes* does not necessarily imply that u has been inserted in the filter. All the bits v_i might have been set during insertions of other elements in the filter. Such a u is often called a *false positive*. By suitably selecting s (the size of the bit table) and n (the number of elements inserted in the filter), we can make the probability of false positives as low as we desire.

Example

Suppose we want to store 3-digit integers in a bloom filter. Take a bit array A of size $s = 16$. For an input $x = (x_2 x_1 x_0)_{10}$ (in base 10), we compute the hash values $h_0(x) = (3^{x_0} \pmod{17}) - 1$, $h_1(x) = (5^{x_1} \pmod{17}) - 1$, and $h_2(x) = (7^{x_2} \pmod{17}) - 1$. Initially, the filter does not contain any element, so the bit array A is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Now, we insert 825 in the filter. We have $h_0(825) = 4$, $h_1(825) = 7$ and $h_2(825) = 15$. So the insertion of 825 updates A as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1

Then, we insert 357 for which the hash values are 10, 13 and 2, so A changes to:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	0	1	0	0	1	0	0	1	0	0	1	0	1

Finally, let us insert 471 with hash values $2, 9, 3$. The bit at position 2 in A is already set. Setting the bits at positions 3 and 9 leaves A as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	1	0	0	1	0	1	1	0	0	1	0	1

Let us now see how we can search in the filter. Let us first search for 789. Applications of the hash functions give the indices 13, 15 and 11. The bits at indices 13 and 15 are set, but the bit at index 11 is not set, implying that 789 has not been inserted in the filter.

Then, we search for 513. The hash values in this case are 9, 4 and 10. The bits of A at all these three positions are set, so we output *Yes*. This is a false positive, since 513 has not been inserted in the filter. The bits at indices 4, 9 and 10 were set during the insertions of 825, 471 and 357, respectively.

Let me now make an analysis of the probability of false-positive errors. Let s be the size of the bit table, k the number of hash functions, and n the number of elements inserted in the filter. During each insertion, each bit in A is set by each hash function with probability $1/s$. Therefore, the probability that a bit in A is set (at least once) during n insertions is

$$1-\left(1-\frac{1}{s}\right)^{kn}.$$

Now, consider an element u not inserted in the filter is searched. The k hash functions produce k uniformly random locations in A, each containing a 1-bit with the above probability. Therefore, the probability that u is a false positive is

$$\left(1-\left(1-\frac{1}{s}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/s}\right)^{k}.$$

This probability is minimized with respect to k for

$$k \approx \frac{s}{n}\ln 2,$$

and the corresponding false positive probability is about

$$2^{-k} \approx (2^{-\ln 2})^{s/n} = p,$$

where p is the allowed error probability suitable for an application. This lets us choose the size s of the bit table A as

$$s \approx -\frac{n \ln p}{\ln^2 2}.$$

25.3 Backtracking and branch-and-bound algorithms

A non-deterministic algorithm is associated with a computation tree for every instance of input. Branching in the tree represents non-deterministic choices. Each leaf in the tree is marked by *Yes* or *No* in order to indicate the decision of the unique computation path from the root to that leaf. A systematic traversal of this computation tree visits all the leaves of the tree and lets us conclude about the decision on the given input instance. Indeed, this is precisely how we emulated a non-deterministic algorithm by a deterministic one (See Chapter 24). This emulation may lead to exponential running time (and even exponential space), but may be the only known way to solve certain problems in NP. Use of suitable heuristic strategies during the traversal may often curtail the search considerably, achieving significant practical speedup.

25.3.1 Backtracking

I now elaborate this idea formally. Each node in the computation tree is uniquely specified by the path from the root to that node, that is, by the non-deterministic choices made so far. We represent