Simplified Analyses of Randomized Algorithms for Searching, Sorting, and Selection*

Michael T. Goodrich†
Dept. of Info. & Computer Science
University of California
Irvine, CA 92697-3425
goodrich @ acm.org

Roberto Tamassia‡
Dept. of Computer Science
Brown University
Providence, RI 02912
rt @ cs.brown.edu

Abstract We describe simplified analyses of well-known randomized algorithms for searching, sorting, and selection. The proofs we include are quite simple and can easily be made a part of a Freshman-Sophomore Introduction to Data Structures (CS2) course and a Junior-Senior level course on the design and analysis of data structures and algorithms (CS7/DS&A). We show that using randomization in data structures and algorithms is safe and can be used to significantly simplify efficient solutions to various computational problems.

1. INTRODUCTION

We live with probabilities all the time, and we easily dismiss as "impossible" events with very low probabilities. For example, the probability of a U.S. presidential election being decided by a single vote is estimated at 1 in 10 million[1]. The probability of being killed by a bolt of lightning in any given year is estimated at 1 in 2.5 million[2]. And, in spite of Hollywood's preoccupation with it, the probability that a large meteorite will impact the earth in any given year is about 1 in 100 thousand[3]. Because the probabilities of these events are so low, we safely assume they will not occur in our lifetime.

Why is it then that computer scientists have historically preferred *deterministic* computations over randomized computations? Deterministic algorithms certainly have the benefit of provable correctness claims and often have good time bounds that hold even for worst-case inputs. But as soon as an algorithm is actually implemented in a program P, we must again deal with probabilistic events, such as the following:

- P contains a bug,
- we provide an input to P in an unexpected form,
- our computer crashes for no apparent reason,
- P's environment assumptions are no longer valid.

Since we are already living with bad computer events such as these, whose probabilities are arguably much higher than the bad "real-world" events listed in the previous paragraph, we should be willing to accept probabilistic algorithms as well. In fact, fast randomized algorithms are typically easier to program than fast deterministic algorithms. Thus, using a randomized algorithm may actually be safer than using a deterministic algorithm, for it is likely to reduce the probability that a program solving a given problem contains a bug.

*This work was announced in preliminary form in the Proceedings of the 13th SIGCSE Technical Symposium on Computer Science Education, 1999, 53–57.
†The work of this author was supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grant CCR–9625289.
‡The work of this author was supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grants CCR–9732327 and CDA–9703080.
[1]wizard.ucr.edu/polmeth/working_papers97/gelma97b.html
[2]www.nassauredcross.org/sumstorm/thunder2.htm
[3]newton.dep.anl.gov/newton/askasci/1995/astron/AST63.HTM

1.1 SIMPLIFYING ANALYSES

In this paper we describe several well-known randomized algorithms but provide new, simplified analyses of their asymptotic performance bounds. In fact, our proofs use only the most elementary of probabilistic facts. We contrast this approach with traditional "average-case" analyses by showing that the analyses for randomized algorithms need not make any restrictive assumptions about the forms of possible inputs. Specifically, we describe how randomization can easily be incorporated into discussions of each of the following standard algorithmic topics:

- searching in dictionaries,
- sorting,
- selection.

We discuss each of these topics in the following sections.

2. SEARCHING IN DICTIONARIES

A *dictionary* is a data structure that stores key-value pairs, called items, and supports search, insertion and deletion operations. We consider *ordered dictionaries* where an ordering is defined over the keys. An interesting alternative to balanced binary search trees for efficiently realizing the ordered dictionary abstract data type (ADT) is the *skip list* [3, 4, 7, 5, 6]. This structure makes random choices in arranging items in such a way that searches and updates take $O(\log n)$ time *on average*, where n is the number of items in the dictionary. Interestingly, the notion of average time used here does not depend on any probability distribution defined on the keys in the input. Instead, the running time is averaged over all possible outcomes of random choices used when inserting items in the dictionary.

2.1 SKIP LISTS

A *skip list* S for an ordered dictionary D consists of a series of sequences $\{S_0, S_1, \ldots, S_h\}$. Each sequence S_i stores a subset of the items of D sorted by nondecreasing key plus items with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in D and $+\infty$ is larger than every possible key that can be inserted in D. In addition, the sequences in S satisfy the following:

- Sequence S_0 contains every item of dictionary D (plus the special items with keys $-\infty$ and $+\infty$).
- For $i = 1, \ldots, h-1$, sequence S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in sequence S_{i-1}.
- Sequence S_h contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 0.1. It is customary to visualize a skip list S with sequence S_0 at the bottom and sequences $S_1, \ldots, S_{h-1}$ above it. Also, we refer to h as the *height* of skip list S.

Intuitively, the sequences are set up so that S_{i+1} contains more or less every other item in S_i. As we shall see in the details of the insertion method, the items in S_{i+1} are chosen at random from the items in S_i by picking each item from S_i to also be in S_{i+1} with probability $1/2$. That is, in essence, we "flip a coin" for each item in S_i and place that item in S_{i+1} if the coin comes up "heads." Thus, we expect S_1 to have about $n/2$ items, S_2 to have about $n/4$ items, and, in general, S_i to have about $n/2^i$ items. In other words, we expect the height h of S to be about $\log n$.

Using the *position* abstraction used previously by the authors [2] for nodes in sequences and trees, we view a skip list as a two-dimensional collection of positions arranged horizontally into *levels* and vertically into *towers*. Each level corresponds to a sequence S_i and each tower contains positions storing the same item across consecutive sequences. The positions in a skip list can be traversed using the following operations:

after(p): the position following p on the same level.

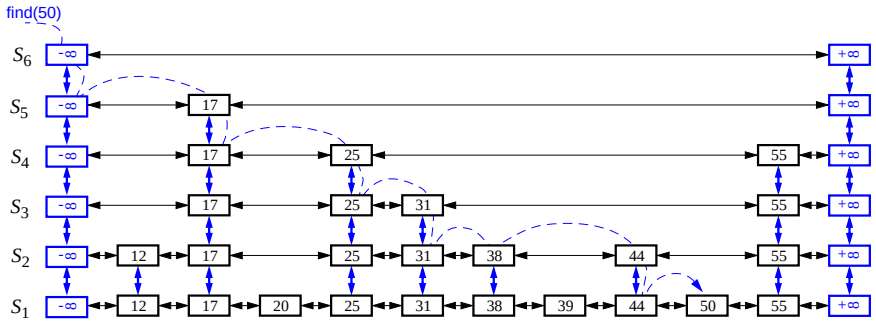

Figure 0.1: Example of a skip list. The dashed lines show the traversal of the structure performed when searching for key 50.

before(p): the position preceding p on the same level.

below(p): the position below p in the same tower.

above(p): the position above p in the same tower.

Without going into the details, we note that we can easily implement a skip list by means of a linked structure such that the above traversal methods each take $O(1)$ time, given a skip-list position p.

2.2 SEARCHING

The skip list structure allows for simple dictionary search algorithms. In fact, all of the skip list search algorithms are based on an elegant SkipSearch method that takes a key k and finds the item in a skip list S with the largest key (which is possibly $-\infty$) that is less than or equal to k. Suppose we are given such a key k. We begin the SkipSearch method by setting a position variable p to the top-most, left position in the skip list S. That is, p is set to the position of the special item with key $-\infty$ in S_h. We give a pseudo-code description of the skip-list search algorithm in Code Fragment 1 (see also Figure 0.1).

Algorithm SkipSearch(k):

Input: A search key k

Output: Position p in S_0 such that the item at p has the largest key less than or equal to k

```
Let p be the topmost-left position of S (which should have at least 2 levels).
while below(p) ≠ null do
  p ← below(p)            {drop down}
  while key(after(p)) ≤ k do
    Let p ← after(p)      {scan forward}
  end while
end while
return p.
```

Code Fragment 1: A generic search in a skip list S.

2.3 UPDATE OPERATIONS

Another feature of the skip list data structure is that, besides having an elegant search algorithm, it also provides simple algorithms for dictionary updates.

Insertion

The insertion algorithm for skip lists uses randomization to decide how many references to the new item (k, e) should be added to the skip list. We begin the insertion of a new item (k, e) into a skip list by performing a SkipSearch(k) operation. This gives us the position p of the bottom-level item with the largest key less than or equal to k (note that p may be the position of the special item with key $-\infty$). We then insert (k, e) in this bottom-level list immediately after position p. After inserting the new item at this level we "flip" a coin. That is, we call a method random() that returns a number between 0 and 1, and if that number is less than $1/2$, then we consider the flip to have come up "heads;" otherwise, we consider the flip to have come up "tails." If the flip comes up tails, then we stop here. If the flip comes up heads, on the other hand, then we backtrack to the previous (next higher) level and insert (k, e) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new item (k, e) in lists until we finally get a flip that comes up tails. We link together all the references to the new item (k, e) created in this process to create the *tower* for (k, e).

We give the pseudo-code for the above insertion algorithm in Code Fragment 2. The uses an operation insertAfterAbove$(p, q, (k, e))$ that inserts a position storing the item (k, e) after position p (on the same level as p) and above position q, returning the position r of the new item (and setting internal references so that after, before, above, and below methods will work correctly for p, q, and r). We note that this pseudo-code assumes that the insertion process never goes beyond the top level of the skip list. This is not a completely reasonable assumption, for some insertions will most likely continue beyond this level. There are two simple ways of dealing with this occurance, however. The first is to extend the height of the skip list as long as an insertion wishes to go to higher levels. We leave it as a simple exercise to show that it is very unlikely that any insertion will go beyond level $3 \log n$ in this case. The second possibility is to cut off any insertion that tries to go beyond a reasonable notion of what should be the top level of the skip list. For example, one could maintain the top level to be at height $3\lceil \log n \rceil$, and then stop any insertion from going beyond that level. One can show that the expected number of elements that would ever be inserted at this level is actually less than 1.

Algorithm SkipInsert(k, e):
  $p \leftarrow$ SkipSearch(k)
  $q \leftarrow$ insertAfterAbove$(p, null, (k, e))$
  while random$() < 1/2$ **do**
    while above$(p) = null$ **do**
      $p \leftarrow$ before(p) {*scan backward*}
    end while
    $p \leftarrow$ above(p) {*jump up to higher level*}
    $q \leftarrow$ insertAfterAbove$(p, q, (k, e))$
  end while

Code Fragment 2: Insertion in a skip list, assuming random() returns a random number between 0 and 1, and we never insert past the top level.

Removal

Like the search and insertion algorithms, the removal algorithm for a skip list S is quite simple. In fact, it is even easier than the insertion algorithm. Namely, to perform a remove(k) operation, we begin by performing a search for the given key k. If a position p with key k is not found, then we indicate an error condition. Otherwise, if a position p with key k is found (on the bottom level), then we remove all the positions above p,

which are easily accessed by using above operations to climb up the tower of this item in S starting at position p (see Figure 0.2).

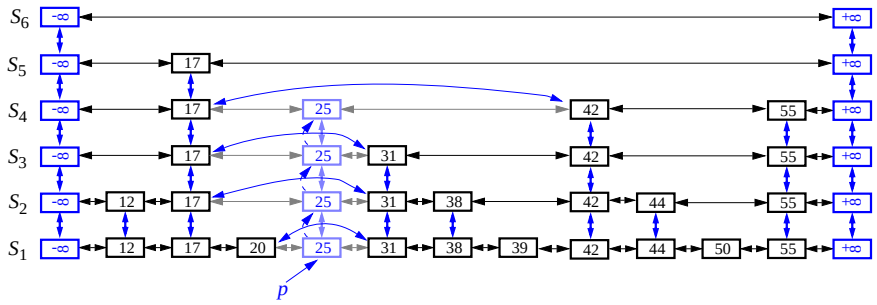

Figure 0.2: Removal of the item with key 25 from a skip list. The positions visited are in the tower for key 25.

2.4 A SIMPLE ANALYSIS OF SKIP LISTS

Our probabilistic analysis of skip lists, which is a simplified version of an analysis of Motwani and Raghavan [4], requires only elementary probability concepts, and does not need any assumptions about input distributions. We begin this analysis by studying the height h of S.

The probability that a given item is stored in a position at level i is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is $1/2^i$. Thus, the probability P_i that level i has at least one item is at most

$$P_i \leq \frac{n}{2^i},$$

for the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one item, that is, it is no more than P_i. This means that h is larger than, say, $3 \log n$ with probability at most

$$P_{3\log n} \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}.$$

More generally, given a constant $c > 1$, h is larger than $c \log n$ with probability at most $1/n^{c-1}$. Thus, with high probability, the height h of S is $O(\log n)$.

Consider the running time of a search in skip list S, and recall that such a search involves two nested **while** loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is $O(\log n)$ with high probability, the number of drop-down steps is $O(\log n)$ with high probability.

So we have yet to bound the number of scan-forward steps we make. Let n_i be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level $i + 1$. If any of these items were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in n_i is $1/2$. Therefore, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. This expected value is 2. Hence, the expected amount of time spent scanning forward at any level i is $O(1)$. Since S has $O(\log n)$ levels with high probability, a search in S takes the expected time $O(\log n)$. By a similar analysis, we can show that the expected running time of an insertion or a removal is $O(\log n)$.

vi Finally, let us turn to the space requirement of a skip list S. As we observed above, the expected number of items at level i is $n/2^i$, which means that the expected total number of items in S is

$$\sum_{i=0}^{h} \frac{n}{2^i} \;=\; n \sum_{i=0}^{h} \frac{1}{2^i} \;<\; 2n.$$

Hence, the expected space requirement of S is $O(n)$.

3. SORTING

One of the most popular sorting algorithms is the quick-sort algorithm, which uses a *pivot* element to split a sequence and then it recursively sorts the subsequences. One common method for analyzing quick-sort is to assume that the pivot will always divide the sequence almost equally. We feel such an assumption would presuppose knowledge about the input distribution that is typically not available, however. Since the intuitive goal of the partition step of the quick-sort method is to divide the sequence S almost equally, let us introduce randomization into the algorithm and pick as the pivot a *random element* of the input sequence. This variation of quick-sort is called *randomized quick-sort*, and is provided in Code Fragment 3.

Algorithm quickSort(S):

Input: Sequence S of n comparable elements

Output: A sorted copy of S

if $n = 1$ **then**
  return S.
end if
pick a random integer r in the range $[0, n-1]$
let x be the element of S at rank r.
put the elements of S into three sequences:

- S_L, storing the elements in S less than x
- S_E, storing the elements in S equal to x
- S_G, storing the elements in S greater than x.

let $S'_L \leftarrow \mathsf{quickSort}(S_L)$
let $S'_G \leftarrow \mathsf{quickSort}(S_G)$
return $S'_L + S_E + S'_G$.

Code Fragment 3: Randomized quick-sort algorithm.

There are several analyses showing that the expected running time of randomized quicksort is $O(n \log n)$ (e.g., see [1, 4, 8]), independent of any input distribution assumptions. The analysis we give here simplifies these analyses considerably.

Our analysis uses a simple fact from elementary probability theory: namely, that the expected number of times that a fair coin must be flipped until it shows "heads" k times is $2k$. Consider now a single recursive invocation of randomized quick-sort, and let m denote the size of the input sequence for this invocation. Say that this invocation is "good" if the pivot chosen is such that subsequences L and G have size at least $m/4$ and at most $3m/4$ each. Thus, since the pivot is chosen uniformly at random and there are $m/2$ pivots for which this invocation is good, the probability that an invocation is good is $1/2$.

Consider now the recursion tree T associated with an instance of the quick-sort algorithm. If a node v of T of size m is associated with a "good" recursive call, then the input sizes of the children of v are each at most

$3m/4$ (which is the same as $m/(4/3)$). If we take any path in T from the root to an external node, then the length of this path is at most the number of invocations that have to be made (at each node on this path) until achieving $\log_{4/3} n$ good invocations. Applying the probabilistic fact reviewed above, the expected number of invocations we must make until this occurs is at most $2\log_{4/3} n$. Thus, the expected length of any path from the root to an external node in T is $O(\log n)$. Observing that the time spent at each level of T is $O(n)$, the expected running time of randomized quick-sort is $O(n \log n)$.

4. SELECTION

The *selection* problem asks that we return the kth smallest element in an unordered sequence S. Again using randomization, we can design a simple algorithm for this problem. We describe in Code Fragment 4 a simple and practical method, called *randomized quick-select*, for solving this problem.

Algorithm quickSelect(S, k):

Input: Sequence S of n comparable elements, and an integer $k \in [1, n]$

Output: The kth smallest element of S

if $n = 1$ **then**
  return the (first) element of S.
end if
pick a random integer r in the range $[0, n-1]$
let x be the element of S at rank r.
put the elements of S into three sequences:

- S_L, storing the elements in S less than x
- S_E, storing the elements in S equal to x
- S_G, storing the elements in S greater than x.

if $k \leq |S_L|$ **then**
  quickSelect(S_L, k)
else if $k \leq |S_L| + |S_E|$ **then**
  return x {*each element in S_E is equal to x*}
else
  quickSelect$(S_G, k - |S_L| - |S_E|)$
end if

Code Fragment 4: Randomized quick-select algorithm.

We note that randomized quick-select runs in $O(n^2)$ *worst-case* time. Nevertheless, it runs in $O(n)$ *expected* time, and is much simpler than the well-known *deterministic* selection algorithm that runs in $O(n)$ worst-case time (e.g., see [1]). As was the case with our quick-sort analysis, our analysis of randomized quick-select is simpler than existing analyses, such as that in [1].

Let $t(n)$ denote the running time of randomized quick-select on a sequence of size n. Since the randomized quick-select algorithm depends on the outcome of random events, its running time, $t(n)$, is a random variable. We are interested in bounding $E(t(n))$, the expected value of $t(n)$. Say that a recursive invocation of randomized quick-select is "good" if it partitions S, so that the size of S_L and S_G is at most $3n/4$. Clearly, a recursive call is good with probability $1/2$. Let $g(n)$ denote the number of consecutive recursive invocations (including the present one) before getting a good invocation. Then

$$t(n) \leq bn \cdot g(n) + t(3n/4),$$

where $b \geq 1$ is a constant (to account for the overhead of each call). We are, of course, focusing in on the case where n is larger than 1, for we can easily characterize in a closed form that $t(1) = b$. Applying the linearity of expectation property to the general case, then, we get

$$E\left(t(n)\right) \leq E\left(bn \cdot g(n) + t(3n/4)\right) = bn \cdot E\left(g(n)\right) + E\left(t(3n/4)\right).$$

Since a recursive call is good with probability $1/2$, and whether a recursive call is good or not is independent of its parent call being good, the expected value of $g(n)$ is the same as the expected number of times we must flip a fair coin before it comes up "heads." This implies that $E(g(n)) = 2$. Thus, if we let $T(n)$ be a shorthand notation for $E(t(n))$ (the expected running time of the randomized quick-select algorithm), then we can write the case for $n > 1$ as $T(n) \leq T(3n/4) + 2bn$. Converting this recurrence relation to a closed form, we get that

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i.$$

Thus, the expected running time of quick-select is $O(n)$.

5. CONCLUSION

We have discussed some simplified analyses of well-known algorithms and data structures. In particular, we have presented simplified analyses for skip lists and randomized quick-sort, suitable for a CS2 course, and for randomized quick-select. These simplified analyses, in slightly different form, along with further discussions of simple data structures and algorithms, can be found in the recent book on data structures and algorithms by the authors [2].

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[2] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, New York, 1998.

[3] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31:775–792, 1994.

[4] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[5] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[6] P. V. Poblete, J. I. Munro, and T. Papadakis. The binomial transform and its application to the analysis of skip lists. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 554–569, 1995.

[7] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[8] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.