INSERTION SORT is $O(n \log n)$ *

Michael A. Bender[†] Martín Farach-Colton[‡] Miguel Mosteiro[§]

Abstract

Traditional INSERTION SORT runs in $O(n^2)$ time because each insertion takes $O(n)$ time. When people run INSERTION SORT in the physical world, they leave gaps between items to accelerate insertions. Gaps help in computers as well. This paper shows that GAPPED INSERTION SORT has insertion times of $O(\log n)$ with high probability, yielding a total running time of $O(n \log n)$ with high probability.

Keywords

Sorting, Library Sort, Insertion Sort, Gapped Insertion Sort. ACM-class: F.2.2, E.5. arXiv: cs.DS/0407003. CoRR Subj-class: DS-Data Structures and Algorithms.

1 Introduction

Success has its problems. While many technology companies are hemorrhaging money and employees, Google is flush with money and hiring vigorously. Google employees are cheerful and optimistic, with the exception of G—.

G— maintains the mailboxes at Google. The mailbox technology consist of trays arranged in linear order and bolted to the wall. The names on the mailboxes are alphabetized. G— is grumpy after each new hire because, to make room for the nth new employee, the names on $O(n)$ mailboxes need to be shifted by one.

University graduate programs in the US have also been growing vigorously, accepting as students those talented employees downsized from high-tech companies.

At Stony Brook S— implements the mailbox protocol. Each time a new student arrives, S— makes room for the new student's mailbox using the same technique as G—. However, S— only needs to shift names by one until a gap is reached, where the empty mailbox belonged to a student who graduated previously. Because the names have more or less random rank, S— does not need to shift many names before reaching a gap.

Both S— and G— are implementing INSERTION SORT. However, while S— is blissfully unaware that INSERTION SORT is an $O(n^2)$ algorithm, G— continually hopes that each new hire will be named Zhang, Zizmor, or Zyxt.

R—, the librarian at Rutgers, is astonished by all the fuss over insertions. R— inserts new books into the stacks[1] every day. R— plans for the future by leaving gaps on every shelf. Periodically, R— adds stacks to

*In Proceedings of the Third International Conference on Fun With Algorithms, FUN 2004.

[†]Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794-4400, USA; bender@cs.sunysb.edu.

[‡]Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA; farach@cs.rutgers.edu.

[§]Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA; mosteiro@cs.rutgers.edu.

[1]Throughout, we mean *library stacks*, an ordered set of stacks, rather than lifo stacks.

accommodate the growing collection. Although spreading the books onto the new stacks is laborious, these rearrangements happens so infrequently that R— has plenty of time to send overdue notices to hard-working students and professors.

This paper shows that GAPPED INSERTION SORT, or LIBRARY SORT, has insertion times of $O(\log n)$ with high probability, yielding a total running time of $O(n \log n)$ with high probability.

Standard INSERTION SORT

In standard INSERTION SORT we maintain an array of elements in sorted order. When we insert a new element, we find its target location and slide each element after this location ahead by one array position to make room for the new insertion. The ith insertion takes time $O(i)$, for a total of $O(n^2)$. Finding the target position of the ith element takes time $O(\log i)$ using binary search, though this cost is dominated by the insertion cost.

LIBRARY SORT

We achieve $O(\log n)$-time insertions with high probability by keeping gaps evenly distributed between the inserted elements and randomly permuting the input. Then we only need to move a small number of elements ahead by one position until we reach a gap. The more gaps we leave, the fewer elements we move on insertions. However, we can tolerate a small-constant-factor overheadin the space occupancy.

The remainder of this paper is organized as follows. We present the details of the algorithm in Section 2 and show in Section 3 that the algorithm runs in $O(n \log n)$ time with high probability. In Section 4 we conclude with a few comments and some related work.

2 LIBRARY SORT: Algorithm and Terminology

Let A be an n-element array to be sorted. These elements are inserted one at a time in random order into a *sorting array* S of size $(1+\varepsilon)n$. The insertions proceed in $\log n$ *rounds* as follows. Each round doubles the number of elements inserted into S and doubles the prefix of S where elements reside. Specifically, round i ends when 2^i elements have been inserted and the elements are *rebalanced*. Before the rebalance, the 2^i elements are in the first $(1+\varepsilon)2^i$ positions. A rebalance moves them into the first $(2+2\varepsilon)2^i$ positions, spreading the elements as evenly as possible. We call $2+2\varepsilon$ the *spreading factor*.

During the ith round, the 2^{i-1} elements in S at the beginning of the round are called *support elements*, and their initial positions are called *support positions*. The 2^{i-1} elements inserted before the end-of-round rebalance are called *intercalated elements*. It is important to remark that the support positions are the positions occupied by the support elements at the beginning of a round because, as explained in the next paragraph, while a round proceeds support elements might be shifted to make room for new insertions.

The insertion of 2^{i-1} intercalated elements within round i is performed the brute force way: search for the target position of the element to be inserted by binary search (amongst the 2^{i-1} support positions in S), and move elements of higher rank to make room for the new element. Not all elements of higher rank need to be moved, only those in adjacent array positions until the nearest gap is found.

3 Analysis

For the sake of clarity, we divide the time complexity into four parts: the rebalance cost, the search cost, the insertion cost for the first $\sqrt{n}$ elements, and the insertion cost for the remaining elements. Let m be the number of elements inserted at any time.

3.1 Insertion cost for $m = O(\sqrt{n})$, rebalance cost, and search cost

The following straightforward lemmas prove the upper bounds on the rebalance and search costs for any m and on the insertion cost for $m = O(\sqrt{n})$.

Lemma 1 *The insertion time for the first $O(\sqrt{n})$ insertions is $O(n)$.*

Proof. By the quadratic running time of INSERTION SORT. □

Lemma 2 *For a given input of size n, the cost of all rebalances is $O(n)$.*

Proof. Since the number of elements doubles in each round and the spreading factor is constant, the cost of spreading m elements on each rebalance is amortized over the previous $m/2$ insertions, for an amortized rebalance cost of $O(1)$ per insertion. □

Lemma 3 *The cost of finding the location to insert a new element in the sorting array is $O(\log m)$.*

Proof. Binary search among the $O(m)$ support positions takes time $O(\log m)$. A final search between two support positions takes time $O(1)$, since the spreading factor is constant. □

3.2 Insertion cost for $m = \Omega(\sqrt{n})$

We now bound the number of elements moved per insertion when $m = \Omega(\sqrt{n})$ elements have already been inserted. We show that with high probability, for sufficiently large c, all sets of $c \log m$ contiguous [2] support elements have fewer than $(1+\varepsilon)c \log m$ intercalated elements inserted among them by the end of the round. The $c \log m$ support elements are spread among $(2+2\varepsilon)c \log m$ sorting array positions at the beginning of a round. Therefore, after $(1+\varepsilon)c \log m$ intercalated elements are added, there will still be gaps — indeed, there will be $\varepsilon c \log m$ empty array positions. Thus, each insertion takes time $O(\log m)$ because shifts propagate until the next gap, which appears within $O(\log m)$ positions. This observation establishes our result.

The direct approach

Let D be a set of $c \log m$ contiguous support elements. We would like to compute the number of intercalated elements that land among the elements of D. Notice that if there are k elements in the sorting array, then the $(k+1)$-th intercalated element is equally likely to land between any of those k elements. Thus, if an intercalated element is inserted within D, the probability of further insertions within D increases, and conversely, if an intercalated element is inserted outside of D, the probability of further insertions within D decreases.

We formalize the problem as follows. Consider two urns, urn A starting with $c \log m$ balls and urn B starting with $m - c \log m$ balls. Throw m additional balls, one after another, into one of the two urns with probability proportional to the number of balls in each urn. Let random variable $X_i = 1$ if ball i lands in urn A and let $X_i = 0$ if ball i lands in urn B. We now need to bound the tails of $\sum X_i$.

Because these X_i are positively correlated, bounding the tail of their sum is awkward. We analyze a simpler game below.

[2] Throughout, we use *contiguous* to refer to elements that are consecutive within their class, i.e. support or intercalated, which does not imply consecutive within the sorting array. More formally, if an ordered sequence of support (resp. intercalated) elements within the sorting array is $e_1, e_2, \ldots, e_k$, for some $1 \le k \le m$, two support (resp. intercalated) elements e_i and e_j with $1 \le i < j \le k$ are contiguous if and only if $j = i + 1$. However, if the sorting array positions where e_i and e_j are stored are $A[i']$ and $A[j']$ respectively, then $j' \ge i' + 1$, because there might be some intercalated (resp. support) elements or empty positions between them.

The arrival permutation

We first set up the problem. Consider $2m$ elements to sort. Each of the $(2m)!$ orders of insertion is equally likely. We refer to each insertion order as an *arrival permutation*. The first half of the arrival permutation consists of support elements, and the second half consists of intercalated elements. Thus, the probability of being a support (resp. intercalated) element equals the probability of being in the first (resp. second) half of the arrival permutation.

Our goal is to show that for sufficiently large c, with high probability in every set of $(2+\varepsilon)c\log m$ contiguous elements, there are at least $c\log m$ support elements at the end of a round. Thus, with high probability there are also at most $(1+\varepsilon)c\log m$ intercalated elements in every set of $(2+\varepsilon)c\log m$ contiguous elements. Because the at least $c\log m$ support elements are spread out in a subarray of size $(2+2\varepsilon)c\log m$, there is room to add the at most $(1+\varepsilon)c\log m$ intercalated elements while still leaving gaps. Therefore, with high probability no insertion will move more than $(2+\varepsilon)c\log m$ elements.

Theorem 4 *In any set C of $(2+\varepsilon)c\log m$ contiguous elements, there are at least $c\log m$ support elements with high probability.*

Proof. Consider choosing an arrival permutation $\mathcal{P}$ of length $2m$ uniformly at random by placing the elements one-by-one into $\mathcal{P}$, selecting an empty slot uniformly at each step. We place the elements of set C into $\mathcal{P}$ before placing the elements of $\overline{C}$ in $\mathcal{P}$. We give an upper bound on the number of elements in C that are support elements, that is, the number of elements that fall in the first half of $\mathcal{P}$.

Let s_i be the number of elements already inserted into the first half of $\mathcal{P}$ just before the ith insertion. The probability p_i that the ith insertion is in the first half of $\mathcal{P}$ is then $(m-s_i)/(2m-i+1)$.

Let random variable $X_i = 1$ if element i is a support element, and let $X_i = 0$ otherwise. Random variables $X_1, \ldots, X_{2m}$ now depend on the remaining blank spaces in the permutation and are negatively correlated. Furthermore, $|C| = (2+\varepsilon)c\log m$ is small compared to m, and so $E[X_i] = p_i$ is very close to $1/2$ for the first $|C|$ elements. Given this bound, we can prove our theorem with a straightforward application of Chernoff bounds.

Here we prove the theorem using elementary methods, as follows. The probability that a given element in C is a support element is at most

$$\frac{m}{2m-|C|+1} = \frac{m}{2m-(2+\varepsilon)c\log m+1}$$

Let p be the probability that there are at most $c\log m$ support elements in the set C. Then,

$$\begin{aligned} p &\leq \sum_{j=0}^{c\log m} \binom{|C|}{j} \left(\frac{m}{2m-|C|+1}\right)^{j} \left(\frac{m}{2m-|C|+1}\right)^{|C|-j} \\ &\leq \left(\frac{m}{2m-|C|+1}\right)^{|C|} \sum_{j=0}^{c\log m} \binom{|C|}{j}. \end{aligned}$$

Bounding the summation by the largest term, we obtain

$$p \leq \left(\frac{m}{2m-|C|+1}\right)^{|C|} c\log m \binom{(2+\varepsilon)c\log m}{c\log m}.$$

Using Stirling's approximation [4], for some constant c' we obtain

$$p \le c' \left(\frac{m}{2m - |C| + 1}\right)^{|C|} \sqrt{c \log m} \left(\frac{(2+\varepsilon)^{(2+\varepsilon)}}{(1+\varepsilon)^{(1+\varepsilon)}}\right)^{c \log m}.$$

Manipulating on the first term, we obtain

$$p \le c' \left(1 + \frac{|C|}{m}\right)^{|C|} \left(\frac{1}{2}\right)^{|C|} \sqrt{c \log m} \left(\frac{(2+\varepsilon)^{(2+\varepsilon)}}{(1+\varepsilon)^{(1+\varepsilon)}}\right)^{c \log m}.$$

Since $|C| \ll m$, we replace the first term by a constant less than e (indeed, $1 + o(1)$) and fold it, along with c', into c''. We get

$$p \le c'' \left(\frac{1}{2}\right)^{|C|} \sqrt{c \log m} \left(\frac{(2+\varepsilon)^{(2+\varepsilon)}}{(1+\varepsilon)^{(1+\varepsilon)}}\right)^{c \log m}.$$

Since $|C| = (2+\varepsilon)c \log m$, the previous equation simplifies to

$$p \le c'' \sqrt{c \log m} \left(\frac{(2+\varepsilon)^{(2+\varepsilon)}}{2^{(2+\varepsilon)}(1+\varepsilon)^{(1+\varepsilon)}}\right)^{c \log m}.$$

This last factor is 1 when $\varepsilon = 0$, and decreases with increasing ε. Then, using that $m \in \Omega(\sqrt{n})$, calling $\delta = (2^{(2+\varepsilon)}(1+\varepsilon)^{(1+\varepsilon)})/(2+\varepsilon)^{(2+\varepsilon)}$ and b the base of the logarithm, we obtain

$$\begin{aligned} p &\le c'' \sqrt{\frac{c}{2}} \sqrt{\frac{\log n}{n^{c/\log_\delta b}}} \\ &\in O(n^{-\gamma}) \text{ where } \gamma = \frac{c}{2 \log_\delta b} - \frac{\log \log n}{2 \log n}. \end{aligned}$$

Thus, for any constant $\varepsilon > 0$, there exists a big enough constant c such that the probability p is polynomially small when m is $\Omega(\sqrt{n})$. □

Note that since C is split evenly in expectation, the expected insertion cost is constant.

Corollary 5 *There are at least $c \log m$ support elements in each set of $(2+\varepsilon)c \log m$ contiguous elements with high probability.*

Proof. We are interested only in nonoverlapping sets of contiguous elements and there are $m/((2+\varepsilon)c \log m)$ such sets. By Theorem 4 and the union bound, the claim holds. □

3.3 Summary

We summarize our results as follows: The overall cost of rebalancing is $O(n)$ by Lemma 2. By Lemma 3, the cost of searching for the position of the ith element is $O(\log i)$, and then the overall searching cost is $O(n \log n)$. We proved in Lemma 1 that the insertion cost of the first $O(\sqrt{n})$ elements is $O(n)$ in the worst case. Finally, Theorem 4 shows that for sufficiently large c, no contiguous $c \log m$ elements in the support have $(1+\varepsilon)c \log m$ intercalated elements inserted among them at the end of any round, with high probability. Thus, there is a gap within any segment of $(2+\varepsilon)c \log m$ elements with high probability. Therefore, the insertion cost per element for $m = \Omega(\sqrt{n})$ is $O(\log m)$ with high probability. The overall cost of LIBRARY SORT is $O(n \log n)$ with high probability.

4 Conclusions and related work

We have shown that LIBRARY SORT outperforms traditional INSERTION SORT. There is a trade-off between the extra space used and the insertion time given by the relation between c and ε. The lower the desired insertion cost, the bigger the required gap between elements at rebalance.

LIBRARY SORT is based on the priority queue presented in Itai, Konheim, and Rodeh [6]. Our analysis is a simplification of theirs. Moreover, we give high probability and expectation bounds for the insertion cost, whereas they only give expectation bounds.

An algorithm similar to LIBRARY SORT was presented by Melville and Gries in [7]. This algorithm has a $1/3$ space overhead as compared with the ε space overhead of LIBRARY SORT. They point out that their running time analysis was too complicated to be included in the journal version. To quote the authors: *"We hope others may develop more satisfactory proofs."*.

The idea of leaving gaps for insertions in a data structure is used by Itai, Konheim, and Rodeh [6]. This idea has found recent application in external memory and cache-oblivious algorithms in the *packed memory structure* of Bender, Demaine and Farach-Colton [1] and later used in [2, 3, 5].

Acknowledgments

We would like to thank Erik Demaine and Ian Munro for helpful discussions. MAB was supported in part by the Sandia National Laboratories and NSF Grants ACI-0324974, CCR-0208670, and EIA-0112849. MFC was supported in part by CCR-9820879.

References

[1] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st IEEE Ann. Symp. on Foundations of Computer Science*, pages 399–409, 2000.

[2] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th ACM-SIAM Ann. Symp. on Discrete Algorithms*, pages 29–38, 2002.

[3] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th ACM-SIAM Ann. Symp. on Discrete Algorithms*, pages 39–48, 2002.

[4] W. Feller. Stirling's Formula. In *An Introduction to Probability Theory and Its Applications*, Vol. 1, 3rd ed., II.9, John Wiley and Sons Inc., New York, pages 50–53, 1968.

[5] G. Franceschini. An in-place sorting algorithm performing $O(n \log n)$ comparisons and $O(n)$ data moves. In *Proc. 44th IEEE Ann. Symp. on Foundations of Computer Science*, pages 242–250, 2003.

[6] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th EATCS Int. Colloquium on Automata, Languages and Programming*, pages 417–431, 1981.

[7] R. Melville, D. Gries. Controlled density sorting. In *Information Processing Letters*, 10:4, pages 169–172, 1980.