Skylinesort

a new sorting algorithm

Daniel Cussen

September 6, 2018

1 Introduction

To sort is to change the order of a list of numbers so that each value is larger than the value that precedes it. In Spanish and French, in fact, it used to be called an "ordenador" or "ordinateur", a sorter, because that's what it primarily did at the beginning of the digital age. IBM sold dedicated sorting machines in those days, huge punch-card rearranging devices, made to sort the cards as quickly as possible.

Sorting. Such a simple thing. Everybody has to do it, to check that no playing cards are missing, for instance, or separating colors in a heap of clothes for the wash. It's basic.

And it's expensive. Estimates from 1980 are that computers spent 25% of their time sorting. These days it is harder to get a precise percentage, but sorting is time- and energy-intensive enough to warrant avoiding it where possible, and either keeping data sorted or using algorithms that altogether don't rely on sorting. So sorting faster would be very useful: It would not only avoid the large penalty that slow sorting incurs, but also to open up approaches that are currently infeasible. Faster sorting would ultimately mean less time and energy spent both directly in tasks that require sorting, and in a cascade of tasks that rely on processes or subprocesses that are limited by the speed of sorting.

So that's what skylinesort is. A new sorting algorithm, created to save time and energy. But to understand it, we must first take a look at the current state of the art in sorting.

2 Problem

To understand skylinesort, we must first see that integers in a range can be sorted in two ways. Two of the basic operations determines how the elements will be rearranged in different places such that they are in order. The first basic operation is comparison. It takes one cycle, one step, and tells which one of two elements is larger. The algorithm that uses this operation is quicksort. The second basic operation is the lookup: for an element n, we read or write to the nth element of an auxiliary array. This operation forms the basis of counting sort. The comparison sorting algorithms and the array sorting algorithms both have their plusses and minuses, but if we rely on both operations we can get the best of both worlds in many ways, in an amount of time that exceeds each of these operations's individual speed limits. Running at the speed limit of comparison sorting, quicksort sorts n elements in O(n*log(n)) operations. Counting sort, the lookup sorting algorithm par excellence, sorts in O(N+n) operation, where N is 2^b, b being the bit-width of the elements.

Quicksort sets the first element of an array as a pivot value, then separates a list into a left part (less than a pivot value) and a right part (greater than the pivot value). It then applies recursively on each sublist, until the sublists are one element long. When it finishes, sorted left half, pivot, and sorted right half are put back together to form a sorted list. This continues until the original list is put back together from its constituent halves, sorted. Note that to accomplish all this quicksort only needs to be able to compare two elements to see which is greater.

The advantages of quicksort are numerous: it scales with the size of the list it has to sort, so small lists take negligible time; it can compare elements of any size, and so is useful for instance when comparing strings by using the order of the alphabet; and it takes place without requiring additional memory. Its disadvantages are that it has a speed limit that is asymptotically greater than counting sort (where lower is better). That speed limit is O(n*log(n)). Besides that, quicksort can't really be parallelized. Finally, if the list is sorted or reversed already, it takes O(n**2) time instead of loglinear time. This last point is the most painful. This disadvantage is frequently triggered, to the point that there are techniques devised to overcome this. The best of these is called median-of-medians, which corrects the quadratic worst-case performance to loglinear time, but worsens the average performance by a coefficient that usually makes it useless. In practice nobody uses median-of-medians, but quicksort is still generally good enough to be the traditional computer scientists's go-to algorithm; it has a strong reputation

as the fastest sorting algorithm.

However, counting sort sometimes improves on quicksort's performance by sorting with an auxiliary array whose every element is zero. It gets its name because it iterates over the list to be sorted, incrementing the element of the array indexed by the element of the list. This first phase, known also as "scattering," thereby counts the occurrences of each element in the list. Counting sort then iterates over the auxiliary array, adding up the values of nonzero array elements in order. This phase is known as "gathering." Then it iterates again through the list, moving each element to its new position in a new list, where each position is given by the corresponding element in the array. The new list is the sorted version of the old list.

The advantages in performing counting sort over quicksort are, first, that there is no logarithmic coefficient to make the sorting time loglinear (O(n) vs. O(n*log(n))). This is a huge difference for larger lists of thousands or millions of elements, as it alone can cut down on the time it takes to sort the list by a factor of 10 or 20, respectively. In addition, the scattering phase of counting sort is optimal. And, when the elements themselves are small, counting sort is the way to go for these reasons. But when it isn't, it's the gather phase that really hurts performance. Not the part of adding elements, but the fact that you have to iterate through all the elements of the array. In so many cases, the overwhelming majority of the elements of the array will be zero because no element indexed them, meaning they will not contribute to the final order of the array in any way. This is because the array will have a size on the order of 2**b elements. It grows exponentially with the size of the elements of the list. But if that were all, it wouldn't be much of a problem; memory is cheap these days. The problem is having to access all those unused, barren elements in the auxiliary array stored in memory. True, there are cases where nearly all elements of the array are in use. But if not, it's that walk over empty elements that ruins the performance. For 16 bit elements, themselves quite small, this walk will take roughly two cycles for each of the 65,536 empty elements, putting a minimum run time on counting sort that does not scale with the length of the list but rather the size of its elements. At 32 bits, it's unthinkable to use counting sort: though the amount of memory is available for modern workstations, taking seconds to traverse it all is much too slow. If for any reason the stages of counting sort take a lot of time, this walk will be the bottleneck.

So how do we avoid that walk? Is there a way to skip over the long segments of empty elements and get to the good parts? At the same time, how do we ensure we don't miss anything?

It turns out there is a way to do this, with a bit of extra work in the

scattering of every element. And that's what skylinesort is. It is really just a fancy way of avoiding the walk.

So, to avoid the walk, you have to be able to not even look at most of the array. If you look at an element to see if it's zero, you already lost the time you were trying to save. On the other hand, if you skip over an element that was nonzero the whole sorting process is ruined totally. Mistakes mean it's not sorting anymore.

To skip entire segments of the array at a time, while making sure not to skip any actual nonzero elements, these segments must in some capacity be marked as empty. It's not enough for them to just be empty, because then we would have to check. They have to be marked collectively. But they also have to be able to mark with a high resolution, so we can be sure we don't skip any. That's the problem statement: how do we resolve this apparent contradiction?

3 Solution

The way to resolve this is by jumping to an element when it is the mark of a segment, and progressively zooming out as we find consecutive marked segments. The way empty sectors get marked is by having an out-of-place element in the usual place to check, and knowing that we set things up so they mean everything between that place and the elements normal place is empty.

How do we do that?

We run in-place tournaments. Every element, after insertion into its own position, starts checking certain places in the array to see if it is larger than the element that is already there. If the element in that array is zero, that means the whole segment between them was empty and is now marked for skipping. If there's an element there, and it's smaller than our current element, it gets overwritten: the array it marked as empty now has another element in it. Finally, if there *is* an element in there, we stop the marking process because that element will have marked the following segments empty, or they won't even be empty.

Then, when we're gathering, we look for positions that are the first place for the nearest, smallest tournament. If that tournament isn't marked, meaning its top element is zero, it's empty, but we can't skip to more than the top of the next tournament. If that's zero too, we continue the process until we find something. When we find something, we know that everything between our current position and that element is empty, so we add that element to the

list and move our cursor to the element following it, beginning this process again.

In doing so, if we draw straight horizontal lines marking jumps between vertical lines marking the tournament height (the height of each bracket) we get a figure that looks like a city skyline. And just from the nature of the algorithm, the lines will meet but never intersect. They will always produce the same skyline regardless of the order of the elements. What makes the skyline unique is what the elements *are*, what their values are, not their order.

4 Results

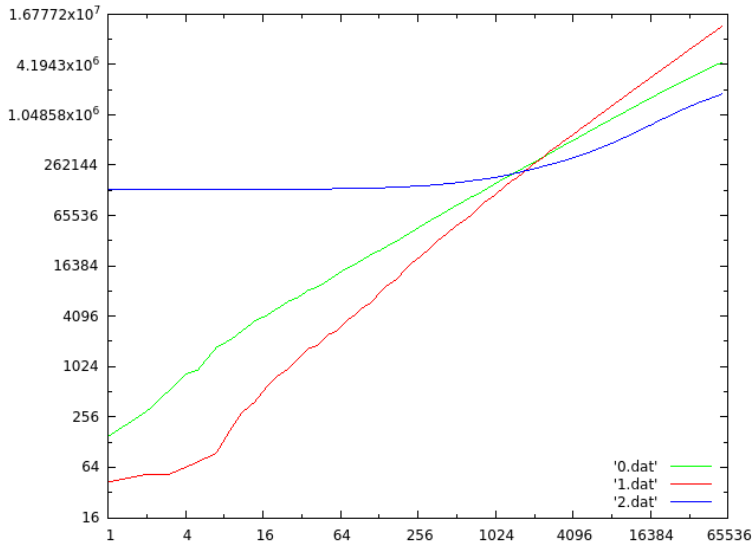
So what can we say about this algorithm? How does it compare to counting sort, and most of all, to quicksort?

Well, like counting sort, it needs an auxiliary array, but unlike counting sort, it is advantageous that this array be large. That is alright provided we can use an existing empty array in memory, and promise to return it empty after clearing it. If we do this, the only way it hurts to use an array is if we only use skylinesort once and never again. (If, on the other hand, it is inside a loop, this cleared array can be used again and again, without having to be allocated or deallocated. After all, memory management is too expensive in time and space to use it arbitrarily for something that has to be as quick as sorting.) With that out of the way, we can look at results.

So, results are divided into six different test cases, which represent different distributions of random numbers.

4.1 Uniform random distribution

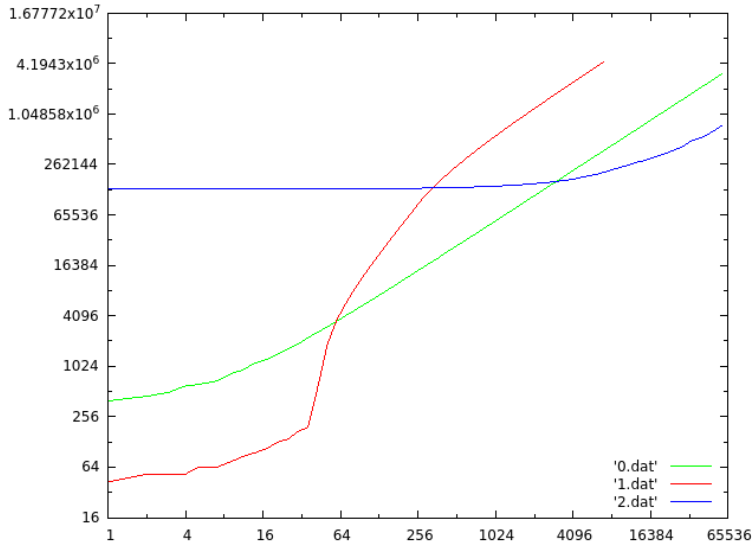
This is the base case, and the textbook case for sorting algorithms, though rare in practice. It is the uniform distribution of elements of the maximum bit-width. So, here, 0 will, on average, appear as often as 65,535 and any other 16-bit number. And the numbers appear in totally random order.



Here, skylinesort doesn't manage to beat both quicksort and counting sort at any point in the range from 2 to 65,536 elements. It is only third place in a very narrow part of the range, and otherwise comes second to both.

4.2 Linear, non-random distribution

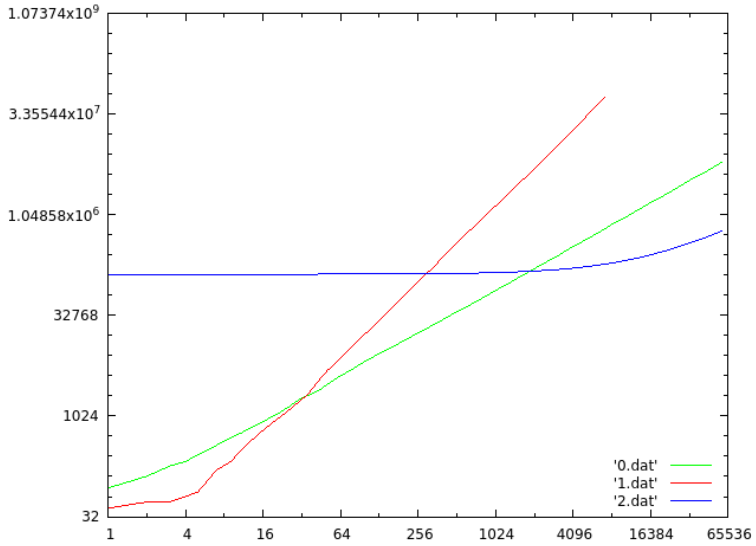
This is a linear, non-random distribution. Just f(x)=x for all positive integers, sorted elements in forwards order.



Here quicksort's curve shows a sharp bend at 32 elements, which is about the boundary where this implementation of quicksort degenerates into insertion sort, which is much faster in constant time. Insertion sort takes O(n), and quicksort takes O(n**2), so after a point–right at that bend–skylinesort overtakes quicksort. Here, skylinesort is linear, taking two comparisons per element. Skylinesort's "territory", the segment of the integers in which it dominates both counting sort and quicksort, is from 60 to 3040 approximately. In other words, an optimal range of 5.663 binary orders of magnitude, out of a possible 16 (2**0 to 2**16). In addition, at 332 elements, it's 7.54x faster.

4.3 Linear, non-random distribution in reverse

This is linear but in reverse. f(x)=length-x. Elements from 0 to length, one of each, sorted high to low, not low to high.

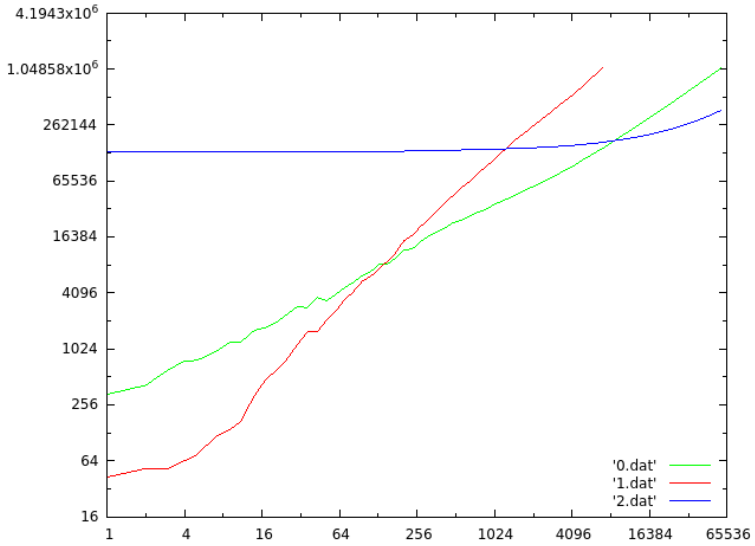


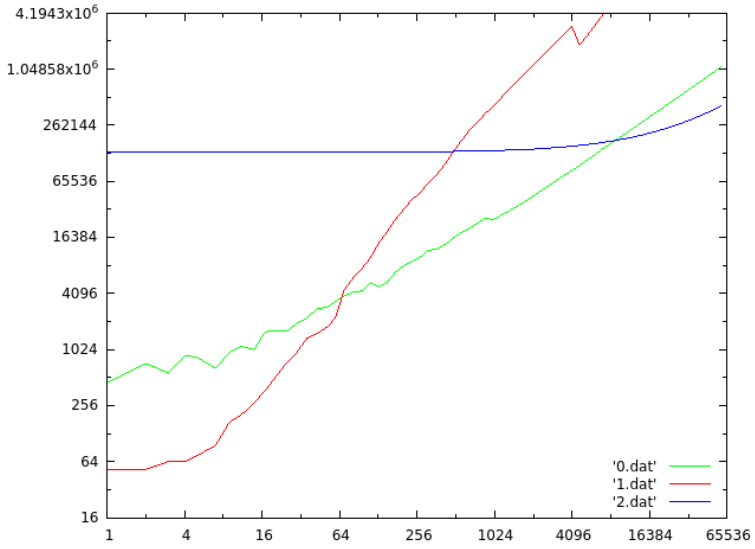
Here, quicksort performs at its worst, taking O(n**2) time. The penalty is also high, as the internal insertion sort algorithm now too works in O(n**2). Skylinesort itself performs at its worst case, O(n*k). Overall, skylinesort is much faster and gets to be 6.748x faster than quicksort and counting sort when they are equally fast. Range is 35 to 1991.

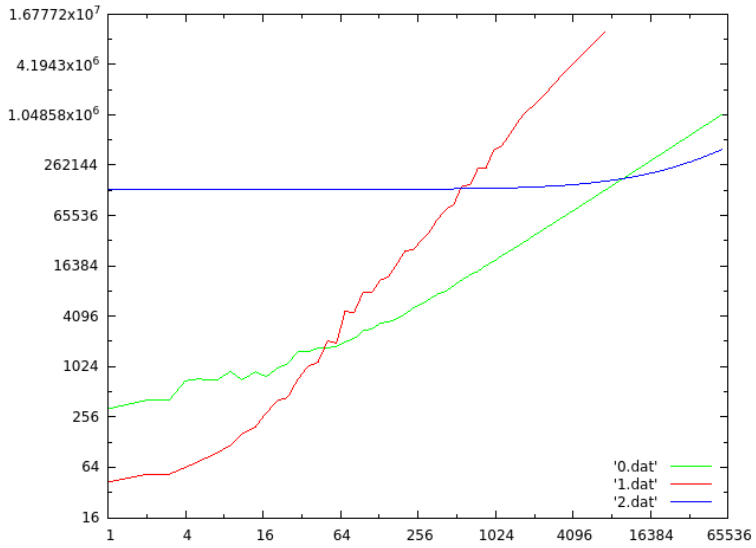
4.4 Duplicate-heavy sets

These graphs are of samples with a lot of duplicate elements. In this case the elements are also small, but that doesn't really change anything. The idea

is the list is sparse, i.e. its elements may contain up to 16 bits, but don't for whatever reason.







Here skylinesort dominates over a very large range, completely excluding quicksort. This is largely because having duplicate elements hurts quicksort's performance but not skylinesort's performance. In fact, for skylinesort, duplicates are all O(1), so they're less expensive than a new unique element. Skylinesort is ideal for this kind of distribution.

5 Conclusion

The final, resounding results are not in yet, so this is tentative. You *can* use skylinesort today. Provided you meet some system requirements, you should—caveats applying, both fingers crossed—see a nontrivial speedup, all *without assembly or parallelization.* Skylinesort has a low enough combined K and asymptotic time that it outdoes quicksort (and all the others) by—no guarantees here, your mileage will surely vary—5x. Or more. If your use case is really pathological, you can get more.

Unless you know precisely that in your particular application, quicksort will be advantageous, it is now almost never worthwhile to implement it. It's just not fast enough.

So what should you use? Usually you should be able to know how many elements a list has ahead of time, and base your choice off that in runtime. If you can't spare the memory, or the elements are uniformly distributed over the whole range, or the elements can't readily be slotted, then yes, choose at runtime between quicksort and insertion sort. Otherwise, consider using skylinesort.