Definition

Given an array A containing $n - b$ elements and a buffer B containing b elements, where the buffer contains insertions to be committed into A. The problem is to calculate the final positions that the inserted elements will occupy in A' and to sort the insertions such that they are in the relative order they will appear in in A'.

An element $B[i] = (j, v)$ where $i \in [0, b)$, $j \in [0, n - b + i]$ and v is an arbitrary value to be inserted, and irrelevant to the problem of buffer sorting.

Additionally it is assumed that $b \ll n$.

Example

Insertions are stored in the buffer, in the order they occure. Below are insertions to indexes $1, 1, 1, 2, 1$ in the order they arrived.

$$B = [(1, 1), (1, 2), (1, 3), (2, 4), (1, 5)]$$

After processing we want

$$B' = [(1, 5), (2, 3), (3, 4), (4, 2), (5, 1)]$$

Background for solutions

Any algorithm that does this processing can also be used to sort an arbitrary list, therefore the limits for sorting apply here as well, namely we can not expect a general solution better than $b \log b$. That said, b is a compile time constant and practically limited to at most 2^{10}. Thus, practical optimizations may be possible.

A version of this problem where $b = n$ is a well known competetive programming problem with 2 commonly seen solutions. The solutions below using self-balancing binary trees and bit-vectors are direct applications of these.

I implemented 7 different versions of solutions algorithms, that are described below.

Since both b and the maximum balue for n are known at complie tie **nothing is ever allocated or deallocated** during buffer sorting. Space for B' along with any needed support structures are statically allocated before running the sorting algorithms.

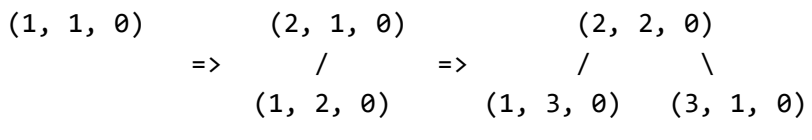
Algos

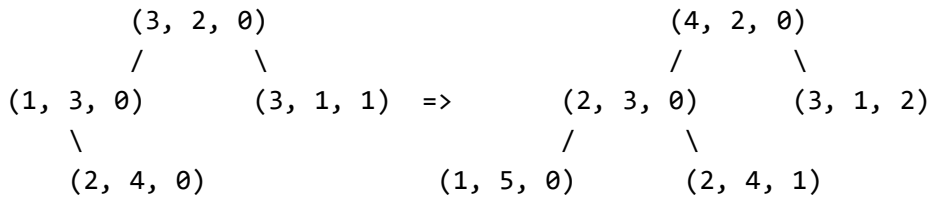
1 modified AVL tree ($b \log b$)

The insertions are added in order into a modified AVL tree, where the operation `increment` is added to nodes. This operation, increments a counter associated with the node, indicating that all elements in the sub-tree have their locations incremented by the value of the counter. Any time a non-zero counter is encountered while walking down the tree or doing rotations, the value of the counter is "pushed down" to the children of the node.

On inserting value j into the tree all elements $k \geq j$ are incremented by one. Otherwise the tree works exactly like a normal AVL tree.

Simulating on the B given above. alues are (i, v, counter).





The final B' can be read "for free" simply by doing an in-order walk of the tree while pushing down encountered increment counters.

The implementation stores node structs in an array and child relations are stored as vector indexes instead of pointers, this saves space and keeps the tree in contiguous memory. No array reallocations are needed since b is a compile time constant and each sorting opreation requires a tree with exactly b nodes.

The tree operations are implemented recursively in code, iterative versions for inserting and walking the tree where tried, but it turns out that the compiler is better at optimizing simple tree recursions than I am. (Even if recurrences are not tail-calls so the compiler can't eliminate the calls, a minimal call stack is probably kept and branch predicition for upwards traversal is good)

Performance is a seemingly slow $\mathcal{O}(b \log b)$. Probably due to the random access nature of a tree embedded in an array, and the constant factors of tree rotations.

The solution takes $\mathcal{O}(b)$ space with not-insignificant constant factors. $\sim 160 \cdot b$ bits.

2 Modified red-black tree ($b \log b$)

Exactly the same points apply here as with the AVL tree.

3 Bit-vector backed by segment tree

The idea is to iterate over B backwards, andfor each (j, v) look for the j^{th} "free space" in the bit-vector and to then mark the space occupied. Practically, this is simply

```
uint32_t t = bv.select(j);
bv.set(t);
j = t;
```

With the given B and $n = 10$ for demonstration:

```
BV          j  select(j) v
0000000000  1  1         5
0100000000  2  3         4
0101000000  1  2         3
0111000000  1  4         2
0111100000  1  5         1
```

Done in-place this gives

$$B = [(5, 1), (4, 2), (2, 3), (3, 4), (1, 5)]$$

that still needs to be sorted.

This solution requires a bit-vector of size n, i.e. the largest possible final value for any j. Additionally, a fast `select` operation for a partially dynamic bit vector is required. This is here provided by a segment tree, where each node stores the number of 1-bits in the left part of it's subtree. The segment tree is built on top of 64-bit words of the bit-vector, and select$_0(j_w)$ for the correct word w_i is done in constant time with

```
63 - __builtin_clzll(_pdep_u64(uint64_t(1) << j_w, ~data[w_i]));
```

With segment tree traversal `select` and `set` take $\log_2(n/64) = \mathcal{O}(\log n)$ time. And since the oprations are done b times, the final complexity is $\mathcal{O}(b \log n)$. In addition, the sorting using `std::sort` takes $\mathcal{O}(b \log b)$.

This solution takes $1.25n + \mathcal{O}(1)$ bits of space. n for the bit-vector and $2 \cdot 16 \cdot n/(2 \cdot 64) = 0.25n$ for the segment tree.

Practical performance for random insertion is fairly poor compared to other solitions. For sorted inputs however, the benefits to cache locality and branch prediction when accessing indexes close to each others makes this solutino somewhat competetive.

4 Bit-vector backed by b-tree

Since the maximum value for n is known at complie time, I tried using and implicit B-tree instead of the segment tree, and hoped this would provide a benefit in word selection and updating the bit-vector. The idea is still to keep track of the number of set bits in subtrees.

Otherwise the same points hold for this as for the previous solution. Including performance and space requirements.

5 Brute force

```
for (int i = 0; i < b; i++) {
    auto element = B[i];
    for (int j = i + 1; j < b; j++) {
        if (B[i].index >= element.index) {
            element.index++;
        }
    }
}
```

The above is a trivial, in-place algorithm to update the values. This solution is $\mathcal{O}(b^2)$ and very slow in practice.

However the slowness for small values for b is not due to the asymptotic complexity, but due to a hard data dependence in the solution.

The result of the comparison in the current iteration of the inner loop depends on the result of the previous iteration of the inner loop. This data dependence is evidently something the compiler can not reorder away even with b known at compile time.

By flipping the iteration to update all previous elements based on the original value of the current element, most of the data dependences are resolved, and gives us

```
void sort(B_type* buffer) {
    for (uint16_t i = 1; i < buffer_size; i++) {
        for (uint16_t j = 0; j < i; j++) {
            buffer[j].first += uint16_t(buffer[i].first <= buffer[j].first);
        }
    }
    std::sort(buffer, buffer + buffer_size);
}
```

A simple and competetive solution for small (≤ 512) values of b.

6 Juhas blocking idea

An array of size $\sqrt{b}$ can be sorted using the brute force solution in $\mathcal{O}(b)$ time. In addition there are no data dependences between blocks, so comparison operations between blocks can be intrelieved to completely eliminate data dependences (assuming a sufficient number of blocks fit in cache).

For an original array of size b, we get $\sqrt{b}$ blocks. Now the least element can be found in $\sqrt{b}$ time, and we can assign $\sqrt{b}$ values to keep track of block offsets. If we select and eliminate (put into the correct position) the least element b times from $\sqrt{b}$ sorted blocks, we are done.

This yields a solution that takes $\mathcal{O}(\sqrt{b} \cdot b)$ time to sort the blocks and a further $\mathcal{O}(b \cdot \sqrt{b})$ to merge the blocks to form the final solution.

Demo with $b = 16$ (without the v values):

```
B  = [ 3, 30,  3,  3,  4, 16, 15, 20, 13, 11,  7, 12, 16, 14, 19,  4]
B' = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Brute force block sort:
B  = [ 3,  4,  5, 32,  4, 15, 17, 20,  7, 12, 13, 16,  4, 15, 18, 20]
B' = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Get first element with block offsets [0, 0, 0, 0]
B  = [  ,  4,  5, 32,  4, 15, 17, 20,  7, 12, 13, 16,  4, 15, 18, 20]
B' = [ 3,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Get second element with block offsets [0, 0, 0, 0]
B  = [  ,  4,  5, 32,  4, 15, 17, 20,  7, 12, 13, 16,   , 15, 18, 20]
B' = [ 3,  4,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Get second element with block offsets [1, 1, 1, 0]
B  = [  ,  4,  5, 32,   , 15, 17, 20,  7, 12, 13, 16,   , 15, 18, 20]
B' = [ 3,  4,  5,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Get third element with block offsets [2, 1, 1, 0]
B  = [  ,   ,  5, 32,   , 15, 17, 20,  7, 12, 13, 16,   , 15, 18, 20]
B' = [ 3,  4,  5,  6,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     And so on until
B  = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]
B' = [ 3,  4,  5,  6,  7,  8, 13, 14, 15, 18, 19, 20, 23, 25, 28, 44]
     With final block offsets [12, 8, 4, 0]
```

All in all this is a practically fast $\mathcal{O}(b\sqrt{b})$ solution. Faster than any of the previously seen solutions with 512 or 1024 elements. However, the slightly worse scaling is visible and would probably mean that this is lower than the balanced BST solutions for larger buffer sizes. Takes $\sim 64b$ bits of statically allocated extra space.

7 merge-sort with block offsets

Based on the idea of counting list inversions using merge-sort. So simply run merge-sort, but instead of sorting by the values themselves, update values with block offsets as needed.

The implementation for this can be very fast even when using standard recursive definitions for merge-sort, since the recurrences call different template functions.

Merge-sort is not in-place, so the data is swapped back and forth between the buffer and statically allocated scratch space. To make sure that the end result in sin the correct location, either 2- or 4-element lists are sorted in-place with a sorting network.

```
// Recursive calls with different template values get inlined
template<uint16_t block_size>
void sort(B_type* source, B_type* target) {
    if constexpr (block_size == 2) {
        // always sort pairs in place
        twosort(target);
        return;
    } else if constexpr (block_size == 4 && __builtin_clz(buffer_size) & uint16_t
(1)) {
        // Sort blocks of 4 in-palce with sorting network if this makes
        // the full sorted list end up in the correct place
        foursort(target);
        return;
    } else {
        const constexpr uint16_t sub_block_size = block_size / 2;
        sort<sub_block_size>(target, source);
        // Do merges...
    }
}
```

Demo with $b = 16$ (without v values):

```
B = [ 3, 30,  3,  3,  4, 16, 15, 20, 13, 11,  7, 12, 16, 14, 19,  4]
S = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Blocks of 4 in-place:
B = [ 3,  4,  5, 32,  4, 15, 17, 20,  7, 12, 13, 16,  4, 15, 18, 20]
S = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]

     Merge to blocks of 8 to scratch
B = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]
S = [ 3,  4,  5,  6, 15, 17, 20, 36,  4,  8, 13, 14, 15, 18, 19, 20]

     Merge final result back to B
B = [ 3,  4,  5,  6,  7,  8, 13, 14, 15, 18, 19, 20, 23, 25, 28, 44]
S = [  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ]
```

Easily the fastest solution so far. No recursive calls get comiled into the binary.

Further ideas

Sorting network

I already use a very simple sorting network for blocks of 4 in the final solution. So why not more.

A sorting network should be possible since b is constant. However, **good** sorting networks for large (b > 16) lists seem somewhat difficult to generate, especially with the added complexity of calculating offsets.

And the last solution essentially already almost makes the compiler generate a sorting network due to the templateing. A "real" sorting network would work in-place I guess.

Bucketing

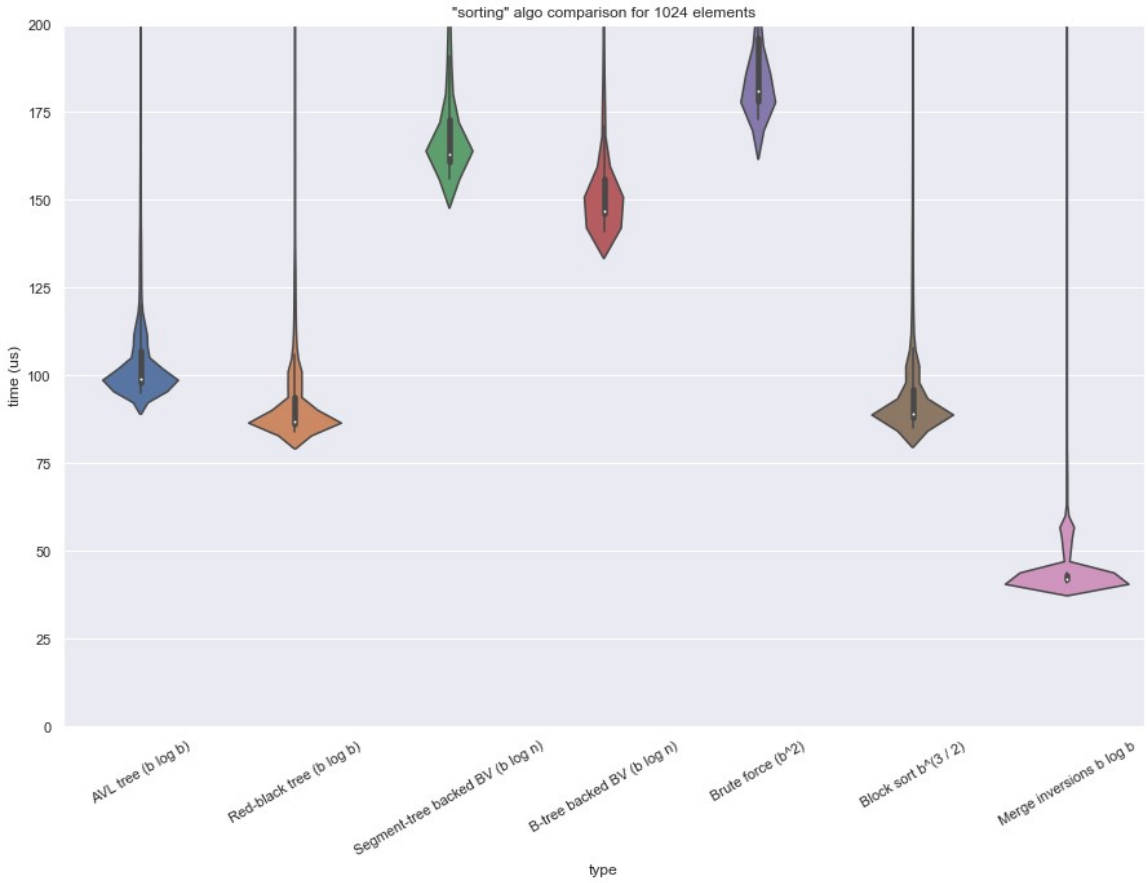
Both Tuukka and Simon talked about some sort of bucketing for splitting the initial buffer and then sorting the buckets.

I still don't get how this would work.

Index compression for the bv-based solution....

Won't work for bigger buffer sizes at least. Could save space for $b \leq 2^6$, guaranteed to fit in bv of 2^12 bits without corrupting calculations. Probably not relevant. Index compression could be done in $b \log b$, simply by sorting and then compressing gaps to be at most b. This would make the $b \log n$ of the bv-based approaches $b \log b^2$ instead.

Results



Merge-sort-based approach is the fastest so far.