emuminov
Posted on Nov 13, 2024 • Edited on Dec 31, 2025

Ford-Johnson Algorithm: Human Explanation & Visualisation

#mergeinsertionsort
#fordjohnsonalgorithm
#42school #cpp09

Ford-Johnson sorting algorithm, or merge-insertion sort, is not a very well known

emuminov

JOINED
Oct 28, 2024

Trending on DEV Community 🔥

Duplicate “Follow” Button Text in User Profile Hover Card
#devbugsmash #devops #discuss #architecture

What was your win this week??
#weeklyretro #discuss

or popular algorithm, which is optimised for making the least amount of comparisons between elements. The reason on why this algorithm is not well known is that it's neither fast, nor the best minimum sort algorithm, which is why there are almost no good resources on this algorithm, between scientific papers, hard to read computer science books, or obscure threads on Reddit or Stack Overflow.

This algorithm has little to do with merge sort but is often confused with it. In the 42 school for cpp09 it's a common mistake, or sometimes a way to cheat, to implement much simpler hybrid sort between merge sort and binary insertion sort. This article aims to give an understanding of the algorithm using simple explanations and step-by-step execution of the algorithm on a set of 22 numbers.

My name is Eldar, and I'm a student at 42 Lyon. Hence, the article will also have the context of one of the school assignments. I decided to write an article on this algorithm because almost no one does this algorithm correctly and understandably since it is hard, and easily one of the more complex sorting algorithms out there.

Steps of the algorithm

I recommend to read first Wikipedia's description of the algorithm, as well as Knuth's. Both of those sources describe the steps of the algorithm a little bit differently. I intend to give my own explanation as well, as I believe that rewording helps to develop understanding; I will also later explain each step individually in depth. Hence, my explanation of the steps for sorting numbers using this algorithm in ascending order:

1. Recursively divide into pairs of numbers, pairs of pairs of numbers, pairs of pairs of pairs of numbers... etc, and sort them by the biggest number, until we can't form any pair anymore. If there is an unpaired odd element, leave it be.
    - I will also refer to the units on which we operate on as `elements`. `elements` can be numbers, pairs of numbers, pairs of pairs of numbers etc...
    - We will refer to the smallest element in each pair as `b`, and to the biggest as `a`. Depending on the index of the pair, we will call them `b1`, `a1`, `b2`, `a2` ... `bx`, `ax`.

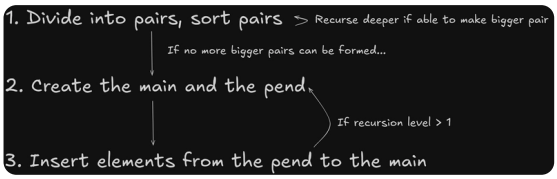
2. Create a sequence (also referred to as `S` or `the main chain` in Wikipedia and Knuth's book respectively) out of the smallest element of the

smallest pair (`b1`) and the rest of `a`s. If the first step was done correctly, this sequence will be sorted. Create the sequence that consists of the rest of `b`s (also referred to as `the pend`), and the odd element if there is any.
    - I will refer to those sequences as `the main` and `the pend` from now on.

3. Binary insert the elements from the pend into the main, in the order based on Jacobsthal numbers. I will explain later why and how it works. If we can't insert elements into `the main` using Jacobsthal numbers anymore, we insert them in reverse order, using similar approach to the plain good ol' binary insertion.

Due to the fact that step 1 is recursive, it will create a bunch of recursion levels, to each of which steps 2 and 3 will be applied. It will look as

something like this:



If everything is done correctly, the sequence will be sorted. Of course, I omitted some of the details, for the sake of giving the general description on what we will be doing. The algorithm isn't easy to understand, and you aren't expected to understand it from the first read. After you'll see the visualisation of the algorithm using the established nomenclature, you will have a much better idea.

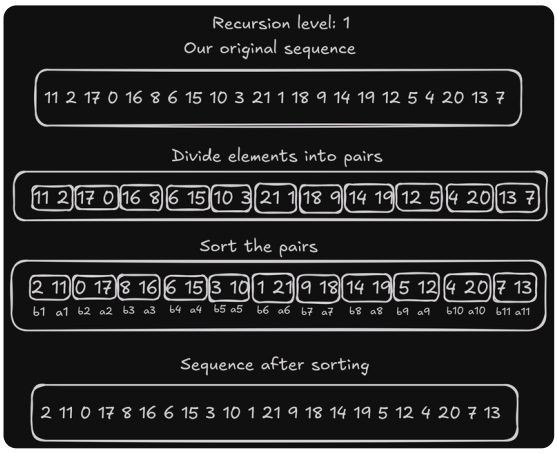
Step 1: the division into the pairs & sorting

This step is pretty confusing to understand, and also tricky to implement in code, but good visualisation helps. The difficulty is that with each recursion call we operate on bigger and bigger

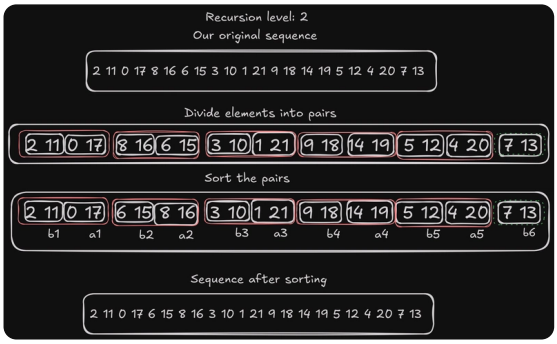
groups of elements: pairs of numbers on the first call, pairs of pairs of numbers on the second call, pairs of pairs of pairs of numbers on the third etc, until we can no longer divide our original sequence into a single pair of elements.

On top of this, we also need to preserve the original pairing: we can't break pairs formed on the first recursion call on the fifth one. This is important for this algorithm, as it relies on certain guarantees about the elements: that the biggest element of `b2` is smaller than the biggest element of `a2`, `b3` is smaller than `a3`, `b4` is smaller than `a4`, `bx` is smaller than `ax`. This is important for the algorithm's nature of a minimum comparison sort. The easiest way to achieve this is to simply swap entire groupes of numbers with each other when we sort them. It'll be much clearer in a minute.

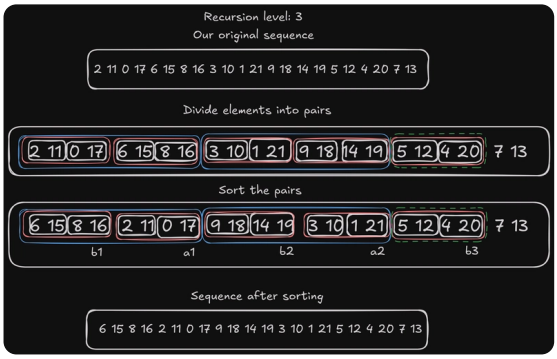
For the visualisation I'll denote at the top, the recursion depth we are currently it. Because the algorithm requires recursive sorting of pairs before the insertion part, I can demonstrate this process in this section.



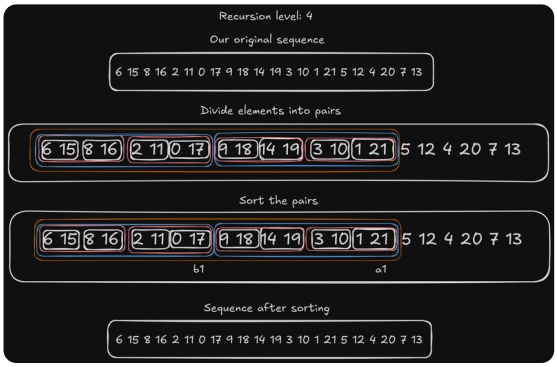
This level of recursion depth is easy to wrap head around: here we operate just on numbers. If the number in the pair is bigger than the other one, we simply swap them in place. After sorting the pairs, I put the `b` and `a` labels that I described earlier underneath each element in the new sequence: now it should be clear how the labels work.



This is where it starts to get funky: now our element is a pair of numbers instead of just a number. Also, we have an odd element that is left without its pair. We name it `b6`.
On top of the white borders denoting pairs, here there are red borders for the pairs of pairs. Notice also that I placed the labels against the last number of each element: this is no coincedence, the labels reside under the number that we actually use for comparison/pair sorting: `11` against `17`, `16` against `15` etc, but the swapping happens on an entire subsequence of numbers. Green dotted border denotes an odd pair. Here, only one swap occures: of `8 16` and of `6 15`.



Now we have a blue border to denote pairs of pairs of pairs... It's already mouthful to say. But we must go deeper.
Notice that there are numbers without any borders: I chose this way to separate numbers who can't even form an element. At this level, the smallest unit on which we operate consists out of 4 numbers, and the pairs are formed out of 8 numbers. What we are gonna do with those numbers?
Simple: we ignore them.
At this level, two swaps occured: `2 11 0 17` and `6 15 8 16`, `3 10 1 21` and `9 18 14 19`.



Now we are at the level of pairs of pairs of pairs of pairs. We have 22 elements, but the element here is formed out of 8 numbers: hence only one pair of two elements. `17` is lesser than `21`. Lucky us! No swaps happened at this level.

There is nothing to be done at the level 5 of recursion: because the amount of numbers in the element doubles with each recursion call, at one point we won't be able to form a pair of elements. This is where we break from this level. Because at level 5 one element consists of 16 numbers, and we have only 22 numbers overall, pair of 2 elements is impossible to form. From now on, we

proceed with the steps starting with the level 4, then we go to level 3, 2, 1... until we got our sorted number sequence.

To really understand what's going on at the sorting of pairs level, I invite you to come up with a random sequence of numbers and try to sort them on paper. After doing it a couple of times, you will probably have some ideas on how to implement this step in code.

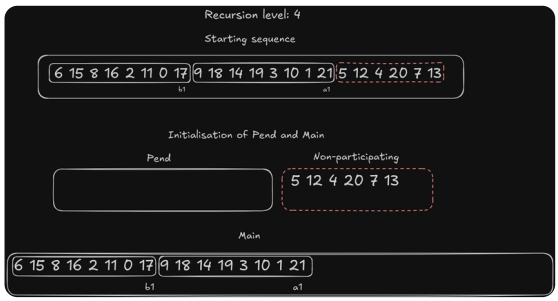
Steps 2 and 3: the initialization and insertion

We will talk about those two steps together, as due to the recursive nature of the algorithm, they will alternate between each other. Step 1 operated on its own and created a 4 recursion levels, which are for steps 2 and 3 to resolve. Those steps are applied to each of the levels.

Here is where the labels we defined earlier will come in

handy.

This step is pretty easy: `The main` is initialised with the elements `{b1, a1}` and then with the rest of `a` s. `The pend` is initialised with the rest of `b` s starting from `b2`. `The main` at this point should be sorted, but not entirely. Like with comparing and swapping at the step 1, only the biggest number in each element counts for "being sorted". Let's demonstrate this with the example we already have. This is where we're at after continuing on the level 4 of recursion:



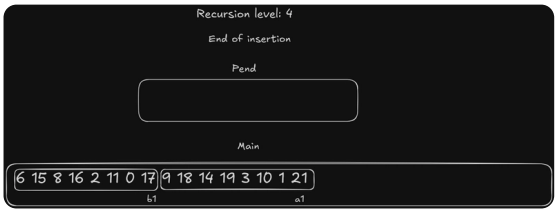
Not a lot of things to talk about at this recursion level: the pend is empty, as there are only two elements which go directly to `the main`, since `b1` and `a1` both start as part of `the main`. The

reason for this is that we already know that `b1` is smaller than `a1`, and because of the logic of the algorithm, and what you will see from further visualisations, `the main` is sorted, so `a1` is smaller than any of the other `a`s, hence no additional comparisons are needed. What counts as "sorted" are the biggest numbers in each element: in this case, in `b1` and `a1` those are `17` and `21` respectively. That is one of the optimisations of this algorithm to minimize comparisons.

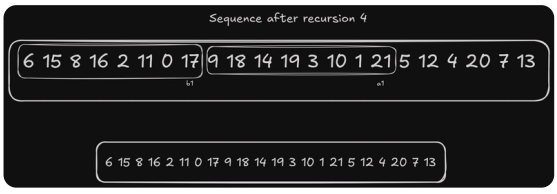
The numbers in the dotted red border `5 12 4 20 7 13` are numbers which can't form even a single element. Just like in the step 1, they are still part of the sequence, they just don't participate in the insertion.

One important thing to understand is that on that step, when we initialise `the pend` and `the main` the

labels change. Elements can and will change their positions compared to their step 1 position because of all the swapping and insertion. On this step, you can think about it as if the labels refer the position of the elements: for example, on this level element consists of 8 numbers, so `b1` will always consist of numbers at positions of 0-7, while `a1` of numbers at positions of 8-15, `b2`, and so on.



Nothing to say about step 3 at this recursion level, as there are no elements in `the pend` at all. No `pend` - no insertions. Move on. I will discuss insertion logic when we'll have something to insert.



We are done with level 4: nothing else to do here.

At this level our element consists of 4 numbers. From the start, we divide elements into `b`s and `a`s. We initialize `the pend` and `the main` as I said before: `the main` is `b1` and the rest of `a`s, `the pend` is the rest of `b`s. Notice that `b`s and `a`s are changed. Remember what I told you about relabeling? We change the labels at the step 2 of each recursion level. Notice, however, that the actual number pairings remained unbroken.

This time we have two elements in `the pend`: `b2` + the odd element that didn't participate in step 1, but is a full participant here. There is something to insert, so we

can discuss how step 3 works.

Step 3 can be pretty confusing. This is where binary insertion of our merge-**insertion** algorithm comes in. What do I mean by binary insertion? It means that we use binary search algorithm on our `main`, which is a sorted sequence, to find a place for insertion for our `pend` elements. Insertion itself works very similarly to how it's done in the binary insertion sort, but with a twist: the insertion order from `the pend` is dictated by mathematical sequence called Jacobsthal numbers.

Before we proceed further, let's spend a little bit of time to understand what's going on here. Why Jacobsthal numbers? Why bother with them, what do they do with sorting?

Why Jacobsthal numbers?

Ford-Johnson algorithm uses

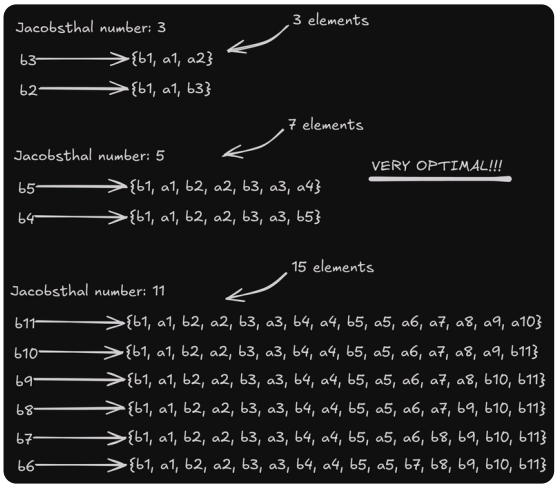
an important binary search algorithm optimisation that further cuts the number of times we compare our elements to each other. The maximal number of comparisons for the binary search is the same for `2^k` as for `2^(k+1)-1`. That means that the maximal number of comparisons is the same for 4(`2^2`) and 7(`2^3-1`). Same thing for 8 and 15, for 16 and 31... etc. This algorithm uses this property of the binary search during the insertion phase, if it can. What do I mean by "if it can"? Well, as you remember, we didn't used any Jacobsthal numbers during unrolling of our level 4 and 3 recursions. Wait, but how are those damn numbers related to this optimization? Well, turns out, that if you insert `the pend` elements into `the main` using a specific order dictated by the Jacobsthal numbers and we respect the bound elements, the surface

area for insertions is `2^(k+1)-1` for (nearly) all elements, as opposed to `2^k` for the first element and God knows what for next elements, since we would naively insert numbers as in regular binary insertion (like we did during the insertion at the recursion level 3).

The way Jacobsthal numbers dictate the order of insertion is like this: we start from the Jacobsthal number of 3. We start insertion from element `b3`. We insert elements in the reverse order starting from this element, until we hit `b` element of number of previous Jacobsthal number. In other words, the amount of inserted elements is `current_jacobsthal - previous_jacobsthal`.
For the Jacobsthal number of 3, we insert 2 elements (3 - 1), `b3, b2`.
For the Jacobsthal number of 5, we insert 2 elements (5 - 3), `b5, b4`.
For the Jacobsthal number of

11, we insert 6 elements (11 - 5), `b11, b10, b9, b8, b7, b6`.
I hope that you got the idea. And that you understand now that we can't always insert numbers this way. If there's not enough elemets to insert, for example, the Jacobsthal number is 11 (we should insert 6 elements), but we have only 3 elements in `the pend`, then we need to handle it somehow, and the simplest way to do this is to insert `the pend` elements in order, as we did in recursion level 3.

Ok, but now again, how exactly this order optimises the amount of comparisons for our binary insertion? Let's look at what happens if we insert pend elements in this specific order! Time for another visual example! Yay!



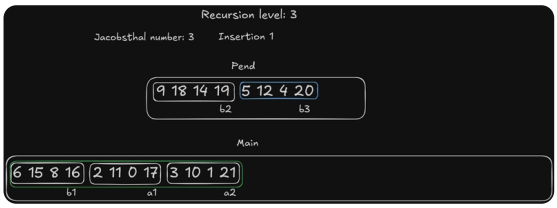
The elements are ordered by labels in these exapmples, but the actual order with actual numbers might be more mixed, so it's like this in the example for clarity.
As you can see, the order dictated by the Jacobsthal numbers is what allows us to use this optimisation in the algorithm: we can insert large amount pend elements into the main with binary search only on `2^(k+1)−1` elements. This works because due to how the algorithm works, we know that the bound for `bx` is `ax`. So when we insert, for example `b5` instead of searching in `{b1, a1, b2, a2, b3, a3, a4, a5, a6, a7, ...}` we can search in `{b1, a1, b2, a2, b3, a3,`

`a4}` , because we know that there is no point to include into the search range `a5` and anything after that since they are guarenteed to be bigger. After we inserted `b5` and it is time to insert `b4` , the bound is decreased by one (from `a5` to `a4`), but the amount of elements is increased by one (we inserted `b5`), so the amount of elements in this range stayed the same. Note: *however*, sometimes, as we will see soon, the Jacobsthal number insertions guarantee `2^(k+1)-1` in the *worst* case. In the best case the range of search may be even smaller, if inserted element is going to be bigger than any other number in the search range. Then, for the next number the search range will actually be lower. You will see this in the continuation of the execution.

Now that we understood what purpose the Jacobsthal numbers have here, I want to

bring up that there is also an optimal way to calculate the n'th Jacobsthal number. Rounded result of `(2^(n+1) + (-1)^n) / 3` will produce an offsetted for our use case (it normally starts from 0) Jacobsthal number.

Back to where we left



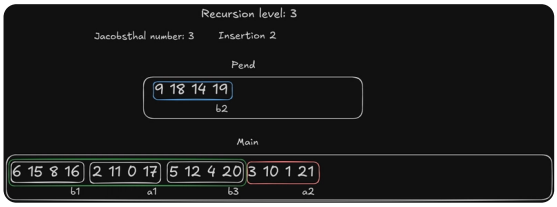
I will mark the current element for insertion from `the pend` with the blue border. The range of elements that we will try to insert this element into with the green border, and the "bound" element, ie corresponding `a` element, with the red border. "Bound" marks the end of the area of search.

The current Jacobsthal number is 3, which means that we start with `b3` from `the pend`. The bound

element for `b3` is `a3`, and we should try to insert it into `{b1, a1, a2}`. Since there is no `a3` element, because `b3` is an odd numer, we compare it with the entire `main`.

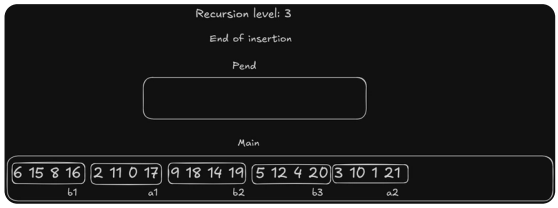
In other words, we should compare `20`, the biggest number in the element, to `{16, 17, 21}`, the biggest numbers in the elements of our search area, and then insert the whole element after that in it's corresponding place.

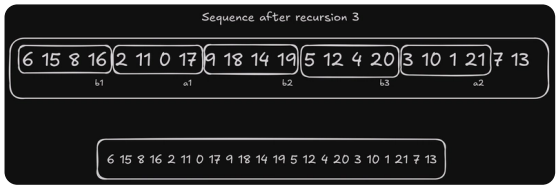
Since `20` is bigger than `16` or `17`, we insert the entire element `b2 (5 12 4 20)` between `a1 (2 11 0 17)` and `a2 (3 10 1 21)`.



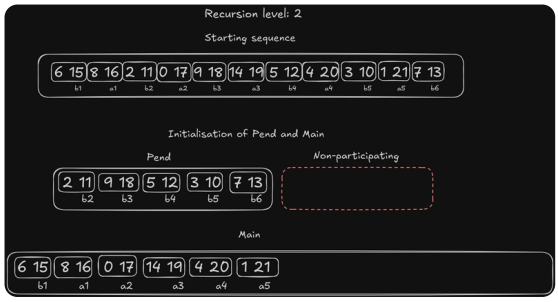
The Jacobsthal number is still 3, and the next element for insertion is `b2`. Its bound element is `a2`: we already know that it's smaller that

this element, so we try to insert it into `{b1, a1, b3}`.



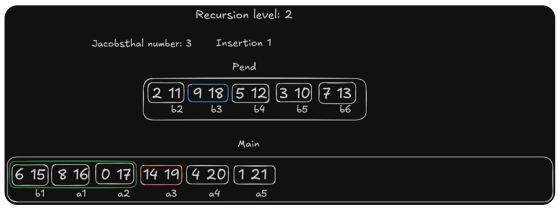


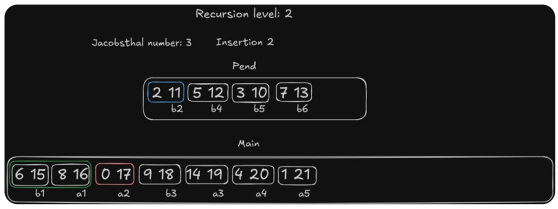
So, at the moment, our sequence is this: `6 15 8 16 2 11 0 17 9 18 14 19 5 12 4 20 3 10 1 21 7 13`. When we go down a recursion level, `the main` and `the pend` initialisation, as well as insertion, will go from this sequence.



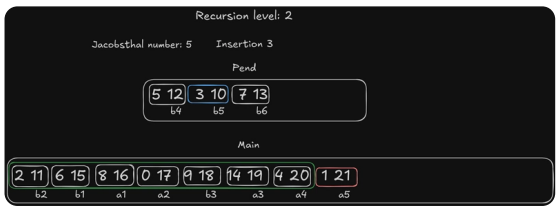
We divide the resulting sequence into the elements again, and initialize our `main` and `pend`. The sequence has changed since the recursion

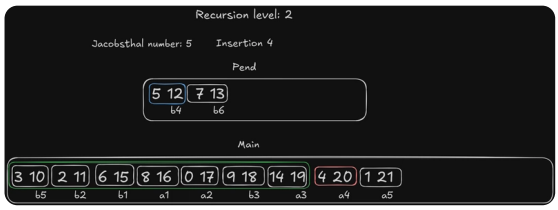
level of 3, and the labels too: but the original pairs (like `((2 11) (0 17))`) remained unbroken. So the guarantee that the `(2, 11)` (now `b2`) element is smaller than `(0, 17)` (now `a2`) is still valid.

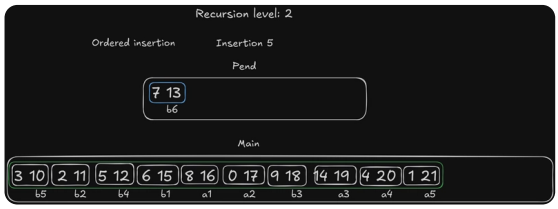




So, we inserted our `b3` into `the main`, but as you can see, it shrinks the search area for the `b2`, as this element turned out bigger than `a2`, so the area of search shrank. Remember the note from the "Why Jacobsthal numbers?" part? This is the example. It may happen sometimes, and it's another tricky thing of this algorithm that needs to be accounted for.



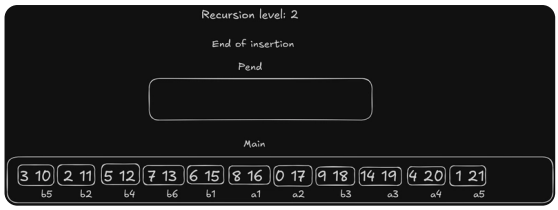




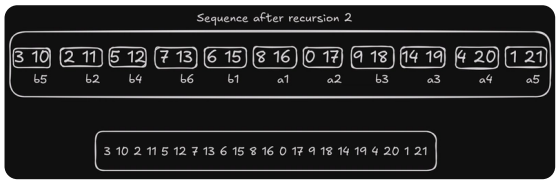
Oops! We run out of the Jacobsthal numbers. There is only one element left in `the pend`! What to do? We do almost the same thing as we do on the binary search: we insert `the pend` elements, but in the **reverse order**, meaning that we start from the end. We do it like that in order to keep the search area the same. Of course, you still have to respect the bound element: for example, if we insert `b3` in this way, the area of search still doesn't include `a3` and further. In this case, it's an odd element, so there is no corresponding bound

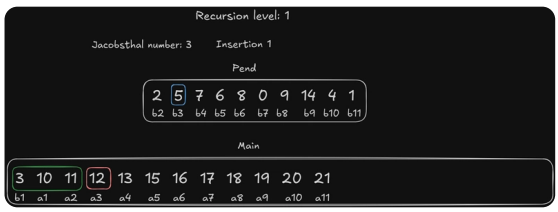
element.

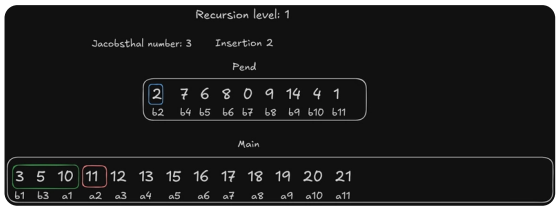
Ok, so now, I think, we are fully equipped to comprehend the rest of the algorithm, so let's proceed with its steps without any more yapping!



With this, the second level of the recursion was resolved. The resulting sequence after this level looks like this:

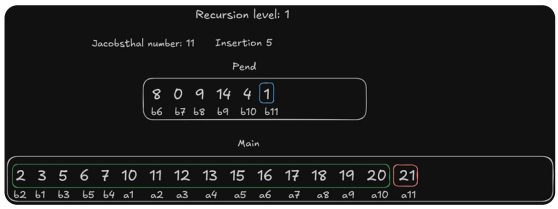


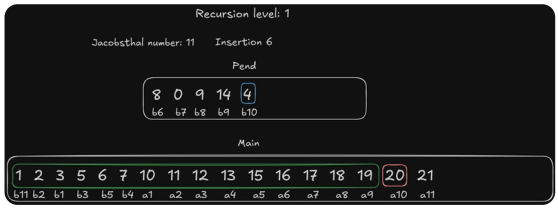



Recursion level: 1
Jacobsthal number: 3
Insertion 2
Pend
2 7 6 8 0 9 14 4 1
b2 b4 b5 b6 b7 b8 b9 b10 b11
Main
3 5 10 11 12 13 15 16 17 18 19 20 21
b1 b3 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11

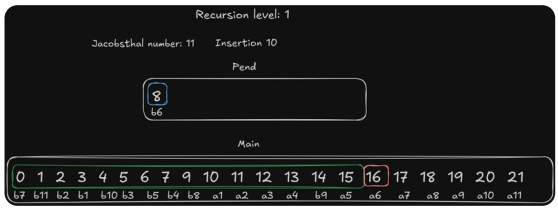

Recursion level: 1
Jacobsthal number: 5
Insertion 3
Pend
7 6 8 0 9 14 4 1
b4 b5 b6 b7 b8 b9 b10 b11
Main
2 3 5 10 11 12 13 15 16 17 18 19 20 21
b2 b1 b3 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11

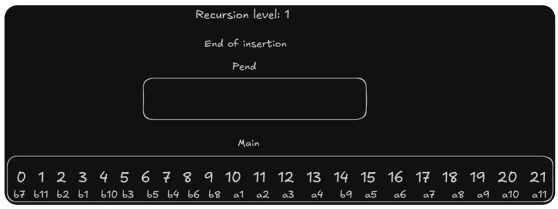

Recursion level: 1
Jacobsthal number: 5
Insertion 4
Pend
7 8 0 9 14 4 1
b4 b6 b7 b8 b9 b10 b11
Main
2 3 5 6 10 11 12 13 15 16 17 18 19 20 21
b2 b1 b3 b5 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11


Recursion level: 1
Jacobsthal number: 11
Insertion 5
Pend
8 0 9 14 4 1
b6 b7 b8 b9 b10 b11
Main
2 3 5 6 7 10 11 12 13 15 16 17 18 19 20 21
b2 b1 b3 b5 b4 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11


Recursion level: 1
Jacobsthal number: 11
Insertion 6
Pend
8 0 9 14 4
b6 b7 b8 b9 b10
Main
1 2 3 5 6 7 10 11 12 13 15 16 17 18 19 20 21
b11 b2 b1 b3 b5 b4 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11


Recursion level: 1
Jacobsthal number: 11
Insertion 7
Pend
8 0 9 14
b6 b7 b8 b9
Main
1 2 3 4 5 6 7 10 11 12 13 15 16 17 18 19 20 21
b11 b2 b1 b10 b3 b5 b4 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11





And with this, our sequence became sorted. This algorithm is not an easy one to understand and also to implement, but both

swapping and insertion follow specific patterns, peculiarities of which you can calculate, if you do this algorithm on paper/in your favorite text editor again and again, which I encourage you to do, as doing is the only way to really cement theory.

Testing the algorithm implementation

This is a minimal comparisons sort, so your algorithm should perform no more than `k` comparisons for the input of `n` numbers. For example, for 21 numbers the maximal number of comparisons is 66, so if you run this algorithm 1000 times on 21 random numbers, it will never exceed 66 comparisons.

There is a mathematical way to calculate `k` given the `n`. The formula was described by Knuth in his book, which in turn was discovered by A. Hadian in 1969.

$$F(n) = \sum_{1 \le k \le n} \lceil \log_2(\tfrac{3}{4}k) \rceil,$$

For people who aren't on friendly terms with math: this formula is actually very simple to implement in code. Math people just like terse scary symbols.
Using C++, `F(n)` can be written as:

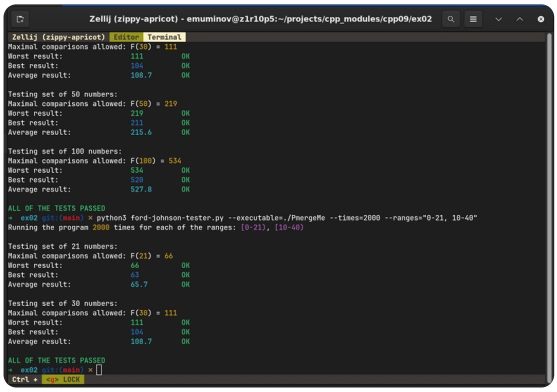
```
#include <cmath>

int F(int n)
{
    int sum = 0;
    for (int k = 1; k
        double value =
        sum += static_
    }
    return sum;
}
```

The next step after that is to implement in your code a comparison function, which goal is simply to increment some global/static variable each time its called. You use it every time you need to perform a comparison

between numbers, including passing it to `std::upper_bound` or a similar function, if you are going to use that.

Run functions a lot of times for a given `n`, using different `n`'s, and compare the biggest their worst in term of number of comparisons result to the expected maximal number of comparisons. If it exceeds this number, your implementation is not correct yet.



[I also wrote a single-file testing script] for benchmarking the algorithm. It requires your executable to print a sorted sequence and number of comparisons at the end of program.

This is an example of how I count the number of comparisons in my code:

```
////////////// ADD COMPA
// PmergeMe.hpp
class PmergeMe
{
  public:
// ...
    static int nbr_of_
// ...

template <typename T>
    PmergeMe::nbr_of_c
    return *lv < *rv;
}

// PmergeMe.cpp don't
int PmergeMe::nbr_of_c

////////////// USE _comp
// PmergeMe.hpp
// ... at the step 1
        if (_comp(next
        {
            _swap_pair
        }
// ...

// ... at the step 3 w
            typename s
                std::u
// ...
```

The code

I have implemented this

algorithm in the context of doing the last C++ exercise of module 09, in the 42 school; and you may find my take on this algorithm here, in cpp09/ex02. There are many ways to do this algorithm, my way is optimised for vectors and benefits a lot from random access.

Last note

Thank you epolitze for proofreading the first version of the article, sponthus for proofreading the second edition, and eandre for helping me to update the diagrams.
This is a second version of an article, where I simplified some explanations (hopefully made them more clear) and fixed a mistake which was pointed out by bwerner.

Original description of the algorithm, when runs out of the Jacobsthal numbers, inserts elements from `the pend` in the reverse order, and in the previous version of

the article I said that this doesn't make a difference according to my benchmarks; I was wrong! After writing the tester that rigorously tests ranges of numbers many-many times I found out that actually it matters whether it's in order or in reverse order. With this algorithm I had to run into unexplored waters, as there were no accessible articles or videos at the time of me doing the algorithm.

Top comments (17)

Benjamin Werner • Feb 10 '25 • Edited on Feb 11

Thank you for creating this great guide, it's very well explained and visualized and easy to follow.

However, if you implement the logic following this guide, and also implement a

counter to count the number of comparisons, you will find that you have too many comparisons. For example, it is stated in Art Of Computer Programming, Vol. 3 on page 184, that for sorting 21 numbers, merge insertion will take **at most** 66 comparisons. However, it took up to 70 comparisons with my implementation following this guide.

I found out that the issue lies in the insertion order. You wrote that if we have no more elements corresponding to a jacobsthal number in the pend, we have to sort the remaining elements *in order*, and that the 'odd element' should always be inserted last. This is incorrect. This became clear to me after looking into another paper which was also

referenced in the 'Art of Computer Programming' book: A Tournament Problem

Here we find a perfect example:

You can imagine A-J to be the main chain, and all the dots with the numbers under them would be the pend elements. (in this case b2 - b10). You will notice that the 'odd element' has no special significance here, it is *not* inserted last.

Note that the 'Order of Insertion' is clearly specified in that example:
b3, b2, b5, b4, b10, b9, b8, b7, b6

This means that we can simply add the 'odd element' into the pend and insert the pend elements back into the main chain according to the

jacobsthal numbers. Once we have no more elements that correspond to a jacobsthal number in the pend, we simply insert the remaining elements in reverse order starting at the last one (b10 in this case).

Here is another example to make it more clear:
Pend: b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16
Main: b1, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15
insertion order: b3, b2, b5, b4, b11, b10, b9, b8, b7, b6, b16, b15, b14, b13, b12

And with this change, you will never exceed 66 comparisons for sorting 21 numbers. In the 'Art of Computer Programming' book, on page 186 you will find a table listing the

maximum number of comparisons for sorting 1-33 numbers in the worst case, so you can double check and verify that you never exceed any of those. (There is also a formula to calculate the maximum for any given sequence length)

emuminov

Mar 21 '25 • Edited on Nov 3

Hello!

I finally fixed the mistakes in the article and mentioned your contribution.

Aymane Zgaoua

Mar 20 '25

Great details and resources, thank you so much man !!

ayouz0 • Jan 7

dwz dwz

emuminov • Feb 14 '25

Thank you for the detailed feedback.

Indeed, you are totally right. My mistake was that I made an odd number 'special'. When I'll have time, I'll review the article and update it.

Qi Ter Tay • Feb 13 '25

Hey @emuminov great article! But just like @bewerner I also implemented some changes to your existing code so that the nums of comparisons match those mentioned in the book. Would you be open to reviewing them?

emuminov • Feb 13 '25

Hello! Thank you. Yes, would love to see the changes. I can include them into the code and credit you and @bewerner if you want.

emuminov • Feb 16 '25

Didn't receive a reply, but still fixed this little issue in my code.

Qi Ter Tay • Feb 18 '25

Oh, my bad didn't get notice your reply. Anyway, I just looked through your code and think it's really similar to what I have. I would like to propose some minor simplifications on top of that and have created an issue on ur repo!

anders • Sep 4 '25 • Edited on Sep 4

Amazing article! This really helped me wrap my head around the recursive nature of main chain sorting!! <3

One tiny thing that might help people starting the project:

It can be a bit overkill to track every single element in the main chain just to ensure that `b_x` is never compared with `a_x`. The Ford-Johnson algorithm only requires that each insertion happens using at most `k` comparisons.

You can still achieve the maximum allowed comparisons by inserting into a "naive" main chain -- you don't need to care whether the element is an `a` or a `b`, etc. I wrote about this approach here ;)

emuminov • Sep 5 '25

Thank you for you feedback!
Yep. I think tbh that even the recursion is not mandatory for this algorithm, as long as the sorting principle is the same and the number of comparisons is under the threshold.
Great to see alternative approaches.

Peter Vivo • Nov 14 '24

Ineresting algorithm, thx for detailed describing.

soulayman bouabid • Dec 7 '24

thanks man keep going

si • Mar 14 '25

Thank you very much. It was very helpful for me!

Theau • Jan 4 '25

Excellent explanation, thank you so much, you really helped me here. I just spotted a little mistake you made : the jacobsthal formula is (2^(n+1) - -1^n) / 3 and not (2^(n+1) + -1^n) / 3.

emuminov • Jan 14 '25 • Edited on Jan 14

Thank you for the feedback.
The presented formula was taken from the Knuth's book, on the next page right after the algorithm was introduced.

Haithem Bendjaballah • Nov 4 '25

Actually The same thing (2^(n+1) + (-1)^n) / 3. = = 2^(n+1) - -1^n) / 3

---

because in the Wikipedia :

and let's replace the n with the k +1
j k+1 = 2k+1 - (-1)k+1 /3
Then :
-(-1)k * (-1)1. = + (-1)k
the Result :

The same in the book

Correct me if im wrong Thank you

Bright Data PROMOTED

SOC-CERT: Automated Threat Intelligence System with n8n & AI

Check out this submission for the AI Agents Challenge powered by n8n and Bright Data.

Read more →