UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2022-2023

Projeto 3 – Ordering

Versão (1.1)

Introduction

A Software Engineer must be able to implement a state-of-the-art algorithm from its description and/or pseudocode. In the 3rd Project we will learn and implement a relatively advanced sorting algorithm. To make the project easier, we're dividing the work into two main tasks/problems. In problem A we will implement a very simplified first version of the algorithm, but which already has some of the ideas of the final algorithm. In problem B we will implement another variant which is very close to the final version of the algorithm. The algorithm to be implemented is based on a combination of the MergeSort and InsertionSort algorithms.

To implement the project's functions, we recommend that you do it in the order presented here, as functions specified later may depend on functions defined earlier. Source code that you should use as basis for your project was provided in Tutoria, including the skeleton of the main classes to implement for problem A and B. We recommended that you carefully read the source code provided.

The tests in Mooshak are designed in a way for you to obtain points after each of the specified methods is correctly implemented. It is not necessary to have a fully implemented issue to get the corresponding quote on the project.

Problema A: *MergeInsertionSort* (7 valores)

Problem A starts with the implementation of a simplified version of the algorithm, which nevertheless has an interesting temporal complexity. We call this simplified variant MergeInsertionSort, as it will combine the MergeSort Bottom Up algorithm with the InsertionSort. Implement the requested functions in the MergeInsertionSort class.

1) traditionalBottomUpSort

Implement this method using the MergeSort Bottom Up algorithm, as taught in theoretical classes. No optimizations should be applied to the algorithm. This method will serve to make some comparisons later.

`void traditionalBottomUpSort(T[] a)`
Generic method of type T that receives an array of comparable elements, and that performs a MergeSort Bottom Up, sorting the array. This method requires additional memory on the order of O(n).

2) insertionSort

Implement the method using the Insertion Sort method, as taught in lectures. However, in this method the sorting is done only between the low index (leftmost element), and the high index (rightmost element to sort). Elements outside these indexes, if any, must not be altered.

void insertionSort(T[] a, int low, int high)
Generic method of type T that receives an array of comparable elements, and that performs an InsertionSort on a subset of the array, sorting it. Low represents the lowest subarray index to sort and high the highest subarray index to sort.

3) determineRunSize

Unlike traditional MergeSort Bottom Up, this algorithm does not start with subarrays of size 1. Instead it chooses a minimum size in the range [32, 64[.

As discussed in classes, one of the problems with MergeSort Bottom Up is that it can be up to 10% slower than the recursive version when the size of the array to sort is not a power of 2. This is due to the use of extra merges.

To mitigate this problem, the algorithm to be implemented uses an interesting technique to determine the minimum size that we will use for the initial subsets. Given the size n of the array, if n is a power of 2, we will choose 32, otherwise we will choose an integer value k between 32 and 64 so that ⌈n/k⌉ (the number of subsets at the initial level) is close of a power of 2. If ⌈n/k⌉ is an exact power of 2 we are in the ideal case, but if ⌈n/k⌉ is greater than the power of 2 closest to ⌈n/k⌉, we are in the worst case possible, as there will be many inefficient merge operations. Fortunately this worst case is easy to solve by choosing k+1 for the minimum size of the subsets. The following image illustrates the problem of ⌈n/k⌉ > 2^b, and illustrates how increasing k by 1 solves the problem, and effectively decreases the number of merge operations.

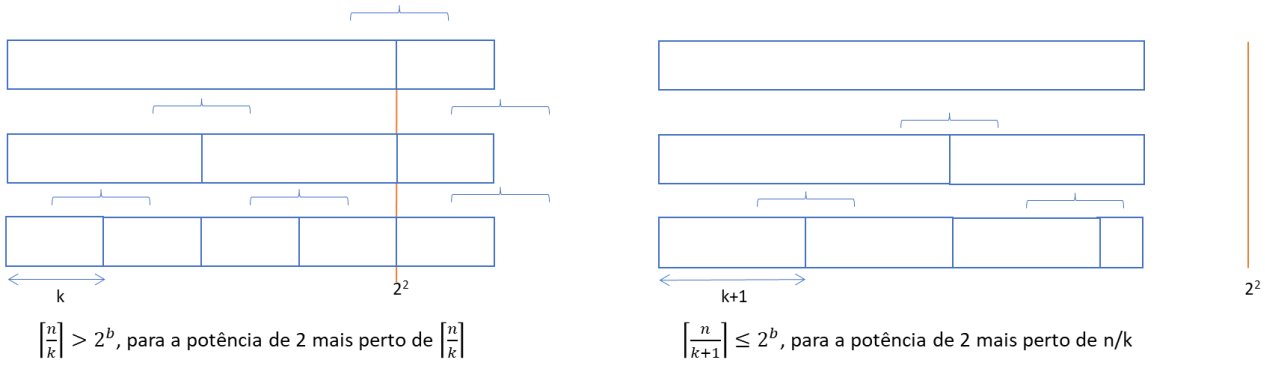


Figura 1 – Example of the problema when for a chosen k, $\left\lceil \frac{n}{k} \right\rceil > 2^b$, where 2^b is the power of 2 closest to $\left\lceil \frac{n}{k} \right\rceil$. In this situation, choosing k+1, ensures $\left\lceil \frac{n}{k+1} \right\rceil \leq 2^b$, leading to a fewer number of merges.

We can now specify how the determineRunSize method works, which will calculate this minimum size of the initial subsets.

We start with k = n, with n being the size of the array

We successively divide k by 2 (integer division), until we get a k in the interval [32, 64 [

If during the integer divisions of k by 2, there was at least one in which the division did not have a remainder of 0, this means that n is not an exact power of 2 (and neither is n/k), and that we are in the condition where ⌈n/k⌉>2^b. Fortunately, to solve this problem, just return as the chosen size k+1.

Otherwise, we return k. In this situation, n is an exact power of 2, and therefore k and k/n as well. This is the best possible case, where $n/k = 2^b$.

Example: given n = 500

Internaly k = 500, 250, 125, 62

Output: 63

`Int determineRunSize(int n)`
Method that calculates and returns the size in the range [32,64[to use for the smallest subarrays in order to minimize merge operations. It takes as an argument n, the size of the array to sort.

4) *sort*

Using the previous methods, we can now finally implement a more efficient version of sorting, by combining the MergeSort Bottom Up with the InsertionSort. The improved algorithm works like this:

If the size of the array n is smaller than 64, it's not even worth worrying about merges. We just do an InsertionSort and exit.

If it is greater than or equal to 64, we determine which size to use for the initial subsets according to the previous function, and we will sort each of the subsets obtained using the InsertionSort method.

Once we have all the initial subsets sorted, we use the MergeSort Bottom Up sorting technique, but instead of starting with groups of size 1, we start with the groups corresponding to the subsets already sorted in the previous step.

`void sort(T[] a)`
Generic method that takes an array of comparable elements of type T, and sorts the array using the method described.

5) Method for example generation

Next we will implement a set of methods that are useful to generate examples of arrays to sort with a certain size n. Although our algorithm works for any comparable object (eg Strings, Dates), for testing we will use arrays of integers. Implement the following methods as described.

`Integer[] generateRandomExample(Random r, int n)`
Returns an array of integers of size n. In each position of the array, a random number between 0 and n must be inserted. You must use the random number generator r received.
`Integer[] generateMostlySortedExample(Random r, int n)`
Returns an array of integers of size n. Approximately 90% of the elements placed in the array should be sorted in ascending order, and the remaining 10% are not. Any decision based on random numbers must use the random number generator r received.
`Integer[] generateAlmostSortedExample(Random r, int n)`
Returns an array of integers of size n. Approximately 99% of the elements placed in the array should be sorted in ascending order, and the remaining 1% are not. Any decision based on random numbers must use the random number generator r received.
`Integer[] generateAscendingExample(Random r, int n)`
Returns an array of integers of size n. The returned array must be sorted in ascending order. Any decision based on random numbers must use the random number generator r received.
`Integer[] generateDescendingExample(Random r, int n)`

Returns an array of integers of size n. The returned array must have a strictly descending order. Any decision based on random numbers must use the random number generator r received.

6) Analysis and comparison of sorting methods

Use empirical tests and doubling-ratio trials to study and compare the running time and temporal complexity of the traditionalBottomUpSort and sort methods. You can use the methods provided in the TemporalAnalysisUtils class. Compare both algorithms in different types of examples, and using sizes that are powers of 2, and sizes that are not powers of 2. You can use the example generators implemented in the previous exercise, but you can also create new types of example generators, if you wish.

7) Analysis and comparison of average execution time for small arrays

To confirm the hypothesis that InsertionSort works well for small arrays, analyze the average runtime for relatively small arrays (e.g. size = 32, size = 40) for the traditionalBottomUpSort and sort methods. For this analysis it is not possible to use doubling-ratio tests because we cannot increase the complexity of the problem. We can however measure the execution time of 10 000 (or 100 000) sorts of a small sized array. Do remember that sort methods change the original array, so be careful to ensure that the array to sort in the 10 000 tests is not always the same array (as it will be sorted after the first iteration).

For points 6 and 7, write a brief summary of the tests and results, and a brief analysis of the results as comments in the Java file to be submitted. The main method must invoke the tests and trials used. Although this method is not automatically validated by Mooshak, it will be taken into account in the project validation, and will count towards the final grade of the project.

Technical requirements: The requested methods must be implemented in the provided MergeInsertionSort class (MergeInsertionSort.java file). Other classes or helper methods must be defined in the same file. Submit only **the file MergeInsertionSort.java** to Problema A.

Problema B: SmartMergeSort

In this problem we will create a more complex version of the algorithm created in problem A, introducing new concepts one at a time. We will call this algorithm SmartMergetSort. Start by opening the provided SmartMergeSort class, and place the methods you implemented in Problem A which you consider relevant.

1) insertionSortWithInitialSortedHand

This method is very similar to the insertionSort implemented in problem A, however it has a slight difference. This method assumes that there is already an initial part of the received subarray that is sorted with each other. Assuming that this part of the subarray, which we call the hand (using InsertionSort's card sorting metaphor) is already sorted, we can start with a bigger hand and save some work. If there is no part of the subarray already sorted, we can always start with a hand of size 1, as we can assume that an element is always sorted relative to itself.

Void insertionSortWithInitialSortedHand(T[] a, int low, int n, int high)
Generic method that receives an array of comparable elements of type T, and that sorts the array from the low position to the high position (including). The argument n indicates the size of the initial hand, or the number of elements from the low position that are already sorted.

2) getNaturalRun

To understand this method, we have to introduce the concept of run[1] . A run represents a part of the array in which all elements are sorted together. A run is represented by an initial index (which marks the start of the run) and a size, which represents the number of elements from the initial index that are already sorted. We can externally represent a run using the {start+length} notation. Consider the following examples:

{0+1} – A run of size 1, which starts at index 0 of the array, and only element 0 is considered sorted.

{4+3} – A run of size 3, starting at index 4 of the array. The elements at indices 4, 5 and 6 are sorted between themselves.

The method that we will implement aims to get the next run of the array starting from a received index, using the least possible effort. For this we will simply try to see if there are already naturally ordered[2] elements, and if there are, we return a run with the number of naturally ordered elements found.

For instance, for array [**2, 3, 3, 6, 7**, 5], if we try to look for a run from index 0, run {0+5} would be returned. The first 5 elements are sorted. Element 5 is no longer part of the run, as it is not ordered relative to 7.

For array [**2**,1,3,5,7,4], if we try to look for a run from index 0, run {0+1} would be returned. The element immediately after 2 is not ordered relative to the previous one.

Run	*getNaturalRun(T[] a, int low, int high)*
Generic method that takes an array of comparable elements of type T, and will try to find the largest possible natural run that starts from the low position. The high argument represents the position of the last element that we must consider. This method should not test for any element other than high. The method finds the highest increasing natural run. Returns the run found.	

3) getNaturalOrMakeAscendingRun

In addition to looking for natural runs, another alternative that is also relatively simple is to look for a strictly decreasing natural run, and invert the elements of the array corresponding to the run, making it an increasing run. A strictly decreasing sequence is a sequence of elements of the array a where for all indices i of the sequence a[i-1] < a[i] is verified.

For example, for the array [**5, 4, 2**, 3, 2, 1] there is a strictly decreasing run {0+3}, as the elements 5, 4, and 2 are strictly decreasing. In this situation, the array between positions 0 and 2 is inverted, leaving [**2, 4, 5,** 3, 2, 1] and the run {0+3} is returned.

The getNaturalOrMakeAscendingRun method looks for either an ascending natural run, or alternatively a strictly descending natural run. To do this, start by comparing the 1st with the 2nd element of the sequence (if there is a 2nd element, because if there is no other element besides the first, a run of size 1 is always returned). If 1st >= 2nd, we are in the easiest case and the method will try to find the biggest natural run possible. If 2nd < 1st, then we have at least one strictly decreasing run of size 2, let's try to find the biggest decreasing run possible, invert the elements of the run and return the run.

[1] Implementation of this class was provided with the base source code. We advise the student to carefully read and understand the class Run.java.

[2] Naturally ordered elements are elements already ordered without us having to do anything.

Run	*getNaturalOrMakeAscendingRun(T[] a, int low, int high)*
Generic method that takes an array of comparable elements of type T, and will try to find the biggest possible run that starts from the low position. The high argument represents the position of the last element that we must consider. This method should not test for any element other than high. The method either finds the largest increasing natural run, or alternatively, if it finds a strictly decreasing natural run, it inverts all elements of the run, making it an increasing run. Returns the run found.	

Suggestion: if you have difficulties implementing this method, you can start by just implementing the previous one. This will allow the following methods to work, and you will already get a substantial part of the points in Mooshak. The downside is that your final sort method won't be as efficient for decreasing arrays.

4) getNextRunWithMinimumSize

This method is simple to explain. The goal is to generate a run that has at least a received size n. To do this, start by trying to get a natural run using one of the previous method. If the size of the natural run is greater than or equal to n, our run already has the required size, we can return it. Otherwise, our run is too small. We will have to force the next positions to be also ordered until we have the minimum size (or until we reach the highest index possible). Fortunately we have a good method for this, insertionSort. We use insertionSort to sort all the remaining elements required for the minimum size and return the corresponding run. However, when doing the insertionSort we need to consider that the elements within the natural run are already all sorted with each other. This will allow us to save some time.

As example, for array [2, 4, 5, 6, 7, 1, 2, 9, 8, 3,…], low=0 and for minimum size n = 8, the natural run returned is {0+5}, which is not the minimum size required. We invoke insertionSort for elements from index 0 to index 7, which will sort the first 8 elements. The array becomes [1, 2, 2, 4, 5, 6, 7, 9, 8, 3, …] and the run {0+8} is returned.

In another example, for the array [1, 2, 4, 5, 6, 7, 1, 2], low = 2 and minimum size n = 8, the natural run {2+4} is found which has no size 8. However, it is not possible to create a run of size 8. Anyway, insertionSort is invoked to sort positions 2 to 8. The array becomes [1, 2, 1, 2, 4, 5, 6, 7] and is returned the run {2+6}.

Este método é simples. O objetivo é gerar uma *run* que tenha no mínimo um tamanho recebido *n*. Para isso começa por tentar obter uma *run* natural usando o método anterior. Se o tamanho da *run* natural é maior ou igual a *n*, a nossa *run* já tem o tamanho necessário, podemos retorná-la. Caso contrário, a nossa *run* é pequena demais. Vamos ter de forçar a que as próximas posições estejam também ordenadas até termos o tamanho mínimo (ou até chegarmos ao índice *high*). Felizmente temos um método bom para isso, o *insertionSort*. Usamos o *insertionSort* para ordenar todos os elementos, e sabemos que no final podemos devolver uma *run* com o tamanho desejado. Mas atenção, que agora sabemos que os elementos dentro da *run* natural já estão todos ordenados entre si.

Run	*getNextRunWithMinimumSize(T[] a, int low, int high, int n)*

Generic method that receives an array of comparable elements of type T, and will try to find the largest possible run that starts from the low position, and that has a minimum size n (or high-low+1). The high argument represents the position of the last element that we must consider. This method should not test for any element other than high. If possible, the method tries to find a natural run that satisfies the minimum size, if not, it will sort the elements to meet the size requirement, and then return the desired run.

5) merge

Implement a merge method like the one used for problem A, but this time it takes a left run, and a right run, and merges the elements corresponding to both runs. The right run corresponds to a run that starts immediately after the left run.

`void merge(T[] a, T[] aux, Run left, Run right)`
Método genérico que recebe um array de elementos comparáveis do tipo T, e um array auxiliar com o mesmo tamanho, e vai fazer *merge* de duas *runs* consecutivas do *array*. *Left* corresponde à *run* esquerda e *right* à *run* direita.

6) mergeCollapse

Due to its complexity, we will provide mergeCollapse pseudocode directly. However, it is important to learn what this method does, why it does it, and how this relates to the sorting algorithm that we will implement in the next exercise.

Unlike the implementation made in Problem A, in which the merges are applied through two relatively simple chained cycles, the management of merges in SmartMergeSort is done in a smarter way, through a stack of runs[3]. Whenever the main algorithm extracts a new run from the array, this run is pushed onto the stack through a push operation, and the mergeCollapse method is called.

The mergeCollapse method aims to do merge operations between consecutive runs intelligently (in order to minimize the number of merges), while preserving the following two properties about the stack of runs.

(1) $r_{i+1}>r_i$

(2) $r_{i+2}>r_{i+1}+r_i$

Where r_i is the height of the stack run i, where i=1 is the top run on the stack, i=2 is the second run, and so on. The following figure shows an example of what it means to respect properties 1 and 2 described above.

[3] An implementation for the Stack of Runs is already provided in the base source code, in the file MergeStack.java.

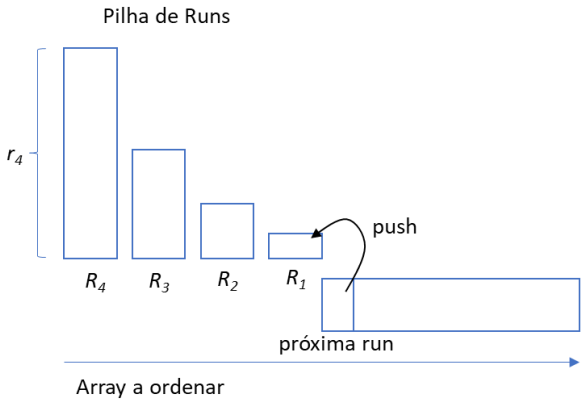


Figura 2 Exemplo de pilha de runs que respeita as propriedades 1 e 2.

The reason why these two properties are important is that they mathematically guarantee that at any given time, the number of runs stored in the stack is at most logarithmic (on the size n of the array). Next, we present the pseudocode of the mergeCollapse algorithm.

Algorithm 1: MergeCollapse

Input: A stack of runs $\mathcal{R}$

Result: changes to the stack and to the array, in order to balance the run lengths and ensure the stack properties:

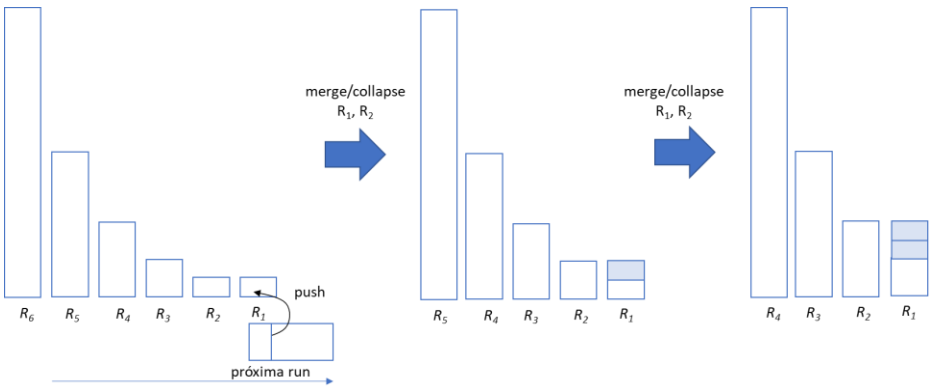
$r_{i+1}>r_i$

$r_{i+2}>r_{i+1}+r_i$

Note: At any time, we denote the height of the stack $\mathcal{R}$ by h, and its i^{th} top-most run by R_i. The length of run R_i is denoted by r_i.

```
1 while h>1 do
2   if h>=3 and r1>r3 then merge and collapse R2,R3
3   else if h>=2 and r1>=r2 then merge and collapse R1,R2
4   else if h>=3 and r1+r2>=r3 then merge and collapse R1,R2
5   else if h>=4 and r2+r3>=r4 then merge and collapse R1,R2
6   else break
```

An interesting feature is that the mergeCollapse method whenever possible tries to make the sizes of older runs double the size of newer runs. When this happens, the next time a new element with the same size is added, it will generate an efficient number of merges/collapses, as we can see in the following figure.



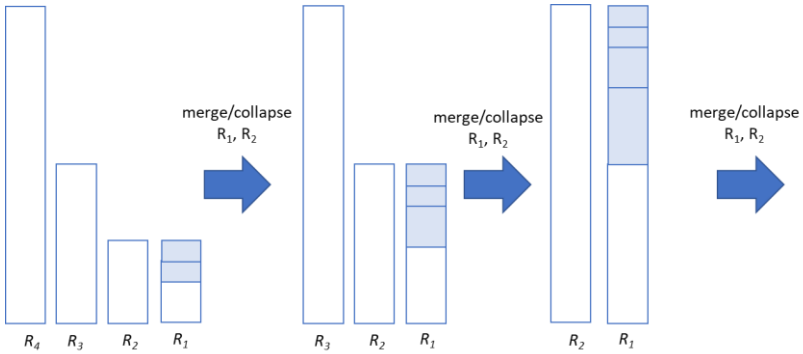


Figura 3 – Illustration of what happens when a new run is pushed into a stack where all runs are twice the size of the next run.

This method is also able to elegantly handle natural runs with a large size using the R2,R3 collapse rule. Note that unlike the sort algorithm in Problem A, in which the initial runs are almost always the same size (except for the last run which may be slightly smaller), in this variant we can at any time place a new natural run of arbitrarily large size, and the MergeCollapse method will try to merge and collapse as efficiently as possible.

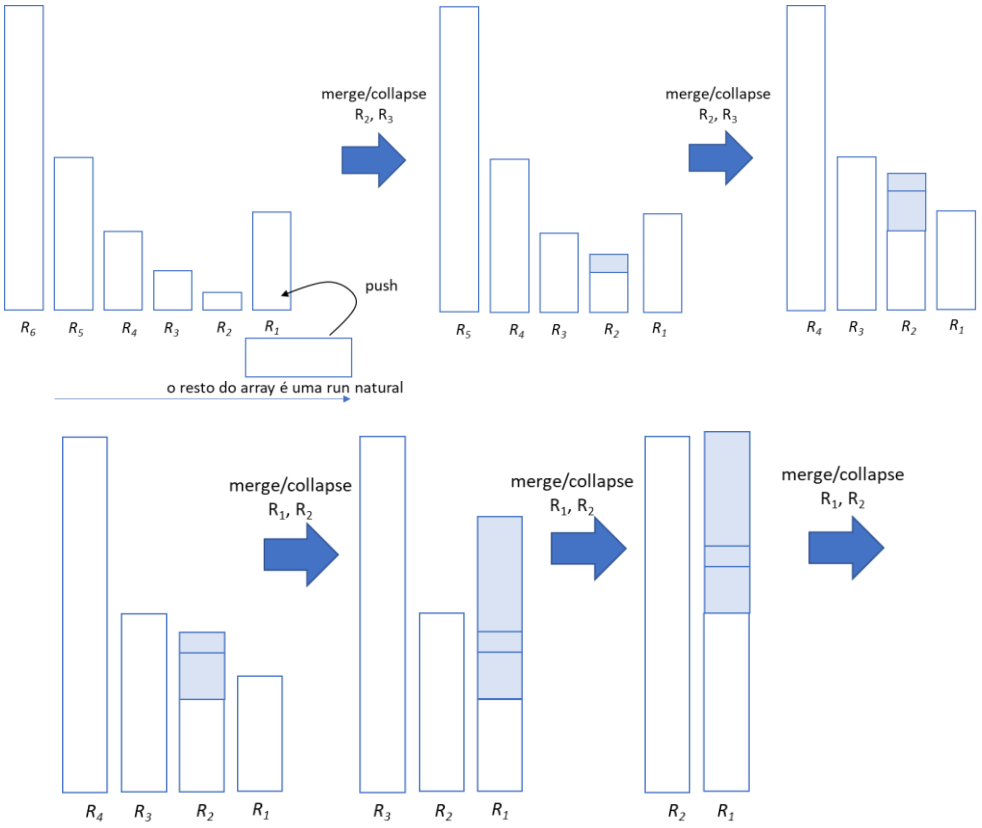


Figura 4 – Pushing a large natural run into the stack

Now that we understand the MergeCollapse method we can implement it. Implement the method according to the above pseudocode.

void mergeCollapse(MergeStack stack, T[] a,T[] aux)
Generic method that receives a stack (MergeStack) of runs for array a, an array a of comparable elements of type T, and an auxiliary array with the same size as a (which can be used as an auxiliary array for merge operations) , will make the merges and collapses intelligently to be as few as possible, and to guarantee the properties: $r_{i+1}>r_i$ $r_{i+2}>r_{i+1}+r_i$

7) sort

Now we have all the helper methods needed to implement the SmartMergeSort algorithm in a relatively simple way:

The algorithm starts by seeing if the array size is less than 64, in which case it's not worth complicating. Call InsertionSort and exit.

If it is greater than or equal to 64, let's roll up our sleeves and start by calculating the best minimum size to use for the algorithm runs, as was done in Problem A. We create auxiliary data structures such as the stack that will store the runs, and a temporary array for merge operations.

We start from the beginning of the array (left side) and move forward, always extracting the next run with the minimum desired size, until there is nothing left to extract from the array.

For each run extracted, we push the run onto the stack, and call the mergeCollapse method. We advance the array to the next position after the extracted run.

At the end of having done all this process for all the extracted runs, it is possible that there are still several runs in the stack for which the merges were not done. This could be because the mergeCollapse method is still waiting for more runs to make the optimal number of merge/collapses. However, since we've reached the end of the array, there are no more runs, and mergeCollapse can't help us anymore (since there are no smart ways to do merges) and we'll have to brute-force merge/collapses. Fortunately this is easy to do: as long as there is more than one run on the stack we merge and collapse the stack runs R1 and R2. When there is only one run on the stack, it means that this run necessarily starts at position 0, and has a size equal to the size of the array. In other words, it's all sorted. There's nothing else to do 😊.

Implement the sort method as described.

void sort(T[] a)
Generic method that receives an array a of comparable elements of type T and that will sort the array efficiently using the SmartMergeSort technique. This algorithm takes advantage of natural runs that may occur in the array to be as efficient as possible.

8) Análise do tempo de execução médio do *SmartMergeSort*

Implement a method for generating examples that has approximately one unordered element in every 1000 elements.

Integer[] generateLargeNaturalRunsExample(Random r, int n)
Returns an array of integers of size n. Approximately 99.9% of the elements placed in the array should be sorted in ascending order, and the remaining 0.1% are not. Any decision based on random numbers must use the random number generator r received.

Compare the average running time of the SmartMergeSort algorithm with the MergeInsertionSort algorithm of problem A. Note that the asymptotic time complexity of SmartMergeSort is equal to that of MergeSort, and therefore the differences will be more relative to the average running time. Compare on various types of examples.

9) Complexidade Temporal Assimptótica do *SmartMergeSort*

When in the previous exercise we said that the asymptotic time complexity of SmartMergeSort is equal to that of MergeInsertionSort, this was not exactly true. It is the same for almost all cases, but there is a type of examples for which the asymptotic time complexity of SmartMergeSort is different. Determine in what kind of situations this occurs and estimate the asymptotic complexity of the SmartMergeSort algorithm for those situations. You may have to increase the number of iterations in the doubling-ratio tests to be able to correctly estimate this value.

For exercises 8 and 9, write a brief summary of the tests and results, and a brief analysis of the results as comments in the Java file to be submitted. The main method must invoke the tests and trials used. Although this method is not automatically validated by Mooshak, it will be taken into account in the project validation, and will count towards the final grade of the project.

10) Utilização de memória eficiente

This is the last exercise, and it is very difficult. Only the most advanced students will be able to implement this point. Worth 1 value of the final grade of the project.

So far, our implementation of SmartMergeSort requires an amount of additional memory equal to the size of the original array. If for instance we want to sort an array of 100 000 000 elements this can be problematic.

We will briefly describe the technique used by the SmartMergeSort algorithm to occupy substantially less additional memory. It is possible to implement a variant of the merge method that only needs an amount of additional memory equal to the size of the smallest run that we want to merge. For example, if we merge a run of size 1024 with a run of size 512, we only need an array of size 512 to be able to merge.

For this we need two merge methods. A merge method that merges from the left of the array to the right of the array - we can call it mergeLeft - and another that merges from the right of the array to the left - mergeRight.

If the left run size is smaller than the right run size, we should use mergeLeft. Here the trick is that we just need to copy the elements from the left run to an auxiliary array with the size of the left run. The elements on the right run stay in the original array and are not copied. This implies that the way indices are used in the traditional merge method must be changed to take this into account. This optimization works because when we are going to change the position of the elements on the right in the array, either the element that was there has already been placed in another position (and therefore we no longer need it) or it will be placed in the same position.

If the size of the run on the right is smaller than the size on the left, we should use mergeRight. In mergeRight we just need to copy the elements from the right run to an auxiliary array with the size of the right run. The left run elements stay in the original array and are not copied. This has several implications for indices, as in the previous method.

Note that: in mergeLeft (and in traditional merge) we move the smallest of the elements between the left and the right to the leftmost unfilled position; but in mergeRight, we move to the rightmost unfilled position the largest of the elements between the left and the right.

If the size of both runs are the same, either use mergeLeft or mergeRight.

Implement this technique so that your sort method uses less memory.

Requisitos técnicos: The requested methods must be implemented in the provided SmartMergeSort class (SmartMergeSort.java file). You can use any class provided (eg Run, MergeStack) with the project's base code in your implementation. However, if you make changes to any of these classes, those changes will not exist in Mooshak. If you need other classes or helper methods, these must be defined in the same file.

Submit **only file *SmartMergeSort*.java** to Problema A.

Condições de realização

The project must be carried out individually. The same or very similar projects will lead to failure in the discipline. The faculty of the discipline will be the only judge of what is considered or not to be copied in a project.

The project code must be delivered electronically, through the Mooshak system, by **11:59 pm on the 18th of November**. The validations will take place the following week, from the **21st to the 25th of November**. Students will have to validate the code together with the teacher during laboratory hours corresponding to the shift in which they are enrolled. The evaluation and corresponding grade of the project will only take effect after the validation of the code by the teacher.