Algorithms For Big Data - Graded Exercise 1.2

Daniel Kaestner - i6119196

February 6, 2020

High-level description

Due to the restiction that $i \leq j$ we cannot brute-force the *BIGJUMP* algorithm efficiently. Without this restriction we could find a solution in linear time by iterating over the array once, finding the minimum and the maximum element. However, due to the restriction this method may result in incorrect solutions and hence we would need two for loops for i and j in order to find a solution. This method has a runtime of $O(n^2)$. This quickly becomes infeasible and we therefore want to find an alternative algorithm with a lower runtime.

Solving the problem in a divide-and-conquer-style results in a runtime of $O(n \cdot log(n))$. The approach chosen for this is very similar as the example provided in the lecture. To begin the algorithm we split the array in the middle into two parts L and R for the left and right half. This is repeated recursively for the sub-arrays until we obtain atomic elements which we cannot split anymore as seen in Figure 1. The method allows us to divide the problem into smaller sub-problems which are easier to solve.
Consider the atomic case. The solution of the biggest jump problem for an array with only one element is clear. Since we only have one element, $i = j$ and hence $a[j] - a[i] = 0$. If we now want to merge two sub-arrays back to the initial, we can find the best i and j of the merged array using the following three options:

1. The biggest jump is in L (the left sub-array)
2. The biggest jump is in R (the right sub-array)
3. The i of the biggest jump is in the left sub-array and the j is in the right sub-array

We only have these three options because $i \leq j$ and therefore j cannot be in L while i is in R. When merging the sub-arrays we simply have to compute the jump size of the three options and keep the i and j of the best option for the next merge. Since we apply a recursive method we already know the jump size of option 1. and 2. because we have computed them in the step before. Furthermore, we can compute 3. by finding the smallest element in the L ($\min(L)$) and the biggest in R ($\max(R)$). This is the case, because the biggest jump consists of the smallest number at index i and the biggest number at index j, where $i \leq j$. Reusing the elements from the prior results is an atomic process of runtime $O(1)$ and finding the minimum and the maximum in an array of size n can be done in $O(n)$ time.

Pseudocode

```
function BigJump(i,j)
    if $i == j$ then
     return i, j                                                          ▷ O(1)
    else
        $i_L, j_L$ = BigJump(i,M) // Split the array in the middle, where $M = \lfloor \frac{i+j}{2} \rfloor$   ▷ O(1)
        $i_R, j_R$ = BigJump(M,j)                                          ▷ O(1)
        $i_{LR}, j_{LR}$ = MinMax(i,j)                                     ▷ O(n)
    end if
     return best i, j
end function
```

```
function MinMax(i,j)
    min = 0;
    max = 0;
    for k = 0 until n do
        if a[k] < a[min] then
            min = k
            if a[k] > a[min] then
                min = k
            end if

     return min, max
```

Runtime analysis

The total runtime of the algorithm is $O(n \cdot log(n))$. Breaking this down into all components we have two parts. First the splitting process and second the merging process. The splitting process takes is linear and takes $O(n)$ time. Finding the middle of the array takes $O(1)$ and we have to do this for every position in the array. After the splitting, we obtain $log_2(n)$ layers of sub-arrays. When merging we need $O(n)$ time for each layer as it contains in total n elements and because we have to find the minimum and the maximum of the sub-arrays we have to iterate over all n elements. Combined, the merging takes $O(n)$ for $log_2(n)$ layers, hence the total runtime is: $O(n) + O(n \cdot log_2(n))$ which can be summasired to:

$$\begin{aligned} &\mathrm{O(n}\ \cdot log_2(n)) + O(n \cdot log_2(n)) \\ =&\mathrm{O(2}\ \cdot (n \cdot log_2(n))) \\ =&\mathrm{O(n}\ \cdot log_2(n)) \end{aligned}$$

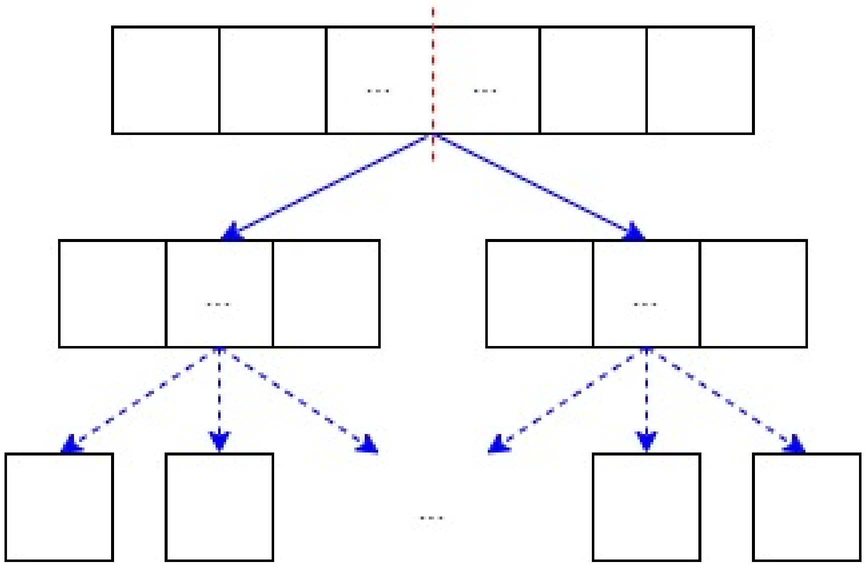

Figure 1: Illustration of divide process

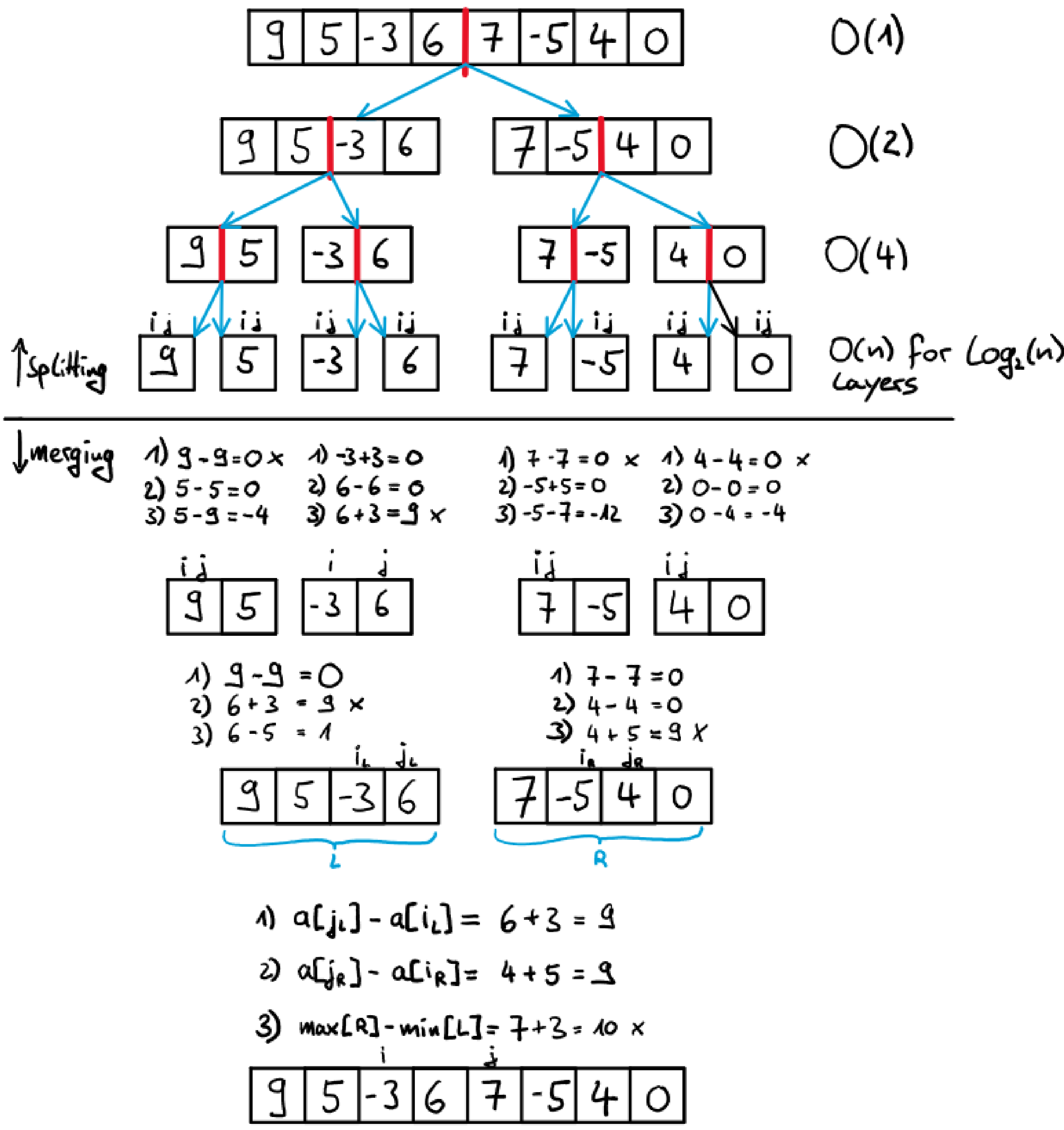

Figure 2: Example of the algorithm on the provided example