EXERCISE 8: SYNTENY BLOCKS

Programming in Bioinformatics

The Breakpoint Sort

Genomes are affected by large structural modifications such as inversions, transpositions, translocations, deletions or duplications of small or large portions etc. The chromosomal regions affected by these rearrangements are called breakpoints, while those which have not been rearranged are called synteny blocks.

The breakpoint sort algorithm is a technique used to detect rearrangement breakpoints in chromosomes. It is a refinement of the synteny blocks construction method, which focuses on the breakpoints themselves. The algorithm involves narrowing down the breakpoints as precisely as possible given a set of synteny blocks. The technique is based on the concept of signed permutations, which assigns numbers and signs to denote orientation to the synteny blocks. The most common genomic rearrangement event is reversal, where a contiguous section of a chromosome is reversed, and the orientation of all synteny blocks within the section changes. The sorting by reversals problem is a related problem that involves converting one signed permutation into the identity permutation.

Sorting by Reversals

Let $\pi = \pi_1\ \pi_2 \ldots \pi_n$ be an unsigned permutation of n distinct numbers and $1 \le \pi_i \le n$. The reversal $\rho = \rho(i, j)$ for $1 \le i < j \le n$ applied to π reverses the values of $\pi_i\ \pi_{i+1} \ldots \pi_{j-1}\ \pi_j$ and thus transforms π into a permutation $\pi \cdot \rho(i, j) = \pi_1 \ldots \pi_j\ \pi_{j-1} \ldots \pi_{i+1}\ \pi_i \ldots \pi_n$, for example: π = 1 2 **4 3 7 5** 6, then $\pi \cdot \rho(3, 6)$ = 1 2 **5 7 3 4** 6. The identity permutation I is the permutation where each $\pi_i = i$ for $1 \le i \le n$. The distance $d(\pi)$ between π and I is the minimum number of reversals ρ that convert π to I.

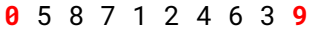
The Breakpoint Sort

The breakpoint sort is a greedy algorithm that minimizes the number of breakpoints between synteny blocks. The reversals are performed in such a way that at each step the number of breakpoints is reduced or is unchanged. All steps of the algorithm for breakpoint sort are described below in the example.

Example

π = 5 8 7 1 2 4 6 3

1) permutation extension
   Add at the beginning by 0 value and at the end by n + 1 value.

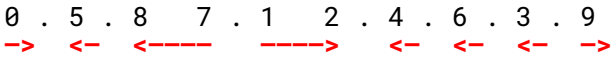


2) breakpoints

Mark parts that are not consecutive by adding breakpoints between inconsecutive blocks.

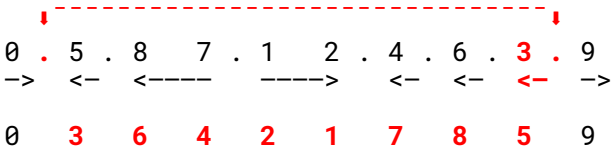
3) ascending and descending parts

The marginal values are always marked as ascending (→). Isolated values (single value between two breakpoints) are always marked as descending (←).

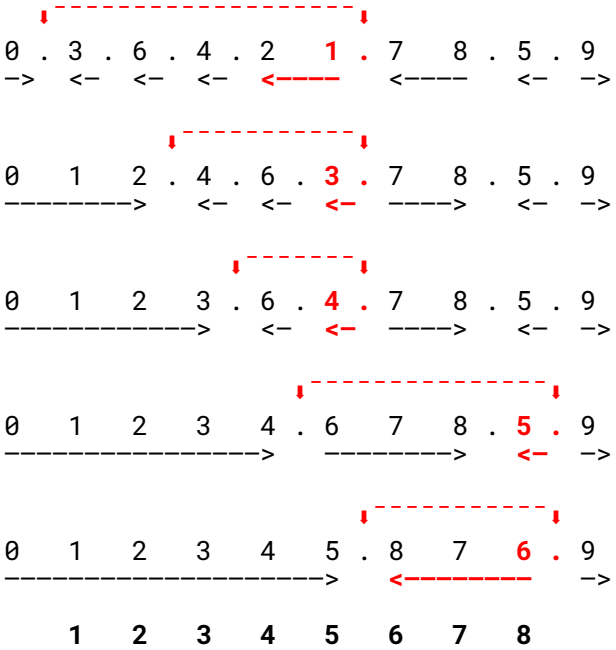


4) reversal

Find the descending part with the smallest value at the end. Reverse the region between the first and the breakpoint following the selected descending part.

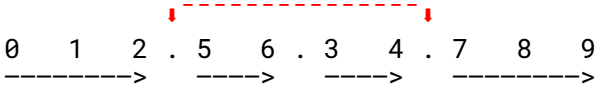


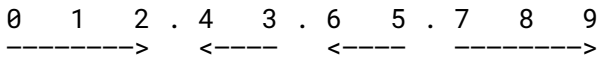
5) repeat steps 2 - 4 until the permutation is sorted



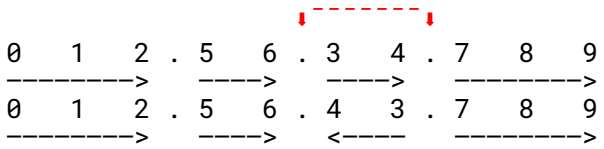
No descending part?

- Reverse the entire block between the first and last breakpoint.



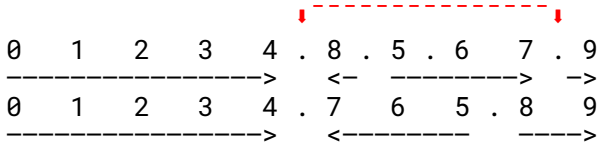


- Reverse the ascending part that starts with the value "end of sorted part" + 1



One descending value in front of the sorted part?

- Reverse the entire region between the first and last breakpoint.



More examples:

- π = 5 3 2 1 8 7 4 6
- π = 3 6 4 2 1 7 8 5
- π = 7 8 1 2 3 9 10 6 5 4

Algorithm

Implement the algorithm for the breakpoint sort that consists of three functions. First function `FindSorted()` will find an index where the unsorted part of the permutation starts e.i. it will mark the position of the first breakpoint. The second function `IndicateAscending()` will create a vector of the same length as the permutation and mark ascending parts by ones and descending parts by zeros. Finally, the last function `BreakPointSort()` will perform sorting by reversals according to the steps described above.

Task 1

- In R, create a function `FindSorted()` to find an index, at which the unsorted part starts.
- Input:
  - a permutation of integers e.g. `0 1 2 3 6 7 4 5 8`
- Output:
  - a position where the unsorted part starts at e.g. 5

Hint:

Compare successively values of the permutation with an increasing number starting at zero (`0`, 1, 2, ...) and ending at length of the permutation - 1. The comparison ends when the value in permutation is not the same as the tested value or when the tested value is equal to the length of the permutation - 1.

Task 2

- In R, create a function `IndicateAscending()` to mark ascending and descending parts of the permutation.
- Input:
  - a permutation of integers e.g. `0 4 5 3 2 1 6 7 8`
- Output:
  - a vector of zeros and ones, where ascending parts are marked by `1` and descending by `0` e.g. `1 1 1 0 0 0 1 1 1`

Hint:
Create an indication vector of the same length as the permutation containing only `0` values, and then set the first and last values to 1. The ascending parts of the permutation vector will be marked with `1` values in the indication vector. Create a loop that iterates through the permutation and if two values next to each other are ascending, i.e. the second is the first + 1, then the indication vector is set to 1 at the given indexes.

Task 3

- In R, create a function `BreakPointSort()` to sort a permutation using breakpoints.
- Input:
  - permutation of integers e.g. `5 1 4 3 7 8 9 2 6`
- Output:
  - sorted permutation of integers e.g. `1 2 3 4 5 6 7 8 9`

Hint:
Add marginal values to the permutation and the following steps are repeated in the loop:

- find the start of the unsorted region,
- mark ascending/descending parts,
- find the smallest value that is marked as descending part,
- reversal between the start of the unsorted region and the smallest value marked as descending part.

The loop ends when the permutation is sorted. Watch out for collision situations i.e. no parts marked as descending or there is a single value marked as descending in front of the sorted part of the permutation.