SoC 2023: Competitive Programming

Week-4: Divide-Conquer, Greedy, DSU

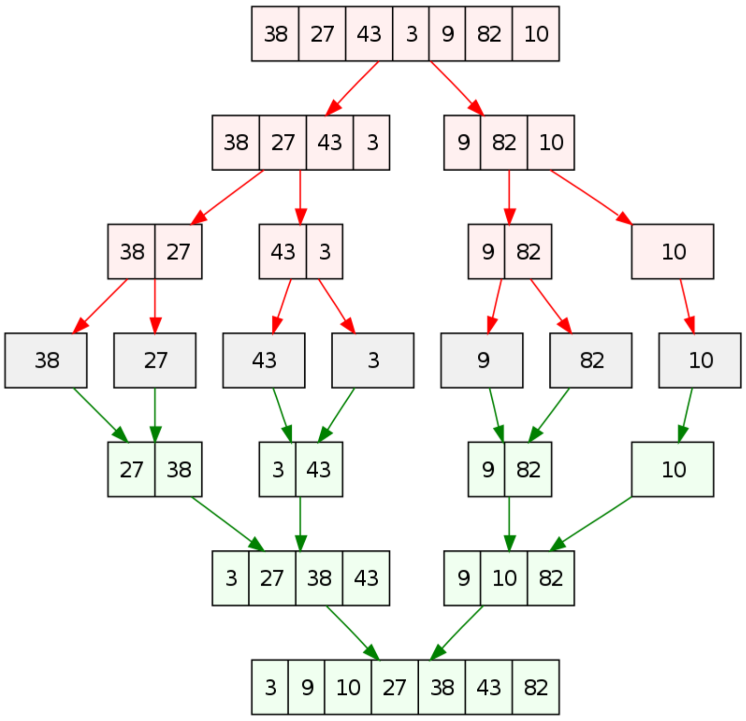
Mentor: Virendra Kabra

Summer 2023

Contents

1 Divide and Conquer

- Idea is to split the problem into subproblems, solve them recursively, and finally merge the results.
- Mergesort is a good example

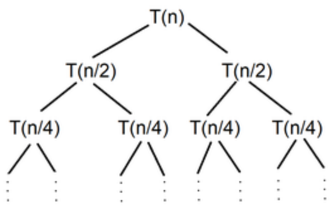


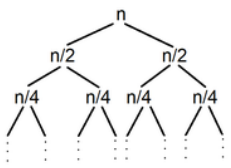
1.1 Time Complexity

- The solution can be thought of as a tree.
- For mergesort, height of this tree is $O(\log n)$, where n is the size of array. In each step, we do $O(n)$ work of splitting and merging. So, overall complexity is $O(n \log n)$.
- This can also be done with a recurrence. Let $T(n)$ be the time complexity taken for an input of length n. Then,

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

This can be solved with a tree as above

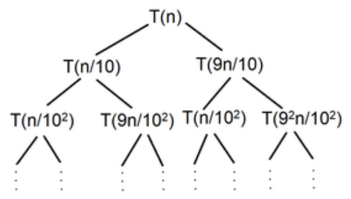


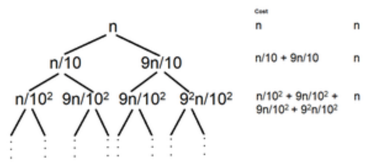


Each level adds up a cost of n, and there are $\log n$ levels.

- Another example

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + O(n)$$





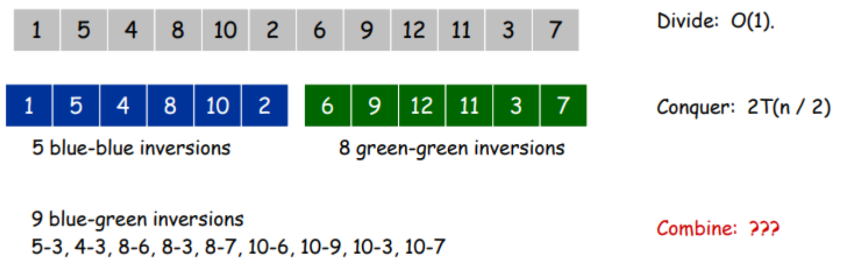
Right-most branch is the longest, with a length $O(\log_{\frac{10}{9}} n)$.

- More examples in CLRS Section 4.4

1.2 Examples

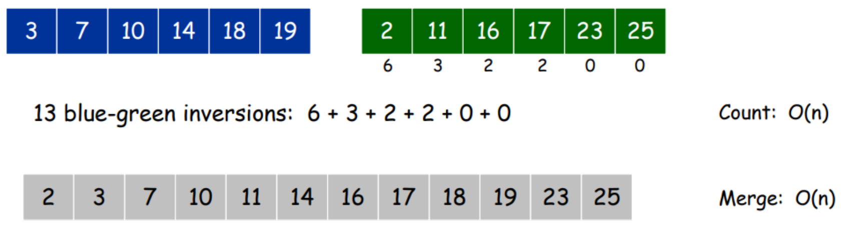
Code in files.

- Counting Inversions. Reference.
  For array `a`, `(i,j)` is an inversion pair if `i<j` and `a[i]>a[j]`. $O(n^2)$ approach is to iterate over all index pairs.



To combine, considering pairs from the two sub-parts gives a recurrence $T(n) = 2 \cdot T(\frac{n}{2}) + O(n^2)$, that is $T(n) = O(n^2)$.

What if the two sub-arrays are sorted? Inversion pairs can be counted in $O(n)$. To maintain the 'sorted' invariant for earlier layers of the recurrence tree, we also need to merge these arrays. This can be done in $O(n)$ (merge routine of mergesort).
In the merging process, we compare first elements of subarrays. For example, $3 > 2$. Since arrays are sorted, all elements in left are greater than 2, and we add 6 to the inversion count.



- Codeforces 1400E

You have a multiset containing several integers. Initially, it contains a_1 elements equal to 1, a_2 elements equal to 2, ..., a_n elements equal to n.

You may apply two types of operations:

1, 4, 1, 1
{1, 2,2,2,2, 3, 4}

- choose two integers l and r ($l \le r$), then remove one occurrence of l, one occurrence of $l+1$, ..., one occurrence of r from the multiset. This operation can be applied only if each number from l to r occurs at least once in the multiset;
- choose two integers i and x ($x \ge 1$), then remove x occurrences of i from the multiset. This operation can be applied only if the multiset contains at least x occurrences of i.

type-1
0, 3, 0, 0
type-2
0, 0, 0, 0

type-2
1, 1, 1, 1
type-1
0, 0, 0, 0

What is the minimum number of operations required to delete all elements from the multiset?

Input

The first line contains one integer n ($1 \le n \le 5000$).

The second line contains n integers $a_1, a_2, ..., a_n$ ($0 \le a_i \le 10^9$).

[1, 0, 2, 0, 3]

Examples: `[1, 4, 1, 1]` outputs `2`, and `[1, 0, 1, 0, 1]` outputs `3`. The array represents frequencies, and goal is to make all elements zero.

- $n \le 5000$, so $O(n^2)$ would work.
- Observation: For any subarray, it is not optimal to perform Type-1 operations (on any of its parts) after any Type-2 operation. Consider the array `[2, 3, 1, 4]`. Apply operations in the order 2 ($i = 2, x = 1$), 1 to get `[1, 1, 0, 3]`. This could be obtained with 1, 2 ($i = 2, x = 1$) as well. Operations 2 ($i = 2, x = 3$), 1, 1 on the initial array give `[1, 0, 0, 3]`, which can be obtained with lesser number of operations 1, 2 ($i = 2, x = 2$).

2, 2, 1, 4

2, 0, 1, 4
1, 0, 0, 3

1, 2, 0, 3

- Further, solution with only Type-2 takes n operations.
- A divide-and-conquer approach: We have two choices
  1. Use Type-1 operations to bring the smallest element to 0, then recursively find min ops for subarrays
  2. Use Type-2 operations only

1, 2, 3, 4
0, 1, 2, 3
0, 0, 1, 2
0, 0, 0, 1
T(n) = T(n-1)+O(n)

Complexity: $O(n^2)$ in the worst case.

2 Greedy

Make locally optimal solutions to get to a globally optimal solution. There can be many greedy strategies to a problem, most (or all) of them not being optimal.

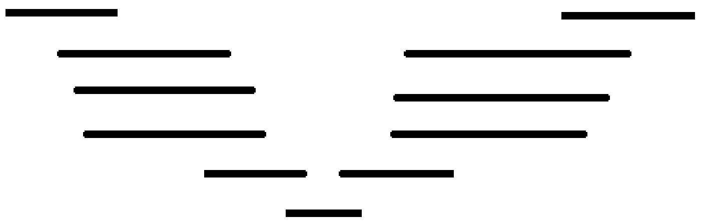
2.1 Examples

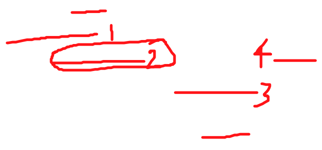
- Activity Scheduling. We saw parts of this in Sorting. Given a set $S = a_1, \ldots, a_n$ of n activities, and can do only one activity at a time. Each activity has a start time and a finish time. Activities are compatible if their intervals do not overlap.

  We wish to select a maximum-size subset of mutually compatible activities. An exponential-time algorithm is to iterate over all possible subsets.

  Greedy strategies that do not work: Counterexamples to prove them non-optimal.

  - Select activities greedily by smallest start time.
    Counterexample: $\{(0, 10), (1, 2), (3, 4), (5, 6)\}$ - first interval overlaps with rest all.
  - Select an interval with the least number of overlaps with other intervals



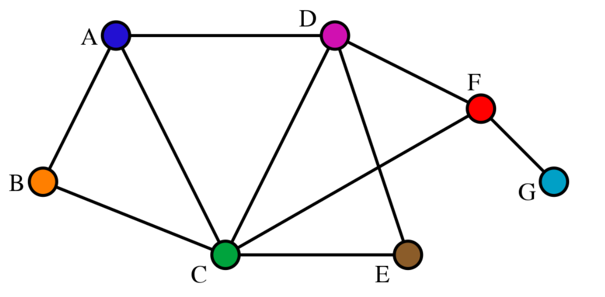


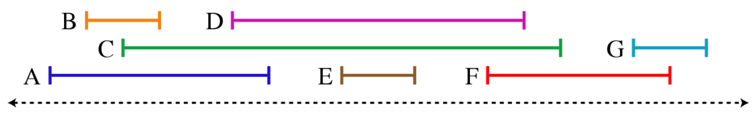
    Get $\{1, 6, 11\}$, optimal is $\{1, 5, 7, 11\}$.

  - Select the shortest interval (and repeat). Counterexample: $\{(1, 10), (8, 12), (11, 20)\}$.

  Optimal greedy strategy is to select an interval with smallest *end* time. Remove any intervals that overlap with this, and repeat.
  Proof: For any optimal solution, we can always replace the first interval with the interval having smallest end time. Doing this repeatedly will give us an optimal interval set. Alternatively, this would also work with largest start times.

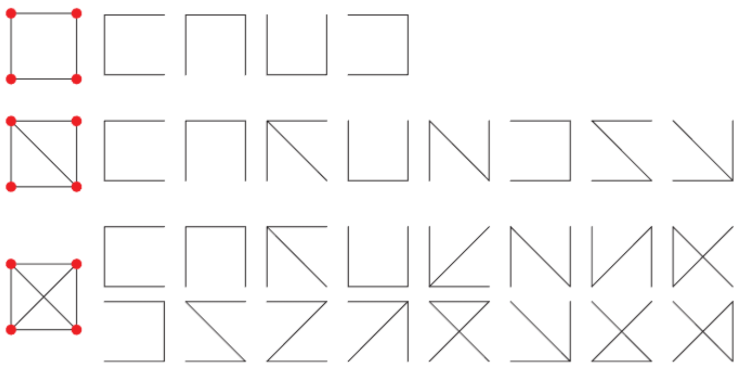
- Interval-graph coloring problem. Find the minimum number of lecture halls for all classes.



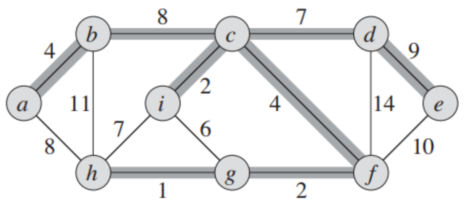


  Start with A in hall 1. Halls 2, 3, and 4 for B, C, and D. Reuse 1 for E, and so on.
  Greedy strategy: Sort activities by start time. For each activity, schedule it in the first empty lecture hall. If no such hall exists, assign a new hall.
  Optimality Proof: The m^{th} hall is added only when the first $m - 1$ are occupied at the moment. So, there are at least m overlapping intervals, requiring at least m halls.

- Dijkstra's algorithm is greedy, choosing the node with *current* smallest distance.
- Minimum Spanning Tree (MST) of a graph.
  Spanning Trees:



MST: Spanning tree with minumum weight.



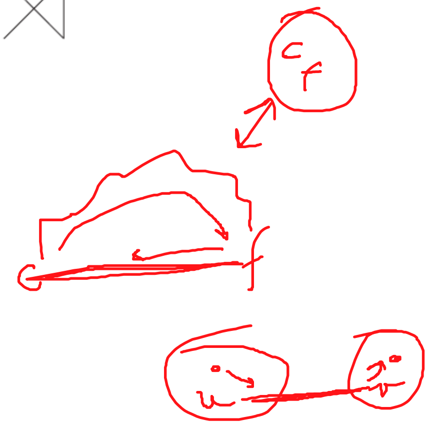


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

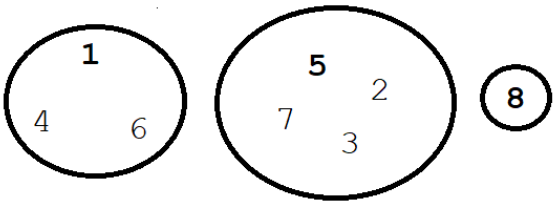
Kruskal's algorithm for MST: Select the smallest-weight edge that does not form a cycle with edges selected earlier.

1. Sort edges in increasing order of weights.
2. At each step, select an edge with the above property.

1 takes $O(m \log m)$. 2 can be done in $O(m^2)$ with cycle-detection ($O(m)$ traversal) at each step. This is sped up using Disjoint-Set Union data structure.

3 Disjoint-Set Union (DSU)

Aka Union-Find, owing to its two main operations.
We are given several sets of elements. Each set has a *representative* element (boldface).



Operations:

- `make_set(v)` - Initialization
- `get(v)` - Return representative of set that `v` belongs to
- `union_sets(a, b)` - Combine the sets that `a` and `b` belong to

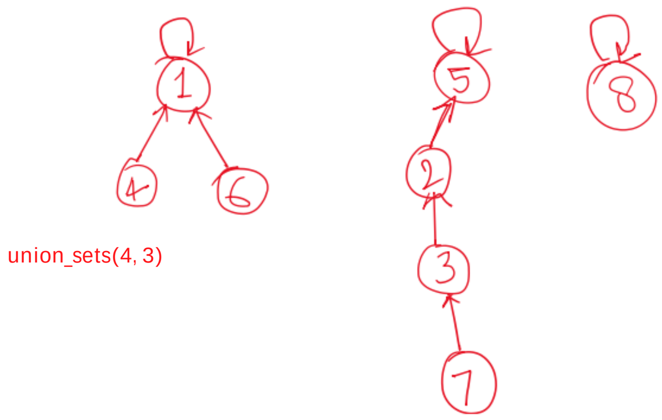
3.1 Array Implementation

Index	1	2	3	4	5	6	7	8
Repr	1	5	5	1	5	1	5	8

Representative	**Elements**
1	1, 4, 6
5	2, 5, 7, 3
8	8

- `get(v)` - Access the representative array.
- `union_sets(a, b)` - Find representatives, and combine sets. This is costly, as we need to copy entire arrays for each union.

3.2 Tree Implementation



The implementation just uses a `parent` array:

Index	1	2	3	4	5	6	7	8
Parent	1	5	2	1	5	1	3	8

Note: `parent[i]` is `i` if and only if `i` is a representative element.

- `get(v)` - Traverse up the tree with `parent`, and return the root. This is costly for long branches.
- `union_sets(a, b)` - Find representatives `r` and `s` using `get`. To prevent long branches, we merge the smaller (by size) set into the larger one. Let `r` represent the smaller set. Then, set `parent[r] = s`. This is called "Union by size". Another optimization "Union by rank" gives an equivalent complexity.

This gives $O(\log n)$ complexity for both operations.

It is further improved using "Path Compression":

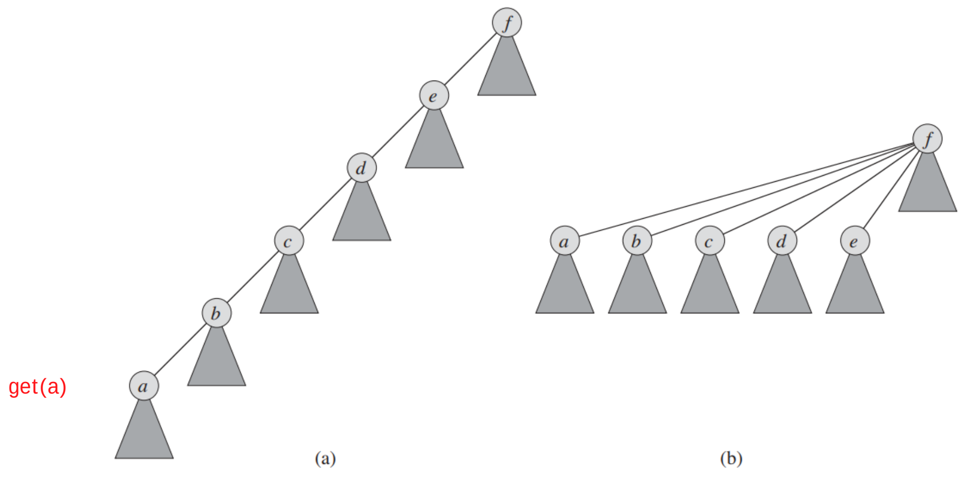


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Code in file. Overall time complexity: Reference.

4 Todos

Check sheet.