Divide and conquer methodology and applications to data structure problems

Junhua Chen 14/12/2020

The red tape

- Legally obliged to encourage you to get tested if you feel iffy

Table of contents

- Part I: Divide and conquer ideas:

    -Mergesort: an refresher

    -Answering range queries: range knapsack

    -Parallel binary search

    -Divide and conquer dp

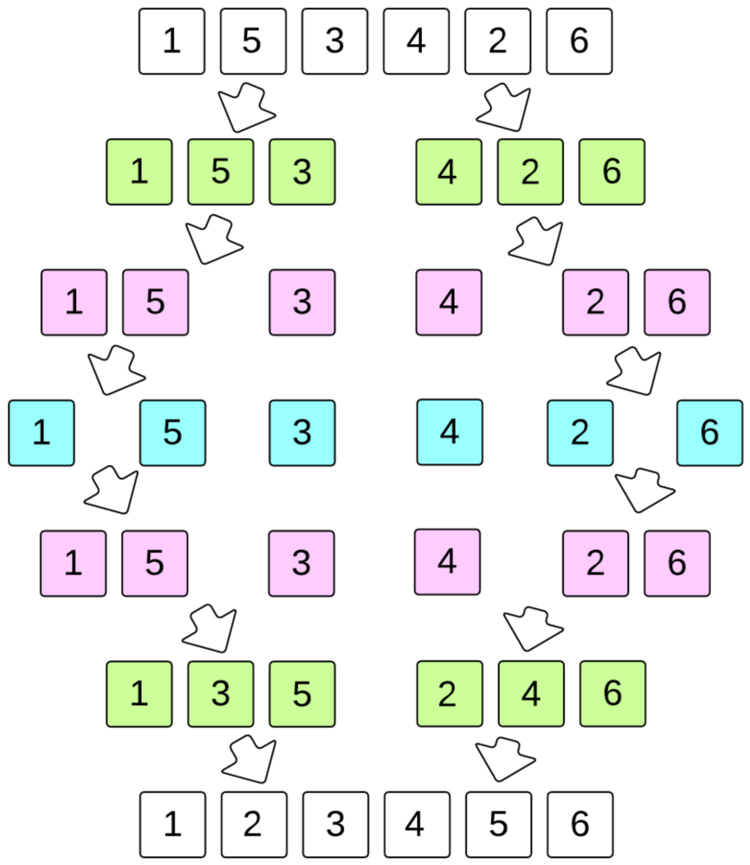
Aim: examine common D and C design ideas

Divide and conquer

- Main idea is to split a problem up into smaller and pieces of similar nature, and use work done on smaller pieces to piece together a solution to the bigger problem
- By its own virtue, splitting a problem up into smaller pieces also reduces unnecessary computation as will be seen soon

Mergesort: the original divide and conquer

Do we all know this?



Working space

Before we go any deeper: The maths

- Invariants and complexity analysis (often through recursions) are essential to understanding divide and conquer
- We introduce the Master theorem which helps use analyse divide and conquer algorithms

Master theorem for D & C complexities

- Basically just remember this:

a	b	k	comments
1	2	0	Binary search
2	2	1	O(NlogN), basically every divide and conquer you'll ever do (mergesort)
2	≥ 4	1	gg

Combining the three cases above gives us the following "master theorem".

Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

Useful note: if there are log factors attached to the n^k term, ignore them and at the end multiply T(n) by these log factors

Explainer

- Mergesort:

  $T(N) = 2T\left(\frac{N}{2}\right) + O(N)$ -> T(N) is complexity for mergesort on an array of size N

Explain why this recursion is so?

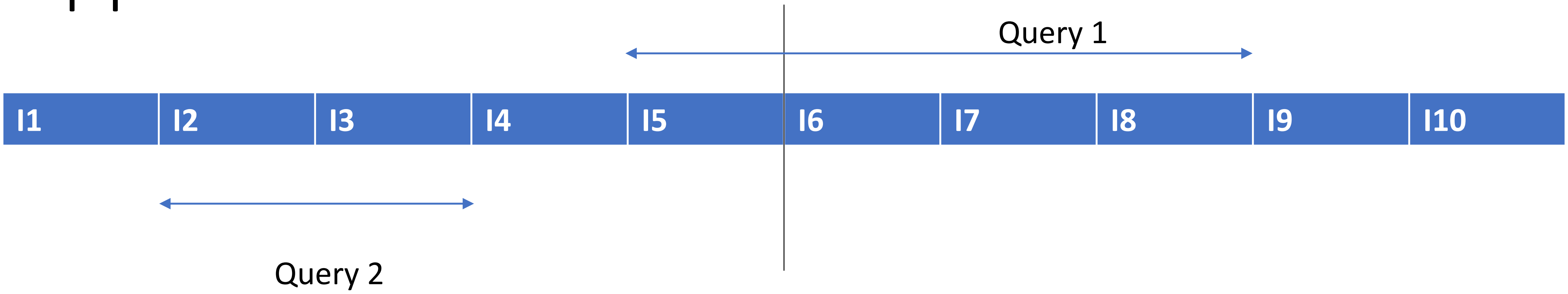
Problem 1: Smaller problem = lighter work

- You have N items 1 … N, each one with weight and value
- Answer offline queries of form: Given a knapsack capacity ci and a range li,ri, find the answer to the 0-1 knapsack problem using items li … ri.
- TL DR: range knapsack query
- Let maximum capacity and weight be C.
- Try to find a $O(CNlogN + QC)$ algorithm (3 minutes thought)

Approach 1: segtree of knapsack dp tables

- Let each ST node store a dp table with the maximum values for each query capacity in a range.

Approach 2: D & C



Lets recursively split the items in the middle (like above)

Eventually, any query range will be cut in half (in above case query 1 is split)

Consider the following D & C: slv(l,r, Q): Q is the set of queries contained wholly in [l,r]. Thus slv(1, N, all the queries) solves all the queries (yay!)

Solution on next page

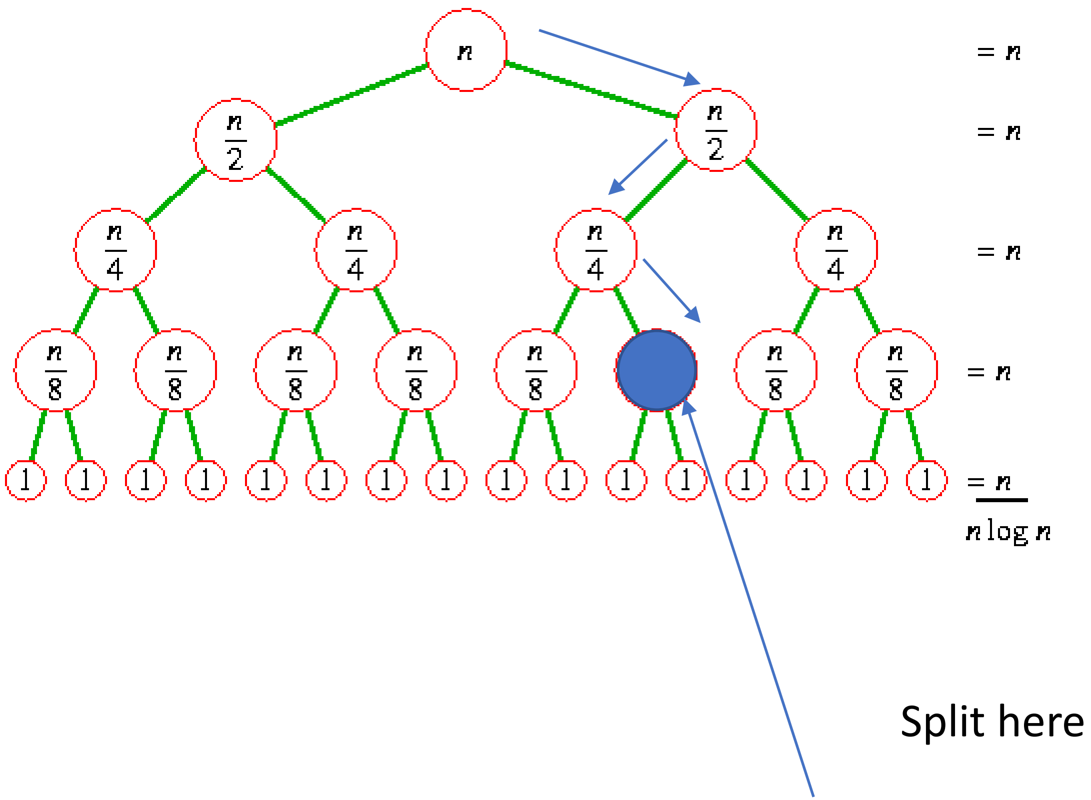
Solution outline

```
void slv(int l, int r, queryset Q) {
    //process all queries Q
    //all queries contained in [l,r]
    if l == r or Q empty //base case
        LMAO TRIVIAL //just math it or return
        return
    ENDIF

    int mid = (l + r) / 2; //split in middle
    COMPUTE best1[i][j] AND best2[i][j] where
    best1[i][j] is the best value obtained FOR capacity
    j AND items i ... mid AND best2 is similarly defined
    FOR mid+1 ... i //this takes O((r-l)C) total trivially
    //by classic knapsack

    for each query q in Q split by mid DO
        for i in range 0 ... q.cap DO
            q.ans = max(q.ans, best1[q.l][i] + best2[q.r][q.cap-i])

    divide remaining queries into qleft and qright
    slv(l, mid, qleft)
    slv(mid+1, r, qright)
}
```

Analysis of D and C and why it works

- A common way to analyse divide and conquer is to use recursion. In this case let $T(N) = cost\ to\ call\ slv\ on\ a\ range\ of\ size\ N$
- $T(N) = 2T\left(\frac{N}{2}\right) + O(CN)$ (two equal sized subproblems recursed on)
- Regarding queries: Think about recursion tree for the D & C. it is a shallow complete binary tree. Each query moves down the tree in one path, taking $O(logN)$ time per query to move to its split point. Meanwhile to solve each query at the split point takes $O(C)$.
- Now back to recursion: do maffs

Queries visualised



The recursion

- In our case we have the classic case (C is constant multiplier as it is independent of N) so we get $T(N) = O(CNlogN)$
- Combined together gives complexity $O(CNlogN + QC)$
- Key takeaways: recursion complexity analysis, use invariant thinking

CDQ: D and C makes things easier

Consider the following problem:

N by N grid ($N \leq 10^6$)

update: add x to a rectangular range

query: what is value of a cell? (offline) (Q operaions total)

Lets examine how D and C is a potent approach to **offline** solve a dynamic data structure task

Offline = operation sequence known in advance

Static = no update operations. Antonym is dynamic

This technique is known as CDQ after its populariser (and maybe inventor?)

Familiar with CDQ? Try this exercise instead

- JOI 2020 Spring Camp: Sweeping
- Feel free to discuss the solution with me in your spare time

A closer look at this problem

- Offline (as in problem)
- Dynamic (there are updates, so its not static)
- Standard approaches (involving 2D segment tree) are slow and painful to code.
- We should reduce the **dimensionality** of the problem

Okay – So 2d segtrees are a pain

- Source: everyone who's ever coded one
- D and C offers a way to circumvent this
- **Key idea:** visualise set of operations as a timeline
- For Instance:

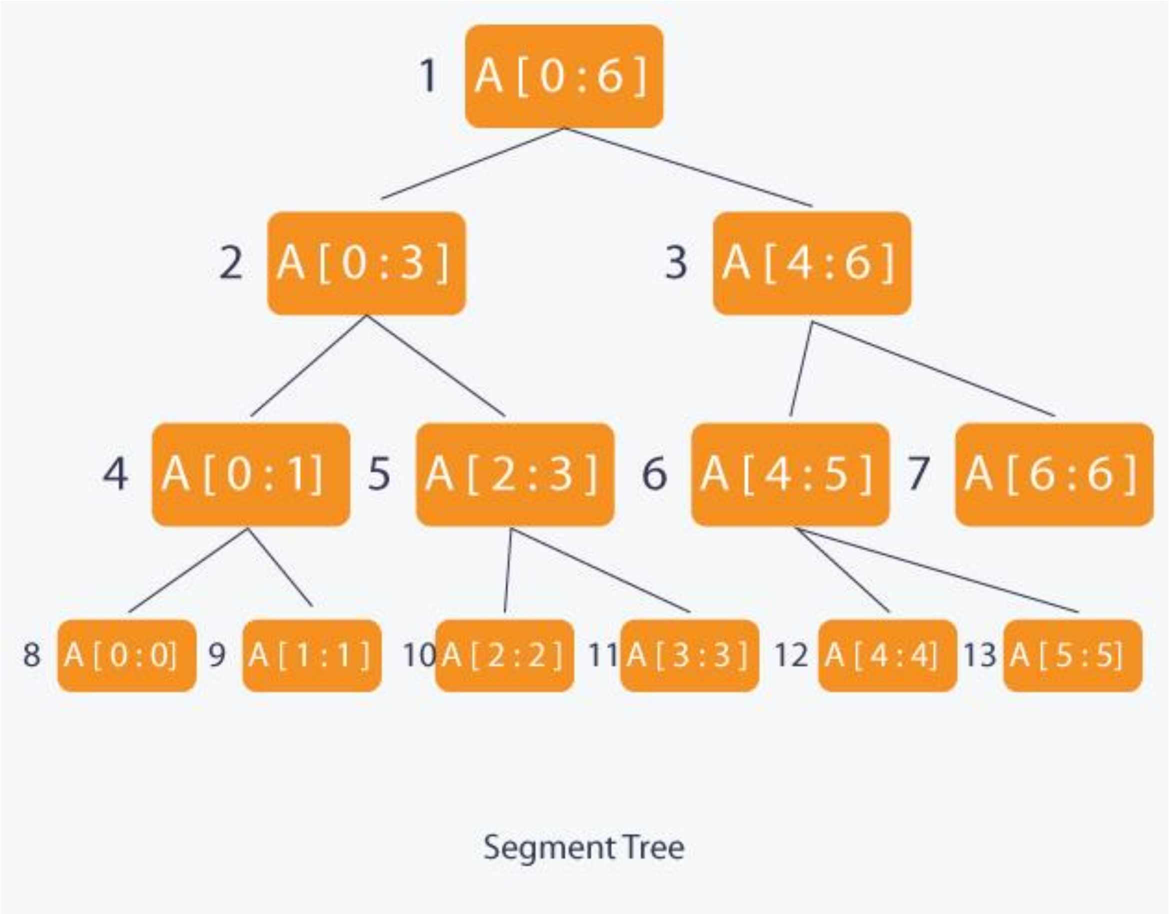
U	Q	U	U	Q	Q	U	Q	Q	Q

- Anything that looks like an array is an array – Confucius
- So lets do D and C on this timeline!

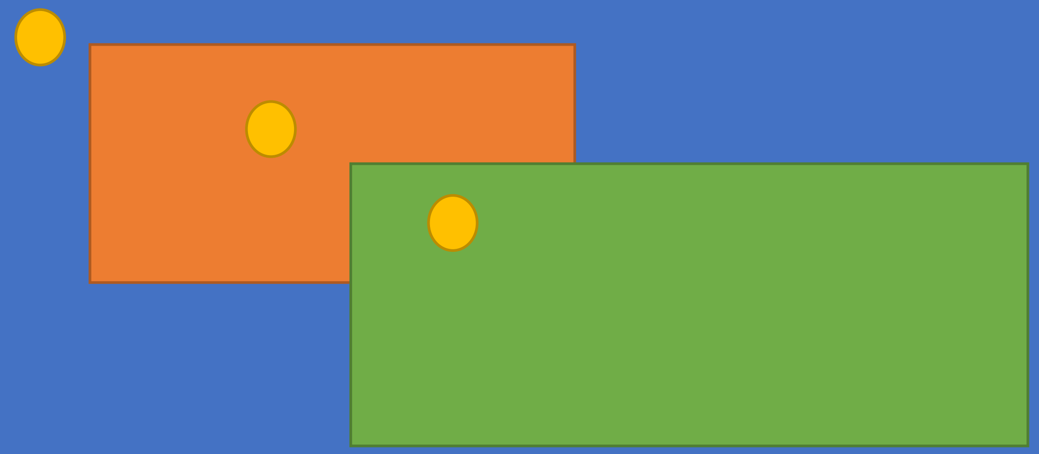
The D and C

```
void CDQ(int l, int r) {
    //goal: once this returns, all updates in l ... r will have their effects
    //applied to all queries in this range
    //CDQ is OP when updates dont really affect each other (so here)
    if l == r
        return //we are done, nothing to do

    CDQ(l, mid) //if you consider the CDQ as traversing the recursion tree of the D and C
    CDQ(mid+1,r)  //then this CDQ does a postorder traversal of the tree
    int mid = (l + r) / 2; //split
    //now we have applied the effects of updates in [l ... mid] to queries in that range
    //and effects of [mid ... r ] to queries in same range
    //it remains us to calculate the effects of updates in [l ... mid] to queries in [mid+1 ... r]
    let S = updates in l ... mid, T = queries in mid+1 ... r
    maintain sweepline and segtree to see effect
}
```

CDQ as preordering a segment tree built on timeline– Lets draw it out!



Filling in details: Example of affecting queries



The circles are queries and rectangles are updates. Here, the green update only effects 1 query whereas the orange one affects 2

What has the problem become?

- How do we sweepline and segtree to apply the updates [l … mid] to queries [mid+1 … r]?
- Note that now the problem is static, all updates applied before queries.
- Maintain segtree on y axis and sweep: at a point in time, it represents its cells represent the values of the grid column cut by sweepline

  -hit left edge of update: range add onto segtree

  -hit right edge of update: range decrement onto segtree

  -arrive at query: access segtree cell

Complexity of CDQ

- $T(Q) = 2T\left(\frac{Q}{2}\right) + O(QlogN) \rightarrow T(N) = O(QlogNlogQ)$ using master theorem
- Faster than segtree
- While not applicable here, mergesort is often combined with CDQ

The mentality of seeing D and C as traversing a segment tree can also come useful in many places, but alas we don’t not have time for all these applications

The Core power of CDQ

- **CDQ:** shaves operations off a DS, eventually turning a dynamic into a static problem (can be nested)
- Done by converting problem into such that all updates are made before any queries

- Exercise: maintain fully dynamic CHT (i.e with deletions as well as insertations of line) (offline)

Divide and conquer DP: a motivator

5	1	1
1	5	2
2	2	5

- Suppose there is a N by N grid of unknown integers. For all columns i you IF x,y are such that: (x,i) and (y,i+1) are positions with maximum value in their column THEN $x \leq y$.
- Find the maximum values on each grid by making $O(NlogN)$ queries of form: What is the value at a position in the grid? (2 minutes)

Solution

Observations:

-Let the maximum value of column i be $opt(i)$

-Then $opt(i) \leq opt(i+1)$ is given. So monotonicity!!!

-How do we use this to prune our search?

A solution

```
int lo = 0; //lowest point that might be optimal
For i = 1 … N DO
    max[i] = find best in lo … N
    lo = lowest value that is maximum

//Does this work? Why or why not?
```

Attempt 2: Lets use D and C

```
void slv(int l, int r, int olo, int ohi) {
    //find max for columns l ... r, know optimum is in olo ... ohi
    int mid = (l + r) >> 1, opt = -1, Max[mid] = -INF; // find max for mid
    for i in olo ... ohi DO
        if Max[mid] < query(mid, i) THEN
            Max[mid] = answer_to_query
            opt = i
        ENDIF
    ENDFOR

    if (l == r)
        return //we're done
    ENDIF

    slv(l, mid-1, olo, opt);
    slv(mid+1, r, opt, r);
}
```

Correctness

- Is every potentially optimal location evaluated? (check tiebreaking)

- Is it fast?
- Hint: Yes, but we will prove it in a bit

D and C dp

- Lets consider it visually: Different colours represent a different recursion call (also different layers)

Layer 1								
Layer 2								
Layer 3								
Layer 4	Insert	8	colours	here	I	cant	be	stuffed

- Recursion is log levels deep and at each level each row position is evaluated once or twice so geometrically complexity is NlogN

How do we use this in other problems?

- See monotonicity? This
- Can use to optimise dp recursions (especially 2D dp)
- Might not want to prove the monotone property (its usually a pain) but guessing + writing D and C doesn’t take much time (low risk)
- Mentality behind it (of avoiding doing necessary evaluations you know will be suboptimal or degenerate) is often a key element to a problem

Parallel binary search

- Interesting and very elegant technique coming in more useful than you think making use of the previous diagram
- Basically run lots of binary searches at the same time
- Example problem: https://codeforces.com/contest/484/problem/E
- 5 minutes read and think

What do we do?

- For a single query consider the following solution (bsearch):
- What if we can run binary searches for all queries at the same time and avoid scanning the array multiple times?

```
Int lo = 0, hi = MAX
While lo + 1 != hi DO
      mid = (lo + hi) / 2
      check if there is a run of W fence posts
      with height at least mid in [l,r] //O(N)
      if so DO
            lo = mid
      else
            hi = mid - 1
Answer lo
```

Problem left as exercise

- Based on our discussion, try to write a D & C invariant that works
- The solution will be discussed afterwards
- Hint: Preorder traverse a segment tree!

Summary

- Things we looked at today:
  - -CDQ
  - -Divide and conquer DP
  - -D and C as traversing a segment tree
  - -applications to offline query problems
  - -parallel binary search: exercise, solution will be given later
  - -the power of invariants in designing these approaches
- Tackle problemset after working on this

Solving parallel bsearch

The ideas

- A useful approach in data structure problems is wishful thinking
  - -So what if we have a black box (BB) that can do:
    - -Given array, positions are on and off (BB.on(x))
    - -operations: turn position on and off (BB.off(x))
    - -query longest run of on positions in a range (BB.ask(l,r))
  - -in O(logN)
- We will solve this issue later

The D and C

```
void slv(int lo, int hi, vector <query> Q) {
    //the answer to this set of queries is in [lo, hi]
    //the function finds the exact answer to each query
    //see, its like doing a lot of bsearches at the same time
    //assume heights in 1 ... N (coordinate compress)
    //precondition: exactly the cells with heights 1 ... lo-1 on
    if l == r THEN
        ANSWER THE QUERIES //cuz theres only 1 answer
        return //WE DONE!!
    ENDIF

    int mid = (lo + hi) / 2;

    TURN on every cell with values in [lo, mid] //O((hi-lo)logN)
    ENDFOR

    vector <query> Ql, Qr;

    FOREACH query q in Q DO
        if BB.ask(q.l, q.r) >= q.w THEN
            place q in Qr; //answer to q greater than mid
        else
            place q in Ql;
        ENDIF
    ENDFOR
    slv(mid+1, r, Qr);
    TURN off every cell with values in [lo, mid] //O((hi-lo)logN)
    slv(l, mid, Ql);
}
```

Discussion

- Argue the correctness of the algorithm
- Argue that the algorithm runs in $O(Nlog^2N)$
- What ideas of D and C we saw earlier are present in this solution?

    -segment tree inorder traversal

    -queries making their way down the call tree
- Exercise: find a way to implement black box Hint: Consider the segment tree lecture in Data structures I

Problemset

Priority	Problem
1	Counting Inversions (basic D and C) and implementing the CF problem
2	Battleship II: Electric boogaloo
3	mobile phone
4	Meteor
5	Arranging heaps

- Extension problem: IOI 2014 Holiday

Thank you for your attention!

- Further reading:
  - Centroid decomp: divide and conquer on tree
  - CP-algorithms: offline dynamic graph connectivity
  - Codeforces pages
  - D and C has numerous applications in interactives