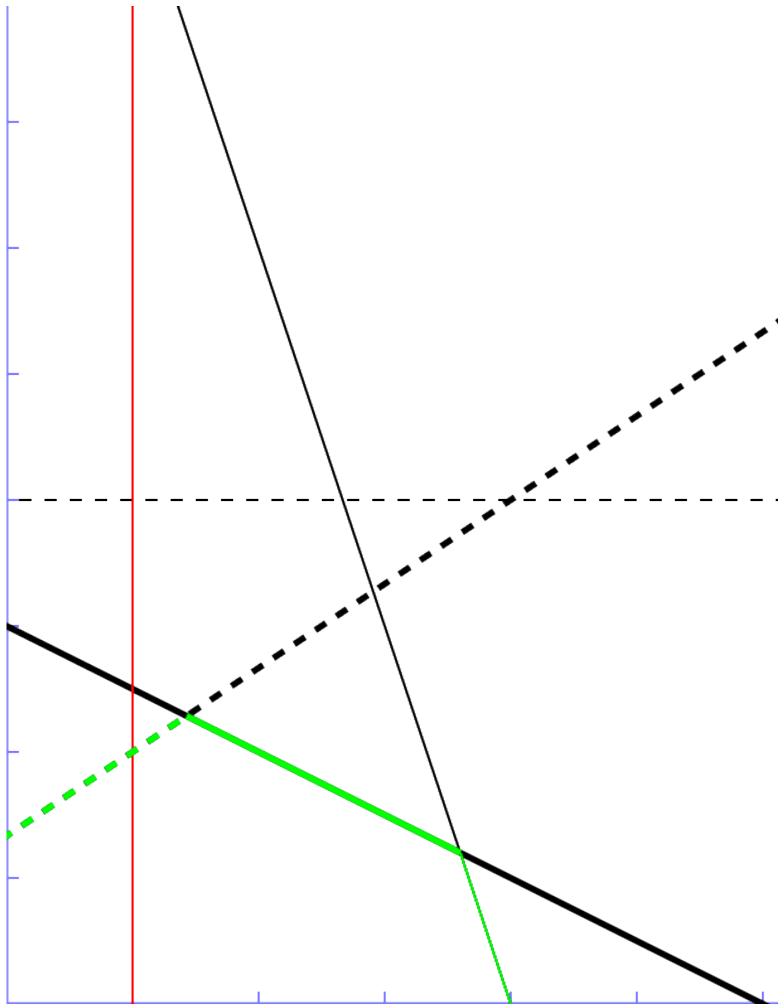
DP Optimizations

1. Convex Hull Trick
2. Divide & Conquer
3. Knuth optimization

A geometry problem

- Given n lines: $y_i = m_i * x + c_i$
- Query: For a given x (find min(y_i))
- Naive: O(NQ)

Observation 1: lower envelope



Source of Image: wcipeg

Construction of lower hull

- Sort lines by decreasing order of slope
- Similar to convex hull, maintain a list/stack of relevant lower hull lines so far and their interval
- Initially add line with interval (-oo, oo)

Construction – part 2

- For each new line l3 and top 2 stack lines l1 and l2 (l2 is top)
  - If size of stack < 2?
    - X = intersection(l2, l3)
      - If parallel?
- Consider intersection xold = intersection(l1, l2) and xnew = intersection(l1, l3)
  - If l1 and l3 parallel?
- When should we remove l2?
- After removing all unnecessary lines, add (xnew, oo)

Answering queries

- At end we will have array of (x1, l1), (x2, l2) ... (xm, lm)
- X1=-oo, xm = oo, x1 < x2 < x3 ... < xm
- Anwering for a particular x?
  - Binary search

Problem #1: CF 319C

- N trees with heights a1, a2, ... an
- A1 < a2 < a3 < ... < an
- Cost of cutting one unit of a tree:
  - Recharge after each cut
  - Let k be the maximum fully cut tree (current height ak' = 0) then recharge cost for ONE unit is bk
  - B1 > b2 > ... > bn
- An = 1, bn = 0
- N <= 10^5

O(N^2) dp solution

- Cut last tree in minimum cost
- Since bn = 0 all cuts after that are free!
- Let dpi = min cost to cut ith tree
- Dpi = For each j < i try to cut j first then cut i
  - Cost to cut i after cutting j will be bj * ai
- Min over j (dpj + bj * ai)
- O(N^2), N = 10^5 – rekt!

Observation

- dpj + bj * ai
- yj = cj + mj * x
- Mj = bj, cj = dpj for 1 <= j < i
- Minimum y for x = ai
- Profit!

Algorithm – O(N lg N)

- Mj = bj, cj = dpj for 1 <= j < i
- Minimum y for x = ai
- Bj < bj+1 – slopes already sorted in required order
- Maintain a deque
  - Stack + array access
- Dp1 = b1
- Add l1
- For each i, calculate minimum value for x = ai
  - Binary search over intervals in the deque
- Then add li (mi = bi, ci = dpi)
- Note: O(N) can also be achieved by observing that x query is always increasing so we can keep a pointer to the last interval and search from there. We have to maintain the pointer when we pop/insert into the stack

Problem #2: USACO MAR08 acquire

- N <= 50000 rectangles
- “Acquire” all of them.
- Each acquire operation can acquire a subset of rectangles
- Cost of each operation – max_width * max_height
- Minimum cost?

Problem continued

- Given (w1, h1), (w2, h2) ... (wn, hn)
- Cost for an operation: max(w) * max(h) of a subset of rectangles
- How to reduce to convex formulation?
- Hint 1: Two rectangles r1 and r2 when will r2 become irrelelvant (i.e. if we acquire r1 in an operation, we can always acquire r2 in that operation)
- Hint 2: How would you remove all such irrelevant rectangles

Solution

- When r2.h <= r1.h and r2.w <= r1.w then r2 is irrelevant – we can always remove it along with r1
- Sort by h r1.h <= r2.h <= r3.h ... <= rn.h
- Maintain a deque of relevant r
- For ri, remove from back of deque whenever r.w <= ri.w

Solution – part 2

- At end we will have r1...rm s.t. r.h is ascending and r.w is descending
- Each operation is on a contiguous subarray of this.
- Dpi = min cost to acquire first 1..i rectangles
- Dpi = min over $j < i$ (dp[j-1] + ri.h * rj.w)
- Familiar?

D&C - Problem

- Given n objects with weight w1...wn, divide them into groups of m contiguous objects
- Weight of a group = (sum(w))^2
- Minimum cost division
- N <= 1000, m <= 1000
- Sample: n = 5, m = 3
  - W = {1, 1, 2, 3, 1}
  - Sol = ?

D&C – normal dp solution

- Let dp[i][j] = min cost of dividing first i objects into j groups
- dp[i][j] = min over k<i of (dp[k][j-1] + cost(k+1,i))
  - We try to from a group of objects from k + 1 to i. The remaining objects will now have to be grouped into j-1 groups (dp[k][j-1])
- cost(i, j) = (csum[j] – csum[i-1])^2
- Loop over j = 1 to m:
  - Loop over i = 1 to n
    - Loop over k = 1 to i - 1

D&C – optimal k

- dp[i][j] = min over k<i of (dp[k][j-1] + cost(k+1,i))
- Let us store opt[i][j] = the optimal k
- Key observation: opt[i][j] <= opt[i+1][j]
  - Intuition – since cost(k, i) < cost(k, i + 1) if opt[i+1][j] is earlier than opt[i][j] then we could have chosen opt[i+1][j] for dp[i][j]
- Opt[.][j] is monotonically increasing

D&C trick

- Opt[.][j] is monotonically increasing
- For each j let us calculate dp[.][j]
- Compute(i1, i2, kleft, kright) – calculate dp[i1...i2][j] given that kleft <= opt[i1][j] <= opt[i2][j] <= kright
- Initially compute(1, n, 1, n)

D&C trick - solution

- compute(i1, i2, kleft, kright);
  - Handle small i2 - i1
  - Mid = (i1+i2) / 2
  - Calculate dp[mid][j] by normal method but with limits on k
    - Loop from kleft to kright
  - compute(i1, mid – 1, kleft, opt[mid][j])
  - compute(mid + 1, i2, opt[mid][j], kright)

D&C time complexity

- Since dividing range(i1, i2) in middle each time - # elements is halved
- Height of recursion tree is log n
- Each call loops over (kleft to kright)
- The initial range for k (1, n) is split in a single level with each neighbouring ranges sharing 1 endpoint – O(n) per level
- So calculating dp[mid][j] taks O(n) across a level
- There are log n level so total O(n lg n) for given j
- Overall O(mn lgn) down from O(m * n^2)

Proving monotonicity

- opt[i][j] <= opt[i+1][j]

Knuth opti - Problem

- Given a string we need to split it into various parts
- “abcdefghijk” parts = [3,7]
- “abc”, “defg”, “hijk”
- Cost of splitting a string at any point = length of the string
- Minimum cost of splitting given string at given parts
- len(string) <= 10^5, |parts| <= 5000

Normal solution

- dp[i][j] = min cost of splitting substring parts[i]..parts[j]
- dp[i][j] = split at k, i <= k <= j
  - dp[i][j] = min over k (dp[i][k] + dp[k][j] + (parts[j+1] – parts[i])
  - Since splitting (i, j) at k we will have 2 subproblems (i, k) and (k, j). The length of string (i, j) would be parts[j+1] – parts[i]. assume parts[n+1] = len(str)
- O(n^3)

Knuth opti: key observation

- Let opt[i][j] be the optimal split
- opt[i][j-1] <= opt[i][j] <= opt[i+1][j]
- opt[i][j-1] <= opt[i][j]
  - We are removing back part from (i, j) so the optimal k of (i, j-1) will never be to the right of original one. Because otherwise we could have used that optimal one for (i, j) itself
- opt[i][j] <= opt[i+1][j]
  - Same logic as above. We are removing from front. Optimal k of (i + 1, j) will never be right of optimal k for (i, j)

Knuth optimization

- Calculate in increasing order of substring length – calculate for all 1 len substrings dp[i][i+1]
- S = 0: dp[1][1], dp[2][2], dp[3][3]...
- S = 1: dp[1][2], dp[2][3], dp[3][4]...
- S = 2 dp[1][3], dp[2][4], dp[3][5]
  - Note that when calculating dp[2][4], we already have values of opt[2][3] and opt[3][4]
- for(s = 0; s <= n; ++s)
  - for(i = 1; i + s <= n; ++i)
    - J = i + s
    - Handle s <= 2
    - Kleft = opt[i][j-1], kright = opt[i+1][j]
    - Loop over k from kleft <= k <= kright

of operations in inner loop

Calculating (n = 8, s = 4)	kleft	kright
(1, 5)	O14	O25
(2, 6)	O25	O36
(3, 7)	O36	O47
(4, 8)	O47	8

1. So, for each s, the k interval of (1, n) is split across each starting index.
2. So for each s the inner loop only takes O(n) time independent of the second loop for i

Inner loop

- for(s = 0; s <= n; ++s)
  - for(i = 1; i + s <= n; ++i)
    - J = i + s
    - Handle s <= 2
    - Kleft = opt[i][j-1], kright = opt[i+1][j]
    - Loop over k from kleft <= k <= kright
- O(n^2)

References

Quick reference and links for each opti: http://codeforces.com/blog/entry/8219

- Problems for each opti: http://codeforces.com/blog/entry/47932
- 1d1d opti: https://sites.google.com/site/ubcprogrammingteam/news/1d1ddynamicprogrammingoptimization-parti
- Cost opti: http://codeforces.com/blog/entry/49691

Thank you!