Advanced Algorithms for Programming Contests
Lecture 8. Sweeping algorithms

Bakhodir Ashirmatov,
Thilo Stier,
Philip Schär

University of Göttingen
Institute of Computer Science

07.06.2018

Overview

Line sweep algorithms

- General definition

Examples

- Problem: Password
- Problem: Intervals
- Problem: Balance
- Problem: Journey

Line sweep algorithm

- A line sweep algorithm is an algorithm, that uses a coneptual sweep line or window to solve various problems.
- Calculations are only performed at the sweep line.
- In practice we cannot simulate all points in time and so we consider only some discrete points.

Problem: Password

- Given a password b ($1 \leq |b| \leq 10^5$), an algorithm encrypts this password using the following 3 steps (in this given order):
  1. Swap two characters of the given password ≥ 0 times.
  2. Append any number of lower case English letters at the beginning.
  3. Append any number of lower case English letters at the end.

Problem

- Given an encrypted password a and an original password b ($|b| \leq |a|$), check whether a may result from encrypting b by the above algorithm.

Problem: Password

Idea

- Letters may only be added at the end or the beginning.

Naive algorithm

- Iterate over all subdivisions $a = L + M + R$
  - Check if M is a permutation of b
  - If this is the case, return true.
- If no match was found, return false.
- Algorithm works in $|a|^3$ - too slow

Problem: Password

Idea

- We know $|M| = |b|$.
- If we know M, we already know L and R.
- Move a window of size $|b|$ from beginning to end of a.

Algorithm

- Let M be the substring of the first $|b|$ characters of a.
- Iterate over string:
  - Check if M is a permutation of b.
  - Add symbol after end of window to M.
  - Remove first symbol from M.

Problem: Password

Solution

```
int cnta[128], cntb[128], alen;
bool solve(const string &a, const string &b) {
  for (int i = 0; i < b.length(); i++)
    cntb[b[i]]++;
  for (int i = 0; i < b.length(); i++)
    if (++cnta[a[i]] <= cntb[a[i]])
      alen++;
  for (int beg = 0, end = b.length(); end <= a.length(); beg++, end++) {
    if (alen == b.length())
      return true;
    if (end < a.length() && ++cnta[a[end]] <= cntb[a[end]])
      alen++;
    if (cnta[a[beg]]-- <= cntb[a[beg]])
      alen--;
  }
  return false;
}
```

Complexity: $O(|a|)$

Problem: Password

Solution

$a =$	x	x	y	x	z	z	y

$b =$	z	x	y	z

	x	y	z
$cnta =$	0	0	0

	x	y	z
$cntb =$	0	0	0

$alen = 0$

Problem: Password

Solution

a =

x	x	y	x	z	z	y

b =

z	x	y	z

cnta =

x	y	z
3	1	0

cntb =

x	y	z
1	1	2

alen = 2

Problem: Password

Solution

a =

x	x	y	x	z	z	y

b =

z	x	y	z

cnta =

x	y	z
2	1	1

cntb =

x	y	z
1	1	2

alen = 3

Problem: Password

Solution

$a =$

x	x	y	x	z	z	y

$b =$

z	x	y	z

$cnta =$

x	y	z
1	1	2

$cntb =$

x	y	z
1	1	2

$alen = 4$

Problem: Password

Solution

a =

x	x	y	x	z	z	y

b =

z	x	y	z

cnta =

x	y	z
1	1	2

cntb =

x	y	z
1	1	2

alen = 4

Problem: Intervals

- Given N intervals $[l_i, r_i]$ and M points x_i $(1 \leq N, M \leq 10^5, 1 \leq l_i \leq r_i \leq 10^9, 1 \leq x_i \leq 10^9)$
- Check if for every interval $[l_i, r_i]$ you can assign a x_j with $x_j \in [l_i, r_i]$ that is not assigned to any other interval. (Every interval wants to have it's own point only for itself but not all points must be used.)

Problem: Intervals

Naive solution

- Brute force - $O(M!)$ too slow
- Graph matching - $O(N^3)$ too slow

Problem: Intervals

Naive solution

- Brute force - $O(M!)$ too slow
- Graph matching - $O(N^3)$ too slow

Idea

- Sweep over coordinates.
- Keep set of intervals starting to the left of the sweep line that do not yet have their own point.
- When sweep line hits a point, give it to the interval from the set that ends first.
- Check, whether every interval got a point.

Problem: Intervals

Algorithm

- Struct containing: coordinate, type ($0 =$ interval; $1 =$ point), index
- Initialize vector v of those structs.
  - Add $(l_i, 0, i)$ for every interval $[l_i, r_i]$
  - Add $(x_i, 1, i)$ for every point x_i
- Sort v lexicographically by coordinate and type.
- Initialize multiset $s = \emptyset$.
- Iterate over elements $(coord, type, index)$ of v:
  - If $type == 0$ (interval): add r_{index} to s.
  - If $type == 1$ (point): find first $r \in s$ with $r \geq coord$.
    - If it exists, increase number of matched intervals.
    - Remove newly matched interval from s and all smaller ones.
- Return whether number of matched intervals is N.

Problem: Intervals

Solution

```
struct point {
  int coord, type, index;
};
bool operator<(point a, point b) {
  if (a.coord == b.coord)
    return a.type < b.type;
  return a.coord < b.coord;
}
int l[MAXN], r[MAXN]; // intervals
int p[MAXM]; // points
int N, M; // number of intervals / points
```

Problem: Intervals

Solution

```
bool solve() {
  vector<point> v;
  for (int i = 0; i < N; i++)  v.push_back({l[i], 0, i});
  for (int i = 0; i < M; i++)  v.push_back({p[i], 1, i});
  sort(v.begin(), v.end());
  multiset<int> s;
  int cnt = 0;
  for (point pt : v) {
    if (pt.type == 0)
      s.insert(r[pt.index]);
    else while (!s.empty()) {
        int pr = *s.begin();
        s.erase(s.begin());
        if (pr >= pt.coord) {
          cnt++; break;
        }
      }
  }
  return (cnt == N);
}
```

Complexity: $O((N + M) \cdot \log(N + M))$

Problem: Balance

- Given an array a of N integers a_i ($a_i \geq 0, \sum_i a_i = N, 1 \leq N \leq 10^5$)
- You can perform the following operation:
  - Take any c, d with $0 \leq c, d, c + d \leq a_i$.
  - Set $a_i := a_i - (c + d)$ and $a_{i-1} := a_{i-1} + c$ as well as $a_{i+1} := a_{i+1} + d$ (i.e. distribute some quantity from a_i to a_{i-1} and a_{i+1}).
- Find minimum number of steps needed to transform a to an $(1, ..., 1)$.

Problem: Balance

- Given an array a of N integers a_i $(a_i \geq 0, \sum_i a_i = N, 1 \leq N \leq 10^5)$
- You can perform the following operation:
  - Take any c, d with $0 \leq c, d, c + d \leq a_i$.
  - Set $a_i := a_i - (c + d)$ and $a_{i-1} := a_{i-1} + c$ as well as $a_{i+1} := a_{i+1} + d$ (i.e. distribute some quantity from a_i to a_{i-1} and a_{i+1}).
- Find minimum number of steps needed to transform a to an $(1, ..., 1)$.

Idea

- Sweep over array and only look, whether you have to move elements over the sweep line.
- Look, whether you can combine two movements to one.

Problem: Balance

Solution

- Let $balance := 0, prevbalance := 0$ (number of elements that have to be moved from left to right over the sweep line to reach equilibrium).
- Number of operations $cnt = 0$.
- Iterate over a_i (from 1 to n):
  - $balance := balance + a_i - 1$
  - If $balance \neq 0$, we have to move elements
    - If $balance > 0$ and $prevbalance < 0$, we can combine moves.
    - Otherwise $cnt := cnt + 1$.
  - $lastbalance := balance$
- Return cnt.

Problem: Balance

Solution

```
int v[MAXN];
int solve() {
  int prevbalance = 0;
  int balance = 0;
  int cnt = 0;
  for (int i = 0; i < n; i++) {
    balance += v[i] - 1;
    if (balance != 0 && !(balance > 0 && prevbalance < 0))
      cnt++;
    prevbalance = balance;
  }
  return cnt;
}
```

Complexity: $O(N)$

Problem: Journey

- We are in a universe with planets named binary strings of length N.
- We want to fly from planet t to s ($s, t \in \{0, 1\}^N, 1 \leq N \leq 1000$)
- Flights only exist between planets with names differing by one bit.
- There are N universal taxes c_i.
- The cost of flying from planet a to b is the sum of the taxes you have to pay at b, so $\sum_{i=1}^{N} b_i c_i$ (where a and b differ by one bit).
- Find the cheapest route from s to t!

Problem: Journey

Naive solution

- Use Dijkstra to find shortest path
- Number of planets is $O(2^N)$
- N could be as large as 1000 - too slow

Problem: Journey

Naive solution

- Use Dijkstra to find shortest path
- Number of planets is $O(2^N)$
- N could be as large as 1000 - too slow

Idea

- It is always worse to fly on a route that adds a bit twice.
- If we fix, which bits we add / remove on the journey, we can calculate the travel cost greedily.
- If we remove bits, it is best, to remove the most expensive first.

Problem: Journey

Algorithm

- Divide bits in three groups
  - Remove - bits that are in s, but not in t (should be removed)
  - Add - bits that are in t, but not in s (should be added)
  - Common - bits that are both in s and t
- Sort common bits by their cost
- While true
  - Sort remove decreasing
  - Sort add increasing
  - Remove bits in decreasing order, add bits in increasing order
  - Compare with minpath
  - If common bits > 0 erase largest from common bits and add it to remove and add
  - Else break

Problem: Journey

Solution

```
void solve() {
    vector<long long> remove, add, common;
    long long sum = 0;
    for (int i = 0; i < n; i++) {
        if (s[i] == '1')
            sum += cost[i];
        if (s[i] == '1' && t[i] == '0')
            remove.push_back(cost);
        if (s[i] == '0' && t[i] == '1')
            add.push_back(cost);
        if (s[i] == '1' && t[i] == '1')
            common.push_back(cost);
    }
    sort(common.begin(), common.end());
    long long mincost = INF;
```

Problem: Journey

Solution

```
    while (1) {
        sort(remove.begin(), remove.end(), greater<int>());
        sort(add.begin(), add.end());
        long long allcost = 0;
        long long tempsum = sum;
        for (int i = 0; i < remove.size(); i++)
            tempsum -= remove[i], allcost += tempsum;
        for (int i = 0; i < add.size(); i++)
            tempsum += add[i], allcost += tempsum;
        mincost = min(mincost, allcost);
        if (common.size() > 0) {
            remove.push_back(common.back());
            add.push_back(common.back());
            common.pop_back();
        }
        else
            break;
    }
}
```

Complexity: $O(N^2 \log(N))$

Do your homework!