Swapping Operations

znzryb

April 21, 2026

1 暴力的求解方法

呃，我们首先要想的就是，我们怎么暴力呢？串中 ? 的地方，我们直接二进制枚举即可。

但是我们直接拿到了一个全部已知的这个字符串，然后，我们如何求解这个答案呢？

我们想到的第一种方法就是找到最大的这个块 1，然后让别人靠过来。但是可能两个比较小的块 1 之间的距离比较近，然后通过其他地方来的 1，变成了一个大的块 1，然后我们的这个贪心就失效了。

不过这道题目本身都是 $O(n^2)$ 的，我们也没有必要非要想一个 $O(n)$ 的检验方法出来。

我们想到的这个 $O(n^2)$ 的做法是，直接枚举我们最后形成的这个一整个连续块 1 的这个左端点和右端点，那么自然而然，在这个左端点和右端点外的所有 1 都要往这个里面转移。

不过由于这个连续的 1 的长度是确定的，实际上这个做法是可以做到 $O(n)$，因为我们不需要枚举左端点和右端点，只需要枚举左端点即可。当然，我们需要计算这个区间外的 1 的数量，但是这个用前缀和可以轻松的做到。

2 最小化交换次数的串

呃，我们发现啊，这个暴力做法还是给我们了一些启发的。

就是在 1 的数量固定的这个情况下，我们想要最小化这个交换次数，我们首先整一个 $O(n^2)$ 的这个做法，当然，这已经够了。就是，先枚举 1 的数量（下界是串中已经有的这个 1 的数量，上界就是 1 的数量 + 问号的数量）

然后我们去枚举这个左端点，如果里面有 ? 的话，我们就尽最大努力它当成 1 来处理，然后区间外的 ? 的话，我们就尽最大努力它当成 0 来处理，那么我们就可以计算出这个交换次数了。

这个就是一个 $O(n^2)$ 计算这个最小交换次数的这个代码了。

```
void cal_pre() {
  for (int i = 0; i < SZ(s); ++i) {
    preA[i + 1] = preA[i] + (s[i] == '1');
  }
  for (int i = 0; i < SZ(s); ++i) {
    preQ[i + 1] = preQ[i] + (s[i] == '?');
  }
  dbg(preA);
  dbg(preQ);
}

pair<ll, ll> cal_lr_1q(ll l, ll r) {
  ll num1 = preA[r + 1] - preA[l];
  ll numQ = preQ[r + 1] - preQ[l];
  return {num1, numQ};
}

ll cal_mn_step_cnt_1(ll cnt, ll l) {
  auto [num1,numQ] = cal_lr_1q(l, l + cnt - 1);
  // ll rem1 = cnt1 - num1;
  ll needQ = cnt - cnt1;
  ll avaQ = min(needQ, numQ);
  dbg(cnt, l);
  dbg(avaQ);
  dbg(num1);
  cend;
```

```
    return cnt - avaQ - num1;
  }
  Solve() {
    cin >> N;
    cin >> s;
    preA.resize(N + 1);
    preQ.resize(N + 1);
    cnt1 = count(all(s), '1');
    cnt0 = count(all(s), '0');
    cntQ = count(all(s), '?');
    cal_pre();
    ll mn_ans = INF;;
    for (ll cnt = cnt1; cnt <= cnt1 + cntQ; ++cnt) {
      for (int l = 0; l < N - cnt + 1; ++l) {
        mn_ans = min(mn_ans, cal_mn_step_cnt_l(cnt, l));
      }
    }
    cout << mn_ans << " ";
  }
```

3 最大化交换次数的串

然后，我们如何来计算最大交换次数呢？那么应该来说，只要把区间内尽可能多的? 变成 0 即可。这样子尽可能的让外面的 1 交换进来。

```
  ll cal_mx_step_cnt_l(ll cnt, ll l) {
    auto [num1,numQ] = cal_lr_1q(l, l + cnt - 1);
    // ll rem1 = cnt1 - num1;
    ll needQ = cntQ - (cnt - cnt1);
    ll avaQ0 = min(needQ, numQ);
    ll avaQ = numQ - avaQ0;
    if (cnt - avaQ - num1==2) {
      dbg(l, l+cnt-1);
      dbg(avaQ0);
      dbg(avaQ);
      dbg(num1);
      cend;
    }
    return cnt - avaQ - num1;
  }
```

你写了一个大概类似于这样子的代码，但是你会发现，最大化操作，**因为优化方向与选择方向不一致，别人不一定会选择你所选择的区间往里面放东西啊，他可能直接选一个其他区间**。

从这道题目，我们会学到一个东西，所谓我们要”**均匀分布**”，其实意思就是，最小的最大，或者什么最大的最小，那么其实就是二**分**！

那么，实际上，要让所有区间内，最小移动次数最大，就是要**让区间内的 num0 的数量足够多**即可。

```
// 我们要看，这个答案有没有可能实现，就是要看
bool check(ll ans) {
  // 注意，区间长度一定要比 1 大
  for (int all1 = max(1ll, cnt1); all1 <= cnt1 + cntQ; ++all1) {
    ops = s;
    ll s;
    // all1 是我们选择的全部 1 的这个数量
    ll q0 = cntQ - (all1 - cnt1);
    dbg(q0, cntQ, cnt1, all1);
    // 后续的这个区间中的 num1 和 numq 采用这个增量维护。
    ll num0 = 0, numq = 0;
    deque<ll> qDeque;
    for (int i = 0; i < all1 - 1; ++i) {
      if (ops[i] == '?') qDeque.push_back(i), numq++;
```

```
      else if (ops[i] == '0') num0++;
    }
    bool ok = true;
    for (int l = 0; l < N - all1 + 1; ++l) {
      ll r = l + all1 - 1;
      if (ops[r] == '?') {
        numq++;
        qDeque.push_back(r);
      } else if (ops[r] == '0') {
        num0++;
      }
      while (num0 < ans) {
        if (qDeque.empty()) {
          ok = false;
          break;
        }
        // 贪心方法，我们选择的 0，希望以后的这个区间也能吃到
        // 因此选择最右边的 0（也就是双端队列的这个后边）
        ll i = qDeque.back();
        qDeque.pop_back();
        ops[i] = '0';
        num0++;
        q0--;
        if (q0 < 0) {
          break;
        }
        // myAssert(0);
      }
      if (!ok || q0 < 0) {
        ok = false;
        break;
      }
      // myAssert(ans!=2);
      if (ops[l] == '?') {
        numq--;
        myAssert(!qDeque.empty());
        qDeque.pop_front();
      } else if (ops[l] == '0') {
        num0--;
      }
    }
    if (ok) {
      return true;
    }
  }
  return false;
}
```