美好的每一天~不连续的异或

0 pts-10pts?

$O(n^2)$ 枚举两个数 i,j 计算 $\text{popcount}\ (i\ \text{xor}\ j)$ 。

范围开的有点大，想要跑 $5e4$ 可能要一些比较奇怪的技巧。

可以用 avx2 指令集，将每四个 **ull** 合成一个向量，然后将枚举的 a_i 扩张成 4 个，然后通过指令集按位做 xor ，然后统计 **popcount** 即可。

20 pts ~ 60 pts

在二进制下考虑一个数 a_i ，可以认为 a_i 是一个由 $0,1$ 构成的 64 维向量。

不难发现 xor 在每一位上独立，只有 $0\ \text{xor}\ 1$ 才能对答案产生贡献。

可以 $O(n\omega)$ 统计每一位 0/1 的数量，在最后统计答案即可。

随机挑选了 20 分和 60分同学的代码魔改了一下，来让大家看看差别。

20pts

```
for (int i = 0 ; i < 64 ; i++) {
    for (int j = 1 ; j <= n ; j++) {
        s[i] += (a[j] >> i) & 1;
    }
}
```

60pts

```
for (int i = 1; i <= n; i++) {
    uint64 ai = mt_rand();
    for (int j = 0 ; j < 64 ; j++) {
        s[j] += ai & 1 , ai >>= 1;
    }
 }
```

那么差距在哪里呢？实际上是后者对缓存更友好，因为访问的地址都是连续的。当 n 取到 10^7 级别时，这种写法上的不同所引发的常数上的差异会被扩大到肉眼可见的程度。

注意这个时候不要用 **if** 判断是否为 1。同样的也是在常数上的差异，这里不过多赘述，有兴趣大家可以去查阅相关资料。

60 pts

发现每个数都是在 $[0,2^{64})$ 范围内的随机数，那么期望意义下，每一个 a_i 只有 $\frac{\omega}{2}$ 个 1 ，可以考虑每次只统计 **lowbit** 上的信息。

```
for(int i = 1 ; i <= n ; i++) {
    uint64 x = mt_rand();
    while (x) {
        s[__builtin_ctzll(x)]++;
        x &= (x - 1);
    }
}
```

通过 **__builtin_ctzll(x)** 来表示 x 的 lowbit 的位置。

100pts

version 1

发现我们每次只统计 1 位的信息似乎对性能有些浪费，那么我们是否可以多统计几位呢?

我们可以每 8 位当成一个小数据块，然后我们被分成了 $\frac{\omega}{8}$ 个数据。

类似 20pts 的思路， 同样的我们统计每一个数据中每种权值出现的次数。

权值的值域为 $[0, 2^8)$ ， 我们可以枚举两两权值预处理出 xor 后的贡献。

这样就得到了 $O(\frac{n\omega}{8})$ 的实现方法。

```
for (int i = 0 ; i <= S ; ++i) {
    for (int j = i + 1 ; j <= S ; ++j) {
        val[i][j] = __builtin_popcount(i ^ j);
    }
}

for (int i = 0 ; i < n ; ++i) {
    uint64 d = mt_rand() , v;
    v = d & S , cnt0[v]++ , d >>= 8;
    v = d & S , cnt1[v]++ , d >>= 8;
    v = d & S , cnt2[v]++ , d >>= 8;
    v = d & S , cnt3[v]++ , d >>= 8;
    v = d & S , cnt4[v]++ , d >>= 8;
    v = d & S , cnt5[v]++ , d >>= 8;
    v = d & S , cnt6[v]++ , d >>= 8;
    v = d & S , cnt7[v]++ , d >>= 8;
}

for (int i = 0 ; i < S ; ++i) {
    for (int j = i + 1 ; j <= S ; ++j) {
        uint64 x = val[i][j] , r = 0;
        r += cnt0[i] * cnt0[j];
        r += cnt1[i] * cnt1[j];
        r += cnt2[i] * cnt2[j];
        r += cnt3[i] * cnt3[j];
        r += cnt4[i] * cnt4[j];
        r += cnt5[i] * cnt5[j];
        r += cnt6[i] * cnt6[j];
        r += cnt7[i] * cnt7[j];
        ans += r * x;
    }
}
```

version 2

飘神和柴老师的做法应该都是这个版本。回到 20pts 的处理方式，还是统计每一位上有多少 1。

相当于手动模拟一个 64 位并行加法器。

具体可以私聊请教他们 ヾ(·ω·`。)

硬邦邦的冰淇淋

20 pts

最简单的暴力，直接 $O(n \times 2^n)$ 枚举每个音符是否点击

40 pts

注意到如果两个存在影响关系的音符之间有别的音符，那么这些音符也能被这两个音符中的至少一个影响
所以只要考虑相邻两个音符是否收到影响
记 f_i 为第 1 ~ i 个时刻中选择点击第 i 个音符时这部分的最大分数
枚举下一个点击的音符转移，时间复杂度 $O(n^2)$
期望得分：40 ~ 45 pts

extra 20pts

对于一部分数据，所有的 t_i 都相等
这意味着只要满足相邻两个音符的距离不小于 t_i
f_i 可以从 $f_{0,\cdots,i-t_i}$ 转移到
记一个前缀最小值即可优化转移
时间复杂度 $O(n)$
期望得分：20 pts
加上前面部分有 60 pts

100 pts

一般情况就有点不一样了
因为还要考虑到前面的音符可能会影响到后面的音符
f_j 能转移到 f_i 当且仅当 $j \le i - t_i$ 且 $j + t_j \le i$
这是个二维偏序
注意到 i 是递增的，所以我们可以按照 $j + t_j$ 从小到大的顺序加入树状数组维护前缀最大值
时间复杂度 $O(n \log n)$
期望得分：100 pts
如果多一个 log 只有 70 ~ 80 pts
只要常数不是特别特别大都能过

```
int main () {
    int n;
    IO >> n;

    vector<int> t(n) , a(n);
    for (int &ti : t) IO >> ti;
    for (int &ai : a) IO >> ai;
```

```
    vector<int64> f(n + 1);
    vector<vector<int>> add(n + 1);

    vector<int64> s(n + 1);

    auto lowbit = [&](const int &x) -> int {
        return x & -x;
    };

    auto modify = [&](int p , int64 x) -> void {
        for ( ; p <= n ; p += lowbit(p)) {
            s[p] = max(s[p] , x);
        }
    };

    auto query = [&](int p) -> int64 {
        int64 ret = 0;
        for ( ; p ; p -= lowbit(p)) {
            ret = max(ret , s[p]);
        }
        return ret;
    };

    for (int i = 1 ; i <= n ; i++) {
        for (int p : add[i]) {
            modify(p , f[p]);
        }
        if (i - t[i - 1] <= 0) {
            f[i] = 1ll * t[i - 1] * a[i - 1];
        } else {
            f[i] = query(i - t[i - 1]) + 1ll * t[i - 1] * a[i - 1];
        }
        if (i + t[i - 1] <= n) {
            add[i + t[i - 1]].push_back(i);
        }
    }

    cout << *max_element(f.begin() , f.end()) << endl;
    return 0;
}
```