A.

Sáng ngày thứ i: tăng i
Tối ngày thứ i: giảm đi a[(i - 1) % n + 1]

Tìm x nhỏ nhất để sau x ngày, có được giá trị >= k

Gọi F(x) là giá trị đạt được sau x ngày. Tính F(x) trong O(1):
Tổng lượng cộng: 1 + 2 + ... + x = x(x+1)/2
Tổng lượng trừ: Gọi p = x / n (phép chia của C++, làm tròn xuống) và q = x % n (phép modulo). Si = a1 + a2 + ... + ai (S là mảng tiền tố của dãy a). Tổng lượng trừ = p * Sn + Sq

F(x) không có tính chất tăng dần hay giảm dần. Không thể chia nhị phân được.

Xét dãy: F(1), F(1 + n), F(1 + 2n),..., F(1 + kn). Sau khi trải qua n ngày, lượng trừ đạt đúng một vòng (Sn). Như vậy, trong dãy này, lượng trừ tăng dần đều Sn. Lượng cộng càng ngày càng nhiều. Do đó hàm có tính chất lồi (trước giảm, sau tăng). Trên dãy F(1), F(1 + n), F(1 + 2n),... ban đầu lượng cộng có xu hướng nhỏ, tăng chậm. Càng về sau, lượng cộng tăng lên càng nhanh. Trong khi đó, lượng trừ tăng dần đều với tốc độ không đổi.

Dãy F(1), F(1+n), F(1+2n)....: Ban đầu giảm, nhưng giảm càng ngày càng chậm và sau đó sẽ quay đầu tăng (dãy F có đạo hàm bậc 2 F'' > 0)

Để tìm giá trị y nhỏ nhất sao F(1 + y * n) >= k, ta xét hai trường hợp:
Nếu F(1) >= k -> kết luận là y = 0 (hay 1 + y * n = 1)
Ngược lại, dãy ban đầu < k và tới một thời điểm nào đó sẽ >= k, ta dùng chia nhị phân tìm điểm y

Ta dễ dàng chứng minh được với mọi 1 <= i <= n: dãy F(i), F(i+n), F(i+2n),...: ban đầu giảm, sau đó sẽ quay đầu tăng. Vì vậy, với mỗi dãy này, ta có thể tìm điểm j nhỏ nhất để F(i + j*n) >= k theo cách như trên (xét điểm đầu tiên, hoặc chia nhị phân phần còn lại)

B.

Cho một đồ thị gồm n đỉnh trong đó có k đỉnh đặc biệt. Đồ thị vô hướng có trọng số. Ban đầu có m cạnh và có p cạnh khác có thể thêm, Cần xóa các cạnh và thêm vào một số cạnh khác sao cho tổng chi phí nhỏ nhất và: Đồ thị có dạng một rừng các cây sao cho mỗi cây có một đỉnh đặc biệt và mỗi đỉnh đặc biệt chỉ thuộc một cây.

Thuật giải gồm 3 bước:

1. Khởi tạo một disjoint set quản lý đồ thị n đỉnh. Sau đó ta nối tất cả các đỉnh đặc biệt lại với nhau. Sở dĩ phải làm điều này là để khi ta dựng các cạnh, hai đỉnh đặc biệt không bao giờ bị chung vào một thành phần liên thông.

2. Sắp xếp m cạnh đã có theo thứ tự trọng số giảm dần. Làm như thuật toán Kruskal: Nếu một cạnh nối hai đỉnh đang thuộc cùng một TPLT ở disjointset, ta phải xóa cạnh này đi. Ngược lại, giữ nguyên cạnh này.

3. Sắp xếp p cạnh có thể thêm theo thứ tự trọng số tăng dần. Xét từng cạnh, nếu một cạnh nối hai đỉnh thuộc hai thành phần liên thông khác nhau, ta phải thêm cạnh này vào.

Để đảm bảo danh sách cạnh có thứ tự từ điển nhỏ nhất:

+ Ở bước 2: Nếu các cạnh có cùng trọng số, ưu tiên xét các cạnh có chỉ số lớn hơn trước (để những cạnh được xét sau có nguy cơ bị bỏ đi cao là các cạnh có chỉ số nhỏ)
+ Ở bước 3: Với các cạnh có cùng trọng số, ưu tiên xét các cạnh có chỉ số nhỏ trước, chỉ số lớn sau.
+ Sau khi đã tìm ra danh sách các cạnh để thêm và xóa, sắp xếp các dãy theo thứ tự chỉ số tăng dần.

C.

Hai thao tác cập nhật:
D u v: nếu Fu < Fv, gán Fv = min(Fv, Fu + 1). Nếu Fv < Fu, gán Fu = min(Fu, Fv + 1)
P l r: Gọi k = min(Fl, F(l+1),..., F(r-1), Fr); cực tiểu hóa mọi Fl..r với k+1. Nói cách khác, gàn Fl = min(Fl, k+1), F(l+1) = min(F(l+1), k+1)... Fr = min(Fr, k+1)

Nếu xét bài toán tổng quát: Có truy vấn l r v với ý nghĩa gán Fi = min(Fi, v) với mọi l <= i <= r và tính tổng Fi. Bài toán này không giải được bằng Segment Tree lazy update. Tuy nhiên bài này có tính chất đặc biệt: giá trị v trong truy vấn dạng trên luôn thỏa mãn v <= min(Fl...r) + 1.

Ý tưởng của IT lazy update: Để áp dụng thao tác lazy update cho một nút i trên cây IT quản lý đoạn l..r (giả sử ta cần cực tiểu hóa mỗi vị trí đc quản lý bởi nút này cho giá trị v), ta xét hai trường hợp:
+ Nếu v <= giá trị min hiện tại ở nút i: Không khác gì thao tác gán cả nút cho v.
+ Nếu v > giá trị min, từ nhận xét trên suy ra v = min của nút i + 1. Do đó, tất cả các giá trị hiện tại đang bằng min sẽ giữ nguyên, phần còn lại được gán với v

```
sumV = tổng kết quả của các số trong nút
minV = giá trị nhỏ nhất của nút
sumMinC = tổng hệ số ở các vị trí đang có giá trị min
sumC = tổng hệ số ở tất cả các vị trí đang quản lý

void pushDown(int i) { // đẩy thông tin lazy update từ nút i xuống hai nút con
      if (lazy[i] >= n) return;
      for (int j = 2 * i; j <= 2 * i + 1; j++) {
            // cần cực tiểu hóa mọi vị trí ở nút j với giá trị lazy[i],
            // giá trị min >= lazy[i] nên ta cần gán lại mọi vị trí với giá trị lazy[i]
            if (minV[j] >= lazy[i]) {
                  sumV[j] = 1LL * lazy[i] * sumC[j] % MOD;
                  minV[j] = lazy[i];
                  sumMinC[j] = sumC[j]; //mọi vị trí lúc này đều bằng lazy[i] nên đều là min
            } else {
      // lazy[i] > minV[j], mọi phần tử là min giữ nguyên, phần còn lại gán bằng lazy[i]
                  sumV[j] = (1LL * minV[j] * sumMinC[j] +
                              1LL * lazy[i] * (sumC[j] - sumMinC[j] + MOD)) % MOD;
                  // minV[j] và sumMinC[j] không đổi
            }
            lazy[j] = min(lazy[j], lazy[i]);
      }
      lazy[i] = INF;
}

void pushUp(int i) {
      sumV[i] = (sumV[2 * i] + sumV[2 * i + 1]) % MOD;
```

```
	minV[i] = min(minV[2 * i], minV[2 * i + 1]);
	sumMinC[i] = 0;
	for (int j = 2 * i; j <= 2 * i + 1; j++)
		if (minV[j] == minV[i]) sumMinC[i] = (sumMinC[i] + sumMinC[j]) % MOD;
}
```

Khi code IT lazy update, ta tưởng tượng nhóm F-1 là nhóm Fn (do các nhóm kia là F0 F1 ... F(n-1). Tuy nhiên, khi in ra kết quả, ta cần phải tính -1 thay vì n. Để làm được điều này, ta lưu thêm một thông tin nữa vào mỗi nút của cây IT, gọi là sumNegC: Tổng hệ số của những vị trí trong nút đang thuộc nhóm F-1. Khi in ra kết quả ta in giá trị sumV[1] - (n+1) * sumNegC[1]

D.

Gọi F(i, j, k) là độ dài nhỏ nhất của một xâu ký tự sao cho xâu này khớp được i ký tự đầu của xâu A, j ký tự đầu của xâu B và k ký tự đầu của xâu C.

F(0, 0, 0) = 0 (xâu rỗng)

Chuyển trạng thái: thêm một ký tự vào xâu hiện tại. Nhận xét: Ta chỉ thêm một trong hai ký tự tiếp theo của các xâu A và B (ta chỉ thêm vào hoặc A(i+1) hoặc B(j+1))

```
int f[MAX][MAX][MAX];
memset(f, 0x3f, sizeof f);

char a[MAX], b[MAX], c[MAX];
m = strlen(a + 1); // độ dài xâu a
n = strlen(b + 1); // độ dài xâu b
p = strlen(c + 1); // độ dài xâu c

f[0][0][0] = 0;
for (int i = 0; i <= m; i++)
for (int j = 0; j <= n; j++)
for (int k = 0; k <= p; k++) if (f[i][j][k] < INF) {
	// xét ký tự thêm hoặc là a[i+1] hoặc là b[j+1]
	for (int t = 0; t < 2; t++) {
		char tmp = t ? a[i + 1] : b[j + 1];
		// nếu ký tự tmp == 0 có nghĩa là ký tự thêm nằm bên ngoài xâu kí tự,
		// nên bỏ qua
		if (tmp == 0) continue;

		int newI = tmp == a[i + 1] ? i + 1 : i;
		int newJ = tmp == b[j + 1] ? j + 1 : j;
		int newK = tmp == c[k + 1] ? k + 1 : k;

		if (newK < p) // chỉ xét trường hợp chưa khớp hết xâu C
			f[newI][newJ][newK] = min(f[newI][newJ][newK], f[i][j][k] + 1);
	}
}
```

kết quả: min(f[m][n][i]) với 0 <= i < k

E.

Ta duyệt từng đỉnh r, gọi đó là đỉnh xuất phát: Mục tiêu: Với mỗi cung u->v, đếm số đường đi ngắn nhất từ r tới một đỉnh nào đó mà có đi qua cung u->v

Dijkstra từ r. Tính độ dài đường đi ngắn nhất từ r tới mọi đỉnh khác. Giữ lại tất cả các cung u->v thỏa mãn: dv = du + c. Các cạnh giữ lại tạo thành DAG.

Nhận xét: Nếu một cung không được giữ lại, không tồn tại bất kì đường đi ngắn nhất nào từ r có đi qua cung đó. Với cung u -> v được giữ lại: Số đường đi ngắn nhất đi qua cung xuất phát từ r = số đường đi trên DAG chứa cung u->v đó. Số cách đi = (số đường đi từ r đến u) * (số đường đi từ v tới một đỉnh bất kỳ)

Gọi F(u) là số đường đi từ r đến u. Xét mọi đỉnh w có cung w -> u được giữ lại: F(u) = tổng F(w)
Gọi G(v) là số đường đi từ v tới một đỉnh bất kỳ. Giả sử từ v có các cung đi ra các đỉnh w1, w2,..., wk: G(v) = 1 + G(w1) + G(w2) + ... + G(wk)

Với mỗi đỉnh r được chọn làm đỉnh xuất phát, ta cần Dijkstra và QHĐ trên DAG -> Độ phức tạp cho mỗi r là O((m + n)log n). Tổng độ phức tạp chung là O(n * m * logn)

Subtask 3:
Duyệt từng cặp đỉnh (s, t) và một cung u -> v. Đếm số đường đi ngắn nhất từ s đến t có đi qua cung u -> v
Nếu d(s, t) = d(s, u) + c(u, v) + d(v, t), số đường đi ngắn nhất là cnt(s, u) * cnt(v, t). Trong đó cnt(s, u) = số đường đi ngắn nhất từ s đến u. d(s, t) = độ dài đường đi ngắn nhất từ s đến t.
Nếu d(s, t) > d(s, u) + c(u, v) + d(v, t) -> không tồn tại đường đi ngắn nhất từ s đến t đi qua cung u -> v

F.

Phân tích các số ra thừa số nguyên tố: Các ràng buộc GCD cho ta cận dưới của số mũ, các ràng buộc LCM cho ta cận trên của số mũ:

GCD(a1, a2) = 20 = 2^2 * 5
+ Xét số nguyên tố 2: số mũ của a1 và a2 >= 2
+ Xét số nguyên tố 5: số mũ của a1 và a2 >= 1

LCM(a3, a4) = 72 = 2^3 * 3^2
+ Xét số nguyên tố 2: số mũ của a3 và a4 <= 3
+ Xét số nguyên tố 3: số mũ của a3 và a4 <= 2
+ Xét mọi thừa số nguyên tố khác: số mũ của a3 và a4 = 0.

Tiền xử lý: giả sử số ai có các ràng buộc LCM với các gía trị 42 = 2 * 3 * 7, 56 = 2^3 * 7, 63 = 3^2 * 7 -> tìm ước nguyên tố chung của tất cả các số này (7 là ước nguyên tố chung duy nhất) -> số ai chỉ có thể có ước nguyên tố là 7. Nói cách khác, giả sử số ai có ràng buộc LCM với các giá trị x1, x2, ..., xk. Tìm GCD(x1, x2, ..., xk), ta có GCD này phải chia hết cho ai.

Giả sử đồng thời số ai này cũng có ràng buộc GCD với các giá trị y1, y2, ..., yp. Nếu tồn tại một số y nào đó mà y không phải là ước GCD(x1, x2, ..., xk) kể trên, bài toán không có lời giải.

Sau bước tiền xử lý này, ta không còn cần quan tâm đến việc ràng buộc LCM có can thiệp với tất cả các thừa số nguyên khác.

Khi xét một thừa số nguyên tố p nào đó, số ai có các ràng buộc về cận trên và cận dưới của số mũ của p trong phân tích ra thừa số nguyên tố của ai. Nhận xét tham lam: ta luôn chọn số mũ hoặc là cận trên hoặc là cận dưới, chứ không chọn một số lơ lửng ở giữa.

Subtask cuối: sử dụng 2-SAT: