Comentario 1

No llamar a una función (con coste no constante) más de una vez con los mismos valores en sus parámetros.

- Razón: el resultado de todas las llamadas será el mismo y pagaremos su coste temporal en cada una de esas llamadas
- Solución: haremos una única llamada (pagaremos una única vez su coste temporal), almacenaremos el valor de retorno en una variable y la consultaremos tantas veces como sea necesario (coste constante)

Ejemplo sencillo:

Incorrecto:

```
// coste diferente de O(1)
int mi_funcion(int x) {...}

int main() {
    int x = 5;
    cout << mi_funcion(x) << endl;
    cout << mi_funcion(x) << endl;
    cout << mi_funcion(x) << endl;
}
```

Correcto:

```
// coste diferente de O(1)
int mi_funcion(int x) {...}

int main() {
    int x = 5;
    int r = mi_funcion(x);
    cout << r << endl;
    cout << r << endl;
    cout << r << endl;
}
```

Ejemplo en el código del ejercicio de servidors:

```
using Servers = map<string, priority_queue<int>>;

void transfer(Servers& s) {
    string from, to;
    cin >> from >> to;
    if (s.count(from) and s.count(to)) {
        if (not s[from].empty()) {    // s[from]: O(log(n))
            ... s[from].top();        // s[from]: O(log(n))
            ... s[from].pop();        // s[from]: O(log(n))
            ...
        }
    }
    ...
}
```

Como entre una llamada y la siguiente al operador [ ] no cambia el valor de from (y, por tanto, estaremos accediendo siempre a la misma cola de prioridad), haremos lo siguiente:

```
using Servers = map<string, priority_queue<int>>;

void transfer(Servers& s) {
    string from, to;
    cin >> from >> to;
    if (s.count(from) and s.count(to)) {
        priority_queue<int>& p = s[from];  // cuidado: es una referencia!
        if (not p.empty()) {
            ... p.top();
            ... p.pop();
            ...
        }
    }
    ...
}
```

Fíjate que p es una referencia! Eso quiere decir que p y $s[from]$ son la misma cola de prioridad. Si fuera:

```
        priority_queue<int> p = s[from];
```

la asignación haría copia del valor devuelto por $s[from]$. Eso quiere decir que p y $s[from]$ son iguales (en ese momento) pero no la misma cola de prioridad. Por tanto, las modificaciones sobre p no afectarían a la cola de prioridad en $s[from]$.

Siguiendo con este razonamiento, fíjate que $s.count(from)$ también tiene coste $O(log(n))$ y que, si existe $from$ en el diccionario, haremos $s[from]$ para acceder a su valor asociado. En estos casos, es mejor pensar en la instrucción $find$ que con un único coste de $O(log(n))$ nos permite saber si $from$ existe y, en caso positivo, retener el acceso a su cola de prioridad.

Comentario 2

Reestructurar los condicionales que tengan la estructura del código de la izquierda a la estructura del código de la derecha:

```
if (not Expr) {
    A;
    B;
    C;
} else {
    D;
    E;
}
```

```
if (Expr) {
    D;
    E;
} else {
    A;
    B;
    C;
}
```

Ejemplo en servidores:

```
if (not p.empty()) cout << ...;
else cout << "-" << endl;
```

```
if (p.empty()) cout << "-" << endl;
else cout << ...;
```

Comentario 3

Cuidado con las declaraciones fuera de ámbito. Una variable está declarada fuera de ámbito si se declara en un bloque más externo de lo que debería (o dicho de otro modo, si se puede declarar en un bloque más interno).

Ejemplo: en el código de la izquierda la variable id está declarada fuera de ámbito. El código de la derecha muestra el bloque en el que se debería declarar.

```
string id;        // fuera de ámbito
for (int i = 0; i < n; ++i) {
    cin >> id;
    servers[id];
}
```

```
for (int i = 0; i < n; ++i) {
    string id;
    cin >> id;
    servers[id];
}
```