A. Formatação de Campos I

O problema pode ser resolvido por meio do uso das funções `toupper()` e `tolower()` da biblioteca `cctype` do C++: basta invocar a primeira função no primeiro caractere e a segunda em todos os demais.

Como ambas funções tem complexidade $O(1)$, a solução tem complexidade $O(N)$, onde N é o tamanho da string dada na entrada.

B. Onde Está Wally?

É possível resolver o problema percorrendo a matriz dada na entrada, linha a linha, coluna a coluna, até encontrar o caractere '`W`'.

Esta solução tem complexidade $O(NM)$.

C. Consonantelândia

Basta imprimir as vogais seguindo a ordem descrita. Veja que é mais simples imprimir cada caractere da original, seguido da vogal apropriada, do que construir toda a string antes da impressão.

A complexidade é $O(NS)$, onde S é o tamanho máximo entre todas as linhas da entrada.

D. Quebra-cabeças

Para formar um palíndromo, a primeira letra deve ser igual a última, a segunda igual a penúltima, e assim por diante. Desta forma, basta que cada letra apareça um número par de vezes.

A única exceção é o caso onde o tamanho da string S é ímpar: neste caso, uma única dentre as diferentes letras da palavra tem que aparecer uma quantidade ímpar de vezes. Esta letra ocupará a posição central do palíndromo, como acontece, por exemplo, no palíndromo `SAIAS`.

A solução tem complexidade $O(N)$ ou $O(N \log N)$, dependendo de como o histograma de frequência das letras foi implementado.

E. Identificadores

Para cada string s dada na entrada, o problema consiste em aplicar os seguintes testes em s:

1. o primeiro caractere de s não é letra nem *underscore*?
2. algum dentre os demais caracteres não é letra, nem número, nem *underscore*?
3. s é uma das palavras reservadas?

Se qualquer uma destas perguntas tiver resposta afirmativa, s não é um identificador válido e a resposta deve ser "`Nao`"; caso contrário, a resposta deve ser "`Sim`". As funções `isdigit()`, `isalpha()` e `isalnum()` da biblioteca `cctype` de C++ são úteis na implementação destes testes.

Esta solução tem complexidade $O(TS)$, onde S é o tamanho da maior dentre as strings dadas na entrada.

F. Masterchef

Cada reinterpretação da receita é um anagrama distinto da string original. Como $N \leq 10$, é possível gerar todos estes anagramas por meio da função `next_permutation()` da biblioteca `algorithm` da linguagem C++, desde que a string S tenha sido ordenada. É preciso lembrar de subtrair a permutação correspondente à receita original. Esta solução tem complexidade $O(N \times N!)$.

É possível, porém, resolver este problema com complexidade $O(N)$. Seja h o histograma dos caracteres de S, o qual pode ser criado em $O(N)$, e considere o conjunto $\{c_1, c_2, \ldots, c_K\}$ dos caracteres distintos de S. O número de anagramas distintos A de S é dado por

$$A = \frac{N!}{h(c_1)!h(c_2)!\ldots h(c_K)} - 1$$

G. Runs

Uma *run* ascendente pode ser identificada por meio de dois ponteiros L e R: em cada possível possível posição inicial i de uma *run*, faça $L \leftarrow i, R \leftarrow i + 1$ e, enquanto $R \leq N$ e $s[R + 1]$ suceder $s[R]$, incremente R. A substring $s[L, R - 1]$ será uma *run* ascendente e R se tornará a próxima posição inicial.

Este algoritmo permite identificar a maior *run* ascendente. Para identificar a maior *run* descendente, basta aplicar o mesmo algoritmo na string reversa de s. A resposta será o maior dentre este dois valores.

Esta solução tem complexidade $O(TS)$, onde S é o maior tamanho dentre todas as strings listadas.

H. Diversidade Cultural

Sejam L_k e R_k os histogramas de $S[1..k]$ e de $S[(k+1)..N]$, respectivamente. A diferença d será dada por

$$d = ||L_k| - |R_k||$$

onde $|L_k|$ é o número de caracteres distintos em L_k. Se estes histogramas forem construídos com uma estrutura `map` da linguagem C++, estes valores seriam obtidos através do método `size()`.

Computar estes histogramas para cada valor $k = 1, 2, \ldots, N-1$ levaria a uma solução $O(N^2)$, o que resultaria num veredito TLE. Contudo, observe que L_0 é vazio e R_N é o histograma de toda string S, e também que

$$L_k = L_{k-1} \cup \{\ s[k]\ \}$$

e

$$R_k = R_{k-1} \setminus \{\ s[k]\ \}$$

onde $A \setminus B$ é a diferença entre os conjuntos A e B.

Assim, é possível construir os histogramas para o sucessor de k em $O(1)$, levando a uma solução $O(N)$ ou $O(N \log N)$, a depender de como o histograma foi implementado.

I. TEP, TEP, TEP, ...

Checar cada um dos trios (i, j, k) possíveis gera uma solução $O(N^3)$, cujo veredito é TLE.

Seja $p[i]$ o número de caracteres 'P' que aparecem no sufixo $s[i..N]$. Temos que $p[N+1] = 0$ e $p[i] = p[i+1] + \delta_i(\texttt{P})$, onde $\delta_i(c) = 1$ se $s[i] = c$, e zero, caso contrário.

Em seguida, é possível computar $ep[i]$, o número de pares que foram a string "EP" no sufixo $s[i..N]$. Novamente $ep[N+1] = 0$ e $ep[i] = ep[i+1] + \delta_i(\texttt{E})p[i+1]$.

Por fim, seja $tep[i]$ o número de ocorrências de "TEP" no sufixo $s[i..N]$. Temos $tep[N+1] = 0$ e $tep[i] = tep[i+1] + \delta_i(\texttt{T})ep[i+1]$.

A resposta do problema é $tep[0]$, e esta solução tem complexidade $O(N)$.

J. AC ou WA?

É possível identificar o i-ésimo caractere de Fibonacci (ou das strings S_j de Tarcísio) sem construir tais strings explicitamente. Para as strings de Fibonacci o i-ésimo caractere é dado por:

```
char nth_char_F(int n, int i)
{
    if (i == 1)
        return 'B';

    if (i == 2)
        return 'A';

    auto a = Fs[i - 1];

    return n <= a ? nth_char_F(n, i - 1) : nth_char_F(n - a, i - 2);
}
```

A rotina para as strings S_j é praticamente idêntica, mudando apenas a ordem de concatenação:

```
char nth_char_S(int n, int i)
{
    if (i == 1)
        return 'B';

    if (i == 2)
        return 'A';

    auto a = Fs[i - 2];

    return n <= a ? nth_char_S(n, i - 2) : nth_char_S(n - a, i - 1);
}
```

Em ambas rotinas, `Fs` é um vetor com os número de Fibonacci pré-computados, que são utilizados para reduzir o problema de computar o i-ésimo caractere de F_k para o problema de computar o i-ésimo caractere de F_{k-1} (se $i \leq$ a) ou o $(i - a)$-ésimo caractere de F_{k-2}, caso contrário, onde a é o $(k - 1)$-ésimo número de Fibonacci. Vale a mesma interpretação para as strings S_j.

Para o problema, basta pré-computar até o 88º número de Fibonacci, que é o primeiro que excede o valor 10^{18}. A primeira chamada de ambas rotinas tem que começar com parâmetro i igual a k, o qual pode ser determinado através da função `lower_bound()` da STL.