1. A seguir estão comentadas ambas as funções comparando suas implementações.

int fibonacci\_01::fib(int n) {  
 /\*  
 \* Aqui utiliza-se função recursiva, em que são gerados "espaços" dentro de "espaços" análogo a  
 \* uma busca em profundidade. Quando a codição n<=1 é satisfeita, a função chamada dentro do último  
 \* "espaço" retorna um valor e assim as demais chamadas retornam valores sucessivamente obedecendo  
 \* a soma fib(n-1) + fib(n-2). o Valor final retornado é o n-ésimo termo da sequência de fibonacci.  
 \*/  
 if (n<=1){  
 return n;  
 }  
 return fib(n-1) + fib(n-2);  
}  
int fibonacci\_02::fib(int n) {  
 //int f[] = new int[n+2]; declaração original incorreta  
 /\*  
 \* Nesse caso, primero são gerados todos os valores da sequência de fibonacci até o n-ésimo item,  
 \* retornando-o no final  
 \*/  
 int \*f = new int(n+2);  
 int i;  
 f[0] = 0;// o termo zero é atribuído  
 f[1] = 1;// o primeiro termo é atribuído  
 for (i=2;i<=n;i++){  
 f[i] = f[i-1] + f[i-2];// com os primeiros termos inicializados, os demais são gerados até o n-ésimo  
 }  
 return f[n]; //retorna o n-ésimo termo

1. Função que retorna string sem caracteres duplicados e estimativa da complexidade assintótica

#include <map>

#include <algorithm>

#include <iostream>

#include <string>

std::string remove\_multiple\_chars(const std::string& in){  
 std::string out = in;  
 const char \*p\_s = out.c\_str();  
 std::map<char,int> m;  
 int i=0;  
 //N  
 while (i<out.length()) { //1  
 if (m.find(p\_s[i])!=m.end()){ // (log(i)), i<K, K é a quantindade de caracteres não repetidos da string   
 out.erase(i,1);// 1  
 }  
 else{  
 m.insert(std::pair<char, int>(p\_s[i],1)); // 1  
 i++;  
 }  
 }  
 return out;  
}  
/\*  
 \* Para determinação da complexidade assintótica, foi considerado  
 \* a complexidade assintótica do método find da classe std::map,  
 \* uma vez que esse recurso foi utilizado.  
 \* A função foi feita de tal forma que uma única varredura pela  
 \* string fosse o suficiente para encontrar todos os caracteres duplicados.  
 \* Para isso foi utilizado a estrutura std::map para armazenar novos  
 \* caracteres encontrados e para verificar a ocorrência de repetições.  
 \* Caso algum caractere já tiver sido armazenado na estrutura std::map,   
 \* onde isso é verificado pela função find da classe std::map, esse é  
 \* removido da string analisada.  
 \* Ao chegar no último caractere da string, ter-se-á obtido uma string  
 \* sem caracteres repetidos.  
 \* Assim temos:  
 \* N -> número de caracteres totais da string  
 \* K -> número de caracteres não repetidos da string  
 \* W -> número de execuções por iteração  
 \* onde K<=W  
 \* Portanto a complexidade assintótica pode ser estimada da seguinte forma:  
 \*   
 \* W  
 \* +  
 \* W + log(1) // existe um caractere na estrutura que se busca se o caractere analisado já foi ou não encontrado  
 \* +  
 \* W + log(2) // existem dois caracteres na estrutura que se busca se o caractere analisado já foi ou não encontrado  
 \* +  
 \* .  
 \* .  
 \* .  
 \* +  
 \* W + log(K) // existem K caracteres na estrutura que se busca se o caractere analisado já foi ou não encontrado  
 \* +  
 \* W + log(K)  
 \* .  
 \* .  
 \* .  
 \* +  
 \* W + log(K)  
 \*   
 \* Assim podemos dizer que a complexidade assintótica sera de O(N\*log(N)).  
 \* \*/