te fazer projetar o algoritmo errado. Leia o problema diversas se for preciso mas entenda o problema.

Depois de dominar o problema, pense em algoritmos para resolvê-lo. Rabiscar cadernos, utilizar quadros e discutir com outras pessoas faz parte do processo. O objetivo é que, ao final deste processo, você tenha uma ideia de boas soluções para o problema. Nesta etapa é essencial entender os custos dos algoritmos em termos assintóticos de tempo e espaço para poder escolher a solução que melhor te atenda.

Por exemplo, suponha que um problema é encontrar a n-ésima permutação de um conjunto de elementos. Pensando em formas de resolvê-lo, você encontra duas formas: uma em tempo linear e outra em tempo exponencial em relação ao tamanho do conjunto. Qual escolher? A resposta é depende. Se o problema define um número máximo de elementos no conjunto que é baixo (suponhamos 5, que dá 5! permutações) e o algoritmo exponencial é mais rápido de implementar, a escolha do mesmo pode ser uma boa ideia. Entretanto, suponha que o número máximo de elementos no conjunto é alto (suponhamos 1.000, que dá 1.000! permutações). Mesmo que um algoritmo exponencial seja mais rápido de implementar, ele não vai rodar no pior caso. Logo, é importante estar ciente de que o algoritmo linear é a única solução viável dentro das que você enumerou.

Depois de pensar na solução, você tem que implementá-la em alguma linguagem de programação. Nesta etapa, é importante conhecer os tipos básicos da linguagem e as funções da biblioteca padrão (SL). Usá-las é bom pois economiza tempo e te oferece um código de qualidade e que foi bem testado. Entretanto, às vezes, é preciso entender o funcionamento das funções da SL. Por exemplo, você pode considerar que a função de comparar duas strings tem custo constante mas, na verdade, ela é linear no tamanho da menor string. Dependendo do número de operações de comparação de string que você fizer, uma parte considerável do tempo do algoritmo pode estar na SL que você está utilizando. Fique atento nas funções básicas que você utilizar e avalie o impacto que elas têm no algoritmo que você projetou.

Por fim, teste seu código e direito! Muitos problemas vem com restrições. Teste seu algoritmo no limite dessas restrições. Por exemplo, veja o que acontece com seu programa quando a entrada já inicia com um sentinela ou está vazia (não tem casos de teste). Se o exercício for uma permutação e o número máximo de elementos é 1.000, teste seu algoritmo com 1.000 elementos. Não seja preguiçoso e teste. Caso encontre um erro, revise o código. Se você tem certeza que o programa está correto, consulte outras pessoas. É comum outros verem erros que você não vê no seu próprio código.

5 Modularização

Nos "Trabalhos práticos", é essencial que você modularize seu código. Modularizar significa dividir seu código em funções bem definidas. Um exemplo trivial é um trabalho que envolva a soma de dois números, A e B. Inicialmente, você pode fazer da seguinte forma:

```
1 ...
2 printf("%d",A+B);
3 ...
```

Entretanto, você poderia modularizar a soma como uma função. Por exemplo, você poderia criar uma função "Soma_Numeros(int A,int B)", conforme abaixo:

```c
int Soma_Numeros(int A, int B){
  return A+B;
}
```

Dada esta função (Soma_Numeros), você poderia chamá-la diversas vezes, sempre quando necessário. Por exemplo, no código principal:

```c
...
printf("%d",Soma_Numeros(A,B));
...
```

Um terceiro ponto é você separar o código em múltiplos arquivos. Por exemplo, você poderia querer construir uma mini biblioteca de operações matemáticas básicas com soma, multiplicação, divisão e subtração de inteiros. Para isso, você deve criar um arquivo de "header" (cabeçalho) para definir as estruturas presentes na sua mini biblioteca. Para a mini biblioteca de operações matemáticas, podemos criar um arquivo "operacoes.h", conforme abaixo:

```c
#ifndef OPERACAO
#define OPERACAO
int Soma_Numeros(int A,int B);
int Subtrai_Numeros(int A,int B);
int Multiplica_Numeros(int A,int B);
int Divide_Numeros(int A,int B);
#endif
```

Observe que definimos apenas os cabeçalhos das funções neste arquivo. Isso porque ele funciona apenas como uma lista do que existe na mini biblioteca. Além do cabeçalho das funções, é comum colocar no arquivo ".h" as estruturas abstratas de dados, como "struct" e "typedef" (Abordado mais abaixo). No código, existem as flags "#ifndef OPERACAO", "#define OPERACAO" e "#endif". Essas flags são úteis para evitar a inclusão do cabeçalho mais de uma vez pelo compilador, podendo gerar erros. Dessa forma, a primeira flag verifica se o nome definido existe (em nosso caso, "OPERACAO"). Se não existir, ele define este nome através da segunda flag. A última flag serve para definir o escopo de atuação do "#ifndef", ou seja, tudo que estiver entre "#ifndef" e "#endif" será afetado.

Dado o arquivo de cabeçalho, você deve implementar as funções para ele. Neste caso, crie um arquivo com o mesmo nome do header, porém com extensão ".c". No nosso caso, esse arquivo chamará "operacoes.c", que é definido abaixo:

```c
#include "operacoes.h"
int Soma_Numeros(int A, int B){
  return A+B;
}
int Subtrai_Numeros(int A, int B){
  return A-B;
```

```
}
int Multiplica_Numeros(int A, int B){
  return A*B;
}
int Divide_Numeros(int A, int B){
  return A/B;
}
```

Tudo certo. Sua mini biblioteca está pronta. Para utilizá-la em outros programas, basta incluir o arquivo do cabeçalho ".h", conforme abaixo:

```
#include "operacoes.h"
int main(){
  int A=2,B=2;
  printf("%d",Soma_Numeros(A,B));
  ...
}
```

Você já sabe agora o básico para modularizar seus programas. O exemplo acima é didático mas, ao longo dos TPs, você sentirá necessidade de modularizar seu código. Por exemplo, um trabalho pode exigir que você use uma "Lista Encadeada". Para isso, você poderia criar uma "lista.h", que definirá as estruturas e as funções de operações sobre as listas e uma "lista.c", onde ficarão as implementações das operações.

6 Fazendo um Makefile

A primeira pergunta é: O que é um Makefile? É um arquivo que define as diretrizes de compilação do seu código. Muita das vezes, quando o código, em "C" está em um arquivo só, basta executar o comando abaixo:

```
gcc <flags> <arquivo.c> -o <binario>
```

O problema surge quando existem vários arquivos a serem mesclados para gerar o código final. Isso ocorre quando o código está dividido em diversos arquivos para uma melhor modularização (Direcionamento 4). Para isso, existe um programa "Make" que, quando invocado (*make*), chama um arquivo chamado "Makefile" dentro do diretório corrente. O arquivo "Makefile" contem as instruções de compilação do programa. A sintaxe básica de um arquivo "Makefile" é:

```
<nome da regra>: <dependencias>
<TAB> <comando>
```

Cada regra é lida e as dependências são arquivos ou regras. No exemplo da mini biblioteca no Direcionamento 4, com os arquivos "main.c", "operacoes.h" e "operacoes.c", poderíamos fazer criar o seguinte arquivo, com nome "Makefile", dentro do diretório onde estão os fontes do programa:

```
operacoes: operacoes.o main.o
<TAB> gcc operacoes.o main.o -o operacoes
main.o: main.c operacoes.h
```