# Soma acumulada

**Soma acumulada** (ou *prefix sum*) é uma técnica de programação muito utilizada na maratona. Para entender seus benefícios, precisamos primeiro entender o problema ao qual ela está relacionada.

## Soma de intervalo

Suponha que você possui um array $arr$ de $n$ elementos e quer achar a soma de todos os elementos no intervalo $[l,\ r]$ (isto é, $arr[l] + arr[l+1] + \cdots + arr[r-1] + arr[r]$).

Qual a maneira de soluciona esse problema?

## Solução ingênua

A solução mais imediata que você poder ter pensado é fazer exatamente a soma que se encontra ali em cima (utilizando um ciclo `while` ou `for` ).

Vamos mostrar rapidamente sua implementação:

```
const int N = 10e5; // 10.000
int arr[N];

int prefixSum(int l, int r) {
    int sum = 0;
    for (; l <=r; l++) sum += arr[l];
    return sum;
}
```

Estamos sempre considerando que os valores $l$ e $r$ são válidos.

Essa implementação parece muito boa, certo? Não precisamos nem declarar uma variável $i$ como geralmente fazemos dentro de um `for` , basta usar o próprio $l$ como iterador.

Antes de partimos para a solução esperta, vamos ver quantos elementos precisamos acessar para obter a soma acumulada?

Não é difícil de ver que são r-l+1 valores acessados, pois esse é o número de iterações do `for` (se não entendeu o `+1` , conte quantos inteiros existem no intervalo $[1,\ 5]$, por exemplo, e verá que a conta está certa).

Logo, no pior caso (o caso em que essa conta dá o maior número possível), temos que $l = 0$ e $r = n - 1$. Ou seja, estamos somando todo o array! Pelos nossos calculos, faremos $(n - 1) - 0 + 1 = n$ iterações.

# Solução esperta (soma acumulada)

Um modo mais rápido de resolver o mesmo problema é fazendo um **pré-processamento**.

Pré-processamentos são muito comuns quando queremos deixar algoritmos mais rápidos, vocês os verão o tempo todo e, por isso, entender bem Soma Acumulada pode ser algo que te ajude muito no entendimento de algoritmos futuros.

## Pré-processamento

Quando formos ler o input (o array), vamos criar um segundo array que chamarei de $ps$ (do inglês, *prefix sum*), contendo também $n$ elementos. Queremos que esse array possua a seguinte propriedade:

- $ps[i] = arr[i] + arr[i-1] + \cdots + arr[0]$;

(talvez agora você entenda o nome soma acumulada)

Criar esse array é bem fácil, pois rapidamente percebemos um padrão recursivo (uma recorrência). A propriedade acima pode ser escrita como:

1. $ps[0] = arr[0]$;
2. $ps[i] = arr[i] + ps[i-1]; \quad \forall i > 0$

Se quiser treinar a matemática, prove isso usando indução 😉

Vamos agora mostrar como fazer isso no código:

```
const int N = 10e5; // 10.000
int arr[N];         // O meu array original
int ps[N];          // O array auxiliar
int n;              // O número de elementos do array

void leituraDoInput() {
    cin >> n;

    // Aqui vou supor que o array é não nulo, mas
    // você pode adaptar para o seu problema
    cin >> arr[0];
    ps[0] = arr[0]; // *

    // i começa de 1, pois já lemos arr[0]
    for (int i = 1; i < n; i++) {
        cin >> arr[i];
        ps[i] = arr[i] + ps[i-1]; // *
        // Exatamente a fórmula descrita acima
    }
}
```

Como já teríamos que fazer esse trabalho todo para ler o input, praticamente não estamos fazendo trabalho extra, certo?
A diferença é que, ao invés de apenas ler o $arr$, estamos fazendo um trabalho a mais em cada leitura (o trabalho das linhas com asterisco).

## Utilizando o pré-processamento

Tá, mas pra que eu fiz todo esse trabalho extra?

Agora é que vem a beleza dessa técnica. Olhemos para a propriedade de $ps$ novamente:

- $ps[i] = arr[i] + arr[i-1] + \cdots + arr[0]$;

E olhemos para o problema que estamos tentando solucionar de forma rápida:

Encontrar $arr[l] + arr[l+1] + \cdots + arr[r-1] + arr[r]$.
Ou, em outra ordem: $arr[r] + arr[r-1] + \cdots + arr[l+1] + arr[l]$.

Vamos pensar em qual é o valor de $ps[r]$?

- $ps[r] = arr[r] + arr[r-1] + \cdots + arr[l] + arr[l-1] + \cdots + arr[0]$;

Se tivéssemos uma maneira de cortar toda a parte da esquerda de $arr[l]$, então teríamos a nossa resposta rapidamente. Mas nós temos! Observe quem é $ps[l-1]$:

- $ps[l-1] = arr[l-1] + arr[l-2] + \cdots + arr[0]$;

Opa! Então fazer $ps[r] - ps[l-1]$ nos dá a resposta para nosso problema instantaneamente, sem ter de fazer mais nada! Que bacana!

- $ps[r] - ps[l-1] = arr[r] + arr[r-1] + \cdots + arr[l+1] + arr[l]$;

Vamos mostrar a solução na prática:

```
const int N = 10e5; // 10.000
int arr[N];         // O meu array original
int ps[N];          // O array auxiliar
int n;              // O número de elementos do array
int l, r;           // Os parâmetros do problema

int main() {
    cin >> n;

    // Aqui vou supor que o array é não nulo, mas
    // você pode adaptar para o seu problema
    cin >> arr[0];
    ps[0] = arr[0];

    // i começa de 1, pois já lemos arr[0]
    for (int i = 1; i < n; i++) {
        cin >> arr[i];
        ps[i] = arr[i] + ps[i-1];
    }

    cin >> l >> r;
    cout << ps[r] - ps[l-1] << endl;
    return;
}
```

Olha que legal, achamos a resposta em uma linha. Nada de `for's` ou mais nada.

Dizemos que esse tipo de algoritmo é um **algoritmo constante**, pois sempre executará no mesmo tempo, independentemente de $n$, $l$ ou $r$.

## Observação1

Você pode ter percebido que esse algoritmo não funciona no caso de l ser exatamente 0, pois fazemos $ps[0-1]$, que é uma posição inválida. Existem duas formas de contornar isso, uma mais eficiente (mas que deixa as coisas um pouco mais difíceis de entender) e outra mais simples e menos eficiente.

A mais simples é fazer

```
if (l == 0) cout << ps[r] << endl
```

e, caso contrário, fazer o que já estávamos fazendo antes.

O outro jeito é deslocar o array $ps$ em uma posição para a direita. Ou seja, ao invés de utilizar $ps[r] - ps[l-1]$, utilizar $ps[r+1] - ps[l]$. Basta ler de forma diferente:

```
cin >> arr[0];
ps[0] = 0;
ps[1] = arr[0];

for (int i = 1; i < n; i++) {
    cin >> arr[i];
    ps[i+1] = arr[i] + ps[i];
}
```

## Observação2

Suponha que no problema, você precisa achar a soma de $l$ a $r$ só que várias vezes e para vários $l$ e $r$. É então que essa técnica começa a fazer diferença entre um código de maratona que passa e outro que não passa o problema.

*Em ciência da computação, não basta saber resolver os problemas, queremos saber como resolvê-los de forma **rápida**.*

# Créditos

Lucas Paiolla Forastiere, estudante do Bacharelado em Ciência da Computação da USP.

Telegram: @lucaspaiolla

*~ 20/03/2020*