Formando Relações de Recorrência

```
long fibonacci (int n) {
  if( n == 1 || n == 2)
      return 1;
  else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- A relação de recorrência é:

$T(n) = c$ se $n = 1$ ou $n = 2$

$T(n) = T(n-1) + T(n-2) + b$ se $n > 2$

Formando Relações de Recorrência

```
long power (long x, long n) {
   if(n == 0)
        return 1;
   else if(n == 1)
        return x;
   else if ((n % 2) == 0)
       return power (x, n/2) * power (x, n/2);
   else
       return x * power (x, n/2) * power (x, n/2);
}
```

- Caso base onde n=0 ou n=1 o tempo é uma constante *c*
- Levando em conta o pior caso onde n é impar e uma multiplicação adicional *b* é necessária

$$T(n) = c \qquad \text{se } n = 0 \text{ ou } n = 1$$

$$T(n) = 2T(n/2) + b \qquad \text{se } n > 2$$

Formando Relações de Recorrência

```
long fatorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * fatorial (n – 1);
}
```

$T(0) = c$ (1)

$T(n) = b + T(n - 1)$ (2)

$= b + b + T(n - 2)$ substituindo $T(n – 1)$ em (2)

$= b + b + b + T(n - 3)$ substituindo $T(n – 2)$ em (2)

…

$= kb + T(n - k)$

O caso base acontece quando $n – k = 0 \rightarrow k = n$, assim temos:

$T(n) = nb + T(n - n)$

$= bn + T(0)$

$= bn + c$

Logo o fatorial é $O(n)$

Formando Relações de Recorrência

```
int binarySearch (int target, int * array, int low, int high) {
   if (low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if (array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
    }
}
```

$T(1) = a$ se $n = 1$ (vetor de um elemento)

$T(n) = T(n / 2) + b$ se $n > 1$

Formando Relações de Recorrência

- Sem perda de generalidade se pode assumir que o tamanho do problema n é uma potência de 2, $n = 2^k$

$T(1) = a$ (1)

$T(n) = T(n / 2) + b$ (2)

$= [T(n / 2^2) + b] + b = T(n / 2^2) + 2b$ substituindo $T(n/2)$ em (2)

$= [T(n / 2^3) + b] + 2b = T(n / 2^3) + 3b$ substituindo $T(n/2^2)$ em (2)

= ……..

$= T(n / 2^k) + kb$

O caso base acontece quando $n / 2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$, então:

$T(n) = T(1) + b \log_2 n$

$= a + b \log_2 n$

Assim, a pesquisa binária recursiva é O(log n)

Formando Relações de Recorrência

```
int fib(int n)
{
  int a = 1, b = 1, c;
  for (int i = 3; i <= n; i++)
    {
        c = a + b;
        a = b;
        b = c;
    }
  return b;
}
```

Complexidade O(n)

Formando Relações de Recorrência

```
long fibonacci (int n) {
if( n == 1 || n == 2)
      return 1;
  else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

$T(n) = c$ se $n = 1$ ou $n = 2$ (1)

$T(n) = T(n-1) + T(n-2) + b$ se $n > 2$ (2)

Formando Relações de Recorrência

$T(n) = c$ se $n = 1$ ou $n = 2$ (1)
$T(n) = T(n-1) + T(n-2) + b$ se $n > 2$ (2)

Determinando o limite inferior de T(n):

$$\begin{aligned}
\text{Expandindo: } T(n) &= T(n-1) + T(n-2) + b \\
&\geq T(n-2) + T(n-2) + b \\
&= 2T(n-2) + b \\
&= 2[T(n-3) + T(n-4) + b] + b && \text{substituindo } T(n-2) \text{ em (2)} \\
&\geq 2[T(n-4) + T(n-4) + b] + b \\
&= 2^2T(n-4) + 2b + b \\
&= 2^2[T(n-5) + T(n-6) + b] + 2b + b && \text{substituindo } T(n-4) \text{ em (2)} \\
&\geq 2^3T(n-6) + (2^2 + 2^1 + 2^0)b \\
&\ \ldots \\
&\geq 2^kT(n-2k) + (2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0)b \\
&= 2^kT(n-2k) + (2^k - 1)b
\end{aligned}$$

O caso base ocorre quando $n - 2k = 2 \rightarrow k = (n-2)/2$

$$\begin{aligned}
\text{Logo } T(n) &\geq 2^{(n-2)/2}T(2) + [2^{(n-2)/2} - 1]b \\
&= (b + c)2^{(n-2)/2} - b \\
&= [(b + c)/2]*(2)^{n/2} - b && \rightarrow \text{Fibonacci recursivo é exponencial}
\end{aligned}$$