# Análise de Desempenho

Uma imagem com texto, Tipo de letra, captura de ecrã, branco

Descrição gerada automaticamente

Figura 1 – Função *xpto()*

A complexidade para o pior caso:

De acordo o Teorema Mestre:

Logo no pior caso, a função xpto() é constante.

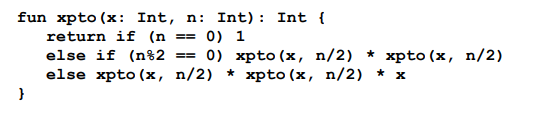


Figura 2 – Função *xpto()*

A complexidade para o pior caso

De acordo o Teorema Mestre:

Logo no pior caso, *xpto()* tem uma complexidade quadrática.

De forma a melhor a nível de complexidade, na Figura 2.1 está representada uma segunda versão da função xpto()

|  |
| --- |
| fun xpto2(x: Int, n: Int): Int{  if(n == 0) return 1  val xpto2 = *xpto2*(x, n/2)  return if(n % 2 == 0) xpto2 \* xpto2  else xpto2 \* xpto2 } |

Figura 2.1 – Versão melhorada da função xpto()

Cálculo da complexidade na segunda versão:

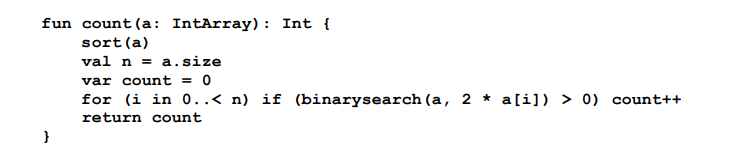


Figura 3 – Função *count()*

De acordo a implementação da função *count()* descrita na Figura 1, a sua complexidade será a seguinte:

|  |  |
| --- | --- |
| Instrução | Complexidade |
| Sort(a) | O(sort) |
| Val n = a.size | O(1) |
| Var count = 0 | O(1) |
| For(i in 0..<n) | O(n) |
| (binarysearch(A, 2\*a[i]) > 0) count++ | O(binarysearch) |
| Return count | O(1) |

Tabela 1 – Complexidade da função *count()*

Para se obter o valor da complexidade da função *count()* é necessário calcular a complexidade da função *sort()* e *binarysearch().*

A função *sort()* escolhida a propósito de deixar o algoritmo o mais eficiente possível foi a função de ordenação, *mergeSort().*

Complexidade da função *mergeSort():*

|  |  |
| --- | --- |
| Instrução | Complexidade |
| If(left < right) | O(1) |
| Val mid = (right + left) / 2 | O(1) |
| Val tableLeft = IntArray(mid – left + 1) | O(1) |
| Val tableRight = IntArray(right – mid) | O(1) |
| Divide(table, tableLeft, tableRight, left, mid, right) | O(divide) |
| mergeSort(tableLeft, 0, mid – left) | C(n/2) |
| mergeSort(tableRight, 0, right – mid – 1) | C(n/2) |
| Merge(table, tableLeft, tableRight, left, mid, right) | O(merge) |
| Total: |  |

Tabela 2 – Complexidade da função *mergeSort()*

Complexidade da função *divide():*

|  |  |
| --- | --- |
| Instrução | Complexidade |
| Var i = 0 | O(1) |
| Var j = 0 | O(1) |
| For(k in left..mid) | O(n/2) |
| tLeft[i++] = t[k] | O(1) |
| For(k in mid + 1..right) | O(n/2) |
| tRight[j++] = t[k] | O(1) |
| Total: | O(n/2)+ O(n/2) = O(n) |

Tabela 3 – Complexidade da função *divide()*

Complexidade da função *merge()*:

|  |  |
| --- | --- |
| Instrução | Complexidade |
| var i = 0 | O(1) |
| var j = 0 | O(1) |
| var k = left | O(1) |
| while (i < tLeft.size && j < tRight.size) | O(n) |
| while (i < tLeft.size) | O(n/2) |
| while (j < tRight.size) | O(n/2) |
| Total: | O(n) + O(n/2) + O(n/2) = O(n) |

Tabela 4 – Complexidade da função *merge()*

Desenvolvimento da equação de complexidade:

Equação geral:

Cálculo Auxiliar:

Para : …

=

=

=

Complexidade da função *binarySearch*:

|  |  |
| --- | --- |
| Instruções | Complexidade |
| var l = left | O(1) |
| var r = right | O(1) |
| while (l <= r) | O(n) |
| val mid = (l + r) / 2 | O(1) |
| if (x == a[mid]) return mid else if (x < a[mid])  r = mid - 1 | O(1) |
| return -1 | O(1) |

Tabela 5 – Complexidade da função *binarySearch()*

De acordo o resultado da equação de recorrência da função *mergeSort* que é linear logarítmica e da função *binarySearch* que é linear como pode ser evidenciado na Tabela 5, a função *count* é linear logarítmica.