Fundação Getulio Vargas
Estruturas de Dados e Algoritmos
Prof. Jorge Poco
Aluno: Davi Sales Barreira

Homework 1
Due: July 21, 2019

Homework 1

20/07/2019

1 Induction [3pts]

Answers should be written in this document.

1. Prove by Induction that: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \forall n \geq 0$
2. Prove by Induction that: $\forall n \geq 7$ it is true $3^n < n!$
3. Prove by Induction that $\forall n \geq 0$

$$\left\lceil \frac{n}{2} \right\rceil = \begin{cases} \frac{n}{2} & \text{si } n \text{ es par} \\ \frac{n+1}{2} & \text{si } n \text{ es impar} \end{cases}$$

4. Prove by induction that a number is divisible by 3 if and only if the sum of its digits is divisible by 3.
5. Prove that any integer greater than 59 can be formed using only 7 and 11 cent coins.
6. Prove by induction that $F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$
7. Prove by induction in n that $\sum_{m=0}^{n} \binom{n}{m} = 2^n$
8. Prove by induction that a graph with n vertices can have at most $\frac{n(n-1)}{2}$ edges.
9. Prove by induction that a complete binary tree[1] with n levels has $2^n - 1$ vertices.
10. A polygon is convex if each pair of points in the polygon can be joined by a straight line that does not leave the polygon. Prove by induction in $n > 3$ that the sum of the angles of a polygon of n vertices is $180(n-2)$.

[1] http://web.cecs.pdx.edu/ sheard/course/Cs163/Doc/FullvsComplete.html

2 Correctness of bubblesort [2pts]

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

Algorithm 1: BUBBLESORT(A)

```
1 for i = 1 to A.length − 1 do
2 |   for j = A.length downto i + 1 do
3 |   |   if A[j] < A[j − 1] then
4 |   |   |   exchange A[j] with A[j − 1]
5 |   |   end
6 |   end
7 end
```

A Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq ... \leq A'[n] \tag{1}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (1).

B State precisely a loop invariant for the **for** loop in lines 2–6, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

C Using the termination condition of the loop invariant proved in part (B), state a loop invariant for the for loop in lines 1–7 that will allow you to prove inequality (1). Your proof should use the structure of the loop invariant proof presented in this chapter.

D What is the worst-case running time of BUBBLESORT? How does it compare to the running time of insertion sort?

3 Growth of Functions [2pts]

A For each of the following pairs of functions, either $f(n)$ is in $O(g(n))$, $f(n)$ is in $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. Determine which relationship is correct and briefly explain why.

- $f(n) = \log n^2$; $g(n) = \log n + 5$
- $f(n) = \log^2 n$; $g(n) = \log n$
- $f(n) = n \log n + n$; $g(n) = \log n$
- $f(n) = 2^n$; $g(n) = 10n^2$

B Prove that $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

C Prove that $n^2 = O(2^n)$.

4 Insertion Sort - Mergesort - Quicksort [3pts]

Implement the insertion sort, merge sort and quicksort using the template `test.py` (use Python 3.X). Create a `test.cpp` file and write the equivalent code from `test.py` in C++, ie., the functions: main, `insertion_sort`, `merge_sort`, `quicksort` and `is_sorted`. For the random number generations you can use the `rand` function from `cstdlib`[2]. Your code should print the tuple (number of objects, time insertion_sort, time merge_sort, time quicksort)

You must submit both `test.py` and `test.cpp`. Graphs and descriptions must be included in this document.

4.1 Random Order

1. Create 10 sets of numbers in random order. The sets must have {10k, 20k, 30k, ..., 100k} numbers.
2. Sort these numbers using the 3 algorithms and calculate the time each algorithm takes for each set of numbers.
3. Generate a plot (using excel or another tool) showing a *linechart*, where the x-axis is the "number of elements", and the y-axis is the time that the algorithms took in C++ and Python. This plot must have 6 lines of different colors with a legend.
4. Write a small paragraph (3 to 4 lines) describing the results.

4.2 Ascending Order

Do the same experiment when the numbers are ordered in ascending order.

4.3 Descending Order

Do the same experiment when the numbers are ordered in descending order.

[2] `http://www.cplusplus.com/reference/cstdlib/rand/`

Respostas

Questão 1

1.1

Prova.
Caso Base: $n = 1$, temos $\sum_{i=1}^{n} i^2 = \frac{1 \cdot 2 \cdot 3}{6} = 1 \therefore$ Satisfeito para $n = 1$.

Hipótese indutiva: Supondo que $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, para $n = k$.

Passo de indução: Seja $n = k + 1$

$$\sum_{i=1}^{k+1} i^2 = (k+1)^2 + \frac{k(k+1)(2k+1)}{6} = \frac{(k+1)(6(k+1) + k(2k+1))}{6} = \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+1+1))(2(k+1)+1)}{6}$$

□

1.2

Prova.
Caso Base: $n = 7$, temos $n! = 7! = 5040$ e $3^7 = 2187 \therefore$ Satisfeito para $n = 7$.

Hipótese indutiva: Supondo que para $n = k, 3^k < k!$

Passo de indução: Seja $n = k + 1$, $3^k < k!$, note que como $k > 3$, temos que $k + 1 > 3$

$\therefore 3^{k+1} = 3^k \cdot 3 < k! \cdot (k+1) = (k+1)!$

□

1.3

Prova.

$$Prop : \left\lceil \frac{n}{2} \right\rceil = \begin{cases} \frac{n}{2} & \text{se } n \text{ é par} \\ \frac{n+1}{2} & \text{se } n \text{ é impar} \end{cases}$$

Caso Base:

$n = 0, \lceil \frac{0}{2} \rceil = 0 = \frac{n}{2} \therefore$ satisfeito para $n = 0$

$n = 1, \lceil \frac{1}{2} \rceil = 1 = \frac{1+1}{2} \therefore$ satisfeito para $n = 1$

Hipótese indutiva: Supondo que para $n = k$ vale a proposição.

Passo de indução: Se k for par, então $\lceil \frac{k}{2} \rceil = \frac{k}{2}$, assim $k+1$ é ímpar, logo

$$\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{k}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil = \frac{k}{2} + 1 = \frac{(k+1)+1}{2}$$

Se k for ímpar, $\lceil \frac{k}{2} \rceil = \frac{k+1}{2}$, assim $k+1$ é par, logo

$$\left\lceil \frac{k+1}{2} \right\rceil = \frac{k+1}{2}$$

□

1.4

Prova. Vamos provar que o resto da divisão de um número por 3 é igual ao resto da divisão da soma dos seus dígitos, ou seja, em P_n temos que $\overline{a_1 a_2 ... a_n} \mod 3 = \sum_{i=1}^{n} a_i \mod 3$

Caso Base: Para P_1 temos os número com 1 dígito, portanto a condição é satisfeita trivialmente já a soma dos dígitos é igual ao próprio número.

Hipótese indutiva: Supondo que P_n é válida.

Passo de indução: Analisando P_{n+1}, temos então A $= \overline{a_1 ... a_n a_{n+1}} = \overline{a_1 ... a_n} + 10 * a_{n+1}$:

$$A \mod 3 = \overline{a_1 ... a_{n+1}} \mod 3$$

$$A \mod 3 = \overline{a_0 ... a_n} + 10 a_{n+1} \mod 3$$

$$A \mod 3 = \overline{a_0 ... a_n} + a_{n+1} + 9 a_{n+1} \mod 3$$

Como $9a_{n+1} \mod 3 = 0$, então, utilizando a propriedade P_n, temos que:

$$A \mod 3 = \overline{a_0 ... a_n} + a_{n+1} \mod 3 = a_0 + ... + a_n + a_{n+1} \mod 3$$

□

1.5

Prova.
Caso Base: $n = 60 = 7 \cdot 7 + 11 \cdot 1 \therefore$ Satisfeito.

Hipótese indutiva: Supondo que para $n = k = a \cdot 7 + b \cdot 11$, onde $a, b \in \mathbb{N} \cup \{0\}$

Passo de indução: Para $n = k + 1$, temos que $k + 1 = a \cdot 7 + b \cdot 11 + 1$

Utilizaremos o fato de que $2 \cdot 11 - 3 \cdot 7 = 1$ e $8 \cdot 7 - 5 \cdot 11 = 1$

- Se $a \geq 3 \therefore k + 1 = a \cdot 7 + b \cdot 11 + 2 \cdot 11 - 3 \cdot 7 = (a-3) \cdot 7 + (b+2) \cdot 11$. Portanto, a proposição vale quando $a \geq 3$.
- Se $a < 3$, ou seja $a \in \{0, 1, 2\}$. Além disso, como $n > 59$ e $a \cdot 7 + b \cdot 11 > 59 \implies b > 4$, já que a é no máximo 2. Logo, podemos fazer:

$$k + 1 = a \cdot 7 + b \cdot 11 + 8 \cdot 11 - 3 \cdot 7 = (a+8) \cdot 7 + (b-5) \cdot 11$$

□

1.6

Prova.
$F = Fibonacci \therefore F_3 = F_2 + F_1$, além disso, $F_1, F_2 = 1$

Caso Base:

- Para $n = 1$, $F_{1+k} = F_k \cdot F_2 + F_1 \cdot F_{k-1} = F_k \cdot 1 + 1 \cdot F_{k-1} = F_{k+1} \therefore$ Satisfeito.
- Para $n = 2$, $F_{2+k} = F_k \cdot F_3 + F_2 \cdot F_{k-1} = F_k \cdot 2 + 1 \cdot F_{k-1} = F_k + F_k + F_{k-1} = F_k + F_{k+1} = F_{k+2} \therefore$ Satisfeito.

Hipótese indutiva: Supondo que $F_{n+k} = F_k \cdot F_{n+1} + F_{k-1} \cdot F_n$ e que $F_{n+k-1} = F_k \cdot F_n + F_{k-1} \cdot F_{n-1}$

Passo de indução: Sabemos que na sequência de Fibonacci $Fn + 1 + k = F_{n+k} + F_{n-1+k}$, portanto,

$$F_{n+1+k} = F_k \cdot F_{n+1} + F_{k-1} \cdot F_n + F_k \cdot F_n + F_{k-1} \cdot F_{n-1}$$

$$F_{n+1+k} = F_k(F_{n+1} + F_n) + F_{k-1}(F_n + F_{n-1}) = F_k \cdot F_{n+2} + F_{k-1} \cdot F_{n+1}$$

□

1.7

Prova.
Caso Base: $n = 1, \sum_{m=0}^{1} \binom{n}{m} = \binom{1}{0} + \binom{0}{0} = 2^1 \therefore$ Satisfeito.

Hipótese indutiva: Supondo que para $n = \sum_{m=0}^{1} \binom{k}{m} = 2^k$

Passo de indução: Seja $n = k + 1$, utilizaremos a seguinte identidade $\binom{k+1}{m} = \binom{k}{m} + \binom{k}{m-1}$.

$$2 \cdot \sum_{m=0}^{k} \binom{k}{m} = \Big[\underbrace{\binom{k}{0}}_{\binom{k+1}{0}} + \underbrace{\binom{k}{0} + \binom{k}{1}}_{\binom{k+1}{1}} + \binom{k}{1} + \binom{k}{2} + ... + \underbrace{\binom{k}{k-1} + \binom{k}{k}}_{\binom{k+1}{k}} + \underbrace{\binom{k}{k}}_{\binom{k+1}{k+1}} \Big]$$

$$2 \cdot 2^k = \left[\binom{k+1}{0} + \binom{k+1}{1} + ... + \binom{k+1}{k+1} \right] = \sum_{m=0}^{k+1} \binom{k+1}{m}$$

□

1.8

Prova.
Caso Base: $n = 1$, temos 0 arestas, portanto, $\frac{n(n-1)}{2} = 0 \therefore$ Satisfeito.

Hipótese indutiva: Supondo que para $n = k$, temos um número máximo de arestas igual a $\frac{k(k-1)}{2}$

Passo de indução: Adicionando um novo vértice, temos que ele pode formar arestas com os demais k vértices, logo o total de arestas aumenta em k, ou seja, temos $\frac{k(k-1)}{2} + k$ arestas.

Portanto, número de arestas para $k+1$ vértices é igual à

$$\frac{k(k-1)}{2} + k = \frac{k(k-1+2)}{2} = \frac{(k+1)(k+1-1)}{2}$$

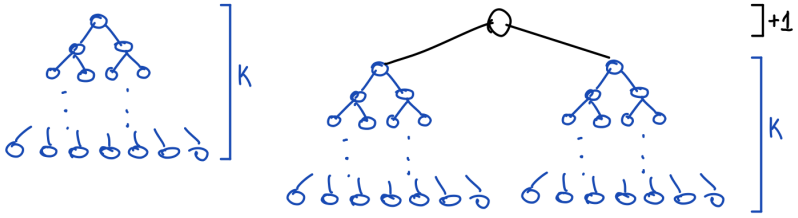
□

1.9

Prova.
Caso Base: $n = 1$ temos que $2^1 - 1 = 1 \therefore$ Satisfeito.

Hipótese indutiva: Supondo que para $n = k$ camadas temos $2^k - 1$ vértices.

Passo de indução: Adicionando uma nova camada às k existentes, ficamos com a estrutura ilustrada abaixo.



Assim, para k camadas a quantidade de vértices era $2^k - 1$, e ao adicionar uma nova camada temos $2(2^k - 1) + 1$ vértices, como pode ser observado na figura acima. Portanto, para $k+1$ camadas, temos que a quantidade de vértices é $2^{k+1} - 1$.

□

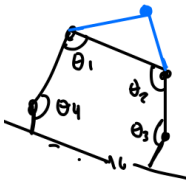
1.10

Prova.
Caso Base: $n = 4$, temos pela definição de polígonos convexoes que podemos traçar uma linha entra dois vértices sem que saia do polígono. Portanto, podemos formar dois triângulo. Como seus ângulos somam 180 cada, temos assim que a soma dos ângulos internos é igual à $180 \cdot 2 = 180(4-2)$

Hipótese indutiva: Supondo que a propriedade é válida para polígonos de $n = k$ vértices.

Passo de indução: Tome um polígono convexo de k vértices. Um polígono convexo de $k+1$ vértices pode ser formado adicionando um novo vértice fora do polígono original.

Portanto, a soma dos ângulos internos será dada por $180(k-2) + 180 = 180(k+1-2)$, como queríamos demonstrar. □



Questão 2

2.A

Se sabemos que $A' = BUBBLESORT(A)$, tal que $A'[0] \leq ... \leq A'[n]$, onde $n = A.length$. Então, o algoritmo termina com seus elementos ordenados. A única coisa que basta garantir é que os elementos de A' são os mesmos elementos de A. Com isso, A' será A ordenado e o algoritmo é eficaz.

2.B

Primeiro, faça $n = A.length$

Assim, o invariante é que no início de cada loop o menor elemento de $A[i, .., n]$ estará no máximo na posição j.

Para o primeiro loop (inicialização), $j = n$, logo o invariante é trivialmente satisfeito, já que o menor elemento de $A[i, ..., n]$ está em no máximo na posição n. Vamos mostrar que esse invariante se mantém.

Suponha que o menor elemento está no máximo na posição $j = k$. Assim, se o menor elemento estiver em $A[k]$, então $A[k] < A[k-1]$, e trocamos os elementos de posição, tal que sabemos que o menor elemento de $A[i, ..., n]$ vai estar no máximo na posição $k-1$. Se $A[k]$ não é o menor elemento, então o menor elemento já estava no máximo na posição $k-1$ e o loop não faz nada. Assim, provamos que o invariante se mantém nas duas condições.

No término, temos que $j = i + 1$, então, garantimos que o menor elemento está no máximo na posição i.

2.C

Para o loop externo, o invariante é que no início de cada loop, $A[1, ...i-1]$ contém os menores $i-1$ elementos e está ordenado.

Na inicialização, temos que $i = 1$, assim, $A[i-1] = A[0]$ e, portanto, o invariante é verdadeiro por vacuidade.

Supondo que para $i = k$, no início do loop, o vetor $A[j, ..., k-1]$ está ordenado com os menores $k-1$ valores de A. Pelo item "B", sabemos que ao final do loop interno (linhas 2 a 6), o menor elemento de $A[k, ...n]$ vai estar na posição k. Portanto, ao final dessa iteração, $A[1, ..., k]$ vai estar ordenado, já que $A[k] \geq A[k-1]$ pelo loop interno, e $A[1] \leq ... \leq A[k-1]$ por hipótese. Logo, provamos que o invariante se mantém.

No término, temos que $i = n + 1$, logo $A[1, ..., n]$ vai estar ordenado, concluindo nossa prova.

2.D

Note que o algoritmo do BUBBLESORT sempre realiza $n-1$ iterações para o loop exterior e $n-i$ iterações para o loop interior, onde i é a iteração do loop exterior. Logo, o total de iterações é dado por $(n-1)+(n-2)+...+1 = n(n-1)/2$. Assim, temos $\Theta(n^2)$ para o melhor e para o pior caso. Comparado ao *Insertion Sort*, o BUBBLESORT tem um pior desempenho, já que o pior caso do *Insertion Sort* também é $\Theta(n^2)$, porém, seu melhor caso é $\Theta(n)$.

Questão 3

Iremos utilizar as seguintes definições:

- $f(n) = O(g(n)) \iff \exists n_o \in \mathbb{N}$, e $\exists c_1 \in \mathbb{R}_+$ tal que $\forall n \geq n_o \implies f(n) \leq c_1 \cdot g(n)$
- $f(n) = \Omega(g(n)) \iff \exists n_o \in \mathbb{N}$, e $\exists c_2 \in \mathbb{R}_+$ tal que $\forall n \geq n_o \implies f(n) \geq c_2 \cdot g(n)$
- $f(n) = \Theta(g(n)) \iff \exists n_o \in \mathbb{N}$, e $\exists c_1, c_2 \in \mathbb{R}_+$ tal que $\forall n \geq n_o \implies c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n)$

A)

- $f(n) = \Theta(g(n))$;

  Prova. $f(n) = \log n^2 = 2\log n$, e $g(n) = \log n + 5$, assim, para $n_o = 2, c_1 = 2, c_2 = 1/10$, temos que $\forall n \geq n_o \implies 1/10 \log n + 5/10 \leq 2\log n \leq 4\log n + 20 \therefore f(n) = \Theta(g(n))$ □

- $f(n) = \Omega(g(n))$;

  Prova. $f(n) = \log^2 n = \log n \cdot \log n$ e $g(n) = \log n$. Tomando $c_2 = 1, n_o = 2$, temos que $\forall n \geq n_o \implies \log^2 n \geq \log n \therefore f(n) = \Omega(g(n))$, pois não existe c_1 e n_o que satisfaçam a condição do $f(n) = O(g(n))$. □

- $f(n) = \Omega(g(n))$;

  Prova. $f(n) = n\log n + n$ e $g(n) = \log n$. Tomando $c_2 = 2, n_o = 2$, temos que $\forall n \geq n_o \implies n\log n + n \geq \log n \therefore f(n) = \Omega(g(n))$, pois não existe c_1 e n_o que satisfaçam a condição do $f(n) = O(g(n))$. □

- $f(n) = \Omega(g(n))$;

  Prova. $f(n) = 2^n$ e $g(n) = 10n^2$. Tomando $c_2 = 1/10, n_o = 4$, temos que $\forall n \geq n_o \implies 2^n \geq 10n^2/10 \therefore f(n) = \Omega(g(n))$, pois não existe c_1 e n_o que satisfaçam a condição do $f(n) = O(g(n))$. □

B)

Prova.
Queremos encontrar n_o, c_1 e c_2 tal que $\forall n \geq n_o \implies c_2 \cdot n^3 \leq n^3 - 3n^2 - n + 1 \leq c_1 \cdot n^3$.

Para isso, basta tomar $n_o = 10, c_1 = 2, c_2 = 0.5$, assim a desigualdade é satifeita, e portanto $n^3 - 3n^2 - n + 1 = \Theta(n^3)$

□

C)

Prova.
Queremos mostrar que $n^2 = O(2^n)$

Basta tomar $n_o = 5, c_1 = 1$, assim $\forall n \geq n_o$, temos que $n^2 \leq 2^n$. Portanto, $n^2 = O(2^n)$

□

Questão 4

Os gráficos abaixo comparam os resultados obtidos para os diferentes algoritmos de sorting implementados tanto em C++ como em Python, sendo os gráficos da direita em escala logaritmica para o eixo y. Como esperado, o código de C++ performa melhor que em Python, com a diferença maior sendo no algoritmo de *insert sort*.

Além disso, observamos que o *quicksort* apresentou o melhor desempenho considerando tanto o vetor com valores aleatórios como o com valores descendentes. No vetor já ordenado (*ascending*), o tempo dos três algoritmos é muito baixo, como era de se esperar, porém, nesse caso o *insert sort* foi quem apresentou o melhor resultado.

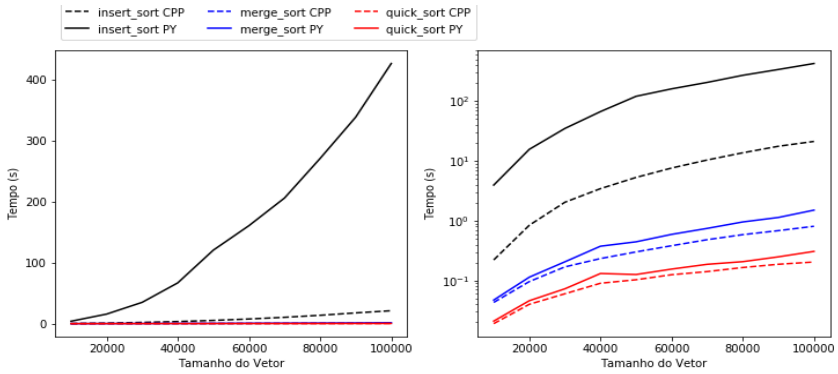


Figure 1: Comparação de performance dos algoritmos de *sorting* para vetores aleatórios

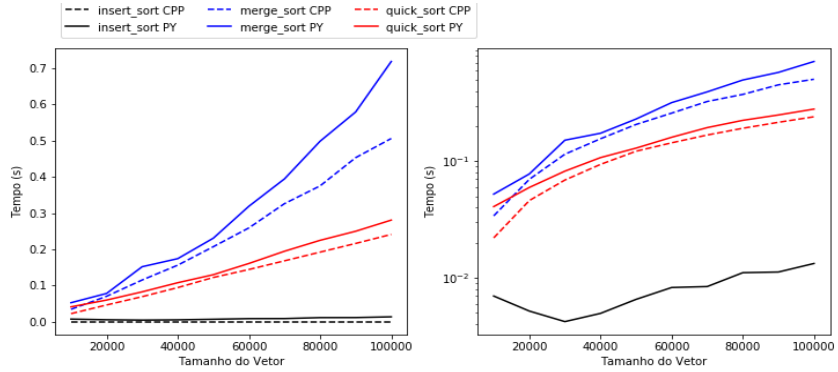


Figure 2: Comparação de performance dos algoritmos de *sorting* para vetores crescentes

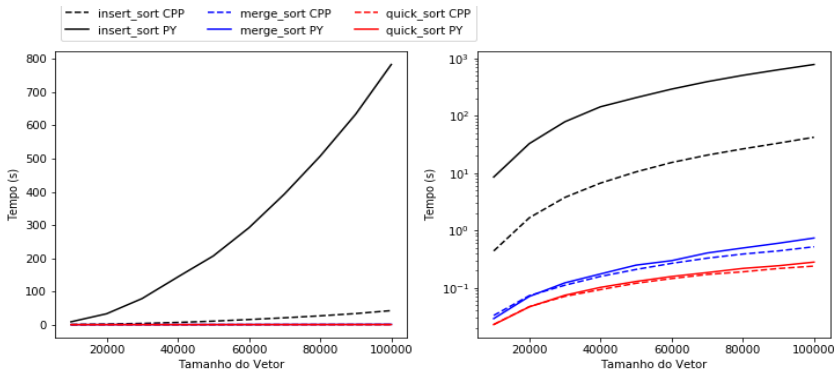


Figure 3: Comparação de performance dos algoritmos de *sorting* para vetores decrescentes