Pontifícia Universidade Católica do Paraná
Escola Politécnica – Bacharelado em Ciência da Computação – 7º Período
Prof. Edson Emílio Scalabrin

Trabalho:	03
Grupo:	até 5 integrantes.
Formato de avaliação:	**F-A** para quem optar por **PROVA + TED + TRABALHO**, **F-B** para quem optar por **TED + TRABALHO**,
Data da entrega:	**F-A** na quinta-feira, dia da 3ª avaliação somativa. **F-B** na quinta-feira, dia da 3ª avaliação somativa.
Data da apresentação:	**F-A** na quinta-feira, dia da 3ª avaliação somativa. **F-B** na quinta-feira, dia da 3ª avaliação somativa.
Peso na nota do RA02:	**F-A** 40%. **F-B** 90%.
Dinâmica da defesa:	assume-se que cada integrante da equipe conhece o projeto como um todo e é capaz de defende-lo na sua amplitude. **Dinâmica:** cada integrante deve apresentar parte do projeto equivalente ao número de integrantes, por exemplo, em uma equipe de 3 integrantes, cada um deles deve apresentar 33%. **Essa participação é compulsória para obtenção da sua nota na atividade avaliativa.**
Tempo máximo:	10 minutos por equipe.
Data de entrega:	24/06/2021

Descrição do problema: um capitulo da análise de complexidade de algoritmos envolve a com escolha aleatória para dar o próximo passo. Neste contexto, dados os códigos em Python, **seção "2. Algoritmos em Python"**, que implementam versões recursivas clássicas e aleatórias dos algoritmos ***quicksort, selectsort e mergesort***, pede-se para realizar as atividades descritas na **seção "1. Detalhamento do Projeto"**, estrutura em ***Parte 01, Parte 02, Parte 03, Parte 04 e Parte 05***. E para ajudar na compressão da atividade, em particular, com relação a noção aleatória associada ao projeto de cada algoritmo, foi incluída na **seção "3. Ordenação: uma versão aleatória**", uma descrição detalhada da análise de ***quicksort*** aleatório. Essa análise pode ser generalizada para os demais algoritmos descritos na seção "**2. Algoritmos em Python".**

1 Detalhamento do Trabalho

1.1 Parte 01

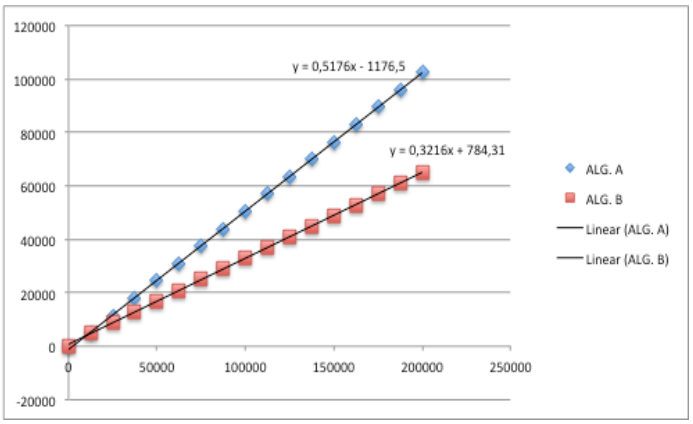
a) Implementar os algoritmos recursivos: ***quicksort***, ***selectsort*** e ***mergesort***. Pode-se usar ***como ponto de partida*** os códigos Python fornecidos na seção **"2. Algoritmos em Python";** e

b) implementar os algoritmos recursivos e aleatórios: ***quicksort***, o ***selectsort***, o ***mergesort***. Pode-se também usar ***como ponto de partida*** os códigos Python fornecidos na seção "**2. Algoritmos em Python".**

1.2 Parte 02

Criar diferentes conjuntos ordenados de valores descrentes (e.g., ...,6,5,4,3,2,1), e crescentes (e.g, 1,2,3,4,5,6,...), assim como conjuntos não ordenados de valores (e.g., 3, 4, 1, 5, 20, 6, 3, ...). Esses conjuntos devem ter diferentes tamanhos (e.g., 10000, 20000, 30000, 40000, 50000); o maior tamanho é aquele o seu computador levar 10 segundos para ordenar 1 único vetor. Os resultados desta experimentação, relacionando os tamanhos dos vetores, e dos tempos de processamento, deve ser apresentados na forma gráficos, onde x é o tamanho do vetor, e y é o tempo necessário para ordenar o vetor (ver exemplo a seguir).

Para cada algoritmo implementado, e para cada vetor (ordenado ou invertido), deve-se realizar uma regressão estatística para prever quanto tempo o computador usado vai levar para ordenar um vetor duas vezes maior que o maior x analisado...



Em resumo, após realização do experimento, deve-se apresentar:

- a fórmula da equação de regressão os experimentos com cada algoritmo;
- os gráficos com os pontos observados plotados em linha/curva de tendência;
- as previsões para vetores cujos tamanhos são 2 (duas) vezes maiores que o maior analisado, 3 (três) vezes maiores e 4 (quarto) vezes maiores.

FONTE: Cormen et al 2002, Algoritmos: Teoria e Prática

- as diferenças, comparativamente, entre as versões aleatórias e não aleatórias das implementações do ***quicksort***, ***selectsort*** e ***mergesort***.
- opcional, fazer algum ajuste (buscando otimizar) em algum dos algoritmos experimentados e confrontar os resultados. Ajustes: reduzir o número de atribuições, substituir um comando por outro, etc.

1.3 Parte 03

Que conclusão a equipe obteve dos algoritmos randomizados ou aleatórios. Eles representam uma opção viável? Não faz nenhum sentido pensar nesta abordagem? Há alguma relação no tocante ao custo médio dos algoritmos randomizados experimentados? Etc.

Para apoiar as discussões e conclusões, a equipe poderá consultar a **seção "Ordenação: uma versão aleatória**"; esse texto foi retirado do Livro Texto (ref. no rodapé da página).

1.4 Parte 04

Para cada algoritmo recursivo, que são: *quicksort*, *selectsort* e *mergesort*, escrever sua equação de recorrência. Aqui, nós pedimos que sejam incluídos todos os custos, ou seja, incluir na contabilização dos comandos executados.

1.5 Parte 05

Preparar uma apresentação para a data definida no plano de ensino. Essa apresentação deve ser feita com slides. Os slides podem conter gráficos, textos, hipóteses, códigos fontes, etc. O tempo de apresentação será limitado a 10 minutos.

2 Algoritmos em PYTHON

O caso aleatório do ***quicksort*** consiste em um elemento escolhido ao acaso a partir do subarranjo A[p, ..., r], e o caso aleatório do ***mergesort*** consiste uma divisão de subarranjo A[p, ..., r] com tamanho ao acaso. No caso do ***quicksort***, essa modificação, em que a amostragem aleatória é feita no intervalo p, ..., r, assegura que o elemento pivô x = A[r] tem a mesma probabilidade de ser qualquer um dos r – p + 1 elementos do subarranjo. O ***selectsort*** segue abordagem precedida ***quicksort***.

2.1 Quick sort

2.1.1 Recursivo

```
def swap(A, i, j):
    aux = A[ i ]
    A[ i ] = A[ j ]
    A[ j ] = aux

def partition(A, p, r):
    x = A[ r ]
    i = p - 1
    for j in range(p, r, 1):
        if A[ j ] <= x:
            i = i + 1
            swap(A, i, j)
    swap(A, i + 1, r)
    return i + 1

def quick_sort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quick_sort(A, p, q - 1)
        quick_sort(A, q + 1, r)

def quick_sort_wapper(A):
    N = len(A) - 1
    quick_sort(A, 0, N)
    return A
```

2.1.2 Recursivo e aleatório

```
from random import randint

def swap_r(A, i, j):
    aux = A[ i ]
    A[ i ] = A[ j ]
    A[ j ] = aux

def partition_r(A, p, r):
    x = randint(p, r)
    swap_r(A, x, r)
    x = A[ r ]
    i = p - 1
    for j in range(p, r, 1):
        if A[ j ] <= x:
            i = i + 1
            swap_r(A, i, j)
    swap_r(A, i + 1, r)
    return i + 1

def quick_sort_r(A, p, r):
    if p < r:
        q = partition_r(A, p, r)
        quick_sort_r(A, p, q - 1)
        quick_sort_r(A, q + 1, r)

def quick_sort_random_wapper(A):
    N = len(A) - 1
    quick_sort_r(A, 0, N)
    return A
```

2.2 Merge sort

2.2.1 Recursivo

```
def merge(A, parte_esq, parte_dir):
    i, j, k = 0, 0, 0
    while i < len(parte_esq) and j < len(parte_dir):
        if parte_esq[ i ] <= parte_dir[ j ]:
            A[ k ] = parte_esq[ i ]
            i = i + 1
        else:
            A[ k ] = parte_dir[ j ]
            j = j + 1
        k = k + 1

    while i < len(parte_esq):
        A[ k ] = parte_esq[ i ]
        i = i + 1
        k = k + 1

    while j < len(parte_dir):
        A[ k ] = parte_dir[ j ]
        j = j + 1
        k = k + 1

def merge_sort(A):
    if len(A)>1:
        divide = len(A)//2
        parte_esq = A[:divide]
        parte_dir = A[divide:]

        merge_sort(parte_esq)
        merge_sort(parte_dir)

        merge(A, parte_esq, parte_dir)

def merge_sort_wapper(A):
    merge_sort(A)
    return A
```

2.2.2 Recursivo e aleatório

```
from random import randint

def merge(A, parte_esq, parte_dir):
    i, j, k = 0, 0, 0
    while i < len(parte_esq) and j < len(parte_dir):
        if parte_esq[ i ] <= parte_dir[ j ]:
            A[ k ] = parte_esq[ i ]
            i = i + 1
        else:
            A[ k ] = parte_dir[ j ]
            j = j + 1
        k = k + 1

    while i < len(parte_esq):
        A[ k ] = parte_esq[ i ]
        i = i + 1
        k = k + 1

    while j < len(parte_dir):
        A[ k ] = parte_dir[ j ]
        j = j + 1
        k = k + 1


def mege_sort_random(A):
    N = len(A)
    if N > 1:
        divide = randint(1, N - 1)
        parte_esq = A[:divide]
        parte_dir = A[divide:]

        mege_sort_random(parte_esq)
        mege_sort_random(parte_dir)

        merge(A, parte_esq, parte_dir)

def merge_sort_random_wapper(A):
    mege_sort_random(A)
    return A
```

2.3 Select

2.3.1 Recursivo

```
def min_indice(A, i, j):
    if i == j:
        return i
    k = min_indice(A, i + 1, j)
    return i if A[i] < A[k] else k

def select_sort(A, n, indice = 0):
    if indice == n:
       return
    k = min_indice(A, indice, n-1)
    if k != indice:
        A[k], A[indice] = A[indice], A[k]
    select_sort(A, n, indice + 1)
    return A

def select_sort_wapper(X):
    return select_sort(X, len(X))
```

2.3.2 Recursivo e aleatório

```
from random import randrange

def partition(x, indice_do_pivo = 0):
    i = 0

    if indice_do_pivo != 0:
        x[0],x[indice_do_pivo] = x[indice_do_pivo],x[0]

    for j in range(len(x) - 1):
        if x[j + 1] < x[0]:
            x[j + 1], x[i +1 ] = x[i + 1], x[j + 1]
            i += 1

    x[0], x[i] = x[i], x[0]
    return x, i

def select_sort_random(x, k):
    if len(x) == 1:
        return x[0]
    else:
        parte_do_x = partition(x, randrange(len(x)))
        x = parte_do_x[ 0 ] # arranjo particionado
        j = parte_do_x[ 1 ] # indice do pivo
        if j == k:
            return x[ j ]
        elif j > k:
            return select_sort_random(x[:j], k)
        else:
            k = k - j - 1
            return select_sort_random(x[(j+1):], k)

def select_sort_random_wapper(x, contador = 0, Q = []):
    if contador == len(x):
        return Q
    else:
        Q.append(select_sort_random(x, contador))
        select_sort_random_wapper(x, contador + 1, Q)
    return Q
```

3 Ordenação: uma versão aleatória

Na exploração do comportamento do caso médio de ***quicksort***, foi feita uma suposição de que todas as permutações dos números de entrada são igualmente prováveis. Adicionando assim um carácter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas.

Hipótese: É preferível a versão aleatória resultante do ***quicksort*** para a ordenação de entrada de dados suficientemente grande.

Como o elemento pivô é escolhido ao acaso, espera-se que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média (verificar no experimento).

As mudanças na partição e no ***quicksort*** são pequenas. Em outras palavras, a modificação no novo procedimento de partição limita-se as trocas antes do particionamento de fato:

partição-aleatória(*A, p, r*)
1. $i \leftarrow$ Randon(*p, r*)
2. trocar *A*[*r*] $\leftrightarrow$ A[*i*]
3. retornar **partição**(*A, p, r*)

quicksort-aleatório(*A, p, r*)
1. **if** $p < r$
2. **then** q $\leftarrow$ **partição-aleatória**(*A, p, r*)
3. **quicksort-aleatório**(*A, p, q* – 1)
4. **quicksort-aleatório**(*A, q* – 1, *r*)

Reflexão
Por que se analisa o desempenho do caso médio de um algoritmo aleatório e não seu desempenho no pior caso?

Pior Caso:
Uma divisão do pior caso em todo nível de recursão do ***quicksort*** produz um tempo de execução igual a $\Theta(n^2)$, que, intuitivamente, é o tempo de execução do pior caso do algoritmo.

Prova:
Método: substituição

Hipótese: tempo de execução de quicksort é $O(n^2)$.

Recorrência: seja $T(n) = \max_{0 \le q \le (n-1)} \left(T(q) + T(n-q-1)\right) + \theta(n)$, o tempo no pior caso para o procedimento ***quicksort*** sobre uma entrada de tamanho n.

Onde o parâmetro q varia de 0 a (n – 1), pois o procedimento ***partição*** produz dois subproblemas com tamanho total n – 1. Supondo que T(n) <= cn^2 para alguma constante c. Pela substituição dessa suposição na recorrência, tem-se:

$$T(n) = \max_{0 \le q \le (n-1)} (cq^2 + c(n-q-1)^2) + \theta(n)$$

$$= c \cdot \max_{0 \le q \le (n-1)} (q^2 + (n-q-1)^2) + \theta(n)$$

A expressão $q^2 + (n - q - 1)^2$ alcança um máxima sobre o intervalo $0 \le q \le (n - 1)$ do parâmetro em um dos pontos extremos, como pode ser visto pelo fato da **segunda derivada** de expressão com relação a q ser positiva:

Max quando: $q = 0$ ou $q = n - 1$

Essa observação nos dá o limite $\max_{0 \le q \le (n-1)} (q^2 + (n-q-1)^2) \le (n-1)^2$ e continuando a definição do limite de T(n) tem-se:

$$T(n) \le n^2 + c(2n-1) + \theta(n)$$

$$T(n) \le n^2$$

Pode-se então escolher um valor para a constante c grande o suficiente para que o termo $c(2n - 1)$ domine o termo $\Theta(n)$. Portanto, $T(n) = O(n^2)$.

Deve-se salientar, como visto em aulas anteriores, quando o particionamento é desequilibrado o ***quicksort*** demora o tempo $\Omega(n^2)$. Assim, o tempo de execução (no pior caso) de ***quicksort*** é $O(n^2)$.

Tempo de execução esperado

Intuitivamente, o tempo de execução do caso médio de ***quicksort-aleatório*** é $O(n \log n)$: se, em cada nível de recursão, a divisão induzida por ***partição-aleatória*** colocar qualquer fração constante dos elementos em um lado da ***partição***, então a árvore de recursão terá a profundidade $O(\log n)$, e o trabalho $O(n)$ será executado em cada nível.

Observação:

- a adição de novos níveis com a divisão ***mais desequilibrada*** possível entre esses níveis, não altera o tempo total $O(n \log n)$.

Tempo de execução e comparações

1. Deve-se notar que o tempo de execução de ***quicksort*** é <u>*dominado*</u> pelo tempo gasto no procedimento ***partição***.
2. Deve-se notar também que toda vez que o procedimento ***partição*** é chamado, um elemento *pivô* é selecionado, e esse elemento nunca é incluído em quaisquer chamadas recursivas futuras: ao ***quicksort*** e à ***partição***. Logo, pode haver no máximo *n* chamadas a ***partição*** durante a execução inteira do algoritmo de ***quicksort***.

```
quicksort (A, p, r)
1. if p < r
2.    then   q ← partição (A, p, r)
3.           quicksort (A, p, q – 1)
4.           quicksort (A, q – 1, r)

partição (A, p, r)
1. x ← A[ r ]
2. i ← p – 1
3. for j ← p to r – 1
4.    do if A[ j ] ≤ x
5.           then i ← i + 1
6.                trocar A[ i ] ← A[ j ]
7. trocar A[i + 1 ] ↔ A[ r ]
8. return i + 1
```

Uma chamada a ***partição*** demora o tempo O(1) mais um período de tempo proporcional ao número de iterações do *loop* **for** das ***linhas 3*** e ***6***. Cada iteração desse *loop* **for** executa uma comparação na ***linha 4***, elemento do arranjo A. Assim, se for possível contar o número total de vezes que a ***linha 4*** é executada pode-se limitar o tempo total gasto no *loop* **for** durante toda execução de ***quicksort***.

Seja *X* o número de comparações executadas na ***linha 4*** de ***partição*** por toda a execução de ***quicksort*** sobre um arranjo de n elementos. Então, o tempo de execução de ***quicksort*** é O(*n* + *X*).

Prova

Dado o que já foi dito, há *n* chamadas a partição, cada uma das quais faz uma proporção constante do trabalho e depois executa o *loop* **for** um certo número de vezes. Cada iteração do *loop* **for** executa a ***linha 4***. □

Portanto, a meta é calcular *X*, *i.e.,* o número total de comparações executadas em todas as chamadas a ***partição***. De forma mais objetiva, será derivado um limite global sobre o número total de comparações.

Procedimento

Reconhecer quando o algoritmo compara 2 elementos do arranjo e quando ele não o faz.

Para facilitar, será renomeado os elementos do arranjo A como $z_1, z_2, ..., z_n$, com z_i sendo o *i-ésimo* menor elemento. Será também definido o conjunto $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$ como o conjunto de elementos entre z_i e z_j inclusive.

Quando o algoritmo compara z_i e z_j?

Deve-se notar que cada ***par*** de elementos é comparado no máximo uma vez. Pois, os elementos são comparados apenas ao elemento *pivô* e, depois que uma chamada específica de ***partição*** termina, o elemento *pivô* usado nessa chamada de nunca é comparado novamente a quaisquer outros elementos.

Usando variáveis indicadoras aleatórias, tem-se:

$$X_{ij} = I\{z_i \text{ é } comparado\ a\ z_j\}, \qquad I(A) = \begin{cases} 1 & se\ A\ ocorre \\ 0 & se\ A\ não\ ocorre \end{cases}$$

Onde está considerando *se* a comparação acontece em qualquer instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chama da ***partição***. Dado que cada ***par*** é comparado no máximo um vez, pode-se caracterizar o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Tomando as EXPECTATIVAS em ambos os lados, e depois usando a linearidade de EXPECTATIVA, tem-se:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ é } comparado\ a\ z_j\}$$

Como calcular $\Pr\{z_i$ é $\boldsymbol{comparado\ a\ z_j}\}$?

Assume-se que cada *pivô* é escolhido ao acaso e de forma independente.

Escolha do pivô

Uma vez que um *pivô x* é escolhido com $z_i < x < z_j$, sabe-se que z_i e z_j não podem ser comparados em qualquer momento subsequente. Se, por outra lado, z_i for escolhido como *pivô* antes de qualquer outro item em Z_{ij}, então z_i será comparado a cada item Z_{ij}, exceto ele mesmo. Analogamente, se z_j for escolhido como *pivô* antes de qualquer outro item em Z_{ij}, então z_j será comparado a cada item em Z_{ij}, exceto ele próprio.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
5	**4**	**10**	**9**	**8**	**7**	**6**	**1**	**2**	**3**

pivô = 7

1	*2*	*3*	*4*	*5*	*6*	*7*
5	**4**	**6**	**1**	**2**	**3**	pivô=**7**
			E			

8	*9*	*10*
10	**9**	**8**
	D	

Deve-se observar que a partir deste ponto, nenhum número do subarranjo *E* será comparado a nenhum outro número do subarranjo *D*. Logo, z_i e z_j são comparados se e somente se o primeiro elemento a ser escolhido como *pivô* de Z_{ij} e z_i ou z_j.

Exemplo

pivô = *x* = 7

$z_i < x < z_j$
5 < 7 < 10

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
5	**4**	**10**	**9**	**8**	**7**	**6**	**1**	**2**	**3**
i		***j***							

Caso 5 fosse escolhido como *pivô*, então z_i e z_j serão comparados. Caso 10 fosse escolhido como *pivô* então z_i e z_j serão também comparados.

Deve-se então calcular a probabilidade de que o evento ocorra: z_i ou z_j ser escolhido com o *pivô*.

Considerações

a) Todo o conjunto Z_{ij} está reunido na mesma ***partição***.
b) Todo elemento de Z_{ij} tem igual probabilidade de ser o primeiro escolhido como *pivô*.
c) Dado que o conjunto Z_{ij} tem $j - i + 1$ elementos e tendo em vista que os *pivôs* são escolhidos ao acaso e de forma independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como *pivô* é:

$$\frac{1}{(j - i + 1)}$$

Tem-se,

$$\Pr\{z_i \text{ é } comparado\ a\ z_j\} = \Pr\{z_i\ ou\ z_j \text{ é } o\ primeiro\ escolhido\ de\ Z_{ij}\}$$

$$= \Pr\{z_i \text{ é } o\ primeiro\ pivô\ escolhido\ de\ Z_{ij}\} + \Pr\{z_j \text{ é } o\ primeiro\ pivô\ escolhido\ de\ Z_{ij}\}$$

$$= \frac{1}{j - i + 1} + \frac{1}{j - i + 1}$$

$$= \frac{2}{j - i + 1}$$

Como os elementos são mutuamente exclusivas, tem-se

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Pode-se então avaliar essa soma usando uma troca de variáveis ($k = j - i$) e o limite sobre a série harmônica.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k + 1}$$

aproximando para chegar na série harmônica

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$

Série harmônica

Para inteiros positivos n, o enésimo ***número harmônico*** é

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

$$H_n = \ln n + O(1)$$