Fundação CECIERJ - Vice Presidência de Educação Superior a Distância
Curso de Tecnologia em Sistemas de Computação
Disciplina: Programação com Interfaces Gráficas
AD1 2^o semestre de 2018.
Professores: Mario Benevides e Paulo Roma

AD1: Calculando Distâncias entre Classificações

1 Introdução

Todos nós já vimos classificações (*rankings*) de algum tipo, sejam de faculdades, telefones inteligentes ou comerciais de Futebol. Os motores de busca produzem rotineiramente, sob demanda, classificações e recomendações de hotéis, restaurantes ou eletrodomésticos de acordo com uma variedade de critérios. É claro que classificações produzidas por meios diferentes, ou em momentos diferentes, podem discordar. A tabela 1 ilustra isso, mostrando duas classificações das cidades mais caras do mundo [1].

Isso leva a uma pergunta natural: como podemos medir o grau de similaridade (ou diferença) entre dois *rankings* para o mesmo conjunto de itens? Acontece que matemáticos e estatísticos estudaram esse problema por mais tempo do que a Internet existe. As medidas de distância entre os *rankings* que eles desenvolveram, agora são usadas amplamente pelos motores de busca [2].

Neste projeto, você implementará algoritmos para calcular duas dessas medidas. Para definir precisamente essas medidas, primeiro precisamos de uma terminologia e notação.

Suponha que são dados um conjunto U de n items a serem classificados - na tabela 1, U = Copenhagen, Geneva, Oslo, Tokyo, Zurich e $n = 5$. Uma classificação para U é uma função σ que atribui a cada item i em U um

[1] Adaptado de: `http://www.nbcnews.com/business/worlds-10-most-expensive-cities-live-eat-big-mac-buy-999380`

[2] C. Dwork, R. Kumar, M. Naor, D. Sivakumar. Rank Aggregation Methods for the Web. WWW10, 2 a 5 de maio de 2001, Hong Kong. `http://www10.org/cdrom/papers/577/`

Cidade	2011 σ_1	2012 σ_2
Copenhagen	4	5
Geneva	3	4
Oslo	1	1
Tokio	5	3
Zurich	2	2

Tabela 1: Cinco cidades classificadas de acordo com o custo de vida em dois anos consecutivos.

inteiro distinto $\sigma(i)$ entre 1 e n. A função σ determina uma ordem

$$\sigma(a_1) < \sigma(a_2) < \cdots < \sigma(a_n)$$

entre os items em U, onde valores-σ menores implicam em *ranking* maiores. Classificações diferentes implicam em ordens diferentes.

Exemplo 1. Classificações de 2011 e 2012 da Tabela 1 são dados pelas funções σ_1 e σ_2, respectivamente, onde:

$$\begin{aligned} \sigma_1(Oslo) = 1 &< \sigma_1(Zurich) = 2 < \sigma_1(Geneva) = 3 \\ &< \sigma_1(Copenhagen) = 4 < \sigma_1(Tokyo) = 5 \\ &\text{e} \\ \sigma_2(Oslo) = 1 &< \sigma_2(Zurich) = 2 < \sigma_2(Tokyo) = 3 \\ &< \sigma_2(Geneva) = 4 < \sigma_2(Copenhagen) = 5. \end{aligned} \tag{1}$$

Defineremos agora as nossas medidas de distância.

Definição 1. A distância footrule de Spearman (ou a distância footrule, para simplificar) entre duas classificações σ_1 e σ_2 de um conjunto U é dada por

$$F(\sigma_1, \sigma_2) = \sum_{a \in U} |\sigma_1(a) - \sigma_2(a)|$$

onde "$|.|$" denota valor absoluto. Então, a distância footrule entre σ_1 e σ_2 é o deslocamento total dos items entre σ_1 e σ_2.

Exemplo 2. Para as classificações da Tabela 1, usando a notação do Exemplo 1, temos:

$$\begin{aligned} F(\sigma_1, \sigma_2) &= |\sigma_1(Copenhagen) - \sigma_2(Copenhagen)| \\ &+ |\sigma_1(Geneva) - \sigma_2(Geneva)| \\ &+ |\sigma_1(Oslo) - \sigma_2(Oslo)| \\ &+ |\sigma_1(Tokyo) - \sigma_2(Tokyo)| \\ &+ |\sigma_1(Zurich) - \sigma_2(Zurich)| \\ &= |4-5| + |3-4| + |1-1| + |5-3| + |2-2| \\ &= 1+1+0+2+0 \\ &= 4 \end{aligned} \tag{2}$$

Definição 2. A distância Kemeny entre duas classificações σ_1 e σ_2 de um conjunto de items U é o número total de pares distintos (i, j) de items de U tais que

$$\sigma_1(i) < \sigma_1(j) \text{ e } \sigma_2(i) > \sigma_2(j).$$

Logo, a distância Kemeny entre σ_1 e σ_2 é o número de pares distintos de items cujas classificações relativas em σ_1 e σ_2 diferem.

Exemplo 3. Continuando com o Exemplo 2, notamos que somente os pares de cidades que possuem posições relativas diferentes em σ_1 e σ_2 são (Copenhagen, Tokyo) e (Geneva, Tokyo), já que:

$$\begin{aligned} &\sigma_1(Copenhagen) = 4 < 5 = \sigma_1(Tokyo) \text{ e} \\ &\sigma_2(Copenhagen) = 5 > 3 = \sigma_2(Tokyo) \end{aligned} \tag{3}$$

e

$$\begin{aligned} &\sigma_1(Geneva) = 3 < 5 = \sigma_1(Tokyo) \text{ e} \\ &\sigma_2(Geneva) = 4 > 3 = \sigma_2(Tokyo). \end{aligned} \tag{4}$$

Logo, $K(\sigma_1, \sigma_2) = 2$. As distâncias Kemeny e footrule entre duas classificações σ_1 e σ_2 estão relacionadas pela seguinte expressão [3].

[3] P. Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. Journal of the Royal Statistical Society, Series B. 39(2): 262–268 (1977).https://www.jstor.org/stable/2984804?seq=1#page_scan_tab_contents

Cidade	2011 σ_1	2012 σ_2
Oslo	1	1
Zurich	2	2
Geneva	3	4
Copenhagen	4	5
Tokio	5	3

Tabela 2: As cidades da Tabela 1, rearranjadas de acordo com a classificação de 2011.

$$K(\sigma_1, \sigma_2) \leq F(\sigma_1, \sigma_2) \leq 2K(\sigma_1, \sigma_2).$$

É fácil calcular a distância footrule entre duas classificações de n itens em tempo $O(n)$ - deixamos por sua conta descobrir como. Uma de suas tarefas neste trabalho é implementar esse algoritmo. A estratégia ingênua para calcular a distância Kemeny é enumerar todos os $C(n, 2) = \binom{n}{2} = n(n-1)/2$ pares de itens distintos, verificando cada um, para ver se há uma inversão. Logo, a complexidade de tempo é $O(n^2)$. Nas próximas seções, descrevemos um algoritmo $O(n\ log\ n)$, que é muito mais rápido que a abordagem ingênua. Outra das suas tarefas neste projeto é implementar este algoritmo $O(n\ log\ n)$.

2 Calculando a Distância Kemeny

Suponha que reorganizemos os itens em U, ordenando-os de acordo com um dos dois *rankings*, digamos σ_1. A tabela 2 mostra o resultado disso para as classificações da Tabela 1. Isso não tem efeito sobre a distância Kemeny (ou a distância footrule) entre os *rankings*, mas oferece um benefício: reduz o cálculo da distância Kemeny a um problema já resolvido, definido a seguir.

Definição 3. Uma inversão de uma sequência $\tau = (\tau(1), \tau(2), \ldots, \tau(n))$ de números distintos é um par de elementos tal que $\tau(i) > \tau(j)$, mas $i < j$. No problema de contagem de inversões, nos é dada uma sequência τ como acima, e somos levados a contar o número de inversões em τ. Assim, uma inversão em τ é um par de números que estão fora de ordem, um em relação ao outro,

e o número de inversões em τ mede o quão longe ela está da classificação. Se você pensar sobre isso, por alguns momentos, vai perceber que a distância Kemeny entre dois *rankings* quaisquer σ_1 e σ_2 é idêntica ao número de inversões na sequência τ, obtido rearranjando σ_2 de acordo com a ordenação dos elementos implicados pelo outro *ranking*, σ_1. Ou seja, calcular a distância Kemeny se reduz a contagem de inversões.

Exemplo 4. Aqui, e no que restar da descrição deste projeto, será útil olhar para o conjunto de items de U como uma lista de números consecutivos, correspondendo aos elementos ordenados de acordo com o primeiro *ranking*. Assim, para os dados da Tabela 2, temos: Oslo $\equiv 1$, Zurich $\equiv 2$, Geneva $\equiv 3$, Copenhagen $\equiv 4$, Tokyo $\equiv 5$. Logo,

$$\begin{aligned} \tau &= (\tau(1), \tau(2), \tau(3), \tau(4), \tau(5)) \\ &= (\sigma_2(1), \sigma_2(2), \sigma_2(3), \sigma_2(4), \sigma_2(5)) \\ &= (1, 2, 4, 5, 3). \end{aligned} \tag{5}$$

A sequência τ possui duas inversões, $4 \leftrightarrow 3$ e $5 \leftrightarrow 3$, que é precisamente a distância Kemeny entre σ_1 e σ_2. Na próxima seção, descrevemos um algoritmo $O(n\ log\ n)$ para contar inversões. Uma das suas tarefas será implementar este algoritmo, como parte da classe Ranking. Você também deverá adaptar este algoritmo para computar a distância Kemeny entre dois *rankings* em tempo $O(n\ log\ n)$.

3 Contando Inversões

O algoritmo 1 apresenta SORTandCOUNT, uma adaptação do mergesort que ordena e conta as inversões de uma sequência em tempo $O(n\ log\ n)$ [4]. Como o mergesort, SORTandCOUNT é baseado no princípio de dividir para conquistar, ou seja:

> O número de inversões de τ é igual ao número inversões na metade esquerda de τ, mais o número de inversões na metade direita de τ, mais o número de inversões entre as duas metades.

[4] O pseudo-código desta seção segue mais ou menos o Capítulo 5, Seção 3, de: J. Kleinberg and E. Tardos. Algorithm Design, Addison-Wesley, 2006. `https://github.com/haseebr/competitive-programming/blob/master/Materials/Algorithm%20Design%20by%20Jon%20Kleinberg%2C%20Eva%20Tardos.pdf`

Algorithm 1: Um algoritmo baseado no mergesort para contar inversões.

SORTandCOUNT ;
Input : Uma lista $\tau = (\tau(1), \tau(2), \ldots, \tau(n))$ de números distintos.
Output: O número de inversões em τ, juntamente com a versão ordenada de τ.
if *n=1* **then**
  retorne 0 e τ.
end
Construa duas listas

$$\tau_{left} = (\tau(1), \ldots, \tau(m)) \text{ e } \tau_{right} = (\tau(m+1), \ldots, \tau(n)),$$

onde m é a parte inteira de $n/2$.
Chame SORTandCOUNT recursivamente para obter inv_{left}, o número de inversões em τ_{left}, e τ'_{left}, a versão ordenada de τ_{left}.
Chame SORTandCOUNT recursivamente para obter inv_{right}, o número de inversões em τ_{right}, e τ'_{right}, a versão ordenada de τ_{right}.
Chame MERGEandCOUNT para contar inv_{cross}, o número de inversões envolvendo um elemento de τ_{left} e um elemento de τ_{right}, juntamente com τ', a combinação ordenada de τ'_{left} e τ'_{right}
Retorne $(inv_{left} + inv_{right} + inv_{cross})$ e τ'.

SORTandCOUNT se baseia no algoritmo MERGEandCOUNT, apresentado no Algoritmo 2, que combina duas listas ordenadas em outra lista ordenada, em tempo linear em relação ao número total de elementos, ao mesmo tempo contando as inversões entre as duas listas. Estude esses algoritmos cuidadosamente, prestando uma atenção especial ao MERGEandCOUNT, certificando-se de que entendeu porque eles estão corretos, e porque os tempos de execução são o que dissemos ser. Você pode achar útil assistir ao vídeo sobre contagem de inversões do Prof. Dan Gusfield, da UC Davis [5]. Para ajudá-lo mais um pouco, aqui está um exemplo.

Exemplo 5. Suponha $\tau = (1, 5, 3, 8, 4, 2, 7, 6)$. As inversões em τ são

$$5 \leftrightarrow 3, 5 \leftrightarrow 4, 5 \leftrightarrow 2, 3 \leftrightarrow 2, 8 \leftrightarrow 4, 8 \leftrightarrow 2, 8 \leftrightarrow 7, 8 \leftrightarrow 6, 4 \leftrightarrow 2, \text{ e } 7 \leftrightarrow 6,$$

então o número total de inversões é 10. SORTandCOUNT conta essas inversões da seguinte forma:

1. Divide τ em duas sublistas: $\tau_{left} = (1, 5, 3, 8)$ e $\tau_{right} = (4, 2, 7, 6)$.

2. Chama recursivamente

   $$(inv_{left}, \tau'_{left}) = SORTandCOUNT(\tau_{left}),$$

   para obter $inv_{left} = 1$ (correspondendo à inversão $5 \leftrightarrow 3$) e a lista ordenada $\tau'_{left} = (1, 3, 5, 8)$.

3. Chama recursivamente

   $$(inv_{right}, \tau'_{right}) = SORTandCOUNT(\tau_{right}),$$

   para obter $inv_{right} = 2$ (correspondendo às inversões $4 \leftrightarrow 2$, $7 \leftrightarrow 6$) e a lista ordenada $\tau'_{right} = (2, 4, 6, 7)$.

4. Chama

   $$(inv_{cross}, \tau') = MERGEandCOUNT(\tau'_{left}, \tau'_{right}),$$

   para obter $inv_{cross} = 7$ (correspondendo às inversões $3 \leftrightarrow 2$, $5 \leftrightarrow 2$, $5 \leftrightarrow 4$, $8 \leftrightarrow 2$, $8 \leftrightarrow 4$, $8 \leftrightarrow 6$, $8 \leftrightarrow 7$) e $\tau' = (1, 2, 3, 4, 5, 6, 7, 8)$, a combinação de τ'_{left} e τ'_{right}. A tabela 3 ilustra a execução do MERGEandCOUNT nas listas τ'_{left} e τ'_{right}.

5. Retorna $inv_{left} + inv_{right} + inv_{cross} = 10$, e τ'.

[5] `http://www.youtube.com/watch?v=Ly-ht7PN9UM`

Algorithm 2: Combinando e contando inversões entre duas sequências.

```
1  MERGEandCOUNT ;
   Input  : Duas listas ordenadas α = (α(1), α(2), ..., α(p)) e
            β = (β(1), β(2), ..., β(q)), cada uma consistindo de
            inteiros distintos, sem nenhum número em comum.
   Output: O número de pares (i, j) tais que α(i) > β(j), juntamente
            com a combinação ordenada de α e β.
2  i=1, j=1, count=0, e γ igual a lista vazia;
3  while i ≤ p e j ≤ q do
4  |  if α(i) < β(j) then
5  |  |  Concatene α(i) a γ;
6  |  |  i++;
7  |  end
8  |  if β(j) < α(i) then
9  |  |  // Encontrada uma inversão: α(i), ..., α(p) ;
10 |  |  // são todos maiores do que β(j). ;
11 |  |  Concatene β(j) a γ;
12 |  |  count = count+(p-i+1);
13 |  |  j++;
14 |  end
15 end
16 if i > p then
17 |  Concatene β(j), ..., β(q) a γ;
18 end
19 if j > q then
20 |  Concatene α(i), ..., α(p) a γ;
21 end
22 Retorne count e γ
```

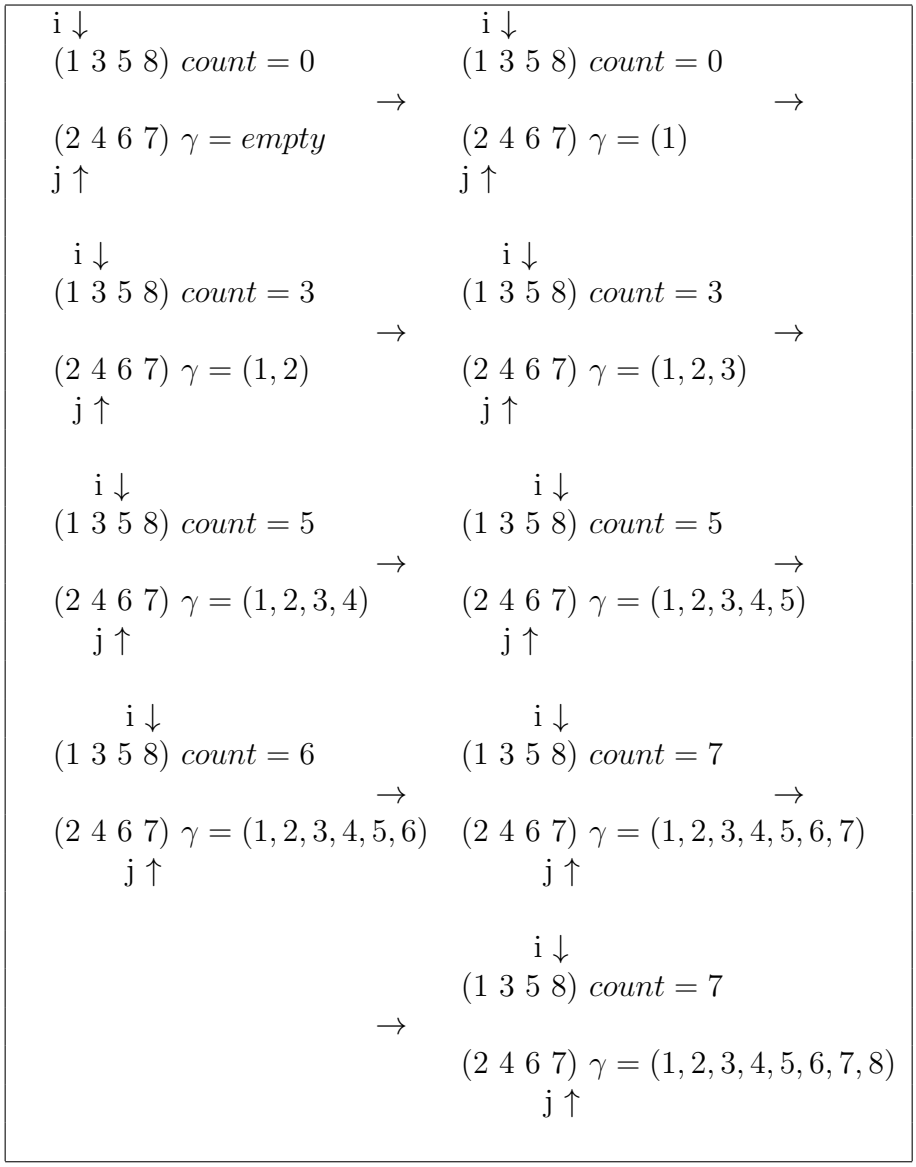

Tabela 3: Execução do MERGEandCOUNT.

4 Tarefas

Sua tarefa é implementar a classe Ranking. Um objeto Ranking representa o *ranking* de um conjunto de items numerados de 1 até n, para algum n. A classe Ranking contém vários métodos para construir *rankings*. Uma vez que um *ranking* é construído, pode-se determinar quantos itens ele classifica e obter a classificação de algum item específico. Também é possível calcular distâncias entre diversas classificações. Ranking de objetos são imutáveis; isto é, uma vez criados, eles não podem ser modificados.

4.1 Métodos Necessários

A seguir, estão especificados os métodos da classe Ranking que devem ser implementados. Como uma parte considerável dos testes será automatizada, você deve seguir exatamente essas especificações. Isso significa, entre outras coisas, que os nomes e tipos de classes e métodos devem ser mantidos como estão. É possíve, claro, definir métodos auxiliares adicionais.

def __init__(self, arg):

> "Python não permite múltiplos construtores. No entanto, eles podem ser simulados pelo tipo do argumento."

```
self.ranking = []
if type(arg) == list:
        self.initializeFromList(arg)
elif type(arg) == tuple:
        self.initializeFromTuple(arg)
elif type(arg) == int:
        self.initializeFromInt(arg)
```

def initializeFromInt(self, n):

> "Constrói um ranking aleatório de números entre 1 e n. Levanta uma exceção ValueError se $n < 1$. Deve rodar em tempo $O(n \; log \; n)$.
>
> **Nota.** Para geração de números aleatórios, use a função randint() ou choice(), do módulo random [6]. Para gerar permutações

[6] `https://docs.python.org/3/library/random.html`

aleatórias de 1 até n, use o algoritmo "shuffle", descrito na referência [7].

def initializeFromTuple(self, rank):

"Constrói um ranking σ do conjunto $U = 1, \ldots, len(rank)$, onde $\sigma(i) = rank[i-1]$. Levanta uma exceção TypeError se $rank$ for None. Levanta ValueError se $rank$ não consistir de elementos distintos entre 1 e $len(rank)$. Deve rodar em tempo $O(n\ log\ n)$, onde $n = len(rank)$."

def initializeFromList(self, scores):

"Constrói um ranking do conjunto $U = 1, \ldots, len(scores)$, onde o elemento i recebe rank k se e somente se $scores[i-1]$ for o k-ésimo maior elemento na lista $scores$. Levanta uma exceção TypeError se $scores$ for None. Levanta ValueError se $scores$ possuir valores duplicados. Deve rodar em tempo $O(n\ log\ n)$, onde $n = len(scores)$."

Exemplo 6. Suponha scores = (0.75, 0.36, 0.65, -1.5, 0.85). Então, o ranking correspondente é $\sigma = (2, 4, 3, 5, 1)$."

def getNumItems(self):

"Retorna o número de items no ranking. Deve rodar em tempo $O(1)$."

def getRank(self, i):

"Retorna o rank do item i. Levanta uma exceção ValueError caso o item i não esteja presente no ranking. Deve rodar em tempo $O(1)$."

def footrule(r_1, r_2):

"Retorna a distância footrule entre r_1 e r_2. Levanta uma exceção TypeError se r_1 ou r_2 forem None. Levanta ValueError se r_1 e r_2 tiverem comprimentos diferentes. Deve rodar em tempo $O(n)$, onde n é o número de elementos em r_1 (ou r_2)."

[7] `http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm`

def kemeny(r_1, r_2):

"Retorna a distância Kemeny entre r_1 e r_2. Levanta uma exceção TypeError se r_1 ou r_2 forem None. Levanta ValueError se r_1 e r_2 tiverem comprimentos diferentes. Deve rodar em tempo $O(n\ log\ n)$, onde n é o número de elementos em r_1 (ou r_2)."

def fDist(self, other):

"Retorna a distância footrule entre self e other. Levanta a exceção TypeError se other for None. Levanta ValueError se self e other tiverem comprimentos diferentes. Deve rodar em tempo $O(n)$, onde n é o número de elementos em self (ou other)."

def kDist(self, other):

"Retorna a distância Kemeny entre self e other. Levanta uma exceção TypeError se other for None. Levanta ValueError se self e other tiverem comprimentos diferentes. Deve rodar em tempo $O(n\ log\ n)$, onde n é o número de elementos em self (ou other).

def invCount(self):

"Retorna o número de inversões neste ranking. Deve rodar em tempo $O(n\ log\ n)$, onde n é o número de elementos em self.

Nota. Como objetos Ranking são imutáveis, de fato, pode-se computar o número de inversões em um ranking apenas uma vez, no momento da criação, e armazená-lo para acessar mais tarde. Nesta implementação, invCount deve rodar em tempo $O(1)$. Você está livre para implementar esta versão ou aquela que computa inversões toda vez que o método for invocado; a documentação deve indicar claramente qual abordagem foi usada."

A representação precisa de objetos Ranking é deixada por sua conta, por exemplo, usando listas. Seu código deve incluir implementações dos algoritmos SORTandCOUNT e MERGEandCOUNT e deve usar essas implementações para contar inversões e calcular a distância Kemeny.

5 Submissão

A submissão deste projeto consiste de duas partes.

Parte1: Envie um arquivo contendo classes de teste, a serem analisadas pelo pacote unittest [8], contendo vários casos de teste para todos os métodos na classe Ranking. Seu código deve:

(a) testar os métodos, levantando exceções sob circunstâncias apropriadas (por exemplo, argumentos nulos),

(b) verificar que o construtor produz classificações válidas (ou seja, permutações de 1 a n) para os três tipos de argumento,

(c) testar casos limite, incluindo o tratamento adequado de *rankings* com um único elemento, contagem de inversões em sequências ascendentes e descendentes, distâncias entre *rankings* idênticos, *rankings* que são inversões entre si, etc.,

(d) executar quaisquer testes adicionais que julgar necessários.

Parte 2: Entregue um arquivo comprimido .tar.gz ou .zip chamado Primeiro_nome.Último_nome.PIG.zip contendo todo o seu código-fonte para a classe Ranking. No arquivo comprimido, todos os arquivos devem estar dentro de um diretório chamado AD1. O nome do arquivo para o projeto deve ser Ranking.py. Inclua o tag do Doxygen @author em cada arquivo. Não inclua arquivos .pyc.

Normalmente, a coisa mais simples a fazer é clicar com o botão direito do mouse no diretório fonte do seu projeto e selecionar "Send To → Compressed/zipped file". Após criar o aquivo zip, verifique-o cuidadosamente. Extraia os arquivos em um diretório temporário vazio, e olhe-os antes de submetê-los. Eles são arquivos .py? Todos os arquivos necessários estão presentes no arquivo comprimido? Estão nos diretórios corretos? Fazem parte da última versão que funciona do seu código?

Todas as submissões devem ser feitas através da plataforma. Por favor, siga as orientações postadas na plataforma.

[8] `https://docs.python.org/3/library/unittest.html`

6 Notas

Sua nota para a Parte 1 será determinada principalmente pelo rigor com que seus testes do unittest verifiquem os métodos necessários. Também podemos executar seus testes em algumas versões corretas e incorretas das várias classes. As versões corretas devem passar pelos testes e as versões incorretas devem falhar, apropriadamente. Uma parte significativa de sua nota para a Parte 2 será baseada na execução de nossos testes em suas classes. Também inspecionaremos seu código para verificar se seus métodos atendem aos requisitos estipulados quanto a complexidade de tempo. Documentação e estilo contarão aproximadamente 15% do total de pontos.

Na AD1, os testes podem ser relativamente simples. Já na AD2, será cobrado um conjunto completo de testes, mais a confecção de uma interface gráfica para o programa.