Instituto Superior Técnico — LMAC/MEBiom

V1

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Existe um número arbitrário de **servidores**, identificados sequencialmente $(0, 1, 2, \dots)$, colocados inicialmente no plano nos seus pontos base. Esta informação (número de servidores e suas posições iniciais) é dada por uma lista como, por exemplo, `base=[(0,0),(10,10),(5,5)]`, que no caso identifica três servidores (o servidor 0 inicialmente colocado em $(0, 0)$, o servidor 1 inicialmente colocado em $(10, 10)$, e o servidor 2 inicialmente colocado em $(5, 5)$).

Há também uma lista de **pedidos** (pontos no plano), que devem ser atendidos sequencialmente. Por exemplo, `requests=[(1,1),(10,9),(4,5),(10,0)]`, sendo que cada pedido deve ser atendido deslocando para o ponto pedido um dos servidores.

Finalmente, há uma lista de **alocação** dos servidores, que determina qual o servidor que deve atender cada pedido. Por exemplo, `allocs=[0,1,2,1]`.

Neste caso, ao atender o primeiro pedido, o servidor 0 desloca-se da sua base $(0, 0)$ para o ponto $(1, 1)$. Depois, ao atender o segundo pedido, o servidor 1 desloca-se da sua base $(10, 10)$ para o ponto $(10, 9)$. De seguida, ao atender o terceiro pedido, o servidor 2 desloca-se da sua base $(5, 5)$ para o ponto $(4, 5)$. Finalmente, ao atender o quarto e último pedido, o servidor 1 desloca-se da sua posição actual $(10, 9)$ para o ponto $(10, 0)$.

Considera-se a habitual distância Euclideana no plano, ou seja, $\text{dist}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, que pode ser facilmente calculada em *Pyhton* pela seguinte definição, que pode usar livremente.

```
from math import sqrt
def dist(p,q):
    return sqrt((p[0]-q[0])**2+(p[1]-q[1])**2)
```

Deste modo o servidor 0 percorre uma distância $\text{dist}((0, 0), (1, 1)) = \sqrt{2}$, o servidor 1 percorre uma distância total $\text{dist}((10, 10), (10, 9)) + \text{dist}((10, 9), (10, 0)) = 1 + 9 = 10$, e o servidor 2 percorre uma distância $\text{dist}((5, 5), (4, 5)) = 1$.

Pretende-se que os vários servidores percorram distâncias o mais aproximadas possível, pelo que o custo final das alocações é a diferença máxima entre as distâncias percorridas por eles, neste caso $10 - 1 = 9$.

(**a**) Defina recursivamente em *Python* uma função `maxmindif` que dada uma lista não vazia de números calcula a diferença entre o máximo e o mínimo da lista.

Por exemplo, `maxmindif([1.41,10,1])` deverá ser `9`.

(**b**) Defina recursivamente em *Python* uma função `calcdist` que dada uma lista base dos servidores, uma lista de pedidos, uma lista de alocações, e a identificação de um servidor, calcula a distância total percorrida por esse servidor face à situação inicial dada pela base, bem como pelas listas de pedidos e alocações.

Por exemplo, `calcdist(base,requests,allocs,1)` deverá ser `10`.

(**c**) Defina recursivamente em *Python* uma função `calcalldist` que dada uma lista base dos servidores, uma lista de pedidos, e uma lista de alocações, calcula a lista das distâncias totais percorridas por cada um dos servidores; e use-a para definir uma função `cost`, com os mesmos argumentos, que calcula o custo total das alocações.

Por exemplo, `calcalldist(base,requests,allocs)` deverá ser `[1.41,10,1]`, e portanto `cost(base,requests,allocs)` deverá ser `9`.

V2

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Existe um número arbitrário de **servidores**, identificados sequencialmente $(0, 1, 2, \dots)$, colocados inicialmente numa linha nos seus pontos base. Esta informação (número de servidores e suas posições iniciais) é dada por uma lista como, por exemplo, `base=[0,-5,10]`, que no caso identifica três servidores (o servidor 0 inicialmente colocado em 0, o servidor 1 inicialmente colocado em -5, e o servidor 2 inicialmente colocado em 10).

Há também uma lista de **pedidos** (pontos na linha), que devem ser atendidos sequencialmente. Por exemplo, `requests=[-7,0,3,20]`, sendo que cada pedido deve ser atendido deslocando para o ponto pedido, ao longo da linha, um dos servidores.

Finalmente, há uma lista de **alocação** dos servidores, que determina qual o servidor que deve atender cada pedido. Por exemplo, `allocs=[1,0,2,2]`.

Neste caso, ao atender o primeiro pedido, o servidor 1 desloca-se da sua base -5 para o ponto -7. Depois, ao atender o segundo pedido, o servidor 0 desloca-se da sua base 0 para o ponto 0 (ou seja, não necessita de se mover). De seguida, ao atender o terceiro pedido, o servidor 2 desloca-se da sua base 10 para o ponto 3. Finalmente, ao atender o quarto e último pedido, o servidor 2 desloca-se da sua posição actual 3 para o ponto 20.

Considera-se a habitual distância na linha, ou seja, $\text{dist}(x, y) = |x - y|$, que pode ser facilmente calculada em *Pyhton* pela seguinte definição, que pode usar livremente.

```
def dist(p,q):
    return abs(p-q)
```

Deste modo o servidor 0 percorre uma distância $\text{dist}(0, 0) = 0$, o servidor 1 percorre uma distância total $\text{dist}(-5, -7) = 2$, e o servidor 2 percorre uma distância $\text{dist}(10, 3) + \text{dist}(3, 20) = 7 + 17 = 24$.

Pretende-se calcular o custo total das alocações, que é a soma das distâncias totais percorridas pelos vários servidores. No exemplo, tem-se custo $0 + 2 + 24 = 26$.

(**a**) Defina recursivamente em *Python* uma função `calcdist` que dada uma lista base dos servidores, uma lista de pedidos, e uma lista de alocações, calcula o custo total.

Por exemplo, `calcdist(base,requests,allocs)` deverá ser `26`.

(**b**) Defina recursivamente em *Python* uma função `posmin` que dada uma lista não vazia de números calcula (um)a posição na lista do valor seu mínimo.

Por exemplo, `posmin([99,5,74,10])` deverá ser `1`.

(**c**) Defina recursivamente em *Python* uma função `custos` que dada uma lista base dos servidores, uma lista de pedidos, e uma lista de listas de alocações, calcula a lista dos custos correspondentes a cada uma das listas de alocações; e use-a para definir uma função `best`, com os mesmos argumentos, que calcula (um)a alocação com menor custo na lista dada.

Por exemplo, `custos(base,requests,[allocs,[1,0,0,2]])` deverá ser `[26,15]`, e portanto `best(base,requests,[allocs,[1,0,0,2]])` deverá ser `[1,0,0,2]`.

V3

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideramos tarefas da forma `(d,p)` onde `d` é um inteiro que indica o prazo máximo em que a tarefa deve ser executada, e `p` é a penalização a pagar se a tarefa não for executada dentro do prazo.

Uma *calendarização* é uma lista de tarefas que estabelece o instante em que cada tarefa é executada. Por exemplo, com a calendarização

```
calend=[(2,30),(1,10),(1,100),(2,5)]
```

está-se a estipular que a tarefa `(2,30)` é executada no instante 0 (o que cumpre o prazo e não paga penalidade), depois a tarefa `(1,10)` é executada no instante 1 (o que também cumpre o prazo e não paga penalidade), de seguida a tarefa `(1,100)` é executada no instante 2 (o que não cumpre o prazo e paga 100 de penalização), e finalmente a tarefa `(2,5)` é executada no instante 5 (o que também não cumpre o prazo e paga 5 de penalização).

A penalização total associada à calendarização é portanto 0+0+100+5=105.

(**a**) Defina recursivamente em *Python* uma função `penal` que dada uma lista de calendarização de tarefas calcula a penalização total associada.

Por exemplo, `penal(calend)` deverá ser `105`.

(**b**) Defina recursivamente em *Python* uma função `melhor` que dada uma lista de listas de calendarização, calcula (um)a calendarização nessa lista com menor penalização total.

Por exemplo, `melhor([calend,[(1,10),(1,100),(2,30),(2,5)]])` deverá ser a calendarização `[(1,10),(1,100),(2,30),(2,5)]` (pois a penalização total que lhe está associada `penal([(1,10),(1,100),(2,30),(2,5)])` é 5).

(**c**) Defina recursivamente em *Python* uma função `atrasaprim` que dada uma lista de calendarização calcula a lista de todas as listas de calendarização que resultam da lista dada atrasando a primeira tarefa de todas as maneiras possíveis mas mantendo a ordem relativa das restantes tarefas; e use-a para definir uma função Booleana `atraso` que dada uma lista de calendarização verifica se a calendarização dada pode ser melhorada atrasando a primeira tarefa.

Por exemplo, `atrasaprim(calend)` resulta na lista `[[(1,10),(2,30),(1,100),(2,5)],[(1,10),(1,100),(2,30),(2,5)],[(1,10),(1,100),(2,5),(2,30)]]`, e portanto o resultado de `atraso(calend)` deverá ser `True`.

V4

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideramos tarefas da forma `d` onde `d` é um inteiro que indica o prazo máximo em que a tarefa deve ser executada, considerando-se uma falha caso a tarefa seja executada fora do prazo.

Uma *calendarização* é uma lista de tarefas que estabelece o instante em que cada tarefa é executada. Por exemplo, com a calendarização

```
calend=[1,2,1,0]
```

está a estipular-se que a tarefa `1` é executada no instante 0 (o que cumpre o prazo), depois a tarefa `2` é executada no instante 1 (o que também cumpre o prazo), de seguida a tarefa `1` é executada no instante 2 (o que não cumpre o prazo e corresponde a uma falha), e finalmente a tarefa `0` é executada no instante 3 (o que não cumpre o prazo e corresponde a outra falha).

O número total de falhas associado à calendarização é portanto 2.

(**a**) Defina recursivamente em *Python* uma função `falhas` que dada uma lista de calendarização de tarefas calcula o número total de falhas associado.

Por exemplo, `falhas(calend)` deverá ser `2`.

(**b**) Defina recursivamente em *Python* uma função `menosfalhas` que dada uma lista de listas de calendarização, calcula (um)a calendarização nessa lista com número mínimo de falhas.

Por exemplo, `menosfalhas([calend,[0,1,2,1]])` deverá ser a calendarização `[0,1,2,1]` (pois o número de falhas associado `falhas([0,1,2,1])` é 1).

(**c**) Defina recursivamente em *Python* uma função `antult` que dada uma lista de calendarização calcula a lista de todas as listas de calendarização que resultam da lista dada antecipando a última tarefa de todas as maneiras possíveis mas mantendo a ordem relativa das restantes tarefas; e use-a para definir uma função Booleana `antecip` que dada uma lista de calendarização verifica se a calendarização dada pode ser melhorada antecipando a última tarefa.

Por exemplo, `antult(calend)` resulta na lista `[[0,1,2,1],[1,0,2,1],[1,2,0,1]]`, e portanto o resultado de `antecip(calend)` deverá ser `True`.

Instituto Superior Técnico — LMAC/MEBiom

V5

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideramos configurações de memória dadas por listas de valores (sem repetições). Por exemplo,

`mem=[23,5,14,77]`.

O custo de aceder a cada valor numa dada configuração da memória é a posição que ocupa. Assim, no exemplo, aceder ao valor 23 tem custo 0, aceder ao valor 5 tem custo 1, aceder ao valor 14 tem custo 2, e aceder ao valor 77 tem custo 3.

Uma lista de *pedidos* é uma lista (com possíveis repetições) de valores em memória, que devem ser acedidos. O custo total de aceder aos pedidos é a soma dos custos associados ao acesso a cada um dos valores. Assim, por exemplo, para a lista de pedidos

`req=[5,14,5,5,77,23]`.

tem-se um custo total igual a 1+2+1+1+3+0=8.

(**a**) Defina recursivamente em *Python* uma função `custo` que dada uma configuração de memória, e uma lista de pedidos, calcula o seu custo total.

Por exemplo, `custo(mem,req)` deverá ser `8`.

(**b**) Defina recursivamente em *Python* uma função `melhormem` que dada uma lista de configurações de memória, e uma lista de pedidos, calcula (um)a configuração de memória na lista dada que minimiza o custo total dos acessos.

Por exemplo, `melhormem([mem,[77,14,5,23]],req)` deverá ser a lista de memória `mem` (pois o custo total `custo([77,14,5,23],req)` é 10).

(**c**) Defina recursivamente em *Python* uma função `inv` que dada uma lista calcula a lista invertida; e use-a para definir uma função Booleana `inverter` que dada uma configuração de memória, e uma lista de pedidos, determina se é vantajoso inverter a configuração de memória.

Por exemplo, `inv(mem)` resulta na lista `[77,14,5,23]`, e portanto `inverter(mem,req)` deverá ser `False`.

V6

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideramos configurações de memória dadas por listas de valores (sem repetições). Por exemplo,

`mem=[16,8,11,20]`.

O custo de aceder a cada valor numa dada configuração da memória é a posição que ocupa. Assim, no exemplo, aceder ao valor 16 tem custo 0, aceder ao valor 8 tem custo 1, aceder ao valor 11 tem custo 2, e aceder ao valor 20 tem custo 3.

Uma lista de *pedidos* é uma lista (com possíveis repetições) de valores em memória, que devem ser acedidos. Por exemplo,

`req=[11,8,11,11,20,16]`.

Dada uma configuração de memória, associa-se a cada lista de pedidos a lista dos custos correspondentes aos acessos. No caso, ter-se-ia

`custos=[2,1,2,2,3,0]`.

O custo total dos acessos é a soma dos valores na lista de custos, neste caso 2+1+2+2+3+0=10.

(**a**) Defina recursivamente em *Python* uma função `custotal` que dada uma lista de custos, calcula o seu custo total.

Por exemplo, `custotal([2,1,2,2,3,0])` deverá ser `10`.

(**b**) Defina recursivamente em *Python* uma função `melhor` que dada uma lista de listas de custos, calcula (um)a lista de custos na lista dada com menor custo total dos acessos.

Por exemplo, `melhor([custos,[1,2,1,1,3,0]])` deverá ser a lista de custos `[1,2,1,1,3,0]` (pois o custo total `custotal([1,2,1,1,3,0])` é 8).

(**c**) Defina recursivamente em *Python* uma função `trocas` que dada uma lista e dois valores `x,y` calcula a lista que resulta de trocar as ocorrências de `x` por `y` e as ocorrências de `y` por `x`; e use-a para definir uma função Booleana `valetrocar` que dada uma lista de custos e dois valores `x,y` determina se seria vantajoso trocar as ocorrências de `x` e de `y` na lista de custos.

Por exemplo, `trocas(custos,1,2)` resulta na lista `[1,2,1,1,3,0]`, e portanto o resultado de `valetrocar(custos,1,2)` deverá ser `True`.

V7

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideram-se listas de valores numéricos. Em muitas situações é importante que trabalhemos com listas ordenadas (por ordem crescente, em sentido lato). Quando numa dada lista o valor numa certa posição é maior que o valor seguinte dizemos que se trata de uma inversão.

Por exemplo, a lista `w=[1,2,3,4,3,5,5,0]` tem 2 inversões (4>3 e 5>0).

Obviamente, uma lista ordenada não tem inversões. Por essa razão dizemos que uma lista está melhor ordenada que outra se tiver um menor número de inversões.

(**a**) Defina recursivamente em *Python* uma função `invs` que dada uma lista de números, calcula o seu número de inversões.

Por exemplo, `invs(w)` deverá ser `2`.

(**b**) Defina recursivamente em *Python* uma função `melhorord` que dada uma lista de listas, calcula (um)a lista (de entre as listas dadas) que está melhor ordenada.

Por exemplo, `melhorord([w,[1,2,3,3,4,5,5,0]])` deverá ser a lista `[1,2,3,3,4,5,5,0]` (pois o seu número de inversões `invs([1,2,3,3,4,5,5,0])` é 1).

(**c**) Defina recursivamente em *Python* uma função `trocas` que dada uma lista calcula a lista de todas as listas que resultam de trocar a posição de 2 elementos adjacentes; e use-a para definir uma função Booleana `valetrocar` que dada uma lista de valores determina se por troca de 2 elementos adjacentes seria possível melhorar a ordenação da lista dada.

Por exemplo, `trocas([1,2,3,4])` resulta na lista `[[2,1,3,4],[1,3,2,4],[1,2,4,3]]`, e `valetrocar(w)` deverá ser `True`.

V8

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

Outubro 2020 Mini-teste 1 Duração: 30m

Nesta avaliação não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Consideram-se listas de valores numéricos. Em muitas situações é importante que trabalhemos com listas ordenadas (por ordem crescente, em sentido lato). Qualquer lista, mesmo que não ordenada, pode ser decomposta em segmentos ordenados.

Por exemplo, à lista `w=[1,2,3,4,3,5,5,0]` corresponde a lista (de listas) de segmentos ordenados `[[1,2,3,4],[3,5,5],[0]]`.

Dizemos que a taxa de ordenação de uma lista é o comprimento do seu maior segmento ordenado, a dividir pelo comprimento total.

Obviamente, uma lista ordenada tem apenas um segmento e a sua taxa de ordenação é 1. A taxa de ordenação da lista `w` é 0.5.

(**a**) Defina recursivamente em *Python* uma função `segs` que dada uma lista de números, calcula a sua lista de segmentos.

Por exemplo, `segs(w)` deverá ser `[[1,2,3,4],[3,5,5],[0]]`.

(**b**) Defina recursivamente em *Python* uma função `maxlen` que dada uma lista de listas calcula o comprimento da maior das listas (na lista dada); e use-a para definir uma função `taxa` que dada uma lista de valores calcula a sua taxa de ordenação.

Por exemplo, `maxlen([[1,2,3,4],[3,5,5],[0]])` deve ser 4, e portanto `taxa(w)` deverá ser `0.5`.

(**c**) Defina recursivamente em *Python* uma função `melhorord` que dada uma lista de listas, calcula (um)a lista (de entre as listas dadas) com maior taxa de ordenação.

Por exemplo, `melhorord([w,[1,2,3,3,4,5,5,0]])` deverá ser a lista `[1,2,3,3,4,5,5,0]` (pois os seus segmentos `segs([1,2,3,3,4,5,5,0])` são `[[1,2,3,3,4,5,5],[0]]`, e portanto a sua taxa de ordenação `taxa([1,2,3,3,4,5,5,0])` é 0.875).