Rep1A

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do 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`), concatenação (+), e replicação (lista*num).

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, `b=[(0,0),(0,5),(3,3)]`, que no caso identifica três servidores (o servidor 0 inicialmente colocado em $(0, 0)$, o servidor 1 inicialmente colocado em $(0, 5)$, e o servidor 2 inicialmente colocado em $(3, 3)$).

Há também uma lista de **pedidos** (pontos no plano), que devem ser atendidos sequencialmente. Por exemplo, `r=[(0,3),(0,2),(1,3),(5,3)]`, sendo que cada pedido deve ser atendido deslocando para o ponto pedido um dos servidores.

Uma lista de **alocação** dos servidores determina qual o servidor que deve atender cada pedido, por exemplo `a=[1,0,2,2]`. Neste caso, ao atender o primeiro pedido, o servidor 1 desloca-se da sua base $(0, 5)$ para o ponto $(0, 3)$. Depois, ao atender o segundo pedido, o servidor 0 desloca-se da sua base $(0, 0)$ para o ponto $(0, 2)$. De seguida, ao atender o terceiro pedido, o servidor 2 desloca-se da sua base $(3, 3)$ para o ponto $(1, 3)$. Finalmente, ao atender o quarto e último pedido, o servidor 2 desloca-se da sua posição actual $(1, 3)$ para o ponto $(5, 3)$.

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 *Python* pela definição abaixo.

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

No exemplo dado, o servidor 0 percorre uma distância $\text{dist}((0, 0), (0, 2)) = 2$, o servidor 1 percorre uma distância $\text{dist}((0, 5), (0, 3)) = 2$, e o servidor 2 percorre uma distância $\text{dist}((3, 3), (1, 3)) + \text{dist}((1, 3), (5, 3)) = 2 + 4 = 6$. A distância total (somada) percorrida pelos servidores é portanto $2 + 2 + 6 = 10$.

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

Por exemplo, `totdist(b,r,a)` deverá ser `10`.

(**b**) Defina recursivamente em *Python* uma função `closer` que dada uma lista base dos servidores, e um ponto pedido, devolve o servidor (ou um dos servidores) cuja distância ao ponto pedido seja mínima.

Por exemplo, `closer(b,(0,3))` deverá ser `1`.

(**c**) Uma **estratégia** é um algoritmo que dada uma lista base dos servidores e uma lista de pedidos, calcula uma lista de **alocação** dos servidores.

Use a função `closer` para definir recursivamente em *Python* uma função `allocbybase` que implemente a estratégia que aloca a cada pedido o servidor (ou um dos servidores) cuja posição base inicial lhe esteja mais próxima.

Por exemplo, `allocbybase(b,r)` deverá ser `a`.

Rep1B

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do 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`), concatenação (+), e replicação (lista*num).

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, `b=[(0,0),(0,5),(3,3)]`, que no caso identifica três servidores (o servidor 0 inicialmente colocado em $(0, 0)$, o servidor 1 inicialmente colocado em $(0, 5)$, e o servidor 2 inicialmente colocado em $(3, 3)$).

Há também uma lista de **pedidos** (pontos no plano), que devem ser atendidos sequencialmente. Por exemplo, `r=[(0,3),(0,2),(0,-1),(5,3)]`, sendo que cada pedido deve ser atendido deslocando para o ponto pedido um dos servidores.

Uma lista de **alocação** dos servidores determina qual o servidor que deve atender cada pedido, por exemplo `a=[1,1,0,2]`. Neste caso, ao atender o primeiro pedido, o servidor 1 desloca-se da sua base $(0, 5)$ para o ponto $(0, 3)$. Depois, ao atender o segundo pedido, o servidor 1 desloca-se da sua posição actual $(0, 3)$ para o ponto $(0, 2)$. De seguida, ao atender o terceiro pedido, o servidor 0 desloca-se da sua base $(0, 0)$ para o ponto $(0, -1)$. Finalmente, ao atender o quarto e último pedido, o servidor 2 desloca-se da sua base $(3, 3)$ para o ponto $(5, 3)$.

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 *Python* pela definição abaixo.

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

No exemplo dado, o primeiro ponto pedido $(0, 3)$ está à distância $\text{dist}((0, 0), (0, 3)) = 3$ da base do servidor 0, à distância $\text{dist}((0, 5), (0, 3)) = 2$ da base do servidor 1, e à distância $\text{dist}((3, 3), (0, 3)) = 3$ da base do servidor 2.

(**a**) Defina recursivamente em *Python* uma função `closest` que dada uma lista base dos servidores, e um ponto pedido, devolve a lista dos servidores cuja distância ao ponto pedido seja mínima.

Por exemplo, `closest(b,(0,2.5))` deverá ser `[0,1]`.

(**b**) Uma **função de custo** é um algoritmo que dada uma lista base dos servidores, uma lista de pedidos, e uma lista de alocação, calcula o custo (não negativo) associado às alocações, ponderando de alguma forma os deslocamentos dos servidores.

Defina recursivamente em *Python* a função de custo `maxdist2base` que devolve a distância máxima a que algum servidor vai estar da sua posição base inicial.

Por exemplo, `maxdist2base(b,r,a)` deverá ser `3` (que corresponde à distância a que o servidor 1 está da sua base `(0,5)` quando se desloca para satisfazer o pedido `(0,2)`).

(**c**) Use a função `closest` para definir recursivamente em *Python* uma função `allocbydist` que dada uma lista base dos servidores e uma lista de pedidos, calcula uma lista de alocação dos servidores que aloca a cada pedido o servidor (ou um dos servidores) cuja posição nesse momento lhe seja mais próxima.

Por exemplo, `allocbybase(b,r)` deverá ser `a`.

Rep2A

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 2 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`), concatenação (+), e replicação (lista*num).

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 `c=[(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 penalização), depois a tarefa `(1,10)` é executada no instante 1 (o que também cumpre o prazo e não paga penalização), 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 3 (o que também não cumpre o prazo e paga 5 de penalização). A penalização total associada à calendarização `calend` é portanto 0+0+100+5=105. Há, no entanto, calendarizações óptimas para as mesmas tarefas, nomeadamente `b=[(1, 10), (1, 100), (2, 30), (2, 5)]` cujo custo total associado é apenas 0+0+0+5=5.

Dada uma lista de tarefas, para obter uma calendarização óptima, em todos os casos, é suficiente fazer o seguinte: começar por considerar uma calendarização parcial em que todos os instantes estão ainda por alocar e depois, sucessivamente, tomar a tarefa ainda não alocada a que corresponda a maior penalização, e alocá-la no instante mais tardio possível dentro do seu prazo, ou caso isso não seja possível, alocá-la no instante mais tardio disponível. Considerando a lista de tarefas `c`, isto corresponde a começar com a calendarização vazia `["free","free","free","free"]`; alocando-se primeiro a tarefa `(1,100)` na posição 1, ficando `["free",(1,100),"free","free"]`; de seguida a tarefa `(2,30)` na posição 2, ficando `["free",(1,100),(2,30),"free"]`; de seguida a tarefa `(1,10)` na posição 0, ficando a calendarização `[(1,10),(1,100),(2,30),"free"]`; e finalmente a tarefa `(2,5)` na posição 3, obtendo `b`.

(**a**) Defina imperativamente em *Python* uma função `worse` que dada uma lista de tarefas a calendarizar, calcula a posição na lista de tarefas da tarefa com a maior componente `p` de penalização (ou a posição de uma delas, caso haja várias).

Por exemplo, `worse(c)` deverá ser `2` (que corresponde à posição da tarefa `(1,100)`).

(**b**) Defina imperativamente em *Python* uma função `where` que dada uma calendarização parcial de tarefas, e uma posição `i`, determina a posição livre mais tardia na calendarização dada, sem exceder `i`. Caso todas as posições até `i` estejam já ocupadas o resultado deve ser `-1`.

Por exemplo, `where(["free",(1,100),"free","free"],2)` deverá ser `2`, assim como `where([(1, 10), (1, 100), (2, 30), "free"],2)` deverá ser `-1`.

(**c**) Usando as funções desenvolvidas nas alíneas anteriores, defina imperativamente em *Python* uma função `docal` que dada uma lista de tarefas a calendarizar, calcula a calendarização óptima obtida pelo método indicado.

Por exemplo, `docal(c)` resulta na calendarização `b`.

Rep2B

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 2 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`), concatenação (+), e replicação (lista*num).

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 `c=[(2,1),(2,10),(1,50),(1,30)]` está-se a estipular que a tarefa `(2,1)` é executada no instante 0 (o que cumpre o prazo e não paga penalização), depois a tarefa `(2,10)` é executada no instante 1 (o que também cumpre o prazo e não paga penalização), de seguida a tarefa `(1,50)` é executada no instante 2 (o que não cumpre o prazo e paga 50 de penalização), e finalmente a tarefa `(1,30)` é executada no instante 3 (o que também não cumpre o prazo e paga 30 de penalização). A penalização total associada à calendarização `calend` é portanto 0+0+50+30=80. Há, no entanto, calendarizações óptimas para as mesmas tarefas, nomeadamente `b=[(1, 30), (1, 50), (2, 10), (2, 1)]` cujo custo total associado é apenas 0+0+0+1=1.

Dada uma lista de tarefas, para obter uma calendarização óptima, em todos os casos, é suficiente fazer o seguinte: começar por considerar uma calendarização parcial em que todos os instantes estão ainda por alocar e depois, sucessivamente, tomar a tarefa ainda não alocada a que corresponda a maior penalização, e alocá-la no instante mais tardio possível dentro do seu prazo, ou caso isso não seja possível, alocá-la no instante mais tardio disponível. Considerando a lista de tarefas `c`, isto corresponde a começar com a calendarização vazia `[None,None,None,None]`; alocando-se primeiro a tarefa `(1,50)` na posição 1, ficando `[None,(1,50),None,None]`; de seguida a tarefa `(1,30)` na posição 0, ficando `[(1,30),(1,50),None,None]`; de seguida a tarefa `(2,10)` na posição 2, ficando a calendarização `[(1,30),(1,50),(2,10),None]`; e finalmente a tarefa `(2,1)` na posição 3, obtendo `b`.

(**a**) Defina imperativamente em *Python* uma função `latefreeupto` que dada uma calendarização parcial de tarefas, e uma posição `i`, determina a posição livre mais tardia na calendarização dada, sem exceder `i`. Caso todas as posições até `i` estejam já ocupadas o resultado deve ser `-1`.

Por exemplo, `latefreeupto([None,(1,50),None,None],1)` deverá ser `0`, assim como `latefreeupto([(1,30),(1,50),(2,10),None],2)` deverá ser `-1`.

(**b**) Defina imperativamente em *Python* uma função `highpen` que dada uma lista de tarefas a calendarizar, e uma lista de Booleanos do mesmo comprimento (em que `True` na posição `i` indica que a tarefa na mesma posição já foi calendarizada, sendo `False` no caso contrário), calcula a posição na lista de tarefas da tarefa ainda não calendarizada com maior penalização (ou a posição de uma delas, caso haja várias).

Por exemplo, `highpen(c,[False,False,True,False]` deverá ser `3` (que corresponde à posição da tarefa `(1,30)`, pois a tarefa `(1,50)` estará já calendarizada).

(**c**) Usando as funções desenvolvidas nas alíneas anteriores, defina imperativamente em *Python* uma função `calendarize` que dada uma lista de tarefas a calendarizar, calcula a calendarização óptima obtida pelo método indicado.

Por exemplo, `calendarize(c)` resulta na calendarização `b`.

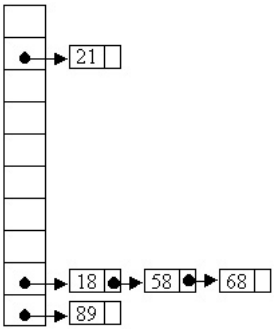
Instituto Superior Técnico — LMAC/MEBiom

Rep3A

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 3 Duração: 30m

Considere *tabelas de dispersão (hash tables)*, estruturas para armazenamento e pesquisa de valores, com dimensão predeterminada `s`, onde valores `x` (sem repetição de valores) são armazenados de acordo com índices `h(x,s)` (necessariamente um valor natural em `range(s)`) que lhes são associados por uma dada *função de dispersão (hash function)*.



Na ilustração, exemplifica-se uma tabela de dispersão de dimensão 10, portanto com índices de 0 a 9 (de cima para baixo na figura), em que estão armazenados os valores 21 (no índice 1), 18, 58, 68 (todos no índice 8), e 89 (no índice 9). Assume-se que os valores são todos inteiros e que à função de dispersão corresponde a seguinte implementação em *Python*.

```
def h(x,s):
    return x%s
```

Identificaram-se as seguintes operações:

- `empty(s)`: tabela de dimensão `s` sem qualquer valor armazenado;
- `put(vals,tab)`: tabela que resulta de armazenar na tabela `tab` todos os valores da lista `vals`;
- `rem(x,tab)`: tabela que resulta de remover da tabela `tab` o valor `x` (se existir);
- `insideQ(x,tab)`: `True` se o valor `x` está armazenado na tabela `tab`, `False` caso contrário;
- `size(tab)`: dimensão da tabela `tab`;
- `get(tab,i)`: lista dos valores armazenados no índice `i` da tabela `tab`;
- `double(tab)`: tabela que armazena os mesmos valores que a tabela `tab` mas cuja dimensão é o dobro da dimensão da tabela `tab`.

Nomeadamente, a expressão `rem(3,put([89,18,21,3,58,21,68],empty(10)))` produziria a tabela da ilustração.

Em *Python*, pretende-se representar uma tabela de dispersão como um par da forma $(\texttt{n}, [\texttt{w}_0, \texttt{w}_1, \ldots, \texttt{w}_{\texttt{s}-1}])$ onde `n` é o número de valores distintos armazenados na tabela, `s` é a dimensão da tabela e cada $\texttt{w}_\texttt{i}$ é a lista dos valores armazenados no índice `i` da tabela. No caso exemplificado, ter-se-ia `(5,[[],[21],[],[],[],[],[],[],[18,58,68],[89]])`.

Apresente implementações eficientes para todas as operações.

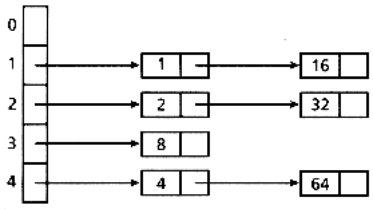
Rep3B

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 3 Duração: 30m

Considere *tabelas de dispersão (hash tables)*, estruturas para armazenamento e pesquisa de valores, com dimensão predeterminada `s`, onde valores `x` (sem repetição de valores) são armazenados de acordo com índices `h(x,s)` (necessariamente um valor natural em `range(s)`) que lhes são associados por uma dada *função de dispersão (hash function)*.



Na ilustração, exemplifica-se uma tabela de dispersão de dimensão 5, portanto com índices de 0 a 4, em que estão armazenados os valores 1 e 16 (ambos no índice 1), 2 e 32 (ambos no índice 2), 8 (no índice 3), 4 e 64 (no índice 4).

Assume-se que os valores são todos inteiros e que à função de dispersão corresponde a seguinte implementação em *Python*.

```
def h(x,s):
    return x%s
```

Identificaram-se as seguintes operações:

- `make(vals,s)`: tabela de dimensão `s` que resulta de armazenar todos os valores da lista `vals`;
- `add(x,tab)`: tabela que resulta de armazenar na tabela `tab` o valor `x`;
- `take(vals,tab)`: tabela que resulta de remover na tabela `tab` todos os valores (que lá existam) da lista `vals`;
- `numelems(tab)`: número de valores armazenados na tabela `tab`;
- `size(tab)`: dimensão da tabela `tab`;
- `getval(tab)`: devolve um dos valores armazenados na tabela `tab` (que se assume não vazia);
- `existsQ(x,tab)`: `True` se o valor `x` está armazenado na tabela `tab`, `False` caso contrário.

Nomeadamente, a expressão `take([20,5],add(16,make([2,1,8,1,16,4,20,64,32],5)))` produziria a tabela da ilustração.

Em *Python*, pretende-se representar uma tabela de dispersão como uma lista da forma $[w_0, w_1, \ldots, w_{s-1}]$ onde `s` é a dimensão da tabela e cada w_i é a lista dos valores armazenados no índice `i` da tabela. No caso exemplificado, ter-se-ia `[[],[1,16],[2,32],[8],[4,64]]`.

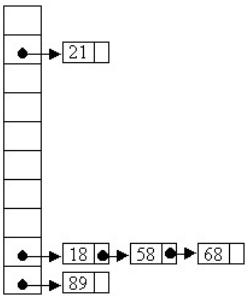
Apresente implementações eficientes para todas as operações.

Instituto Superior Técnico — LMAC/MEBiom **Rep4A**

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 4 Duração: 30m

Considere a camada de abstracção que disponibiliza o tipo de dados *tabela de dispersão (hash table)*, uma estrutura para armazenamento e pesquisa de valores (que se assumem inteiros), com dimensão predeterminada `s`, onde valores `x` (sem repetição de valores) são armazenados de acordo com índices `h(x,s)` (necessariamente valores em `range(s)`) que lhes são associados por uma dada *função de dispersão (hash function)*. A camada disponibiliza as seguintes operações:



- `empty(s)`: tabela de dimensão `s` sem qualquer valor armazenado;
- `put(vals,tab)`: tabela que resulta de armazenar na tabela `tab` todos os valores da lista `vals`;
- `rem(x,tab)`: tabela que resulta de remover da tabela `tab` o valor `x` (se existir);
- `size(tab)`: dimensão da tabela `tab`;
- `get(tab,i)`: lista dos valores armazenados no índice `i` da tabela `tab`;
- `double(tab)`: tabela que armazena os mesmos valores que a tabela `tab` mas cuja dimensão é o dobro da dimensão da tabela `tab`.

Na ilustração, exemplifica-se uma tabela de dispersão de dimensão 10, portanto com índices de 0 a 9 (de cima para baixo na figura), em que estão armazenados os valores 21 (no índice 1), 18, 58, 68 (todos no índice 8), e 89 (no índice 9), que poderia ser produzida por `tab=rem(3,put([89,18,21,3,58,21,68],empty(10)))`.

Defina em *Python*, sempre assegurando a independência relativamente à implementação do tipo de dados (disponibilizada na próxima página), as seguintes funções:

(**a**) uma função `pairscollide`, que dada uma tabela calcula a lista de todos os pares de valores $(\texttt{x}, \texttt{y})$ com $\texttt{x} < \texttt{y}$ que estão ambos armazenados no mesmo índice da tabela dada; por exemplo, `pairscollide(tab)` deverá resultar em `[(18,58),(18,68),(58,68)]`;

(**b**) uma função `halve`, que dada uma tabela calcula a tabela que armazena os mesmos valores que a tabela `tab` mas cuja dimensão é metade da dimensão da tabela `tab`, caso a dimensão seja par, devolvendo a tabela original em caso contrário; por exemplo, `halve(tab)` deverá resultar numa tabela com dimensão 5 armazenando os valores 21, 18, 58, 68, 89;

(**c**) uma função `loadfactor`, que dada uma tabela calcula a sua taxa de ocupação dada pela fracção dos índices da tabela que armazenam algum valor; por exemplo, `loadfactor(tab)` deverá ser 0.3 (correspondendo a 3 dos 10 índices conterem valores); use a função anterior para definir uma outra função `takeout`, que dada uma lista de valores, uma tabela, e um valor de referência (entre 0 e 1), remove todos os valores da lista dada da tabela, e de seguida comprime a sua dimensão (usando `halve`) enquanto a dimensão for par e a taxa de ocupação for inferior ao valor de referência, devolvendo a tabela resultante; por exemplo, `takeout([21,89],tab,0.5)` será a tabela de dimensão 5 armazenando os valores 18, 58, 68.

```
def empty(s):
    return [s]


def put(vals,tab):
    w=tab[1:]
    for x in vals:
          if x not in w:
                w=w+[x]
    w.sort()
    return [tab[0]]+w


def rem(x,tab):
    if x in tab[1:]:
          w=tab[1:]
          w.remove(x)
          return [tab[0]]+w
    else:
          return tab


def size(tab):
    return tab[0]


def get(tab,i):
    def h(x,s):
          return x%s
    return [x for x in tab[1:]  if h(x,tab[0])==i]


def double(tab):
    tab[0]=2*tab[0]
    return tab
```

Rep4B

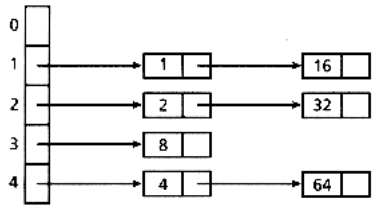
Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 4 Duração: 30m

Considere a camada de abstracção que disponibiliza o tipo de dados *tabela de dispersão (hash table)*, uma estrutura para armazenamento e pesquisa de valores (que se assumem inteiros), com dimensão predeterminada `s`, onde valores `x` (sem repetição de valores) são armazenados de acordo com índices `h(x,s)` (necessariamente valores em `range(s)`) que lhes são associados por uma dada *função de dispersão (hash function)*. A camada disponibiliza as seguintes operações:

- `make(vals,s)`: tabela de dimensão `s` que resulta de armazenar todos os valores da lista `vals`;
- `take(vals,tab)`: tabela que resulta de remover na tabela `tab` todos os valores (que lá existam) da lista `vals`;
- `numelems(tab)`: número de valores armazenados na tabela `tab`;
- `size(tab)`: dimensão da tabela `tab`;
- `getmin(tab)`: valor mínimo armazenado na tabela `tab` (que se assume não vazia);
- `existsQ(x,tab)`: `True` se o valor `x` está armazenado na tabela `tab`, `False` caso contrário.



Na ilustração, exemplifica-se uma tabela de dispersão de dimensão 5, portanto com índices de 0 a 4, em que estão armazenados os valores 1 e 16 (ambos no índice 1), 2 e 32 (ambos no índice 2), 8 (no índice 3), 4 e 64 (no índice 4), que pode ser obtida por `tab=take([5],make([5,4,16,2,1,2,8,4,64,32],5))`.

Assumindo que à função de dispersão corresponde a seguinte implementação,

```
def h(x,s):
    return x%s
```

defina em *Python*, sempre assegurando a independência relativamente à implementação do tipo de dados (disponibilizada na próxima página), as seguintes funções:

(**a**) uma função `repeats`, que dadas duas tabelas (possivelmente com dimensões diferentes) calcula a lista ordenada de todos os valores que estão armazenados em ambas; por exemplo, `repeats(tab,tab)` deverá resultar em `[1,2,4,8,16,32,64]`;

(**b**) uma função `double`, que dada uma tabela calcula a tabela que armazena os mesmos valores que a tabela `tab` mas cuja dimensão é o dobro da dimensão da tabela `tab`; por exemplo, `double(tab)` deverá resultar numa tabela com dimensão 10 armazenando os valores 1, 2, 4, 8, 16, 32, 64;

(**c**) uma função `collisionsQ`, que dada uma tabela devolve `True` se existem colisões na tabela `tab`, ou seja, valores distintos armazenados no mesmo índice, `False` caso contrário.r; por exemplo, `collisionsQ(tab)` deverá ser `True` (nomeadamente, os valores 1 e 16 estão ambos armazenados no índice 1); use a função anterior para definir uma outra função `endcollisions` que, dada uma tabela, expande repetidamente a sua dimensão (usando `double`) até que deixe de haver colisões, devolvendo a tabela resultante; por exemplo, `endcollisions(tab)` deverá resultar numa tabela com dimensão 40 armazenando os valores 1, 2, 4, 8, 16, 32, 64.

```
def h(x,s):
    return x%s


def make(vals,s):
    w=[]
    for x in vals:
          if x not in w:
                w=w+[x]
    w.sort()
    return [s]+w


def take(vals,tab):
    w=tab[1:]
    for x in vals:
          if x in w:
                w.remove(x)
          return [tab[0]]+w
    else:
          return tab


def numelems(tab):
    return len(tab)-1


def size(tab):
    return tab[0]


def getmin(tab):
    return min(tab[1:])


def existsQ(x,tab):
    return x in tab[1:]
```

Rep5A

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 5 Duração: 30m

Nesta avaliação não pode usar recursão, ciclos ou atribuições. Pode usar, sem necessitar de os definir, os combinadores `map, filter, any, all, reduce, nest, fixedpoint`, bem como definições `lambda` e por compreensão.

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

Por exemplo, à lista `w=[1,2,3,4,3,1,0,5,5,0]` corresponde a lista (de listas) de segmentos alternados `a=[[1,2,3,4],[3,1,0],[5,5],[0]]`. O primeiro segmento é sempre ordenado por ordem crescente e deve ser o mais longo possível, depois o segundo segmento é ordenado por ordem decrescente e deve também ser o mais longo possível, e por aí em diante.

(**a**) Defina funcionalmente em *Python* uma função `updown` que dada uma lista de números calcula a sua lista de segmentos alternados.

Por exemplo, o resultado de avaliar `updown(w)` deverá ser `a`.

(**b**) Defina funcionalmente em *Python* uma função `merge` que dada uma lista de listas, todas ordenadas por ordem crescente, calcula a lista ordenada com todos os elementos de todas as listas que resulta de acumular e eliminar, em cada passo, o menor elemento de todas as listas dadas (como no algoritmo de ordenação por fusão, mas não apenas com duas listas).

Por exemplo, o resultado de `merge([[1,2,3,4],[0,1,3],[5,5],[0]])` deverá ser a lista ordenada `[0,0,1,1,2,3,3,4,5,5]`.

(**c**) Defina funcionalmente em *Python* uma função `sort` que dada uma lista de números a ordena por ordem crescente, tirando partido das funções `updown` e `merge` das alíneas anteriores, mas tendo em conta que é necessário inverter alguns dos segmentos da lista.

Por exemplo, o resultado de `sort(w)` deverá ser `[0,0,1,1,2,3,3,4,5,5]`.

Rep5B

Instituto Superior Técnico — LMAC/MEBiom

Elementos de Programação

28 Janeiro 2021 Repescagem do Mini-teste 5 Duração: 30m

Nesta avaliação não pode usar recursão, ciclos ou atribuições. Pode usar, sem necessitar de os definir, os combinadores `map, filter, any, all, reduce, nest, fixedpoint`, bem como definições `lambda` e por compreensão.

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

Por exemplo, à lista `w=[5,1,7,8,9,6,4,2,3,0]` corresponde a lista (de listas) de segmentos alternados `a=[[5,1],[7,8,9],[6,4,2],[3],[0]]`. O primeiro segmento é sempre ordenado por ordem decrescente e deve ser o mais longo possível, depois o segundo segmento é ordenado por ordem crescente e deve também ser o mais longo possível, e por aí em diante.

(**a**) Defina funcionalmente em *Python* uma função `downup` que dada uma lista de números calcula a sua lista de segmentos alternados.

Por exemplo, o resultado de avaliar `downup(w)` deverá ser `a`.

(**b**) Defina funcionalmente em *Python* uma função `fusion` que dada uma lista de listas, todas ordenadas por ordem crescente, calcula a lista ordenada com todos os elementos de todas as listas que resulta de acumular e eliminar, em cada passo, o menor elemento de todas as listas dadas (como no algoritmo de ordenação por fusão, mas não apenas com duas listas).

Por exemplo, o resultado de `fusion([[1,5],[7,8,9],[2,4,6],[3],[0]])` deverá ser a lista ordenada `[0,1,2,3,4,5,6,7,8,9]`.

(**c**) Defina funcionalmente em *Python* uma função `order` que dada uma lista de números a ordena por ordem crescente, tirando partido das funções `downup` e `fusion` das alíneas anteriores, mas tendo em conta que é necessário inverter alguns dos segmentos da lista.

Por exemplo, o resultado de `order(w)` deverá ser `[0,1,2,3,4,5,6,7,8,9]`.