

## question 1: scheduling

A *prereq* is a task that must be performed before a particular other task. A *schedule* is an ordering of all tasks so that so that all prereqs are satisfied.

Example: Before you do your homework, you have to tidy your room and walk the dog. Before you tidy your room, you have to eat lunch. Before you walk the dog, you have to feed the dog. We can represent these prereqs like this:

```
tidy < homework
walkdog < homework
lunch < tidy
feeddog < walkdog
```

Here is a schedule for these prereqs:

```
lunch < feeddog < tidy < walkdog < homework
```

(a) [1 marks] Give another schedule for the above set of prereqs.

Any of these: FLWTH FWLTH FLTWH LFWTH LTFWH

(b) [1 marks] Give a schedule that satisfies these prereqs, or explain why there is none.

```
run < supper
lunch < text
study < supper
email < supper
text < email
lunch < run
LRTES LTRES LTERS ... plus study anywhere before supper
```

(c) [2 marks] Give a schedule that satisfies these prereqs, or explain why there is none.

```
run < supper
lunch < text
study < supper
supper < email
email < text
text < run
```

**there is none:** run < supper < email < text < run

[6 marks] (d) Write code that takes as input a list of prereqs and returns either a schedule or the message that there is no schedule.

```
# this algorithm is known as topological sorting. here is a python program.
# data structure: list of sublists (task, followed by that task's prepreqs)
# to topsort, repeatedly remove task with no prereqs, and update List
import sys

def taskIndex(L,t): #return index of task; append if not in list
    for j in range(len(L)):
        if t==L[j][0]:
            return j          # continue if no return ...
```

```
L.append([t])
return len(L)-1

def newPrereq(L,s,t): #add s<t to list L of prereqs
    j = taskIndex(L,s) #ensures s is in L
    k = taskIndex(L,t) #ensures t is in L, and gets index
    L[k].append(s)

def hasNoPrereqs(L,j): #if the only item in the sublist is the task
    return 1==len(L[j])

def noPrereqsIndex(L): #largest index of task with no prereqs
    x = -1
    for j in range(len(L)):
        if hasNoPrereqs(L,j):
            x = j
    return x # if still -1 then all tasks have prereqs

def removeTask(L,j): # remove task with index j
    task = L.pop(j)[0] # task is first item in sublist L[j]
    for item in L: # remove task wherever it appears as prereq
        for t in range(len(item)):
            if task==item[t]:
                item.pop(t)

def inputPrereqs(L):
    for line in sys.stdin:
        tuple = line.replace('\n','').split(' ')
        newPrereq(L,tuple[0],tuple[2])

def outputSchedule(L):
    S = []
    okSoFar = True
    while okSoFar and len(L)>0:
        x = noPrereqsIndex(L)
        if x<0:
            okSoFar = False
        else:
            S.append(L[x][0])
            removeTask(L,x)
    if okSoFar:
        for j in range(len(S)-1):
            print S[j], '<',
            print S[len(S)-1]
    else:
        print 'no schedule exists'

L = []
inputPrereqs(L)
outputSchedule(L)
```

## question 2: permutations

A *permutation* of a string of characters is an arrangement of the characters in some order. For example, the permutations of string abca are aabc aacb abac abca acab acba baac baca bcaa caab caba cbaa . Here, the permutations are sorted in *lexicographic* (dictionary) order.

a) [1 marks] How many permutations are there of iverson?

$$7! = 7*6*5*4*3*2*1 = 5040$$

b) [1 marks] How many permutations are there of baseballs?

**If the symbols were all different, the answer would be 9!, but the 2 a's are the same, the 2 b's are the same, the 2 s's are the same, and the 2 l's are the same, so the answer is  $9!/(2!*2!*2!*2!) = 22680$  .**

c) [3 marks] Write code that takes as input a string of lowercase alphabetic characters and returns the number of permutations.

1. count letter frequencies, say  $f_1 f_2 \dots f_t$

2.  $n!/(f_1! * f_2! \dots * f_t!)$

d) [2 marks] How many permutations of `baseballs` start with `a`?

$$8!/(2!*2!*2!) = 7! = 5040$$

e) [3 marks] For the string `baseballs`, find the 1982nd permutation (in lexicographic order). Show your work.

```
perms    1 .. 630    aa.....    7!/(2!*2!*2!) = 630
perms   631 .. 1890  ab.....    7!/(2!*2!)    = 1260
perms  1891 .. 1980  aea.....    6!/(2!*2!*2!) =  90
perm   1981                aebabllss
perm   1982                aebablsls
```

here's some python code to check my answer, and the output

fun problem: write the code for `next_permutation( )`

```
list = 'baseballs'.split() # list of characters
list.sort()
count = 1
increments = [0, 630, 1260, 90, 1]
for j in increments:
    for _ in range(j):
        next_permutation(list)
        count += 1
    print "".join(list), count
```

```
aabbellss 1
ababellss 631
aeabllss 1891
aebabllss 1981
aebablsls 1982
```

### question 3: ciphers

```

8      a b c d e f g h i j k l m n o p q r s t u v w x y z
1      b c d e f g h i j k l m n o p q r s t u v w x y z a
3      c d e f g h i j k l m n o p q r s t u v w x y z a b
4      d e f g h i j k l m n o p q r s t u v w x y z a b c
12     e f g h i j k l m n o p q r s t u v w x y z a b c d
2      f g h i j k l m n o p q r s t u v w x y z a b c d e
2      g h i j k l m n o p q r s t u v w x y z a b c d e f
6      h i j k l m n o p q r s t u v w x y z a b c d e f g
7      i j k l m n o p q r s t u v w x y z a b c d e f g h
0      j k l m n o p q r s t u v w x y z a b c d e f g h i
1      k l m n o p q r s t u v w x y z a b c d e f g h i j
4      l m n o p q r s t u v w x y z a b c d e f g h i j k
2      m n o p q r s t u v w x y z a b c d e f g h i j k l
7      n o p q r s t u v w x y z a b c d e f g h i j k l m
8      o p q r s t u v w x y z a b c d e f g h i j k l m n
2      p q r s t u v w x y z a b c d e f g h i j k l m n o
0      q r s t u v w x y z a b c d e f g h i j k l m n o p
6      r s t u v w x y z a b c d e f g h i j k l m n o p q
6      s t u v w x y z a b c d e f g h i j k l m n o p q r
9      t u v w x y z a b c d e f g h i j k l m n o p q r s
3      u v w x y z a b c d e f g h i j k l m n o p q r s t
1      v w x y z a b c d e f g h i j k l m n o p q r s t u
2      w x y z a b c d e f g h i j k l m n o p q r s t u v
0      x y z a b c d e f g h i j k l m n o p q r s t u v w
2      y z a b c d e f g h i j k l m n o p q r s t u v w x
0      z a b c d e f g h i j k l m n o p q r s t u v w x y

```

You might find the above column and array useful in this question. The column on the left gives percentage frequency of English letters, e.g. 8%, 1%, 3% for a, b, c.

A *cipher* takes as input a message, called the *plaintext*, and possibly also a *key*, and returns as output the encrypted message, called the *ciphertext*. The *Caesar-shift* cipher has no key: it shifts each plaintext letter forward 3 positions (with the last 3 letters of the alphabet shifted by wrapping around to the start of the alphabet).

Example: Caesar-shift of plaintext `iverson wxyz` is ciphertext `lyhuvrq zabc`.

More generally, for any integer  $k$ , the  $k$ -*shift* cipher shifts each plaintext character  $k$  positions forward (if necessary, wrap around to the start of the alphabet). Here,  $k$  is the key.

Example: 4-shift of plaintext `iverson wxyz` is ciphertext `mzivwsr abcd`.

Example: 30-shift of `iverson wxyz` is `mzivwsr abcd`.

(a) [2 marks] Caesar-shift `wacky zebra exit`

`zdfnb cheud halw`

25-shift `wacky zebra exit`

`vzbjx ydaqz dwhs`

The *polyshift* cipher works like this. Pick a keyword, say `crazy`. Pick a plaintext, say `too soon old, too late smart`. Create the key by repeating the keyword so that it is as long as the plaintext: `cra zy cr azy, cra zy cr azy cr`. Now use each key letter as the shift for the

corresponding plaintext character, where a is 0, b is 1, . . . , z is 25. So the shift sequence is 2-17-0 25-24-2-17 0-25-24 2-17-0 25-24-2-17 0-25-24-2-17, and the ciphertext is vfo rmqe okb vfo kyvv slytk. Another example: polyshift elephant with keyword baby is ciphertext flfniaor.

(b) [2 marks] polyshift iverson exam with keyword crazy kmeqqqe ewyo

(c) [2 marks] To *crack* a cipher is to find the plaintext from the ciphertext without knowing the key. Crack this shift cipher (English plaintext, spaces omitted). Explain your method.

**method: try every shift, there are only 25 . . . shift is m**

ciphertext: ufezqhqdfaaxmfq

plaintext: itsnevertoolate

(d) [4 marks] Crack this polyshift cipher (English plaintext, keyword length 3). Explain your method. Hint: consider letter frequencies in the substring composed of positions 1,4,7, . . . , and then for positions 2,5,8, . . . , and then for positions 3,6,9, . . .

ciphertext: pvtgirjvjgjq gmc uccrk bgvnp

below are the letter frequencies of the 3 substrings. each substring has only 1 character which appears more than once. it turns out that this corresponds to the most frequent English character, e. so shifting respectively g,v,c back to e undoes the cipher. keyword is cry.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	0	0	0	0	0	4	0	0	1	0	0	0	1	0	1	0	1	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1	1	0	1	0	0	1	0	0	0	0	0	3	0	0	0	0
0	1	2	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0

plaintext: nevertheless eve slept deeply

## question 4: sigma game

Alice and Bob play a game called *sigma*. The input is a list of numbers. Alice goes first. They alternate turns. On a turn, a player takes (and removes) either the first number or last number from the list. The game ends when all numbers are gone. A player's score is the sum of the numbers they took. The player with the greater score wins.

Example: list [2 5 1 4]. Alice takes 4, leaving [2 5 1]. Bob takes 2, leaving [5 1]. Alice takes 5, leaving [1]. Bob takes 1. Alice scores  $4+5=9$ . Bob scores  $2+1=3$ . Alice wins.

A player is *sane* if she maximizes the minimum score that she is guaranteed against all possible opponent strategies.

Example: [5 1]. If Alice is sane then she takes 5 and scores 5 (otherwise she takes 1 and scores 1).

Example: [1 2 4]. If Alice is sane then she takes 4.

The *value* of a list is, for the game with that list, the score of the first player minus the score of the second player, assuming that each player is sane.

Examples: value of [5] is  $5-0=5$ ; value of [5 1] is  $5-1=4$ ; value of [1 2 4] is  $5-2=3$ .

(a) [2 marks] Give the value of each list:

[1 5 2] value -2

[1 5 2 4] value 6

[3 1 5 2] value 5

[3 1 5 2 4] value -1

(b) [2 marks] Bob says that a best strategy is to always take the larger available number, or either one if they are equal. Alice says Bob is wrong. Who is correct? Justify your answer.

**Alice is correct. For [1 1 100 2], Alice's scores -98 if she takes 2, but 98 if she takes 1.**

(c) [3 marks] The value of a list can be computed from the values of its sublists. Fill in the empty cells of the array, which give sublist values of [1 4 5 2 3 3 5 1]. The entry in row  $x$  and column  $y$  is the value of the sublist from position  $x$  to position  $y$ . E.g., the entry in the second row and fourth column is the value of sublist [4 5 2], namely 1.

1	3	2	0	3	0	5	4
	4	1	1	2	1	4	-3
		5	3	4	3	2	7
			2	1	2	3	-2
				3	0	5	4
					3	2	-1
						5	4
							1

(d) [3 marks] Write code that takes as input a list and returns the list's value.

```
def score(L):
    # compute the value of list L

    n = len(L)
    # list L has n numbers, indexed 0..n-1

    for j in range(n):
        # j runs from 0 to n-1

        Value[j][j] = L[j]

    for gap in range(1,n):
        #gap runs from 1 to n-1

        for x in range(n-gap):
            #x runs from 0 to (n-gap)-1

            y = x+gap

            Value[x][y] = max( L[x] - Value[x+1][y], L[y] - Value[x][y-1] )

    return Value[0][n-1]
```