# the Iverson computing science competition
# 27 May 2014
# with solutions

| | |
|---|---|
| name | |
| school | |
| city | |
| grade | |
| cs teacher | |

Are you taking AP or IB computer science? (yes/no) ____________

Have you taken any Advanced Level courses? (3000 level, e.g. CSE3110 Iterative Algorithms I.) (yes/no/currently taking) ____________

**illegible answers will not be marked**

| questions | – – – – | marks | score |
|---|---|---|---|
| 1 | sparse strings | 10 | |
| 2 | tv schedules | 10 | |
| 3 | error correction | 10 | |
| 4 | balancing parentheses | 10 | |
| total | – – – – | 40 | |

## Exam Format

This is a two-hour paper and pencil exam. There are four questions, each with more than one part. Some part(s) might be easy. Solve as many parts of as many questions as you can.

## Programming Language

Questions that require programming can be answered using any language (e.g. C/C++, Java, Python, ...) or pseudo-code. **Pseudo-code should be detailed enough to allow for a near direct translation into a programming language.** Clarify your code with appropriate comments. For full marks, a question must be **correct, well-explained, simple, and efficient.**

Our primary interest is in thinking skill rather than coding wizardry, so logical thinking and systematic problem solving count for more than programming language knowledge.

## Suggestions

- You can assume that the user enters only valid input.
- Sample executions of the desired program are shown for coding questions. Review the samples carefully to make sure you understand the specifications. The samples may give hints.
- Design your algorithm before writing any code. Use any format (pseudo-code, diagrams, tables) or aid to assist your design plan. We may give part marks for legible rough work, especially if your final answer is lacking. We are looking for key computing ideas, not specific coding details, so you can invent your own "built-in" functions for subtasks such as reading the next number, or the next character in a string, or loading an array. Make sure to specify such functions by giving a relationship between their inputs and outputs.
- In the same vein, if you forget the specifics of how to read input and write output in the language you choose (if you are not using pseudocode) then simply provide some pseudocode explaining how the input is stored and what will be output. We are mostly interested in the algorithm you design to solve the problem and are not as concerned about input/output details.
- Read all questions before deciding which ones to attempt, and in which order. Start with the easiest parts of each question.

## question 1: sparse strings

A bit-string is *sparse* if each pair of consecutive bits includes at least one `0` bit. Example: `010` and `101` are sparse, but `011` is not. Below are all sparse strings with length three.

```
000   001   010   100   101
```

(a) [2 marks] List all sparse bit-strings with length `4`.

```
0000  0001  0010  0100  0101  1000  1001  1010
```

(b) [2 marks] For a given positive integer `n`, let `f(n)` be the number of sparse length `n` bit-strings. Example: `f(3)` = `5`. Find `f(5)`.

**`f(5) = 13`, the strings (which you didn't have to reproduce) are**

```
10000  10001  10010  10100  10101

00000  00001  00010  00100  00101  01000  01001  01010
```

(c) [2 marks] For `f(n)` defined above, and for `n` an integer and at least 3, assume that you know the values of `f(k)` for each `k` with `1` $\leq$ `k` $\leq$ `n-1`. Explain how to use this information to compute `f(n)`.

**For `n` $\geq$ 3, we have `f(n) = f(n-1) + f(n-2)`.**

**Reason: Every sparse string of length `n` starts with either a `1` or a `0`. If it starts with `1`, then the next bit must be `0` followed by any one of the `f(n-2)` different length `n-2` sparse strings. If a length `n` sparse string begins with a `0`, the remaining `n-1` bits are equal to one of the `f(n-1)` length `n-1` sparse strings.**

**The solution written above for part (b) illustrates this fact.**

(d) [4 marks] Write a function `numSparse` that takes as parameter a positive integer `n` and returns `f(n)`. For full marks, your algorithm should be efficient (faster than generating all of $2^{n}$ length `n` bit-strings and checking which are sparse).

**Example Input**

```
3
```

**Example Output**

```
5
```

**The idea in this solution is to use the fact that `f(n) = f(n-1) + f(n-2)`. This was demonstrated in part (c).**

```
def numSparse(n):
    f = [1,2,3]                         # f[1] = 2 and f[2] = 3, ignore f[0]
    while len(f) <= n:
        f.append(f[-1] + f[-2])
    return f[n]
```

## question 2: tv schedules

Abbey is switching from cable TV to IPTV (an internet protocol streaming service). Before her cable service ends, she will watch as many TV shows as possible. She will follow these rules:

- She will watch one show at a time (of course!).
- For each show that she watches, she will watch from the start time to the end.

We describe each show its start time `s` and end time `t`. Each time is represented as an integer, in minutes. She can switch instantly from the end of one show to the start of another. So, if a show has start time 30 and end time 60, and a second show has start time 60 and end time 90, then she can watch both. But, if the second show has start time 59 or earlier, then she cannot watch both.

(a) [1 marks] Each line gives a show's start time, followed by its end time. Find the maximum number of shows she can watch, and indicate a maximum size set of shows she can watch by circling the start/end times.

**Abbey can watch 3 shows:**
90 120
**100 150**
**30 70**
60 90
**80 100**

(b) [2 marks] Repeat part (a), for the following shows.

50 60
20 60
**40 55**
**65 80**
**25 35**
**55 65**
**80 95**
70 85
55 65

(c) [3 marks] Write a function `compatible` that takes as parameters four integers `s1,t1,s2,t2` — the start and end times of two shows — and returns a boolean value that is true if and only if Abbey can watch both shows. Example: `compatible(30,60,60,90)` returns `true`. Example: `compatible(30,61,60,90)` returns `false`.

```
def compatible(s1, t1, s2, t2):
    return t1 <= s2 or t2 <= s1
```

(d) [4 marks] Write a function `maxShows` that takes as input a positive integer `n`, and a list of `n` shows (each represented by its start and end time), and outputs the maximum number of shows that Abbey can watch. You may assume that, for each show, its end time is greater than its start time. Your output should be the integer that gives the maximum number of shows that Abbey can watch. For full marks, your algorithm should be faster than exhaustive search.

**Example Input**

```
4
95 110
90 100
105 120
100 110
```

**Example Output**

```
2
```

**One way to get a fast algorithm is to exploit the following fact. Let `1` $\leq$ `i` $\leq$ `n` be the index of a show with the smallest ending time. Then there is a way to watch the maximum number of shows such that show `i` is one of the ones watched.**

**Reason: Suppose Abbey watches some shows but does not watch show `i`. Say show `j` had the earliest end time of the shows that Abbey watched. Since `t[i]` $\leq$ `t[j]` and all other shows Abbey watched started after `t[j]`, then the only show Abbey watched that could have conflicted with show `i` is show `j`. So, Abbey could still watch the same number of shows by watching show `i` instead of show `j`.**

**Once we see this, an efficient algorithm is easy to describe. Watch the show that ends earliest, discard all shows that conflict with this show, and repeat until there are no more shows to process.**

```
def maxShows(n, S, E):
    count = 0
    while len(S) > 0:
        min_E = min(E)
        count += 1
        # find the times that do not conflict with the
        # show ending at time min_E
        Stmp, Etmp = [], []
        for i in range(len(S)):
            if S[i] >= min_E:
                Stmp.append(S[i])
                Etmp.append(E[i])
        S, E = Stmp, Etmp
    return count
```

**In fact, we can get an even faster algorithm if we simply sort all shows by their end time. Once they are sorted, we iterate through them in increasing order of end time and watch a show if it does not conflict with the previously-watched show.**

## question 3: error correction

Data corruption is a real-world problem. For instance, when satellites transmit bit-strings, it is common that 0.01% of the received bits *flip* from 0 to 1, or from 1 to 0. For example, if `1 0 1 1 0 1 1 1` is sent and `0 0 1 1 1 1 1 1` is received, then the 1st and 5th bits were flipped.

By adding extra bits to messages in a clever way, *error correcting codes* allow reconstruction of the correct original bit-string.

Here is one method that error-corrects a 3-bit message $(a, b, c)$ if at most 1 bit flips during transmission: use 3 extra bits, and transmit the message

$$(a, b, c, a \oplus b, a \oplus c, b \oplus c)$$

where $x \oplus y$ is 0 if $x = y$ and 1 if $x \neq y$.

(a) [2 marks] Complete the following table.

| $a$ | $b$ | $c$ | $a \oplus b$ | $a \oplus c$ | $b \oplus c$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

(b) [2 marks] We claim that using the above method, the receiver can reconstruct the original message $(a, b, c)$ if at most one of the six bits $(a, b, c, a \oplus b, a \oplus c, b \oplus c)$ flips during transmission. In one or two sentences, explain why this claim is true.

**For two different 3-bit strings, the corresponding 6-bit messages differ in at least three positions (see the table in part (a)). So, if a single bit flips during transmission then the only row in the above table that differs from the corrupted message in one bit is the original 6-bit message.**

(c) [2 marks]

| e | c | a | d | h | t | k | n |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

Text messages are transmitted using bit-strings, as shown by the above table. The above table gives the bit-strings for some alphabetic characters. For example, '`k`' is represented by '`110`'. To transmit the character '`k`', its 3-bit code is sent (`110`), followed immediately by its 3-bit error-correcting code (`011`). For example, to tranmit the character string '`kc`', send `110011 001011`.

An important message is sent to you in this way. You receive the following sequence. In each 6-bit string, at most 1 bit has flipped. What is the important message?

```
001000 111010 010110 100101 100010 000000

110101 101111 111101 000101 001011 111011
```

`e n d t h e a t t a c k`

(d) [1 marks] Assume that each bit flips with probability 1/10 (a noisy transmission channel!). If we send 3-bits $(a, b, c)$, the probability that this string is correctly received is equal to the probability that each bit does not flip, so $\left(\frac{9}{10}\right)^3 = 0.729$. Now suppose that we send the 6-bits $(a, b, c, a \oplus b, a \oplus c, b \oplus c)$ instead. What is the probability that at most one bit flips (so the receiver can decode the original 3-bit message)?

(For full marks, please express your answer as an arithmetic expression. You do not need to simplify it).

**The probability nothing flips is $\left(\frac{9}{10}\right)^6$. The probability that bit $i$ flips is $\frac{1}{10} \cdot \left(\frac{9}{10}\right)^5$ for any $i$. Thus, the probability at most one bit flips is**

$$\left(\frac{9}{10}\right)^6 + 6 \cdot \frac{1}{10} \cdot \left(\frac{9}{10}\right)^5 = 0.885735$$

**Even though we are sending more bits, we actually have a higher probability of receiving the original message because we can correct one-bit errors. Cool!**

(e) [3 marks] Write a function `decode` that takes as parameters the six received bits $(a, b, c, a \oplus b, a \oplus c, b \oplus c)$, where at most one of the sent bits has been flipped. The function returns the original bits $(a, b, c)$. For full marks, your solution must be clean and concise!

**Example Input**
```
0 1 0 0 1 1
```

**Example Output**
```
1 1 0
```

```
def xor(x, y):
    if x == y:
        return 0
    else:
        return 1

# check if (A,B,C,D,E,F) is one of the rows in the table
def check(A, B, C, D, E, F):
    return xor(A,B) == D and xor(A,C) == E and xor(B,C) == F
    # checking A^B == D and so on also works in Python

def decode(A, B, C, D, E, F):
    if check(1-A, B, C, D, E, F):
        return (1-A, B, C)
    elif check(A, 1-B, C, D, E, F):
        return (A, 1-B, C)
    elif check(A, B, 1-C, D, E, F):
        return (A, B, 1-C)
    else:
        return (A, B, C)
```

**Idea: The first three statements check to see if there was a one-bit error in the first three bits. If not, then either there was no error or the error was in the last three bits. In either case, the first three bits are not corrupted.**

## question 4: balancing parentheses

In an arithmetic expression, multiplication has precedence over addition. Example: `3+4*5` evaluates to `23`, because the multiplication is performed before the addition. If you want to add before you multiply, you must use parentheses. Example: `(3+4)*5` evaluates to `35`. To make it easier to read expressions, we also use { } or `[ ]` as parentheses. Example: `[(6+{3+4}*5)*2 + 1]*(1+3)` evaluates to `332`.

If you take such an arithmetic expression, and remove all characters except `(`,`)`,`[`,`]`,`{`,`}`, then the remaining parentheses are *balanced* if they are in the set $S$ of strings defined by the following rules:

1. `()`, `[]`, `{}` are each in $S$,
2. if `s` and `t` are each in $S$, then `st` is in $S$,
3. if `s` is in $S$, then `(s)`, `[s]`, `{s}` are each in $S$.

Example: Consider the above expression `[(6+{3+4}*5)*2 + 1]*(1+3)`. Removing all characters except parentheses leaves `[({})]()`. Now `{}` is in S by rule 1, so `({})` is in S by rule 3, so `[({})]` is in S by rule 3. Also, `()` is in S by rule 1, so finally `[({})]()` is in S by rule 2.
Example: By the rules, the set of parentheses `(()` is not balanced.

(a) [3 marks]
Which of these strings are balanced? Underline them.

- `(]`
- $\underline{\texttt{([](\{\}))}}$
- `{()[]{([])}()`
- `[]({[]()}[{})]())`
- $\underline{\texttt{[](\{[]()\}[\{\}]())}}$
- `(((())())()))((())(()())`

(b) [2 marks] List all balanced strings containing only `[` and `]`, and with length 6.

`[][][]`    `[][[]]`    `[[]][]`    `[[][]]`    `[[[]]]`

(c) [5 marks] Write a function `balanced` that takes a parameter $s$ — a non-empty string consisting of characters `(`, `)`, `[`, `]`, {, or } — and returns `true` if the string is balanced and `false` if it is not.

For part marks (at most 4 out of 5), you may assume that $s$ is a non-empty string consisting of only `(` or `)` (the other parenthesis symbols do not appear).

**Example Input**
`[({})]()`

**Example Output**
`true`

**Idea: Keep track of a list of characters corresponding to the "stack" of opening characters `(`,`[`,{ that have not yet been matched. This should be such that the most recent opening character is at the front of the list and so on.**

**Go through the characters of `s` from left to right. When an opening character is encountered, add it to the front of the list we are maintaining. When a closing character `)`,`]`,} is encountered, it has to match the opening character at the front of the list otherwise the string is not balanced. Once these are matched, remove the character at the front of the list and continue.**

**Once the string has been processed we have to check to see if the final list is empty. If not, then not all opening characters were matched.**

```
def balanced(s):
    stack = []

    for c in s:
        if c == '[' or c == '{' or c == '(':
            stack.append(c)
        else:
            if len(stack) == 0:
                return False
            elif c == ']' and stack[-1] != '[':
                return False
            elif c == ')' and stack[-1] != '(':
                return False
            elif c == '}' and stack[-1] != '{':
                return False

            stack.pop()

    return len(stack) == 0
```

**Partial Marks Version: If there are only `(` and `)` characters in `s`, then it is easier to check. Keep track of a "count" of how many opening parenthesis have not been closed off yet. When `(` is seen, increase the count. When `)` is seen, decrease the count. This count should never be negative, otherwise the closing parenthesis does not match an opening parenthesis. Finally, it should be 0 at the end to ensure every opening parenthesis was matched.**

```
def balanced_easy(s):
    count = 0

    for c in s:
        if c == '(':
            count += 1
        else:
            count -= 1

        if count < 0:
            return False;

    return count == 0
```