Iverson computing competition
2018 may 23

name ______________________________

school ______________________________

city ______________

grade _______

cs teacher ______________________________

are you taking AP computer science? (yes/no) ____

are you taking IB computer science? (yes/no) ____

illegible answers will not be marked
keep answers brief and to the point

question	– – – –	marks	your score
1	toothpaste	6	
2	run length encoding	7	
3	shuffle	6	
4	bin packing	9	
5	interesting	12	
total	– – – –	40	

Exam prepared by Zachary Friggstad

Exam Format

This is a two-hour paper and pencil exam. There are five questions, each with multiple parts. Some part(s) might be easy and the weight of a part does not always reflect the difficulty of the part. 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, an answer must be correct, well-explained, and as simple as possible.

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

1. You can assume that the user enters only valid input in the coding questions.
2. In somes cases, sample executions of the desired program are shown. Review the samples carefully to make sure you understand the specifications. The samples may give hints.
3. 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 simple subtasks such as reading the next number, or the next character in a string, or loading an array.
4. Read all questions before deciding which ones to attempt, and in which order. Start with the easiest parts of each question.
5. Use the reverse side of the exam sheets for longer answers. Clearly indicate when you have done so, so we don't miss it when grading.
6. Some parts ask you to write a function to solve a problem. You may use additional "helper" functions.

question 1: toothpaste

Kids are quite inefficient at using toothpaste: they only use some of the toothpaste in each tube before they discard it. They claim it is too difficult to get the last part out.

Rather than help them squeeze the last part out, one could combine the contents of discarded tubes to create new tubes. In particular, for some integer $k \geq 2$ one could do the following. For every k tubes of toothpaste the kids discard, create a new tube of toothpaste.

Suppose a particular household starts with n tubes of toothpaste (a bulk pack from Costco). Let `num_tubes(n, k)` be how many tubes the kids will end up using overall when starting with just these n tubes, assuming every k tubes they discard will be combined into a new tube that will also be used.

example: `num_tubes(2, 2)` is 3. After the initial $n = 2$ tubes are used, the parents combine them into one more new tube

example: `num_tubes(3, 2)` is 5. After the initial $n = 3$ tubes are used, the parents combine two of them into a new tube. Once this tube is used, the parents combine it with the remaining leftover tube from the initial tubes to create one final tube.

(a) [2 marks] Compute the following:

- `num_tubes(4, 2)`
  Solution: 7

- `num_tubes(7, 3)`
  Solution: 10

- `num_tubes(23, 4)`
  Solution: 30

- `num_tubes(173, 90)`
  Solution: 174

(b) [3 marks] Provide an implementation of the function `num_tubes(n, k)`.

Solution You can just simulate the process.

```
def num_tubes(n, k):
    used = n
    empty = n

    while empty >= k:
        empty -= (k-1) # use k empty tubes to create one new tube
        used += 1
    return used
```

(c) [1 marks] Show how to compute `num_tubes(n, k)` using a simple one-line expression (no loops).

Solution Look carefully at the previous solution. After the initial n tubes, the question is really how many times we can subtract $k - 1$ from n until it is less than k.

So the answer is just $n/(k-1)$ rounded down if $k-1$ does not divide n. But if $k-1$ divides n then it is $n/(k-1) - 1$ (i.e. we had to stop the process once we reached $k-1$ remaining tubes, not when we reached 0). We can capture both cases by simply $(n-1)/(k-1)$ rounded down. Or, in terms of the original question about toothpaste, the answer is

$$n + \frac{n-1}{k-1} \text{ rounded down}$$

question 2: run length encoding

A very simple algorithm to compress a string of letters is to replace consecutive sequences of the same characters by a number indicating how many occurrences there are of the letter. One does this by breaking a string into the **maximal** substrings consisting of consecutive occurrences of the same character (an example follows). That is, all characters in a substring must be the same, each substring needs at least one character, and characters just before and after the substring (if any) should be different than the character in the substring.

Each such substring is then replaced by first the length of the substring followed by the common character, but **only if** this produces a shorter string. Finally, these replacement strings are concatenated to form the compressed string.

example
Beginning with string `str = aaaaabbbcbbbbaacccccccccccc`:

- Break `str` into substrings `aaaaa`, `bbb`, `c`, `bbbb`, `aa`, `cccccccccccc`.
- Respectively, replace these with substrings `5a`, `3b`, `c`, `4b`, `aa`, `12c`.
- Form the compressed string `5a3bc4baa12c`.

(a) [1 marks] Encode the following strings using this algorithm.

- `bbbbbbaaaaccccccccccdddaaaaa`
  Solution: `6b4a10c3d5a`
- `abbabbaabbbaba`
  Solution: `abbabbaa3baba`

(b) [1 marks] Each string below is a compressed string. For each, find the string consisting only of lowercase letters that would produce the corresponding compressed string.

- `7c3a5c3a`
  Solution: `cccccccaaacccccaaa`
- `3babb4a5baab`
  Solution: `bbbabbaaaabbbbbaab`

(c) [2 marks] Write a function `compress(txt)` that takes, as input, a single string `txt` and returns the string obtained by compressing `txt` according to the above algorithm. You may assume, without checking, that `txt` only contains lowercase letters (in particular, it has no spaces, punctuation, digits, or uppercase letters).

Solution

```
def compress(txt):
    at = 0 #index into the unprocessed part of the string
    output = ""

    while at < len(txt):
        next = at
        while next < len(txt) and txt[next] == txt[at]:
            next += 1
        if next > at+2:
            output += str(next-at) + txt[at]
        else:
            output += txt[at:next]
        at = next
    return output
```

(d) [3 marks] Write a function `decompress(enc)` that takes, as input, a single string `enc` and returns the string `txt` consisting only of lowercase letters such that `compress(txt) == enc`. You may assume such a string `txt` exists. You may use, without providing the implementation, functions `isletter(c)` and `isdigit(c)` to check if a character `c` is a letter or a digit (and similar functions, if desired).

Solution

First is a simpler implementation that works if every number is ≤ 9. It is not a complete solution, but it illustrates the point.

```
def decompress(enc):
    at = 0
    output = ""

    while at < len(enc):
        if enc[at].isdigit():
            output += enc[at+1]*(int(enc[at]))
            at += 2
        else:
            output += enc[at]
            at += 1
    return output
```

A complete solution reads off digits from the current position until it encounters a letter (to get the whole number). Here is one way to do it.

```
def decompress(enc):
    at = 0
    output = ""

    while at < len(enc):
        char_pos = at

        # read off the digits starting at position "at"
        while enc[char_pos].isdigit():
            char_pos += 1

        if char_pos == at:
            # if we read no digits, use a single character
            num = 1
        else:
            # else, get the number
            num = int(enc[at:char_pos])

        # either way, the character is at index char_pos
        output += enc[char_pos]*num

        at = char_pos+1
    return output
```

question 3: shuffle

A common feature of music players is to “shuffle” the songs: to play them in random order. Consider the following pseudocode that shuffles a playlist with just three songs. For brevity, the songs will be called `A`, `B`, and `C`.

```
songs = ["A", "B", "C"]

for i from 0 to 2
  swap(songs[i], songs[random(3)])
```

Here, `swap` is a function that exchanges the values of the two variables and `random(3)` picks and returns either 0, 1, or 2 uniformly at random (i.e. each has probability 1/3 of being chosen). Unfortunately, this is not a good shuffling algorithm! Some rearrangements are more likely than others.

(a) [1 marks] How many different rearrangements of the array `songs` are there? List them all.
Solution: There are 6. `ABC, ACB, BAC, BCA, CAB, CBA`:

(b) [2 marks] What is the probability of the song `"C"` being first in the rearrangement when the array is shuffled using the code above? Express as a fraction for full marks. It will not be $\frac{1}{3}$, as one would expect if all rearrangements were equally likely.
Tip: Consider all ways that `C` could be put in the first position by the algorithm and add their probabilities.
Solution: 8/27. Here are the possible ways:

- `C` is initially swapped to the start and then never swapped out again.
  Probability $= \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{2}{3}$.
- `C` is not swapped out of the last position until the very end, when it is swapped to the first position.
  Probability $= \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}$.

You can check carefully that these are the only ways have `C` be moved to the start.

(c) [3 marks] Describe how to shuffle an array of arbitrary length $n \geq 1$ so each possible arrangement is equally likely. You can either provide pseudocode or a brief written description. Provide a brief argument to justify why your approach works. For full marks, your algorithm should perform at most n swaps.

Solution

Below is pseudocode to shuffle an array named `array` with length `n`

```
for i from n-1 down to 1
  # swap the entry at index i with the
  # entry at a random index from 0, 1, ..., i
  swap(array[i], array[random(i+1)])
```

We'll sketch a proper "proof by induction", though we accepted arguments that were less formal. For $n = 1$, there is only one arrangement so (trivially) all rearrangements are equally likely.

For $n \geq 2$, note each item has the same probability (namely, $1/n$) of being swapped to position $n-1$ in the first iteration. Then the rest of the algorithm behaves identically on the remaining array. By "induction" (i.e. by repeating the argument for smaller arrays), each of the $(n-1)!$ rearrangements of the remaining array are equally likely. Thus, each of the $n!$ arrangements of the original array are equally likely.

question 4: bin packing

In the **bin packing** problem, you are given n items of various weights $w_0, w_2, \ldots, w_{n-1} \geq 0$. You want to ship these items in containers, but each container can only hold up to a total weight of $C \geq 0$ items. Naturally, you want to ship the items using the fewest containers possible. You may assume $w_i \leq C$ for each item i.

Unfortunately this is a hard problem; nobody has discovered an efficient algorithm to solve it. In this question, you will explore some *heuristics*: fast algorithms that tend to perform somewhat well but may not always find the best solution.

(a) [2 marks] Consider the following **first-fit heuristic**. Here, `bins` is an array representing the remaining capacity in each bin we have used so far. Initially, this array is empty because we have not used any bins. We assume the weights are stored in an array `w[]`, indexed from 0 to $n-1$.

```
bins[] = empty array
for each i from 0 to n-1
  find the smallest index j (if any) where w[i] < bins[j]
  if no such j exists, append C-w[i] to bins   # i.e. add item i to a new bin
  otherwise, set bins[j] = bins[j]-w[i]        # i.e. put item i in bin j
```

That is, each item is packed in the first bin that it fits in. If there is no such bin, a new bin is created (i.e. appended to the end of the array `w[]`).

Solution Note

There was a typo in the pseudocode. It should have said `where w[i] <= bins[j]` in the first line in the loop. We accepted answers that used either the original statement or the variant with this `<=` fix. The answers here will use the `<=` interpretation, the answer for part (b) would be different under the other interpretation.

(a) [2 marks] How many bins will be used by this heuristic if `C = 10` and the array of weights `w[]` is `2, 1, 2, 6, 2, 7`? Show the contents of the bins after each iteration.

Solution

3 bins will be used.

1. $[2]$
2. $[2, 1]$
3. $[2, 1, 2]$
4. $[2, 1, 2], [6]$
5. $[2, 1, 2, 2], [6]$
6. $[2, 1, 2, 2], [6], [7]$

(b) [2 marks] People have observed the above heuristic often performs better if the items are first sorted in reverse order. Consider the following algorithm.

```
sort the items in reverse order (so w[0] >= w[1] >= ... >= w[n-1])
pack the items in bins using the first-fit heuristic (above)
```

Now how many bins will be used by this heuristic applied to the same case as last part? Show the contents of the bins after each iteration.

Solution
2 bins will be used.

1. $[7]$
2. $[7], [6]$
3. $[7, 2], [6]$
4. $[7, 2], [6, 2]$
5. $[7, 2], [6, 2, 2]$
6. $[7, 2, 1], [6, 2, 2]$

(c) [2 marks] Find an example showing the algorithm from part (b) does not always use fewest bins possible to pack the items. Make sure you show the final packing obtained from the algorithm **and** a packing using the fewest bins possible.

Solution
One possibility could be if the items have weights $3, 3, 2, 2, 2, 2$ and the bin capacity is 7. A packing with two bins is $[3, 2, 2], [3, 2, 2]$ but the first-fit heuristic would use three bins, namely $[3, 3], [2, 2, 2], [2]$.

(d) [1 marks] The next few parts have you reason that, at the very least, the number of bins used by the heuristics is somewhat close to the optimum.

Let $W = w_0 + w_1 + \ldots w_{n-1}$ and let k^* denote the minimum number of bins such that it is possible to pack the items into k^* bins. Briefly explain why $\frac{W}{C} \leq k^*$.

Solution

The total capacity of the k^* bins is exactly $k^* \cdot C$. By the definition of k^*, there is some way to pack all items in k^* bins so the total capacity of $k^* \cdot C$ holds a total weight of W. Namely, $W \leq k^* \cdot C$.

(e) [1 marks] Briefly explain why the algorithm from part (a) will result in *at most* one bin being less than half full (i.e. at most one index j such that `w[j] < C/2`).

Solution Suppose two bins were less than half full and say these are bins $i < j$. Then at the point of the algorithm when some item that was put into j was considered, bin i was already available and had enough space for the item. More precisely, this item has weight $< C/2$ because it is one of the items in bin j. When this item was being added, bin i has at least $C/2$ leftover capacity so the item should have been put there instead.

Thus, it is impossible to have more than one bin that is less than half full.

(f) [1 marks] Finally, explain why the number of bins used at the end of the algorithm from part (a) is at most $2 \cdot k^* + 1$. You may assume any of the statements above, even if you could not find a good argument.

Solution Say the algorithm used k bins. On one hand, we know $W \geq (k-1)/2 \cdot C$ because at most one of the k bins is less than half full. On the other hand, we know $W \leq k^* \cdot C$. So,

$$(k-1)/2 \cdot C \leq W \leq k^* \cdot C.$$

Rearranging, this means $k \leq 2 \cdot k^* + 1$. Algorithm (a) is guaranteed to use barely more than twice the number of bins than the optimum.

Note

The algorithm from part (b) is known to to use at most $\frac{11 \cdot k^* + 6}{9}$ bins, which is even better than what we saw from (a). But the analysis is far more involved.

question 5: interesting

A **bitstring** is a string consisting only of characters `0` and `1` and is not the empty string. A bitstring is said to be **interesting** if it does not contain three consecutive occurrences of `0` or three consecutive occurrences of `1`.

some interesting bitstrings
`1`, `00`, `00110011`, `100110010101001`.

some bitstrings that are not interesting
`1011101`, `000111000111`, `111`, `101000101101`.

all interesting bitstrings of length 4 ending with a 0
`0100`, `1100`, `0010`, `1010`, `0110`.

(a) [1 marks] For $n \geq 1$, let $f_0(n)$ be the number of interesting bitstrings with length exactly n that end with 0. We see from the example above that $f_0(4) = 5$.

Compute $f_0(1)$, $f_0(2)$ and $f_0(3)$.

Solution $f_0(1) = 1, f_0(2) = 2$ and $f_0(3) = 3$.

(b) [2 marks] Compute $f_0(5)$ and $f_0(6)$.

Solution $f_0(5) = 8$ and $f_0(5) = 13$.

(c) [3 marks] For $n \geq 3$, express $f_0(n)$ as a simple function of some values $f_0(k)$ for various $k < n$.
Hint: Consider the different ways the 0s can appear at the end of a bitstring. Notice ending with a 0 or 1 does not really matter, the counts would be the same.

Solution
In any interesting bitstring that ends with a 0, either the preceding character is a 1 or the preceding two characters are 10. If we let $f_1(n)$ be the number of interesting bit strings that end with a 1, then we have just seen for $n \geq 3$ that

$$f_0(n) = f_1(n-1) + f_1(n-2).$$

Of course, $f_1(n) = f_0(n)$ which can be seen by noting we can just flip the bits in all interesting bit strings of length n ending with a 1 to get all interesting bit strings of length n that end with a 0. So,

$$f_0(n) = f_0(n-1) + f_0(n-2).$$

These are just Fibonacci numbers.

(d) [3 marks] Write a function `interesting(n)` that accepts a single integer $n \geq 1$ as a parameter and returns the number of interesting bitstrings of length exactly n (including ones that end with a 1). For example `interesting(3)` should return 6. For full marks, a reasonable implementation of the function you describe would compute `interesting(1000)` in less than one second. Do not worry about overflow if you provide the implementation in a language that only supports fixed-size integers.

```
def interesting(n):
    f = {1:1, 2:2}

    for i in range(3, n+1):
        f[i] = f[i-1] + f[i-2]

    # double the answer because f[n] is just f_0(n) from the exercises
    return 2*f[n]
```

(d) [3 marks] Now consider bitstrings with unknown entries: some characters are `*`. Write a function `can_complete(bitstring)` that takes a single string that you may assume only contains characters `0`, `1`, and `*`.

The function should return `True` if it is possible to replace each `*` character with a `0` or `1` so the resulting bitstring is interesting, otherwise the function should return `False`. Again, for full marks a reasonable implementation of the function you describe would finish in less than one second even if the string had 1000 characters.

examples

- `can_complete("00**1")` returns `True`. Replacing the first `*` with `1` and the second with `0` results in the interesting string `00101`.
- `can_complete("00*11")` returns `False` because any setting of `*` would result in a bitstring that is not interesting.
- `can_complete("001*0*1*")` returns `True`. There is more than one way we could replace the `*` characters with bits to get an interesting string: one such way is `00110011`.

Solution
This was, by far, the most challenging question in the exam. There are a number of approaches, here is one.
Repeat the following until neither applies.

- If the string already contains `000` or `111`, then return `False`.
- If the string contains `*00` or `*11`, then replace the `*` with the only valid character (eg. for `*00`, replace `*` with `1`).

Once this phase is done, just return `True` if we did not already return `False`.

To see why we could return `True`, just greedily fill in the remaining asterisks in a left-to-right fashion, where each asterisk would be replaced by the bit that is different than it's preceding bit (and if `*` is at the start, pick anything). For example, if the string is `00*10**1` then the first `*` would be replace by `1` (because it follows a `0`).

Such a replacement cannot create a triple to the left of the old `*` because the character is different than the preceding character. It cannot create a triple to the right of the old `*` because we exited the “repeat the following...” loop.

So this process will successfully fill in all `*` to get an interesting string. Of course, since you just needed to return `True` or `False` you did not actually have to do this. This is just an argument justifying why you could return `True`.

The code below does the loop slightly differently, but it results in the same thing: a string with no `*00` or `*11` if it does not fail.

```
def can_complete(bitstring):
    # ‘processed’ will eventually be ‘bitstring’ with asterisks replaced
    # so *00 or *11 is not a substring, it is built "right-to-left"
    processed = ""

    for c in bitstring[::-1]: # process characters in reverse order
        if c == "*" and processed[:2] == "00":
            processed = "1"+processed
        elif c == "*" and processed[:2] == "11":
            processed = "0"+processed
        else:
            processed = c+processed
            if processed[:3] == "000" or processed[:3] == "111":
                return False
    return True
```