

Interview questions for programmers (& my
answers to them)

Gene Michael Stover

created Thursday, 2007 October 18
updated Monday, 2008 June 2

Copyright © 2007–2008 Gene Michael Stover. All rights reserved. Permission to copy, store, & view this document unmodified & in its entirety is granted.

Contents

1	What is this?	5
2	Find one by occurrences	7
3	Rotate a string	9
4	Distribute spaces equally in a string	11
5	Reverse the words in a string	13
A	Other File Formats	17

Chapter 1

What is this?

I enjoy solving programming problems that are often given in job interviews. Here are some I've seen or that people have told me. Here are also my answers to them (though not always the answers I produced during the interview).

Chapter 2

Find one by occurrences

Problem: Given an array of N integers, find one which occurs at least X times.

1. I suspect the fastest solution at run-time is to construct a dictionary of the keys & their number of occurrences, as you would for a bucket sort. Unlike the bucket for a bucket sort, you could exit as soon as you know that some integer occurs X times.
2. In a language where you can implement the dictionary easily, it shouldn't be a difficult program to write. Since the values might vary wildly, your best bet might be a hash table or tree instead of a true array.
3. Here's an implementation in Lisp:

```
(defun fnf fn (a x)
  "Return some integer in array A which occurs at
  least X times. If there is no such integer,
  return NIL."
  (declare (type array a) (type (integer 0) x))
  (let ((ht (make-hash-table)))
    (dotimes (i (length a))
      (incf (gethash (aref a i) ht) 0))
      (when (<= x (gethash (aref a i) ht))
        (return-from fnf (aref a i))))
    nil)
```

4. Test it on (10 11 12 11 13) where $X = 2$. It should return 11, since that's the only value which occurs two or more times.

```
> (fnf # (10 11 12 11 13) 2)
11
```

5. Test it on an empty array.

```
> (fnfnfn #() 10)
NIL
```

6. Test it when two numbers occur at least X times. We could return any of the values which occur at least X times, but we know that our function will return the first one which occurs X times.

```
> (fnfnfn #(1 2 1 2 1 2 1) 3)
1
```

7. Test it when no numbers occur X or more times.

```
> (fnfnfn #(1 2 3 1 1 1) 100)
NIL
```

8. Since lookups in a hash table are $O(K)$, & we visit each element in the array at most once, the solution is $O(KN)$. If, as with searching through an unsorted list, we find a satisfactory element in position $\frac{N}{2}$ on average, the solution is $O(KN/2)$, but that's still $O(KN)$, which is still $O(N)$.
9. If we implemented the same solution in C++ with STL's class template `map`, the worst-case lookup would be $O(\log N)$, & that particular cost would occur only if all N values were unique. So this same algorithm in C++ would be better than $O(N \log N)$.

Chapter 3

Rotate a string

A coworker gave me this one. I think he encountered it on an interview at Microsoft.

Problem: Given a string of N characters, rotate it by factor I .
For example if the string is “abcdefgh”, $N = 8$, rotate by $I = 3$,
the output is “defghabc”.

In Lisp:

```
(defun rotate-string (x count)
  (do ((y nil (cons (char x (mod (+ i count) (length x))) y))
      (i 0 (1+ i)))
      ((<= (length x) i) (coerce (reverse y) 'string))))
```

No solution is complete without tests:

```
(defun test-rotate-string ()
  (assert (equal "defghabc" (rotate-string "abcdefgh" 3)))
  (assert (equal "abcdefgh" (rotate-string "abcdefgh" 0)))
  (assert (equal "fghabcde" (rotate-string "abcdefgh" -3)))
  (assert (equal "abcdefgh" (rotate-string "abcdefgh" 8)))
  'test-rotate-string)
```

Easy.

In Delphi Pascal:

```
function RotateString (s : string; i : integer) : string;
var
  x : string;
  j, k : integer;
begin
  if s = '' then begin
```

```
    // The input string is the empty string, so we do nothing.
    Result := ''
end else if i = 0 then begin
    // The rotation amount is zero, so we do nothing.
    Result := ''
end else begin
    SetLength (x, Length (s));
    for j := 1 to Length (s) do begin
        k := (i + j) mod Length (s);
        k := k + 1;
        WriteLn ('x[' , k , '] gets s[' , j , '].');
        x[k] := s[j]
    end;
    Result := x
end;
end;
```

Chapter 4

Distribute spaces equally in a string

Problem: Distribute spaces equally in a string.

For example, if the string is “it####rains#####in#####Seattle####” & using # as white-space so we can see our spaces, the output should be “it#####rains#####in#####Seattle”. Notice that the spaces are equally distributed among the words.

The example contains 21 spaces & four words, so the spaces are divided into three sections. Three divides 21 integrally, but what if the number of sections doesn't divide the number of spaces integrally? I'd bet that, after making an initial attempt at a solution in an interview, the interviewer would ask this question.

In Lisp:

```
(defvar *space-char* #\Space)

(defun read-token (strm)
  "Return next token or NIL"
  ;; Consume leading spaces
  (loop while (eql (peek-char nil strm nil) *space-chars*)
    do (read-char strm nil))
  ;; Accumulate until next character is a space or end-of-input
  (do ((lst nil (cons c lst))
      (c (read-char strm nil) (read-char strm nil))
      (end (list nil *space-char*)))
      ((member c end)
       ;; At end of input, you get NIL.
       ;; Otherwise, you get the token as a string.
       (and lst (coerce (reverse lst) 'string)))))
```

```

(defun tokenize (x)
  "Return list of tokens in the string"
  (with-input-from-string (strm x)
    (do ((tokens nil (cons token tokens))
        (token (read-token strm) (read-token strm)))
        ((null token) (reverse tokens))))))

(defun count-spaces (x) (count *space-char* x))

(defun distrib (x)
  "Return new string with redistributed spaces, same
total number of spaces"
  (let* ((tokens (tokenize x))
        (gaps (1- (length tokens)))
        (count (count-spaces x))
        (spaces (coerce
                  (loop for i from 0 below (floor (/ count gaps))
                        collect (first *space-chars*)
                        'string))
              (rem (mod count gaps))))
    (format t "~&";; ~A is ~S." 'tokens tokens)
    (format t "~&";; ~A is ~S." 'gaps gaps)
    (format t "~&";; ~A is ~S." 'count count)
    (format t "~&";; ~A is ~S." 'spaces spaces)
    (format t "~&";; ~A is ~S." 'rem rem)
    (with-output-to-string (strm)
      (format strm "~A" (first tokens))
      (dolist (token (rest tokens))
        (format strm "~A~A~A"
                  spaces
                  (cond ((plusp rem) (decf rem) (first *space-chars*))
                        (t ""))
                  token))))))

```

This DISTRIB function is too long for me to be very happy with it, though the other functions aren't too bad.

```

(defun test-distrib ()
  (let ((*space-char* #\#))
    ;; example from the problem statement
    (assert (equal "it#####rains#####in#####Seattle"
                  (distrib "it####rains#####in#####Seattle####")))
    ;; 22 spaces -- indivisible by 3
    (assert (equal "it#####rains#####in#####Seattle"
                  (distrib "it####rains#####in#####Seattle####"))))
  'test-distrib)

```

Chapter 5

Reverse the words in a string

For some reason I can't guess, I encountered this problem no fewer than three times while interviewing for a contract at Microsoft in 2008 February.

Problem: Given a string containing words, reverse the words in the string.

For example, convert “one two three four” to “four three two one”.

The interviewers didn't say whether words were always separated by a single space or could be separated by multiple spaces. Neither did they specify what to do to groups of spaces (if they were allowed): reverse the groups; preserve the groups, reversing the words between them?

Since they didn't say, & since it was easier on me if single spaces always separated words, I decided to assume single spaces *so I said so*.¹

If you assume single spaces between words (or that you can convert multiple consecutive spaces to a single space on output), the main difficulty in this problem is tokenizing. In a high level language, it's easy.

Here it is in Lisp:

```
(defun consume-spaces (strm)
  "Consume characters until the next character isn't a space
or we're at end-of-input"
  (loop while (eql (peek-char nil strm nil) #\Space)
    do (read-char strm nil)))

(defun read-token (strm)
  (consume-spaces strm))
```

¹Interviewers often leave things unspecified to see when you'll realise it's unspecified & to see what you do when you realise it.

```

;; Accumulate until next character is a space or end-of-input
(do ((lst nil (cons c lst))
    (c (read-char strm nil) (read-char strm nil))
    (end (list nil #\Space)))
    ((member c end)
     (and lst (coerce (reverse lst) 'string)))))

(defun tokenize (x)
  "Return list of tokens in the string"
  (with-input-from-string (strm x)
    (do ((tokens nil (cons token tokens))
        (token (read-token strm) (read-token strm)))
        ((null token) (reverse tokens)))))

(defun reverse-words (x)
  (format nil "~{~A~^ ~}" (reverse (tokenize x))))

(defun test-rotate-words ()
  (assert (equal "four three two one"
                (reverse-words "one two three four")))
  ;; works with a single word
  (assert (equal "one"
                (reverse-words "one")))
  ;; How about a trailing space? Is this correct?
  ;; The problem didn't say, & I think I never asked.
  (assert (equal "one"
                (reverse-words "one ")))
  (assert (equal "two one"
                (reverse-words "one two ")))
  ;; Leading spaces. Is this correct? Problem didn't say.
  (assert (equal "two one"
                (reverse-words " one two")))
  'test-rotate-words)

```

If you can see C++ & the STL (which I couldn't, said all the interviewers), you get a similar solution.

```

#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using std::cout;
using std::istringstream;
using std::ostringstream;
using std::string;
using std::vector;

```

```

/*
 */
static vector<string>
S_Tokenize (string const &x)
{
    vector<string> tokens;
    istringstream strm (x);
    string token;

    while (strm >> token) tokens.push_back (token);
    return tokens;
}

/*
 */
static string
S_ReverseWords (string const &x)
{
    ostringstream strm;
    vector<string> tokens (S_Tokenize (x));
    size_t i;
    bool is_first = true;
    // Gather the tokens from tokens[] starting at the end,
    // as if it were a stack. That reverses the tokens.
    for (i = tokens.size (); 0 < i; --i) {
        if (is_first) is_first = false;
        else strm << " ";
        strm << tokens[i-1];
    }
    return strm.str ();
}

/*
 */
static bool
S_Test (string const &input, string const &expect)
{
    bool is_good;
    string result;

    result = S_ReverseWords (input);
    is_good = expect == result;
    if (!is_good) {
        cout << "error: S_ReverseWords (\\"" << input << "\")\n"
             << "    returned \"" << result << "\",\n"

```

```
        << "    expected \"" << expect << "\".\n";
    }
    return is_good;
}

int
main ()
{
    bool is_good =
        // basic case
        S_Test ("one two three four", "four three two one") &&
        // one word
        S_Test ("one", "one") &&
        // two words
        S_Test ("one two", "two one") &&
        // odd number of words
        S_Test ("one two three", "three two one") &&
        // empty input
        S_Test ("", "");

    return is_good ? 0 : 1;
}
```

Appendix A

Other File Formats

- This document is available in multi-file HTML format at <http://cybertiggyr.com/gene/iq/>.
- This document is available in Pointless Document Format (PDF) at <http://cybertiggyr.com/gene/iq/iq.pdf>.

Bibliography