Analysis of Puzzles

Vol. 3: Programming Puzzles

Umesh Nair

Introduction

Solving puzzles has been my favorite hobby since childhood. Initially, puzzles came in the form of questions and answers. Nobody thought about *how* to find the answer. After learning a little mathematics, I started to solve some of these puzzles by some systematic way. Some were solved by trial and error, but whenever I came across a method by which a puzzle could be solved systematically, I was delighted.

This delight increased when I met smarter people and watched them solve the same puzzles in a different, often more elegant, way. Watching different methods leading to the same solution was a really great thing.

At some point, I started to collect some of the puzzles I came across with *all* the solutions I heard. This is that collection.

I do not know the source of most of these puzzles. I mentioned from where I heard it first, if I remember.

The solver's name is mentioned with most of the solutions. All solutions without a solver's name mentioned are mine. However, this does not mean that they are my *original* solutions. Some of them are; others I read or heard somewhere and just reproduced here.

Another goal of this work is to try to find the most general solution to many popular puzzles for which we normally heard only the particular problem.

The book is divided into three volumes:

Volume 1: Simple Puzzles that can be understood and solved by simple logic and High School Mathematics.

Volume 2: Mathematical Puzzles that require specialized knowledge in some branch of Mathematics.

Volume 3: Programming Puzzles related to computer programming and algorithms.

This is an ongoing effort. Please send your comments to `umesh.p.nair@gmail.com`.

Umesh Nair
Portland, OR, USA.

Contents

Chapter 1

Questions

Puzzle 1 Loop in linked list (*Solution : Ch. 2, page 11*)

You are given the address of the first node of a singly-linked list. The list may end in `NULL`, or it may have a loop (cycle) in it, so that one of the later elements point to an element already occurred, forming a loop, making traversing infinite.

Your task is to find whether the linked list has a loop or not (it is easy to see that it will not have more than one loops), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the list, and time is measured as the number of jumps from one node to another.

```
typedef struct Node_t {
   int x;
   struct Node_t *next;
} Node;

const Node* has_loop (const Node* head);
```

The function `has_loop` should return `NULL` if the list ends in `NULL` or a node in the loop if there is a loop.

Puzzle 2 Loop-head in linked list (*Solution : Ch. 3, page 15*)

You are given the address of the first node of a singly-linked list. The list may end in `NULL`, or it may have a loop (cycle) in it, so that one of the later elements point to an element already occurred, forming a loop, making traversing infinite.

Your task is to find the head of the loop, if any, in the linked list (it is easy to see that it will not have more than one loops), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the list, and time is measured as the number of jumps from one node to another.

```
typedef struct Node_t {
   int x;
   struct Node_t *next;
} Node;

Node* get_loop_head (Node* head);
```

The function `get_loop_head()` should return `NULL` if the list ends in `NULL` or the first node in the loop if there is a loop.

Puzzle 3 Repeated value in array (*Solution : Ch. 4, page 21*)

A read-only array of n elements (assume a 0-based array) contains numbers from 1 to $(n-1)$, not necessarily all of them. So, by pigeon-hole principle, there must be at least one number repeated. The task is to find one such number (i.e., the value), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the array, and time is measured as the number of array element accesses.

Examples are given below with an array of 9 elements.

Array	Value
6, 4, 3, 8, 4, 2, 1, 5, 7	4
6, 4, 3, 6, 4, 4, 3, 5, 7	6, 4 or 3

Puzzle 4 Bringing zeros to front (*Solution : Ch. 3, page 15*)

Implement the following C function.

```
void bringZerosToFront(int array[], int arraySize);
```

The array `array` contains integers that may be positive, negative or zero. `arraySize` is the size of the array. The task is to change the array in place. so that all zeros are brought to the beginning of the array, and the values in those elements are pushed to some elements in the back. In the end, no element should be lost, but may be re-ordered. All zeros should be in the beginning, and the order of the remaining elements does not matter. Do it in minimum number of data assignments.

Puzzle 5 Reversing bits (*Solution : Ch. 6, page 29*)

Implement the following C function.

```
void bringZerosToFront(int array[], int arraySize);
```

The array `array` contains integers that may be positive, negative or zero. `arraySize` is the size of the array. The task is to change the array in place. so that all zeros are brought to the beginning of the array, and the values in those elements are pushed to some elements in the back. In the end, no element should be lost, but may be re-ordered. All zeros should be in the beginning, and the order of the remaining elements does not matter. Do it in minimum number of data assignments.

Puzzle 6 Get hundred (*Solution : Ch. 7, page 31*)

Insert any number of +s and −s between the digits of, or in front of the first digit, of

123456789

to get a result of 100.

Write a computer program to find all solutions. Will the solution be more complicated if the multiplication sign ∗ also is allowed? What if a power sign ^ also allowed?

Puzzle 7 Circle from three points (*Solution : Ch. 8, page 39*)

Insert any number of +s and −s between the digits of, or in front of the first digit, of

123456789

to get a result of 100.

Write a computer program to find all solutions. Will the solution be more complicated if the multiplication sign ∗ also is allowed? What if a power sign ^ also allowed?

Chapter 2

Loop in linked list

2.1 Question

You are given the address of the first node of a singly-linked list. The list may end in `NULL`, or it may have a loop (cycle) in it, so that one of the later elements point to an element already occurred, forming a loop, making traversing infinite.

Your task is to find whether the linked list has a loop or not (it is easy to see that it will not have more than one loops), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the list, and time is measured as the number of jumps from one node to another.

```
typedef struct Node_t {
   int x;
   struct Node_t *next;
} Node;

const Node* has_loop (const Node* head);
```

The function `has_loop` should return `NULL` if the list ends in `NULL` or a node in the loop if there is a loop.

2.2 Solution

2.2.1 Algorithm

This can be found by using two pointers, one advancing one at a time, and the other advancing two at a time, and checking whether they ever meet. Algo-

rithm 1 illustrates this.

Algorithm 1 CHECKFORCYCLE(`head`)

```
1:  /* Finds whether the linked list starting at head has a cycle */
2:  p1  ⇐ head
3:  p2  ⇐ head
4:  loop
5:     Advance p2
6:     if p2 == NULL then
7:        return NULL
8:     end if
9:     Advance p2
10:    if p2 == NULL then
11:       return NULL
12:    end if
13:    Advance p1
14:    /* No need to check for NULL, because p1 is slower than p2 */
15:    if p1 == p2 then
16:       return p1
17:    end if
18: end loop
```

2.2.2 Complexity analysis

If there is no loop, `two` will find it in n jumps. By the time, `one` will jump $\lfloor \frac{n-1}{2} \rfloor$ times. So, the total number of jumps is $\lfloor \frac{3n-1}{2} \rfloor$, which is $O(n)$.

If there is a loop, let the length of the stem is m and the length of the loop is p. $0 \leq m < n$, $0 < p \leq n$, $m + p = n$. Also let $m = kp + c$, where k is a non-negative integer and $0 \leq c < p$.

Now, when `one` reaches the head the loop, it has jumped m times, and `two` $2m = m + kp + c$ times, so that it is c nodes ahead of `one`, and will take $(p - c)$ steps to catch up with `one`.

Total number of jumps by `two` $= m + kp + c + 2(p - c) = m + (k + 2)p - c$.

Total number of jumps by `one` $= m + p - c$.

Total number of jumps $= 2m + (k + 3)p - 2c = 3m + 3p - 3c = 3(m + p - c)$. This is also $O(n)$, since neither of m, p and c is greater than n.

2.2.3 Implementation

The C implementation is given in Figure 2.1 on the next page.

Figure 2.1: Listing of `has_loop()`

```
Node* has_loop(const Node* head)
{
   if (!head) return 0;
   const Node* one = head;
   const Node* two = head;

   /* Now, check whether they will ever meet */
   do {
      /* Advance 'two' two times */
      two = two->next;
      if (two == 0) {
         return 0;
      }
      two = two->next;
      if (two == 0) {
         return 0;
      }
      /* Advance 'one' one time  */
      one = one->next;
   } while (one != two);
   /* Now, one and two point to a node in the loop */
   return one;
}
```

Figure 2.2 has a less verbose version* of the above function.

Figure 2.2: Listing of `has_loop1()`

```
const Node* has_loop1(const Node* head)
{
  const Node *a = head, *b = (head ? head->next : 0);
  for ( ; b && b->next ; a = a->next, b = b->next->next)
    if (a == b || a == b->next) return a;
  return 0;
}
```

*Due to John Hershberger.

Chapter 3

Loop-head in linked list

3.1 Question

You are given the address of the first node of a singly-linked list. The list may end in `NULL`, or it may have a loop (cycle) in it, so that one of the later elements point to an element already occurred, forming a loop, making traversing infinite.

Your task is to find the head of the loop, if any, in the linked list (it is easy to see that it will not have more than one loops), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the list, and time is measured as the number of jumps from one node to another.

```
typedef struct Node_t {
   int x;
   struct Node_t *next;
} Node;

Node* get_loop_head (Node* head);
```

The function `get_loop_head()` should return `NULL` if the list ends in `NULL` or the first node in the loop if there is a loop.

3.2 Solution 1

3.2.1 Algorithm

This solution* contains three steps.

*Due to John Hershberger.

1. Get a node in the cycle.

   Use Algorithm 1. The function returns when the two pointers are the same, and that node is in the cycle.

2. Find the cycle length.

   Keep one pointer at the node in the loop, and let the other advance one node at a time, till they meet again, and count how many nodes it covered.

   Algorithm 2 illustrates this procedure.

Algorithm 2 FindCycleLength(p)

```
1: /* Finds the cycle length, given a point on the cycle */
2: p1  ⇐ p
3: n   ⇐ 0
4: repeat
5:    Advance p1
6:    Increment n
7: until p == p1
8: return n
```

3. Find the start of the cycle.

   Start with one pointer at the beginning, and the other k nodes ahead, where k is the cycle length found using Algorithm 2, and advance both pointers one at a time. The node at which they meet is the start of the cycle. Algorithm 3 illustrates this.

Algorithm 3 FindStartOfCycle(head, cycleLength)

```
1: /* Finds the start of a cycle of length cycleLength in a linked list starting
   at head */
2: p1  ⇐ head
3: p2  ⇐ head
4: for i in 1 TO cycleLength do
5:    Advance p2
6: end for
7: while p1 <> p2 do
8:    Advance p1
9:    Advance p2
10: end while
11: return p1
```

3.2.2 Complexity analysis

As we saw in section 2.2.2, step 1 will take $3(m + p - c)$ jumps, where m is the length of the stem, p is the length of the loop and $c = (m \mod p)$.

Figure 3.1: Listing of `get_loop_head()`

```
const Node* get_loop_head(const Node* head)
{
   const Node* p = has_loop(head); /* Figure 2.1 on page 13 */
   if (!p) return 0;
   const Node* q = p;
   int loop_length = 0;
   do {
      ++loop_length;
      q = q->next;
   } while (p != q);

   return get_loop_start(head, loop_length);
}

const Node* get_loop_start(const Node* head, int loop_length)
{
   const Node* p = head;
   const Node* q = head;
   int i=0;
   for (; i<loop_length; ++i) {
      q = q->next;
   }
   while (p != q) {
     p = p->next;
     q = q->next;
   }
   return p;
}
```

Step 2 will take p steps, and step 3 will take $p + m + m = 2m + p$ jumps.

So, total number of jumps is $3m + 3p - 3c + p + 2m + p = 5m + 5p - 3c$ steps.

This is also $O(n)$.

3.2.3 Implementation

The C implementation is given in Figure 3.1.

3.3 Solution 2

3.3.1 Algorithm

This solution* is more effiecient than the previous one.

We saw in section 2.2.2 that when the two pointers meet in the loop, `two` is c nodes ahead of `one`, and they will meet in another $(p - c)$ steps. This means a pointer at this position will reach the head of the loop in another c steps. In general, it will be at the head after $lp + c$ steps, where l is an integer. In particular, it will be at the head after $m = kp + c$ steps. This leads to this solution.

1. Get a node in the cycle using Algorithm 1.
2. Keep another pointed at the head of the list. Advance both pointers together, one at a time, till they coincide.
3. When they coincide, that will be the head of the loop.

Algorithm 4 FindHeadOfCycleInLinkedList(`head`)

```
1: /* Returns NULL if there is no cycle, or the node that starts the cycle */
2: p  ⇐ CheckForCycle(head) /* Alg: 1 */
3: if p == NULL then
4:    return NULL
5: end if
6: q  ⇐ head
7: while p <> q do
8:    Advance p
9:    Advance q
10: end while
11: return q
```

3.3.2 Complexity analysis

Number of steps $= 3(m + p - c) + 2m = 5m + 3(p - c)$.

Compared to the previous solution, which takes $5m + 5p - 3c$ jumps, this takes $2p$ less jumps. I haven't seen a more efficient algorithm for this problem.

This is also $O(n)$.

*Due to Cibu C. J..

Figure 3.2: Listing of `get_loop_head2()`

```
const Node* get_loop_head2(const Node* head)
{
   const Node* p = has_loop(head); /* Figure 2.1 on page 13 */
   if (!p) return 0;
   const Node* q = head;
   while (p != q) {
      p = p->next;
      q = q->next;
   }
   return q;
}
```

3.3.3 Implementation

The C implementation is given in Figure 3.2.

Chapter 4

Repeated value in array

4.1 Question

A read-only array of n elements (assume a 0-based array) contains numbers from 1 to $(n-1)$, not necessarily all of them. So, by pigeon-hole principle, there must be at least one number repeated. The task is to find one such number (i.e., the value), in linear ($O(n)$) time and constant ($O(1)$) space, where n is the number of elements in the array, and time is measured as the number of array element accesses.

Examples are given below with an array of 9 elements.

Array	**Value**
6, 4, 3, 8, 4, 2, 1, 5, 7	4
6, 4, 3, 6, 4, 4, 3, 5, 7	6, 4 or 3

4.2 Solution

4.2.1 Algorithm

Since the elements of the array can take only values within the range $1-(n-1)$, we can make a linked list from the array-elements by going to the index given by the value in a particular element (assuming a one-based array). That way, when an element is repeated, the list will create a loop at that element, and value in that element will give the repeated number. This element can be located by the solution of the previous puzzle.

Note that the linked list formed by this method may not include all the elements in the array. However, it will include one repeated element, and that is sufficient

for our solution. This cannot be used to find *all* repeating elements in the array.

One problem here is, what should be the head of the linked list? If we select, for example, the i^{th} element as the head, and if the i^{th} element's value is i, the algorithm will wrongly identify it as the repeated element. We need to start on an element whose index cannot be the value in another element. In zero-based arrays, the only element with this property is at index 0.*

Now, let us translate the meaning of traversing the linked list into accessing the array elements. This is simpler than the other algorithms, because we know that there is a repeated number, so we need not check for NULL as in the other puzzles.

1. Instead of two pointers, we'll use two indexes.
2. `p = p->next` translates to `i = A[i]`, and `q = q->next->next` translates to `q = A[A[q]]`.

Algorithm 5 FINDREPETITION(A)

```
1: /* Finds a repeated value in the array A */
2: /* Translation of Algorithm 1. This is needed to locate an element in the
   loop. */
3: p1 ⇐ 0
4: p2 ⇐ 0
5: repeat
6:    p2 ⇐ A[A[p2]]
7:    p1 ⇐ A[p1]
8: until p1 == p2
9: /* Translation of Algorithm 4. This is the most efficient way to locate the
   head of the loop. */
10: p2 ⇐ head
11: while p1 <> p2 do
12:    Advance p1
13:    Advance p2
14: end while
15: return p1
```

4.2.2 Complexity analysis

The complexity of this algorithm is essentially the same as that of Algorithm 4 discussed in §3.3.2 on page 18.

*For one-based arrays, this is at index n. Everything else is the same.

Figure 4.1: Listing of `repeated_element()`

```
int repeated_element(int A[])
{
   /* Translation of Algorithm 1 */
   int p = 0;
   int q = 0;
   do {
      q = A[A[q]];
      p = A[p];
   } while (p != q);

   /* Translation of Algorithm 4 */
   q = 0;
   while (p != q) {
      p = A[p];
      q = A[q];
   }

  return p;
}
```

4.2.3 Implementation

A C implementation is given in Figure 4.1. An implementation in `FORTRAN`, which uses 1-based arrays, is given in Figure 4.2 on the next page.

Figure 4.2: Listing of `REPVAL`

```
FUNCTION REPVAL(N, A)
DIMENSION A(N)

I1 = N
I2 = N
DO
  I2 = A(A(I2));
  I1 = A(I1);
WHILE (I1 .NE. I2)

I2 = N
DO WHILE (I1 .NE. I2)
   I1 = A(I1)
   I2 = A(I2)
END DO

REPVAL = I1
RETURN
```

Chapter 5

Bringing zeros to front

5.1 Question

Implement the following C function.

```
void bringZerosToFront(int array[], int arraySize);
```

The array `array` contains integers that may be positive, negative or zero. `arraySize` is the size of the array. The task is to change the array in place. so that all zeros are brought to the beginning of the array, and the values in those elements are pushed to some elements in the back. In the end, no element should be lost, but may be re-ordered. All zeros should be in the beginning, and the order of the remaining elements does not matter. Do it in minimum number of data assignments.

5.2 Solution 1

5.2.1 Algorithm

This can be achieved by keeping the position of the first non-zero element in a variable, and whenever a zero element is found, swapping it with that. Also, swapping doesn't need a third temporary variable, because one of them is zero.

Figure 5.1 on the next page shows the implementation.

Figure 5.1: Listing of `bringZerosToFront()`

```
void bringZerosToFront(int array[], int arraySize)
{
   int firstNonZero = 0;
   int curr = 0;
   while (curr < arraySize) {
      if (array[curr] == 0) {
         if (curr == firstNonZero) {
            ++firstNonZero;
         } else {
            array[curr] = array[firstNonZero];
            array[firstNonZero++] = 0;
         }
      }
      ++curr;
   }
}
```

5.2.2 Complexity analysis

The number of swaps is equal to the number of zeros that are not in the beginning. One swap involves two assignments.

5.3 Solution 2

5.3.1 Algorithm

This solution* implements the same algorithm in a slighly different way. Zeros and non-zeros are skipped from the beginning and end of the array, and the first pair of elements obtained are swapped. The same procedure is repeated for the sub-array.

Figure 5.2 on the next page shows the implementation.

5.3.2 Complexity analysis

The number of swaps is equal to the number of zeros that are not in the beginning. One swap involves two assignments.

*Due to Cibu C. J..

Figure 5.2: Listing of `bringZerosToFront2()`

```
void bringZerosToFront2(int array[], int arraySize)
{
   int i = 0;              /* Finding non-zeros from start */
   int j = arraySize - 1;  /* Finding zeros from end */
   for (;;) {
      /* Skip zeros from left */
      for (; array[i] == 0 && i < j; ++i);

      /* Skip non-zeros from right */
      for (; array[j] != 0 && i < j; --j);

      /* We got a non-zero and a zero */
      if (i < j) {
         /* Swap */
         array[j--] = array[i];
         array[i++] = 0;
      } else {
         /* We are done */
         break;
      }
      /* Continue till both pointers meet */
   }
}
```

Chapter 6

Reversing bits

6.1 Question

Reverse the bits in a word of 2^k bits, addressable in one instruction, in $O(k)$ time and constant space.

In other words, reverse the bits in a word of n bits, where n is a power of two, in $O(log(n))$ time and constant space.

6.2 Solution

Figure 6.1 on the next page shows a C++ function to do this.

6.3 Illustration and explanation

Figure 6.1: Listing of `reverse_bits()`

```
template <typename IntType>
void reverseBits(IntType& n)
{
   int wordSize = sizeof (IntType) * CHAR_BIT;
   IntType a = (IntType) ~0;
   while (wordSize > 0) {
      wordSize >>= 1;
      a ^= (a << wordSize);
      IntType b = (IntType) ~a;
      n = ((n >> wordSize) & a) | ((n << wordSize) & b);
   }
}
```

Chapter 7

Get hundred

7.1 Question

Insert any number of +s and −s between the digits of, or in front of the first digit, of

123456789

to get a result of 100.

Write a computer program to find all solutions. Will the solution be more complicated if the multiplication sign ∗ also is allowed? What if a power sign ^ also allowed?

7.2 Solution

One elegent approach* is to use a recursive procedure to build an expression and then evaluate it. Algorithm 6 illustrates this.

Algorithm 6 OneToNine()

```
1: /* Solve the puzzle of 1 to 9 */
2: /* Handle the case that starts with "1" */
3: Solve(2, "1") /* Alg: 7 */
4:
5: /* Handle the case that starts with "-1" */
6: Solve(2, "-1") /* Alg: 7 */
```

*Due to Cibu C. J..

The recursive function is given in Algorithm 7.

Algorithm 7 SOLVE(digit, str)

```
1: /* Recursively form expression and evaluate it */
2: operators  ⇐ {"", "+", "-"}
3: if digit == 10 then
4:    if evaluate(str) == 100) then
5:       Output str
6:       return
7:    end if
8: end if
9: for each op in (operators) do
10:    SOLVE(digit+1, str + op + toString(digit)) /* Alg: 7 */
11: end for
```

Implementing this in an interpreted language like PERL that has a builtin *evalauate()* function to evaluate expressions is easy. A direct translation to PERL is given in Figure 7.1.

Figure 7.1: Listing of perl solve() – version 1

```
sub solve {
   my ($i, $str) = @_;
   if ($i == 10) {
      printf "$str\n" if (eval($str) == 100);
      return;
   }
   foreach $op ("", "+", "-", "*", "**") {
      solve($i+1, "$str$op$i");
   }
}

solve(2, "1");
solve(2, "-1");
```

Figure 7.2 on the next page is another way to write this.*

*This is Cibu's original code.

Figure 7.2: Listing of perl solve() – version 2

```
sub solve {
   my ($i, $str) = @_;
   if ($i == 10) {
      printf "$str\n" if (eval($str) == 100);
      return;
   }
   foreach $op ("", "+", "-", "*", "**") {
      solve($i+1, "$str$op$i") unless ($i == 1 && $op =~ /[+*]/);
   }
}

solve(1, "");
```

This can be implemented in any language. Figure 7.3 on the next page is a C++ version.

but the tricky part is to find a way to evaluate the expression. If the operators are restricted to + and - only, Figure 7.4 on page 35 gives a solution.

This function gets more complicated is multiplication also is included, and still more complex if other operators like exponentiation is included. We'll need an expression-parser to solve the problem then.

Figure 7.5 on page 36 is an alternate elegent solution to do it in C*, making use of the power of the `printf()` function.

Figure 7.6 on page 37 is a modification of Figure 7.5 on page 36, handling multiplication as well.†

*Due to Cibu C. J..

†Cibu claims this can handle exponentiation also if it associates from right to left.

Figure 7.3: Listing of C++ solve()

```
#include <iostream>
#include <iomanip>

using std::cout;
using std::string;
using std::endl;


void solve(int n, string str)
{
   static const string ops[] = {"", "+", "-"};

   if (n == 10) {
      if (evaluateExpression(str) == 100) {
         cout << str << " = 100" << endl;
      }
      return;
   }
   for (int i = 0; i < sizeof ops / sizeof (string) ; ++i) {
      solve(n+1, (str + ops[i] + ('0' + n)));
   }
}

void solveProblem()
{
   solve(2, "1");
   solve(2, "-1");
}

int main()
{
   solveProblem();
   return 0;
}
```

Figure 7.4: Listing of C++ evaluateExpression()

```
int evaluateExpression(const string& str)
{
   int result = 0;
   int number = 0;
   int placeValue = 1;

   string::const_iterator it = str.begin() + str.length();
   for (; it >= str.begin(); --it) {
      char c = *it;
      if (c >= '0' & c <= '9') {
         number += placeValue * (c - '0');
         placeValue *= 10;
      } else if (c == '+') {
         result += number;
         number = 0;
         placeValue = 1;
      } else if (c == '-') {
         result -= number;
         number = 0;
         placeValue = 1;
      }
   }
   result += number;
   return result;
}
```

Figure 7.5: Listing of C solve()–version 1

```
#include <stdio.h >

char str[20];

void
recur( int nextint, int left, int sign, int right, int next_pos )
{
   int cur_value = left + sign * right;

   switch (nextint) {
   case 10:
      if (cur_value == 100) {
         printf("%s\n", str);
      }
      return;

   default:
      sprintf(&str[next_pos], "+%d", nextint);
      recur(nextint+1, cur_value,  1, nextint, next_pos + 2);

   case 1:
      sprintf(&str[next_pos], "-%d", nextint);
      recur(nextint+1, cur_value, -1, nextint, next_pos + 2);

      sprintf(&str[next_pos], "%d", nextint);
      recur(nextint+1, left, sign, 10*right + nextint,
            next_pos + 1);
   }
}

int
main()
{
   recur(1, 0, 1, 0, 0);
}
```

Figure 7.6: Listing of C solve()–version 2

```
#include <stdio.h>

char str[20];

void
recur( int nextint, int a, int s, int b, int c, int next_pos )
{
   int cur_value = a + s*b*c;
   char *strp = &str[next_pos];

   switch (nextint) {
   case 10:
      if (cur_value == 100) {
         printf("%s\n", str);
      }
      return;

   default:
      sprintf(strp, "+%d", nextint);
      recur(nextint+1, cur_value,  1, 1, nextint, next_pos + 2);

      sprintf(strp, "*%d", nextint);
      recur(nextint+1, a, s, b*c, nextint, next_pos + 2);

   case 1:
      sprintf(strp, "-%d", nextint);
      recur(nextint+1, cur_value, -1, 1, nextint, next_pos + 2);

      sprintf(strp, "%d", nextint);
      recur(nextint+1, a, s, b, 10*c + nextint, next_pos + 1);
   }
}

int
main()
{
   recur(1, 0, 1, 1, 0, 0);
}
```

Chapter 8

Circle from three points

8.1 Question

Given three points (x_1, y_1), (x_2, y_2) and (x_3, y_3). Find the center and radius of the circle passing through it.

8.2 General discussion

Three non-collinear points represent a unique circle on a unique plane. It is easy to construct the circle from 3 points by drawing perpendicular bisectors of any two chords. If the 3 points are not collinear, the bisectors should meet at the center of the circle.

Using vector's, it is possible to devise an algorithm to do this. However, an easier method will be substituting the three points into the general equation of the circle

$$x^2 + y^2 + 2gx + 2fy + c = 0 \tag{8.1}$$

and solving for g, f and c. Center is $(-g, -f)$ and the radius is $\sqrt{g^2 + f^2 - c}$.

8.3 Solution 1

This can be solved by substituting the three sets of values in the above equation and solving three simultaneous linear equations. The solution to the above set of simultaneous equations yields the equation of the circle passing through

(x_1, y_1), (x_2, y_2) and (x_3, y_3), and is given by equating the value of the following determinant equation:

$$\begin{vmatrix} x & y & x^2+y^2 & 1 \\ x_1 & y_1 & x_1^2+y_1^2 & 1 \\ x_2 & y_2 & x_2^2+y_2^2 & 1 \\ x_3 & y_3 & x_3^2+y_3^2 & 1 \end{vmatrix} = 0 \tag{8.2}$$

8.4 Solution 2

A modified algorithm is given below:

The task will be much simpler if we apply a linear transformation so that one of the points (say (x_3, y_3)) is $(0, 0)$, and now the constant term vanishes, and now there are only two variables and two unknowns. We can transform the co-ordinates back after solving.

So, transform (x_1, y_1), (x_2, y_2), (x_3, y_3) to

$$x_1 \Leftarrow x_1 - x_3 \tag{8.3}$$
$$y_1 \Leftarrow y_1 - y_3 \tag{8.4}$$
$$x_2 \Leftarrow x_2 - x_3 \tag{8.5}$$
$$y_2 \Leftarrow y_2 - y_3 \tag{8.6}$$

and now the equations are

$$2x_1 \cdot g + 2y_1 \cdot f = z_1 \tag{8.7}$$
$$2x_2 \cdot g + 2y_2 \cdot f = z_2 \tag{8.8}$$

where

$$z_1 = x_1^2 + y_1^2 \tag{8.9}$$
$$z_2 = x_2^2 + y_2^2 \tag{8.10}$$
$$\tag{8.11}$$

Now, we can solve for g and f here.

$$g = \frac{y_2 z_1 - y_1 z_2}{2(x_1 y_2 - x_2 y_1)} \tag{8.12}$$

$$f = \frac{x_1 z_2 - x_2 z_1}{2(x_1 y_2 - x_2 y_1)} \tag{8.13}$$

$$r = \sqrt{g^2 + f^2} \tag{8.14}$$

Finally, applying the tranformations back, the center is $(g + x_3, f + y_3)$.

The following C++ function illustrates this algorithm.

8.5 Solution 3

Plastock and Kalley [1] gives the following formulae for solving this problem:

The center (h, k) and the radius r of a circle passing through the points (x_1, y_1), (x_2, y_2) and (x_3, y_3), are given by:

$$h = \frac{1}{2} \cdot \frac{d_1^2(y_2 - y_3) + d_2^2(y_3 - y_1) + d_3^2(y_1 - y_2)}{d} \tag{8.15}$$

$$k = -\frac{1}{2} \cdot d_1^2(x_2 - x_3) + d_2^2(x_3 - x_1) + d_3^2(x_1 - x_2)d \tag{8.16}$$

$$r = \sqrt{(x_1 - h)^2 + (y_1 - k)^2} \tag{8.17}$$

where

$$d_i^2 = x_i^2 + y_i^2, \qquad \text{for i = 1, 2, 3} \tag{8.18}$$

$$d = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \tag{8.19}$$

These formulae can be used to find the parameters of the circle.

8.6 Solution 4

This can be solved using geometry as well.

The line passing through (x_1, y_1) and (x_2, y_2) is

$$y = m_a(x - x_1) + y_1 \tag{8.20}$$

Figure 8.1: Listing of `solve3PointsCircle()`

```
bool   /* true if valid circle, false if collinear */
solve3PointsCircle (
   double  x1, double  y1,  /* Point 1 */
   double  x2, double  y2,  /* Point 2 */
   double  x3, double  y3,  /* Point 3 */
   double& cx, double& cy,  /* Center  */
   double& r                /* Radius  */
)
{
   double z1, z2, d;

    /* Apply the transforms */
   x1 -= x3;
   y1 -= y3;
   x2 -= x3;
   y2 -= y3;

    /* Temporary variables to avoid some computations */
   z1 = x1 * x1 + y1 * y1;
   z2 = x2 * x2 + y2 * y2;
   d = 2.0 * (x1 * y2 - x2 * y1);
   if (fabs(d) < 0.00000001) {
      /* Collinear */
      return false;
   }

    /* Calculate the center of the transformed circle */
   cx = (y2 * z1 - y1 * z2)/d;
   cy = (x1 * z2 - x2 * z1)/d;
   r = sqrt(cx * cx + cy * cy);

    /* Apply the transforms back */
   cx += x3;
   cy += y3;
   return true;
}
```

where

$$m_a = \frac{y_2 - y_1}{x_2 - x_1} \tag{8.21}$$

The perpendicular bisector of this line passes through $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$ and has a slope of $(-1/m_a)$.

So, the perpendicular bisector is

$$y = -\frac{1}{m_a}\left(x - \frac{x_1 + x_2}{2}\right) + \frac{y_1 + y_2}{2} \tag{8.22}$$

Similarly, the perpendicular bisector of the side passing through (x_2, y_2) and (x_3, y_3) is

$$y = -\frac{1}{m_b}\left(x - \frac{x_2 + x_3}{2}\right) + \frac{y_2 + y_3}{2} \tag{8.23}$$

where

$$m_b = \frac{y_3 - y_2}{x_3 - x_2} \tag{8.24}$$

Now, the center of the circle is the point of intersection of these two perpendicular bisectors. So, equating the right hand sides of (8.22) and (8.23), and solving for x,

$$x = \frac{m_a m_b(y_1 - y_3) + m_b(x_1 + x_2) - m_a(x_2 + x_3)}{2(m_b - m_a)} \tag{8.25}$$

Bibliography

[1] R.A. Plastock and G. Kalley. *Theory and Problems of Computer Graphics*. Shaum's outline series, McGraw-Hill Book Company, 1986.

Index