Analysis: Broken Necklace

Russ Cox

In this problem, the necklace size is small enough (350) that we might as well just try breaking the necklace at each point and see how many beads can be collected. This will take approximately O(n^2) time, but n is small enough that it won't matter.

The code is slightly simple-minded in that we might count the same beads twice if they can be taken off either side of the break. This can only happen if all beads can be taken off the necklace, so we just check for that at the end.

```
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define MAXN 400

char necklace[MAXN];
int len;

/*
 * Return n mod m.  The C % operator is not enough because
 * its behavior is undefined on negative numbers.
 */
int
mod(int n, int m)
{
    while(n < 0)
        n += m;
    return n%m;
}

/*
 * Calculate number of beads gotten by breaking
 * before character p and going in direction dir,
 * which is 1 for forward and -1 for backward.
 */
int
nbreak(int p, int dir)
{
    char color;
    int i, n;

    color = 'w';

    /* Start at p if going forward, bead before if going backward */
    if(dir > 0)
        i = p;
    else
        i = mod(p-1, len);

    /* We use "n<len" to cut off loops that go around the whole necklace */
    for(n=0; n<len; n++, i=mod(i+dir, len)) {
        /* record which color we're going to collect */
        if(color == 'w' && necklace[i] != 'w')
            color = necklace[i];
        /*
         * If we've chosen a color and see a bead
         * not white and not that color, stop
         */
        if(color != 'w' && necklace[i] != 'w' && necklace[i] != color)
            break;
    }
    return n;
}

void
main(void)
{
    FILE *fin, *fout;
    int i, n, m;

    fin = fopen("beads.in", "r");
    fout = fopen("beads.out", "w");
    assert(fin != NULL && fout != NULL);

    fscanf(fin, "%d %s", &len, necklace);
    assert(strlen(necklace) == len);

    m = 0;
    for(i=0; i<len; i++) {
        n = nbreak(i, 1) + nbreak(i, -1);
        if(n > m)
            m = n;
    }

    /*
     * If the whole necklace can be gotten with a good
     * break, we'll sometimes count beads more than
     * once.  this can only happen when the whole necklace
     * can be taken, when beads that can be grabbed from
     * the right of the break can also be grabbed from the left.
     */
    if(m > len)
        m = len;

    fprintf(fout, "%d\n", m);
    exit (0);
}
```

Daniel Bundala had an O(n) solution:

Dynamic Programming is good method for solving this problem in O(N). If we consider two copies of the string we easy transform cyclic configuration of the necklace to linear. Now we can compute for each breaking point how many beads of the same color can be collected on the left and on the right from the breaking point. I show how we can compute it only for the left side. For right side it is analogical. Let r[p] and b[p] be the number of red / blue beads that can be collected, when necklace is broken in point p. If we know this and color of next bead (c) we can compute r[p+1] and b[p+1].

```
 r[0] = p[0] = 0
 If c = 'r' then r[p+1] = r[p] + 1 and b[p+1] = 0
        because the length of the blue beads is 0.
 if c = 'b' then b[p+1] = b[p] + 1 and r[p+1] = 0
 if c = 'w' then both length of the red and length of blue beads
            can be longer.
so r[p+1] = r[p]+1 and b[p+1] = b[p] + 1.
```

The number of beads that can be collected in breaking point p is then max(left[r[p]], left[b[p]]) + max(right[r[p]], right[b[p]]). And the maximum from this value is answer for the problem.

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
```

```
using namespace std;

FILE *in,*out;

int main () {
   in = fopen("beads.in", "r");
   out = fopen ("beads.out", "w");

   int n;
   char tmp[400], s[800];
   fscanf(in, "%d %s", &n, tmp);

   strcpy(s, tmp);
   strcat(s, tmp);

   int left[800][2], right[800][2];
   left[0][0] = left[0][1] = 0;

   for (int i=1; i<= 2 * n; i++){
       if (s[i - 1] == 'r'){
           left[i][0] = left[i - 1][0] + 1;
           left[i][1] = 0;
       } else if (s[i - 1] == 'b'){
           left[i][1] = left[i - 1][1] + 1;
           left[i][0] = 0;
       } else {
           left[i][0] = left[i - 1][0] + 1;
           left[i][1] = left[i - 1][1] + 1;
       }
     }

   right[2 * n][0] = right[2 * n][1] = 0;
   for (int i=2 * n - 1; i >= 0; i--){
       if (s[i] == 'r'){
           right[i][0] = right[i + 1][0] + 1;
           right[i][1] = 0;
       } else if (s[i] == 'b'){
           right[i][1] = right[i + 1][1] + 1;
           right[i][0] = 0;
       } else {
           right[i][0] = right[i + 1][0] + 1;
           right[i][1] = right[i + 1][1] + 1;
       }
   }

   int m = 0;
   for (int i=0; i<2 * n; i++)
       m = max(m, max(left[i][0], left[i][1]) + max(right[i][0], right[i][1]));
   m = min(m, n);
   fprintf(out, "%d\n", m);
   fclose(in); fclose(out);
   return 0;
}
```

Holland's Frank Takes has a potentially easier solution:

```
/* This solution simply changes the string s into ss, then for every starting
// symbol it checks if it can make a sequence simply by repeatedly checking
// if a sequence can be found that is longer than the current maximum one.
*/

#include <iostream>
#include <fstream>
using namespace std;

int main() {
  fstream input, output;
  string inputFilename = "beads.in", outputFilename = "beads.out";
  input.open(inputFilename.c_str(), ios::in);
  output.open(outputFilename.c_str(), ios::out);

  int n, max=0, current, state, i, j;
  string s;
  char c;

  input >> n >> s;
  s = s+s;
  for(i=0; i<n; i++) {
    c = (char) s[i];
    if(c == 'w')
      state = 0;
    else
      state = 1;
    j = i;
    current = 0;
    while(state <= 2) {
      // dont go further in second string than starting position in first string
      while(j<n+i && (s[j] == c || s[j] == 'w')) {
        current++;
        j++;
      } // while
      state++;
      c = s[j];
    } // while
    if(current > max)
      max = current;
  } // for

  output << max << endl;
  return 0;
} // main
```