Relating Runtime to Complexity

Example:

Algorithm is \( \Theta(n) \). Program takes 0.25 seconds on \( n = 1000 \). How long will the program take on \( n = 10^6 \)?

The math:

\[
\begin{align*}
R_1 &= f(n_1) \\
R_2 &= f(n_2) \\
\frac{R_1}{R_2} &= \frac{f(n_1)}{f(n_2)} \quad \Rightarrow \quad R_2 &= \frac{R_1 f(n_2)}{f(n_1)}
\end{align*}
\]

Notice that \( \frac{f(n_1)}{f(n_2)} \) reduces any multiplication constants.

\[
\Rightarrow \quad R_2 = \frac{(0.25)(10^6)}{(10^3)} = (0.25)(10^3) = 250 \text{ seconds}
\]
Relating Runtime to Complexity

**Examples**

\[ R_1 = 4 \text{ms} \quad n_1 = 150 \quad R_2 = ? \]
\[ f(n) = \Theta(n^3) \quad n_2 = 750 \]

\[ R_1 = 9 \text{ms} \quad n_1 = 1000 \quad R_2 = ? \]
\[ f(n) = \Theta(n^2) \quad n_2 = 16,000 \]
Growth of Functions

Linear
\[ f(n) = a \cdot n \]
\[ f(2n) = a \cdot 2n = 2an = 2f(n) \]

Quadratic
\[ f(n) = a \cdot n^2 \]
\[ f(2n) = a \cdot (2n)^2 = 4an^2 = 4f(n) \]

Cubic
\[ f(n) = a \cdot n^3 \]
\[ f(2n) = a \cdot (2n)^3 = 8an^3 = 8f(n) \]
Growth of Functions

Log

\[ f(n) = a \log(n) \]

\[ f(2n) = a \log(2n) = a \log(n) + a \log(2) = f(n) + a \log(2) \]

\[ f(4n) = a \log(4n) = a \log(n) + a \log(4) = f(n) + a \log(2) + a \log(2) \]

Exponent

\[ f(n) = a b^n \]

\[ f(2n) = a b^{(2n)} = a b^n b^n = f(n) \cdot b^n = \frac{1}{a} f(n)^2 \]

\[ f(4n) = a b^{(4n)} = a b^n b^n b^n b^n = \frac{1}{a^3} f(n)^4 \]
Experimental Runtime vs. Complexity

Given: Program that executes algorithm
Find: Runtime complexity of algorithm
\[
\text{Time}(h) = T_0 + f(h) + C_0 \quad \text{eg.} \quad f(h) = h^2
\]
\[
\text{Time}(2h) = T_0 + f(2h) + C_0 \quad f(2h) = (2h)^2 = 4h^2
\]
generate datasets of differing size.

double / order of magnitude increase.

-time execution on each dataset

-table/chart with runtime vs. size

-compare table/chart vs. known function: \( \log(n) \), \( n \), \( n^2, n^3 \), \( 2^n \) . . .
Some small data sets are too fast for resolution.

Need to run multiple times in one execution to get better data.

Launching process/reading data = overhead cost.

Timing:
- Time externally vs. internally
- Real time vs. CPU time.

Sub-second resolution
Charting: Remember G, T. can scale plots to compare shapes.
Problem G: Shuffling

Source: shuffling.{c,cpp,java}
Input: console {stdin,cin,System.in}
Output: console {stdout,cout,System.out}

A casino owns an expensive card shuffling machine which may shuffle up to 520 cards at a
time (there are 52 cards in each deck). For convenience, we will simply label the cards 1, 2,
3, ..., N where N is the total number of cards, and copies of the same card (e.g. Ace of
Spades) from different decks are considered different. Unfortunately, the card shuffling
machine is defective, and it always shuffles the cards the same way. The company that
produces these machines is out of business because of the economic downturn. There is no
one who can fix the machine, and a new machine is too expensive.

Being a brilliant employee of the casino, you realized that all is not lost. You can shuffle the
cards differently simply by using the machine zero or more times. For example, suppose
that the machine shuffles the cards 1, 2, 3, 4 into the order 2, 3, 4, 1. If you put the cards into
the machine, take the shuffled cards out and insert them into the machine again (without
changing the order), you will get the order 3, 4, 1, 2. That way, it is possible to shuffle the
cards in many different ways even though it may take longer. But this is not a significant
issue since decks do not have to be reshuffled often, and used decks can be shuffled while
other decks are being used to avoid any waiting time.

Unfortunately, not all shufflings can be produced in this way in general, and you wish to
know if this procedure "stack the decks" in a favorable way for the casino or the player. As
a first step, you wish to know which shufflings are possible to produce, and how many times
you need to use the machine on the deck in order to produce the shuffling.
Input

The input for each case consists of three lines. The first line consists of a single integer $N$ indicating the number of cards to shuffle. The number of cards is a positive integer up to 520. The second line consists of the integers 1, 2, ..., $N$ listed in some order and separated by a space. The list gives the order of the shuffling performed by the machine when the input cards are ordered 1, 2, ..., $N$. The third line is in the same format as the second line, and gives the shuffling we wish to obtain. The end of input is indicated by a line in which $N = 0$.

Output

For each case, print the smallest number of times (zero or more) you need to pass the deck through the machine to produce the desired shuffling. If it is not possible, print -1. The output for each case should be in a single line. You may assume that the answer will always fit in a 32-bit signed integer.

Sample Input

```
4
2 3 4 1
3 4 1 2
4
2 3 4 1
1 3 2 4
10
2 1 3 5 6 7 8 9 10 4
1 2 3 9 10 4 5 6 7 8
0
```

Sample Output

```
2
-1
12
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_CARDS 1000

int my_read_problem(int shuffle[], int target[])
{
    int c;
    int N;
    int i;
    c = fscanf(stdin, "%d", &N);
    if(c <= 0) { return 0; } /* bad scan */
    if(N == 0) { return N; } /* end of problems */

    for(i = 0; i < N; i++)
    {
        c = fscanf(stdin, "%d", &shuffle[i]);
        if(c <= 0) { return 0; } /* bad scan */
    }

    for(i = 0; i < N; i++)
    {
        c = fscanf(stdin, "%d", &target[i]);
        if(c <= 0) { return 0; } /* bad scan */
    }

    return N;
}

int brute_solve_problem(int N, int shuffle[], int target[])
{
    int deckA[MAX_CARDS];
    int deckB[MAX_CARDS];
    int next[MAX_CARDS]; /* maps where cards go to on shuffle
    * index == card_no-1   where the card was before shuffle
    * value == card_no     where the card will be after shuffle */
    int last[MAX_CARDS]; /* maps where cards should be at end
    * index == card_no-1   the card's number - 1
    * value == card_no     where the card should be at the end */

    int i,j;
    int position;
    int done, looped;
    int MAX_LOOPS = N*N*N; /* N*N*20; needs to be larger than this. */

    done = 1;
    for(i = 0; i < N; i++)
    {
        next[shuffle[i]-1] = i+1;
        last[target[i]-1] = i+1;
        deckA[i] = i+1;
        if(deckA[i] != target[i]) { done = 0; }
    }

    return done;
}
if(done) { return 0; }

for(j = 0; j < MAX_LOOPS; j++)
{
  for(i = 0; i < N; i++)
  {
    deckB[next[i]-1] = deckA[i];
  }
  for(i = 0; i < N; i++)
  {
    deckA[i] = deckB[i];
  }
  done   = 1;
  for(i = 0; i < N; i++)
  {
    if(deckA[i] != target[i]) { done = 0; break; }
  }
  looped = 1;
  for(i = 0; i < N; i++)
  {
    if(deckA[i] != i+1) { looped = 0; break; }
  }
  if(looped || done) { break; }
}
if(looped) { return -1; }
if(done) { return j+1; }
return -1;

int main(int argc, char **argv)
{
  int last_N;
  int N;
  int shuffle[MAX_CARDS];
  int target[MAX_CARDS];
  int count;
  clock_t t1;
  clock_t t2;
  clock_t dt;
  double clocks_per_rep;
  double seconds;
  int total = 0;
  t1 = clock();
  while(0 < (N = my_read_problem(shuffle, target)))
  {
    count = brute_solve_problem(N, shuffle, target);
    printf("%d\n", count);
    total ++;
    last_N = N;
  }
  t2 = clock();
dt = t2 - t1;
clocks_per_rep = ((double)dt)/total;
seconds = clocks_per_rep/CLOCKS_PER_SEC;
fprintf(stderr, "problems: %d size: %d seconds: %g seconds/problem: %g\n",
        total, last_N, ((double)dt)/CLOCKS_PER_SEC, seconds);
    return 0;
}
/* Extended Euclid algorithm for greatest common divisor
 * calc d = gcd(a,b) and x,y such that d = ax + by
 * a >= b >= 0
 * returns the greatest common divisor of a and b, x and y
 */
long gcd(long a, long b, long *xptr, long *yptr)
{
    long q, r;
    long x1, y1;
    long d;
    if(a < 0 || b < 0) { fprintf(stderr, "ERROR: gcd(): %ld %ld\n", a, b); exit(1); }
    if( a < b ) { return gcd(b, a, yptr, xptr); }
    if( b == 0 )
    {
        *xptr = 1;
        *yptr = 0;
        return a;
    }
    q = a/b;
    r = a%b;
    d = gcd(b, r, &x1, &y1);
    *xptr = y1;
    *yptr = x1 - q * y1;
    return d;
}

void test_gcd()
{
    long i, j, d, x, y, z;
    for( i = 1; i < 10; i++ )
    {
        for(j = 1; j < 10; j++)
        {
            d = gcd(i, j, &x, &y);
            z = 1*x + j*y;
            printf("%ld = %ld*%ld + %ld*%ld = %ld\n",
                    d, i, x, j, y, z);
        }
    }
}

/*
 * given:
 * x = r_0 mod b_0
 * x = r_1 mod b_1
 * find r, b such that:
 * x = r mod b
/*
 * using substitution,
 * d0 = gcd(b0, b1)
 * d1 = gcd(d0, r1-r0)
 * c = [b0/d1]^-1 mod b1/d1
 * r = r0 + (r1 - r0)*b0*c/d1
 * b = b0*b1/d1
 * returns 0 on failure, 1 on success (there is an non-invertable number)
 */
long substitution(long r0, long b0, long r1, long b1, long *rptr, long *bptr)
{
    long d0, d1, c;
    long x, y;
    long a, b, d;
    long dr;

    d0 = gcd(b0, b1, &x, &y);
    dr = r1-r0;
    if(dr < 0)
    {
        d1 = gcd(d0, -dr, &x, &y);
        y = -y;
    }
    else
    {
        d1 = gcd(d0, dr, &x, &y);
    }

    /* c = [b0/d1]^-1 mod b1/d1; */
    a = b0/d1;
    b = b1/d1;
    d = gcd(a, b, &x, &y);
    if(d != 1) { return 0; }
    c = x;

    *rptr = r0 + (r1 - r0)*c*(b0/d1);
    *bptr = b0*(b1/d1);
    while(*rptr < 0) { *rptr += *bptr; }
    *rptr %= *bptr;

    return 1;
}

/* returns 0 on failure, 1 on success (there is an non-invertable number) */
long n_substitution(long r[], long b[], long n, long *rptr, long *bptr)
{
    long i;
    long r0, b0;

    r0 = r[0];
b0 = b[0];

for(i = 1; i < n; i++)
{
    if(!substitution(r0,b0, r[i],b[i], rptr, bptr))
    {
        return 0;
    }
    r0 = *rptr;
    b0 = *bptr;
}

return 1;

#define MAX_CARDS 1024

long my_read_problem(long shuffle[], long target[])
{
    long c;
    long N;
    long i;
    c = fscanf(stdin, "%ld", &N);
    if(c <= 0) { return 0; } /* bad scan */
    if(N == 0) { return N; } /* end of problems */

    for(i = 0; i < N; i++)
    {
        c = fscanf(stdin, "%ld", &shuffle[i]);
        if(c <= 0) { return 0; } /* bad scan */
    }

    for(i = 0; i < N; i++)
    {
        c = fscanf(stdin, "%ld", &target[i]);
        if(c <= 0) { return 0; } /* bad scan */
    }

    return N;
}

long solve_problem(long N, long shuffle[], long target[])
{
    long r[MAX_CARDS];
    long b[MAX_CARDS];
    long next[MAX_CARDS]; /* maps where cards go to on shuffle
        * index == card_no-1 where the card was before shuffle
        * value == card_no  where the card will be after shuffle */
    long last[MAX_CARDS];  /* maps where cards should be at end
        * index == card_no-1 the card's number - 1
        * value == card_no  where the card should be at the end */
    long i,j;
    long position;
for(i = 0; i < N; i++)
  
  next[shuffle[i]-1] = i+1;
  last[target[i]-1] = i+1;
}

for(i = 0; i < N; i++)
  
  position = i+1; /* i-th card's position */
  r[i] = -1;
  b[i] = -1;
  for(j = 0; j <= N; j++) /* can't have cycle longer than the number of cards */
  
  if(position == last[i] && r[i] < 0)
  
    r[i] = j;
  }
  if(j > 0 && ((position-1) == i) && b[i] < 0)
  
    b[i] = j;
    break;
  
  position = next[position-1];
}

if(r[i] == -1)
  
  /* this card will never be where we want it. */
  return -1;

long x, base;
if(n_substitution(r, b, N, &x, &base))
  
  return x;
else
  
  return -1;

return N;

int main(int argc, char **argv)
{
  long last_N;
  long N;
  long shuffle[MAX_CARDS];
  long target[MAX_CARDS];
  long count;
  clock_t t1;
  clock_t t2;
  clock_t dt;
double clocks_per_rep;
double seconds;
int total = 0;

t1 = clock();
while(0 < (N = my_read_problem(shuffle, target)))
{
    count = solve_problem(N, shuffle, target);
    printf("%ld\n", count);
    total ++;
    last_N = N;
}
t2 = clock();
dt = t2 - t1;

CLOCKS_PER_SEC = clocks_per_rep/CLOCKS_PER_SEC;
seconds = clocks_per_rep/CLOCKS_PER_SEC;
fprintf(stderr, "problems: %d size: %ld seconds: %g seconds/problem: %g\n",
    total, last_N, ((double)dt)/CLOCKS_PER_SEC, seconds);

return 0;
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX_SIZE 1040

void build_problem(int N) {
    int a[MAX_SIZE];
    int b[MAX_SIZE];
    int i, j, t;
    for (i = 0; i < N; i++) {
        a[i] = i + 1;
        b[i] = i + 1;
    }
    for (i = 0; i < N; i++) {
        j = rand() % N;
        t = a[i];
        a[i] = a[j];
        a[j] = t;
        j = rand() % N;
        t = b[i];
        b[i] = b[j];
        b[j] = t;
    }
    printf("%d\n", N);
    for (i = 0; i < N; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    for (i = 0; i < N; i++) {
        printf("%d ", b[i]);
    }
    printf("\n");
}

int main(int argc, char **argv) {
    int N=10, count=1, opt, i;
    while ((opt = getopt(argc, argv, "n:c:")) != -1)
    {
        switch (opt)
        {
        case 'n':
            
    }
N = atoi(optarg);
break;
case 'c':
    count = atoi(optarg);
    break;
default: /* '?' */
    fprintf(stderr, "Usage: %s [-n size] [-c count]\n",
            argv[0]);
    exit(EXIT_FAILURE);
}
}
srand(time(0));
for(i = 0; i < count; i++)
{
    build_problem(N);
}
printf("\n");
return 0;
#!/bin/bash

make c_brute_shuffling c_shuffling c_case_generator

for N in 1 2 4 8 16 32 64 128 256 512; do
  count=`echo $N | awk '{printf("%d
", 1000000/$1);}'
  ./c_case_generator -n $N -c $count > p.$N.in
  echo -n "$N $count F: "
  ./c_shuffling < p.$N.in > p.out
done

for N in 1 2 4 8 16 32 64 128 256 512; do
  count=`echo $N | awk '{printf("%d
", 10000000/$1/$1);}'
  ./c_case_generator -n $N -c $count > p.$N.in
  echo -n "$N $count B: "
  ./c_brute_shuffling < p.$N.in > p.out
done
<table>
<thead>
<tr>
<th>N</th>
<th>Fast</th>
<th>Brute</th>
<th>Brute Scaled</th>
<th>Log(N)</th>
<th>N</th>
<th>N^2</th>
<th>N^3</th>
<th>2^N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00E+00</td>
<td>3.00E-007</td>
<td>2.69E-007</td>
<td>5.07E-009</td>
<td>0.00E+000</td>
<td>1.40E-007</td>
<td>2.19E-009</td>
<td>3.42E-011</td>
<td>9.71E-025</td>
</tr>
<tr>
<td>2.00E+00</td>
<td>4.80E-007</td>
<td>4.08E-007</td>
<td>7.69E-009</td>
<td>1.49E-006</td>
<td>2.80E-007</td>
<td>8.75E-009</td>
<td>2.73E-010</td>
<td>1.94E-024</td>
</tr>
<tr>
<td>4.00E+00</td>
<td>8.00E-007</td>
<td>6.88E-007</td>
<td>1.30E-008</td>
<td>2.99E-006</td>
<td>5.60E-007</td>
<td>3.50E-008</td>
<td>2.19E-009</td>
<td>7.77E-024</td>
</tr>
<tr>
<td>8.00E+00</td>
<td>1.28E-006</td>
<td>1.34E-006</td>
<td>2.53E-008</td>
<td>4.48E-006</td>
<td>1.12E-006</td>
<td>1.40E-007</td>
<td>1.75E-008</td>
<td>1.24E-022</td>
</tr>
<tr>
<td>1.60E+00</td>
<td>2.56E-006</td>
<td>3.07E-006</td>
<td>5.79E-008</td>
<td>2.24E-006</td>
<td>5.60E-007</td>
<td>1.40E-007</td>
<td>3.18E-020</td>
<td>3.18E-020</td>
</tr>
<tr>
<td>3.20E+00</td>
<td>4.80E-006</td>
<td>2.05E-005</td>
<td>3.86E-007</td>
<td>7.47E-006</td>
<td>4.48E-006</td>
<td>2.24E-006</td>
<td>1.12E-006</td>
<td>2.09E-015</td>
</tr>
<tr>
<td>6.40E+00</td>
<td>8.96E-006</td>
<td>4.75E-004</td>
<td>8.96E-006</td>
<td>8.96E-006</td>
<td>8.96E-006</td>
<td>8.96E-006</td>
<td>8.96E-006</td>
<td>8.96E-006</td>
</tr>
<tr>
<td>1.28E+00</td>
<td>1.79E-005</td>
<td>2.08E-002</td>
<td>3.92E-004</td>
<td>1.05E-005</td>
<td>3.58E-005</td>
<td>7.17E-005</td>
<td>1.65E+014</td>
<td>1.65E+014</td>
</tr>
<tr>
<td>2.56E+00</td>
<td>3.58E-005</td>
<td>1.11E+000</td>
<td>2.09E-002</td>
<td>1.19E-005</td>
<td>3.58E-005</td>
<td>1.43E-004</td>
<td>5.73E-004</td>
<td>5.62E+052</td>
</tr>
<tr>
<td>5.12E+00</td>
<td>7.68E-005</td>
<td>3.31E+001</td>
<td>6.23E-001</td>
<td>1.34E-005</td>
<td>7.17E-005</td>
<td>5.73E-004</td>
<td>4.59E-003</td>
<td>6.51E+129</td>
</tr>
</tbody>
</table>

| 6.40E+00 | 8.96E-006 | 4.75E-004 | 4.75E-004 | 1.81E+000 | 6.40E+001 | 4.10E+003 | 2.62E+005 | 1.84E+019 |
| Normalization | 1.89E-002 | 4.96E-006 | 1.40E-007 | 2.19E-009 | 3.42E-011 | 4.86E-025 |

Diagram showing the comparison of Fast, Brute, Brute Scaled, Log(N), N, N^2, N^3, and 2^N.