NOI 2007—Solutions to Contest Tasks

Martin Henz

20/1, 2007

Problem

The coach of Jacqueline Yo, Olympic swimmer for Singapore, is concerned about her Butterfly stroke. He records her daily timing in milliseconds (a millisecond is one-thousand of a second) and devises a scheme whereby each time she achieves a timing that is lower than the previous day's timing by at least a certain number of milliseconds, he will reward her with a small encouragement gift. Given a list of daily timings, determine how many gifts Jacqueline would have received.

Input and Output

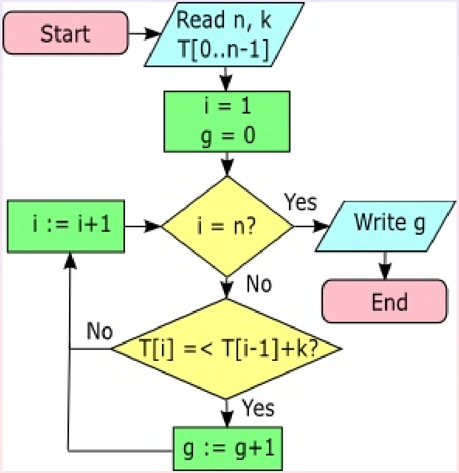
- Input file: `GIFT.IN`

```
4 100
59420
59310
59400
59290
```

- Output file: `GIFT.OUT`

```
2
```

Flow Diagram



Program—Header

```
// Program for GIFT
// gift.c
// NOI Singapore 2007
// Author: Aaron Tan
// Date: November 2006

#include <stdio.h>
#define MAX 100
```

Program—Declarations

```
int main() {

  FILE *infile, *outfile;
  int n,      // number of daily timing records
      k,      // improvement threshold
      gifts,  // number of gifts
      i;
  int T[MAX];
```

Program—Read Input

```
// ************ INPUT **************
infile = fopen("GIFT.IN", "r");
fscanf(infile, "%d %d", &n, &k);

// read n daily timing records
for (i=0; i<n; i++)
  fscanf(infile, "%d", &T[i]);
fclose(infile);
```

Program—Computation

```
// *** Compute number of gifts ***
gifts = 0;
for (i=1; i<n; i++)
  if (T[i] <= T[i-1] - k)
    gifts++;
```

Program—Write Output

```
  // ************ OUTPUT **************
  outfile = fopen("GIFT.OUT", "w");
  fprintf(outfile, "%d\n", gifts);
  fclose(outfile);
  return 0;
}
```

Problem

Given a set of rectangles $\{R_1, R_2, \cdots, R_n\}$, compute the area of their common intersection. i.e.,

$$\text{Area } (R_1 \cap R_2 \cap \cdots \cap R_n)$$

The edges of the rectangles $R_1, R_2, \cdots, R_n$, are either vertical or horizontal lines.

Input Data

```
3
1 4 1 8
0 2 0 5
10 15 22 35
```

Here, there are three rectangles, each described by

```
a1 a2 b1 b2
```

C++ program fits on one slide

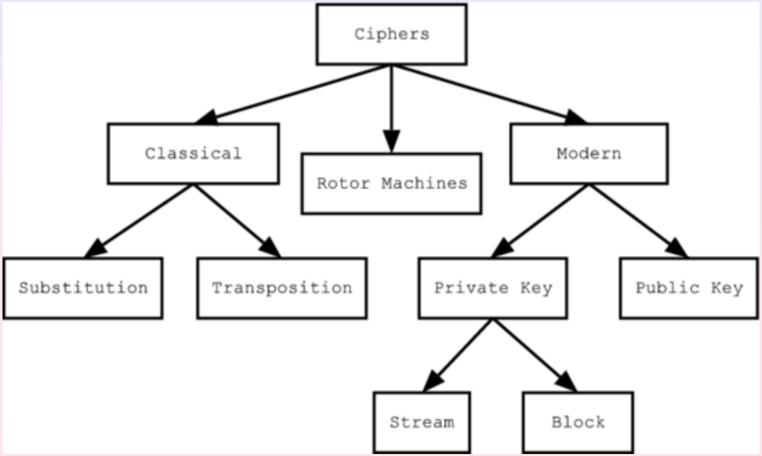
```
#include <stdio.h>
int main (int c, char *v[]) {
 FILE *i=fopen("RECT.IN","r"), *o=fopen("RECT.OUT","w");
 int n, a1, a2, b1, b2, x1=0, x2=10000, y1=0, y2=10000;
 fscanf(i,"%d",&n);
 while(n-->0){ fscanf(in,"%d %d %d %d",&a1,&a2,&b1,&b2);
               x1>?=a1; x2<?=a2; y1>?=b1; y2<?=b2;      }
 fprintf(o,"%d",(0>?(x2-x1))*(0>?(y2-y1)));
 fclose(i); fclose(o);                                  }
```

Problem

You receive messages of the form:

```
XLMW MW OSNEO. M EQ LIVIFC SVHIVMRK XLEX EPP QC QIR QYWX
IEX  TITTIVSRM TMDDEW IZIVCHEC. XLMW SVHIV AMPP FI
VITIEPIH SRPC YTSR QC VIXMVIQIRX. PSRK PMZI XLI GLMTQYROW!
```

Cipher Taxonomy



Caesar Cipher

Plain	A	B	C	D	E	F	G	H	I	J	K	L	M
Cipher	C	D	E	F	G	H	I	J	K	L	M	N	O

Plain	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	P	Q	R	S	T	U	V	W	X	Y	Z	A	B

We have a Crib!

A “crib” is a piece of cipher-text, along with the corresponding plain text message.

Example

A German WWII commander in the Afrika Korps used to encrypt

KEINE BESONDERE EREIGNISSE

using the ENIGMA encoding machine. The British decoders cracked ENIGMA using this crib and a code cracking machine called *bombe*.

We have a Crib!

A “crib” is a piece of cipher-text, along with the corresponding plain text message.

Example

A German WWII commander in the Afrika Korps used to encrypt

KEINE BESONDERE EREIGNISSE

using the ENIGMA encoding machine. The British decoders cracked ENIGMA using this crib and a code cracking machine called *bombe*. The designer of *bombe* was Alan Turing (1912–1954), father of computer science.

We have a Crib!

A "crib" is a piece of cipher-text, along with the corresponding plain text message.

Example

A German WWII commander in the Afrika Korps used to encrypt

KEINE BESONDERE EREIGNISSE

using the ENIGMA encoding machine. The British decoders cracked ENIGMA using this crib and a code cracking machine called *bombe*. The designer of *bombe* was Alan Turing (1912–1954), father of computer science.

Our crib is that the words "CHIPMUNK" and "LIVE" occur in every message.

Code Breaking Algorithm

- Follow “brute-force” approach

Code Breaking Algorithm

- Follow “brute-force” approach
- Try each of the 26 possible Caesar ciphers and stop when we find the crib

Program

```
#define WORD1 "LIVE"
#define WORD3 "CHIPMUNK"
//-----------------------------------------------------
// core routine (adapted from Low Kok Lim's solution)
//-----------------------------------------------------
for ( i = 0; i < 26; i++ ) {
   if ( strstr( str, WORD1 ) &&
        strstr( str, WORD2 ) break;
   increaseByOne( str );
}
```

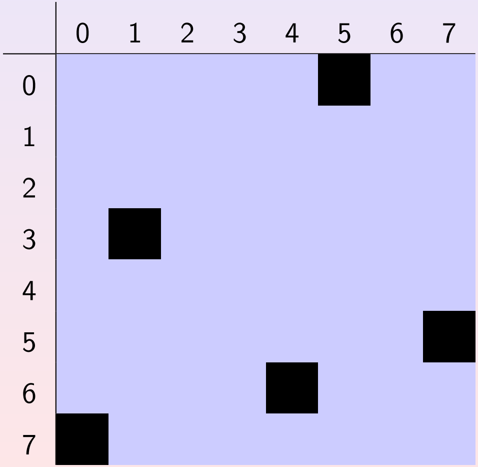
In case you're curious...

```
THIS IS KOJAK. I AM HEREBY ORDERING THAT ALL MY MEN MUST
EAT  PEPPERONI PIZZAS EVERYDAY. THIS ORDER WILL BE
REPEALED ONLY UPON MY RETIREMENT. LONG LIVE THE CHIPMUNKS!
```

Problem

- Many sensors are air-dropped into forest.
- A *hole* is a square region of forest without a sensor.
- Goal: Find largest hole

Example



Efficiency Counts!

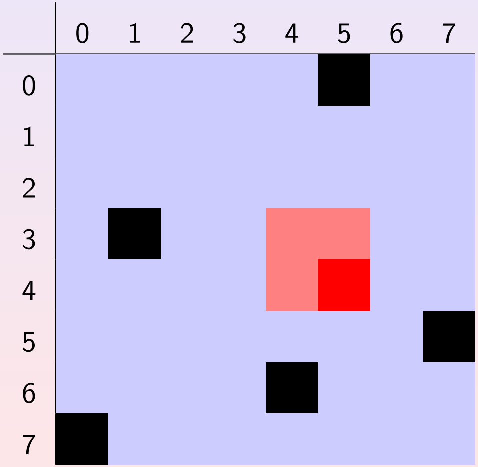
Attention

The forest can be large; your algorithm must be fast!

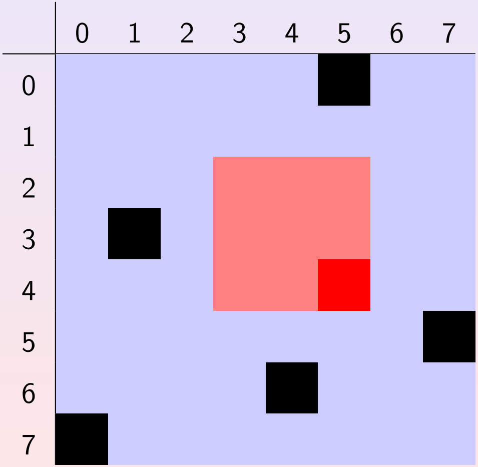
Naive Solution

	0	1	2	3	4	5	6	7
0						⬛		
1								
2								
3		⬛						
4						🟥		
5								⬛
6					⬛			
7	⬛							

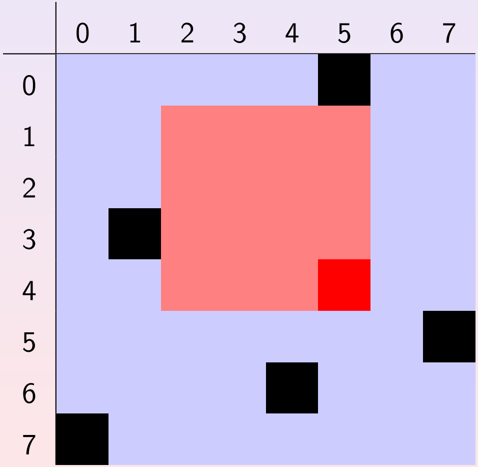
Naive Solution



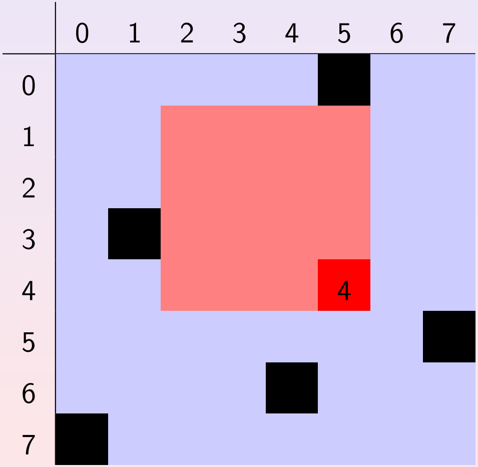
Naive Solution



Naive Solution



Naive Solution



Naive Solution

- For every cell, try possible hole sizes starting with 1 and ending when a sensor is reached.
- Complexity proportional to n^2

Naive Solution

- For every cell, try possible hole sizes starting with 1 and ending when a sensor is reached.
- Complexity proportional to $n^2\ n$

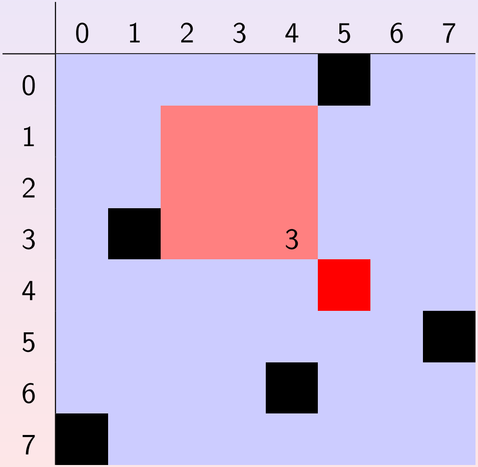
Naive Solution

- For every cell, try possible hole sizes starting with 1 and ending when a sensor is reached.
- Complexity proportional to n^2 n n^2

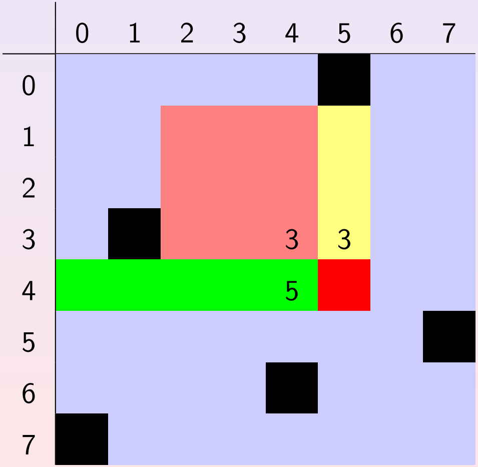
Naive Solution

- For every cell, try possible hole sizes starting with 1 and ending when a sensor is reached.
- Complexity proportional to $n^2 \; n \; n^2 = n^5$

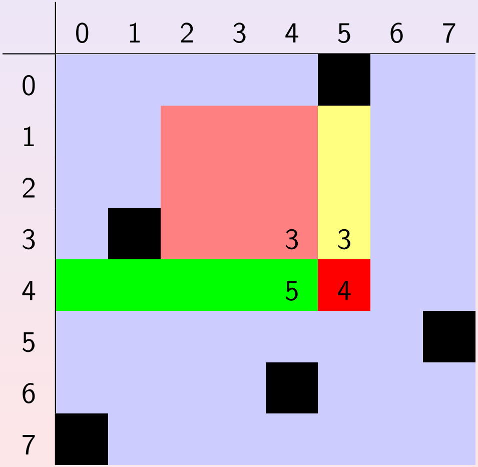
Smarter Way: Learn from Previous Work



Smarter Way: Learn from Previous Work



Smarter Way: Learn from Previous Work



Implementation

C++ program by Chang Ee-Chien

```
const int MAX=1024;
int A[MAX][MAX];  // 1 means sensor, 0 means empty
int B[MAX][MAX];  // explained later
int C[MAX][MAX];  // explained later
int H[MAX][MAX];  // explained later
```

Implementation

```
// B[i][j] represents the number of empty cells
// to the left of (and including) cell (i,j)
for (i=0;i<N;i++) {
   B[i][0] = 1-A[i][0];
   for (j=1;j<N;j++) {
       if (A[i][j]==1) B[i][j]=0;
       else            B[i][j]=B[i][j-1]+1;
   }
}
```

Implementation

```
// C[i][j] represents the number of empty cells
// above (and including) cell (i,j)
for (j=0;j<N;j++) {
   C[0][j]= (1-A[0][j]);
   for (i=1;i<N;i++) {
      if (A[i][j]==1) C[i][j]=0;
      else            C[i][j]=C[i-1][j]+1;
   }
}
```

Implementation

```
// H[i][j] represents the size of the hole above and
// to the left of cell (i,j)

// Initialize first row and first column
for (i=0;i<N;i++) {
   H[0][i]= (1-A[0][i]);
   H[i][0]= (1-A[i][0]);
}
```

Implementation

```
// Main algorithm: Compute H[i][j] based on
// H[i-1][j-1], B[i][j] and C[i][j]
for (i=1;i<N;i++)
   for (j=1;j<N;j++) {
      H[i][j]=H[i-1][j-1]+1;
      if (B[i][j]< H[i][j])  H[i][j] = B[i][j];
      if (C[i][j]< H[i][j])  H[i][j] = C[i][j];
   }
```

Complexity

- Computation of matrix B requires time proportional to n^2

Complexity

- Computation of matrix B requires time proportional to n^2
- Computation of matrix C requires time proportional to n^2

Complexity

- Computation of matrix B requires time proportional to n^2
- Computation of matrix C requires time proportional to n^2
- Computation of matrix H requires time proportional to n^2

Complexity

- Computation of matrix B requires time proportional to n^2
- Computation of matrix C requires time proportional to n^2
- Computation of matrix H requires time proportional to n^2

Overall complexity

Proportional to n^2

A Game Worth More Than 1000 Words

Play Jawbreaker Online

Task Description

A player starts with score 0. Each move adds to score by the square of number of balls removed. Player "wins" Jawbreaker if he/she is able to remove all balls from the board. A win adds a bonus of 1,000 points to the score. Game over if there are no valid moves remaining on the board.

Goal

Output the maximum score that can be achieved given the board.

Solution Outline

- Explore all possible move sequences
- Exponential worst-case complexity!
- Implementation requires careful programming
- Note that the maximum score might be achieved without the bonus!

Main Loop (adapted from Kan Min-Yen's solution)

```
int recurse () {
  state *s = head;
  int score = 0;
  s = popState();
  while (s != NULL) {
    executeMove(s);
    pushNewMoves(s);
    free(s);
    s = popState();
  }
  return topScore;
}
```

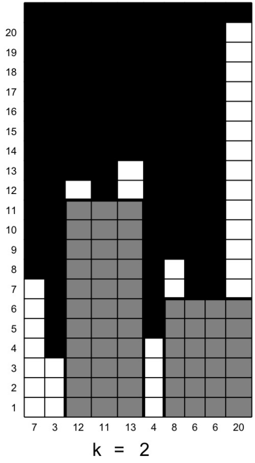
Problem

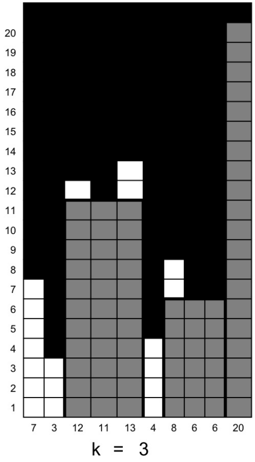
There are n lots on one side of a street (where $n \leq 500$). We would like to erect at most k apartment buildings on these lots. Each building must occupy an interval of at most t consecutive lots. Each lot i has a height restriction $r[i]$.

Goal

Select at most k non-overlapping intervals to erect the buildings such that the total usable facade space is maximized.

Examples: $t = 4$ lots, $k = 2, 3$ buildings





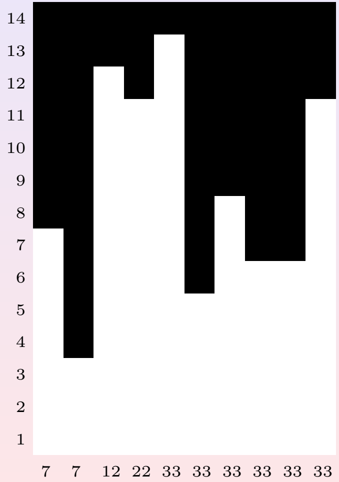
Naive Solution

- Try all possible combinations and positions of buildings that stay within the lots limit.
- Exponential complexity!

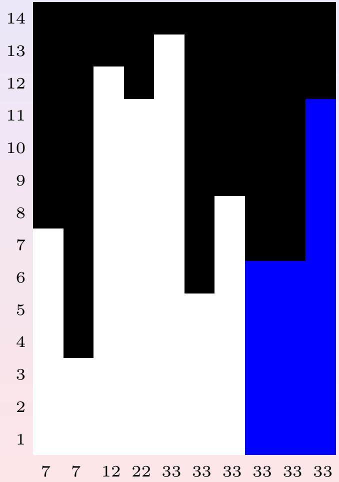
Smart Solution: Idea

- Proceed by trying increasing numbers of buildings κ, starting with 1 and ending with k.
- Each time, proceed from left to right. For each lot, record the best score for erecting buildings up until that lot.
- For a particular lot i for a given κ, make use of score for $\kappa - 1$.

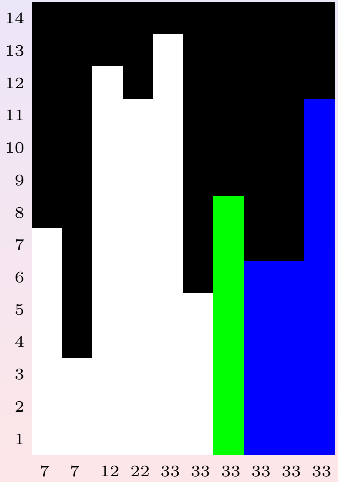
Example: $\kappa = 2$, $t = 4$



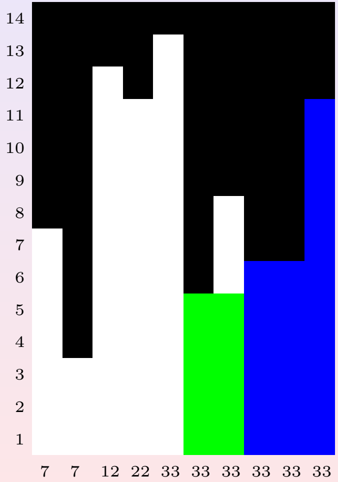
First Case: No New Building, stick with 33



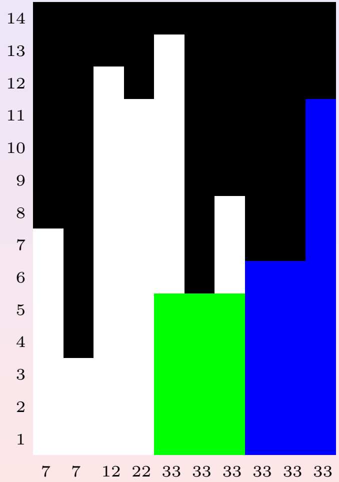
Second Case: New Building with Width 1



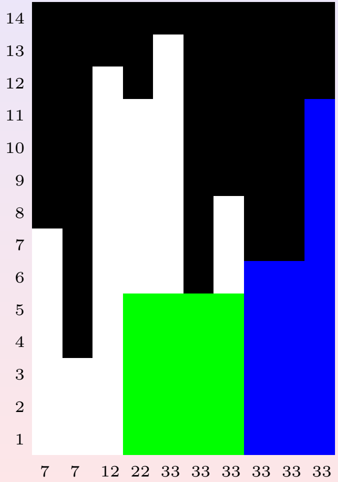
Third Case: New Building with Width 2



Fourth Case: New Building with Width 3



Final Case: New Building with Width 4



Complexity

- Iterate through all $\kappa \leq k$
- Each time iterate through all $i \leq n$
- Each time iterate through all $\tau \leq t$

Overall complexity

Proportional to k

Complexity

- Iterate through all $\kappa \leq k$
- Each time iterate through all $i \leq n$
- Each time iterate through all $\tau \leq t$

Overall complexity

Proportional to $k\ n$

Complexity

- Iterate through all $\kappa \leq k$
- Each time iterate through all $i \leq n$
- Each time iterate through all $\tau \leq t$

Overall complexity

Proportional to $k\ n\ t$