Task Analysis

Let us first reformulate the task. We introduce set $A = 1..M$ of wires and set $B = 1..N$ of switches. (In the task it is given that $M = N$, but we will not use that fact when developing our programs.) The environment of the program knows a (total) function $f: A \to B$, which indicates for each wire $i \in A$ that it is connected to switch $f(i) \in B$. In Pascal terminology this context is set up as follows:

```
const
  MaxM = 90;
  MaxN = 90;

type
  A = 1..MaxM; { wires }
  B = 1..MaxN; { switches }

var
  { input variables }
  M: A; { intended A = 1..M }
  N: B; { intended B = 1..N }

  { output variable }
  f: array [A] of B;
```

The program has to reconstruct function f by doing measurements. With the probe command `T` the program can find out whether $f(i) \in C$ for wire $i \in A$ and set of switches $C \subseteq B$ (we call it the probe command here, because 'test' is an overloaded name). The set of (conducting) switches C is initially empty and is affected by the change command `C`. The main block of a reconstruction program will look like this:

```
var i: A;

begin
readln(M, N) ;
{ reconstruct f } ... ;
write('D') ;
for i := 1 to M do
  write(' ', f[i]:1) ;
writeln
end.
```

We abstract from the details of the probe and change commands by introducing function `probe` and procedure `change`:

```
function probe(i: A): boolean;
  var r: char;
  begin
  writeln('T ', i:1) ;
  readln(r) ;
  probe := (r = 'Y')
  end; { probe }

procedure change(j: B);
  var r: char;
  begin
  writeln('C ', j:1) ;
  readln(r) { ignore }
  end; { change }
```

We will now sketch two 'naive' methods for reconstructing f. In method $W1$ we determine for each wire the (unique) switch to which it is connected. In method $S1$ we determine for each switch all the wires that are connected to it. Here is a procedure for method $W1$:

```
type B0 = 0..MaxN;

procedure W1;
  var i: A; j: B0; p: boolean;
  begin
  for i := 1 to M do begin
    j := 0 ;
    repeat
      j := j + 1 ;
      change(j) ;
      { switch j conducting }
      p := probe(i) ;
      change(j)
      { switch j non-conducting }
      until p ;
    f[i] := j
    end { for i }
  end; { W1 }
```

Here is a procedure for method $S1$:

```
procedure S1;
  var i: A; j: B;
  begin
  for j := 1 to N do begin
    change(j) ;
    { switch j conducting }
    for i := 1 to M do
      if probe(i) then f[i] := j ;
    change(j)
    { switch j non-conducting }
    end { for j }
  end; { S1 }
```

Let us briefly compare these methods. In method *W*1 the change and probe commands are both in the inner loop, whereas in method *S*1 the change command is in the outer loop and only the probe command is in the inner loop. Method *S*1 always requires $2N$ changes and MN probes. For method *W*1 the numbers depend on the cable. In the worst case (all wires connected to switch N), it does $2MN$ changes and MN probes. In the best case (all wires connected to switch 1), $2M$ changes and M probes suffice. On average, method *W*1 does MN changes and $\frac{1}{2}MN$ probes, because, on average, the inner loop is broken off halfway.

Both methods can be refined. In method *W*1, switches are reset after each probe, but this is not necessary. One could probe each wire initially and then continue *changing* switches until the probe result *differs* from the initial probe. This is done in method *W*2:

```
procedure W2;
  var i: A; j: B0; p: boolean;
  begin
  for i := 1 to M do begin
    p := probe(i) ; j := 0 ;
    repeat
      j := j + 1 ;
      change(j)
      until probe(i) <> p ;
    f[i] := j
    end { for i }
  end; { W2 }
```

On average, method *W*2 probes $M + \frac{1}{2}MN$ times and changes $\frac{1}{2}MN$ times. A small further improvement (*big* improvements are kept for later) can be obtained by observing that the initial probe can be omitted in some cases and that once a wire has been probed negatively with $N-1$ switches, it must be connected to the one remaining switch. This is incorporated (cleverly) in method *W*3:

```
procedure W3;
  var i: A; j: B; p: boolean;
  begin
  for i := 1 to M do begin
    if (i=1) or (N=1) then p := false
    else p := probe(i) ;
    j := 1 ; f[i] := N ;
    while f[i] <> j do begin
      change(j) ;
      if probe(i) = p then j := j + 1
      else f[i] := j
      end { while }
    end { for i }
  end; { W3 }
```

Method *S*1 can be improved by skipping, in the inner loop, those wires that have already been determined. This also removes the need to make the switches non-conducting again. Furthermore, when $N-1$ switches have been covered, the remaining undetermined wires must be connected to the remaining switch. Here is the—surprisingly compact—code for method *S*2:

```
procedure S2;
  var i: A; j: B;
  begin
  for i := 1 to M do f[i] := N ;
  for j := 1 to N-1 do begin
    change(j) ;
    for i := 1 to M do begin
      if f[i] = N then
        if probe(i) then f[i] := j
    end { for j }
  end; { S2 }
```

Method *S*2 issues $N-1$ changes and, on average, approximately $\frac{1}{2}M(N-1)$ probes (because on average about $\frac{1}{2}M$ wires are probed in the inner loop) but certainly no more than $M(N-1)$ probes (in case all wires are connected to switch N). A small improvement can be obtained by stopping the outer loop as soon as all wires have been determined (the best case then issues 1 change and M probes).

The tables below summarize the performance of these methods ($W3$ has not been included because it is just a little better than $W2$). So far, method $S2$ seems the best, but for $M = N = 90$ the worst case still requires $90^2 = 8100$ commands (including 'Done').

Number of Changes

	best	average	worst
$W1$	$2M$	MN	$2MN$
$W2$	M	$\frac{1}{2}MN$	MN
$S1$		$2N$	
$S2$		$N-1$	

Number of Probes

	best	average	worst
$W1$	M	$\frac{1}{2}MN$	MN
$W2$	$2M$	$M+\frac{1}{2}MN$	$M+MN$
$S1$		MN	
$S2$	M	$\frac{1}{2}M(N-1)$	$M(N-1)$

How much further improvement can we expect? To answer this question let us look at the matter from another angle. The total number of possible functions f equals N^M (for each of the M wires there is a choice of N switches). Each probe gives at most one bit of information (since there are only two possible results for a probe: `Y` or `N`). On information-theoretical grounds, therefore, the minimum number of probes for reconstructing f is $\log_2(N^M) = M\log_2 N$, where $\log_2$ denotes the logarithm to base two. Whether this lower bound is feasible is another matter. This analysis also does not say anything about the number of change commands.

Observe the appearance of the factor $\log_2 N$, where the preceding methods obtained a factor N or $\frac{1}{2}(N-1)$ or so. This gives the impression that we should be able to squeeze more out of it. Let us concentrate on reducing the number of probes first.

In method $W1$, we used a *linear search* to find the switch to which a given wire is connected. A linear search through N objects requires, on average, $\frac{1}{2}N$ tests. However, when applicable, a *binary search* would need only $\log_2 N$ tests.

In this case, a binary search is indeed possible, because we can make a whole *set* of switches conducting before doing a probe. Say, we start with half the switches conducting and half non-conducting. Then we probe the wire and continue with the set of conducting switches if the probe yields true, and with set of the non-conducting switches otherwise. At every step the set of candidate switches is halved, until a singleton remains. Thus we find the desired switch in about $\log_2 N$ probes.

We can also arrive at this method in a more direct way. The knowledge that the program gathers about f can be captured by stating for each wire $i \in A$ to what *set* of switches it is possibly connected. Let us denote the candidate set for wire i by $G(i) \subseteq B$. Initially, we have $G(i) = B$ for every wire i. Function f is said to be compatible with 'state of knowledge' G when $f(i) \in G(i)$ for all $i \in A$. (In a sense, G generalizes f.)

Let $C \subseteq B$ be the set of conducting switches at the moment of a probe. If probing wire i yields true, then this reduces the set of candidate switches for i to $G(i) \cap C$. If the probe of i yields false, then this reduces the candidate set to $G(i) \cap (B - C) = G(i) - C$.

The number $F(G)$ of functions f compatible with G is readily computed as the product of the sizes of the sets $G(i)$ for $i \in A$. Initially $F(G)$ equals N^M, since $G(i) = B$ for all $i \in A$. The best we can hope to accomplish by a single probe is halving the number $F(G)$ (by halving the candidate set for the probed wire). Therefore, we must do a probe on some wire i with half of the switches in $G(i)$ conducting and the other half non-conducting.

Here is the code that uses a binary search to find the switch for a single wire i . The candidate set always consists of adjacent switches in the range `L..R`. It is assumed that initially all switches are non-conducting.

```
procedure BinarySearch(i: A);
var j, L, R, h: B; p: boolean;
begin
L := 1 ; R := N ; p := false ;
{ switches L..R are p-conducting }
while L <> R do begin
  h := (L + R) div 2 ;
  for j := L to h do change(j) ;
  { sw. L..h are non-p-conducting }
  { sw. h+1..R are p-conducting }
  if probe(i) = p then L := h+1
  else begin R := h ; p := not p end
  end { while } ;
f[i] := L
end; { BinarySearch }
```

Note that the state of the switches after execution of `BinarySeach` is rather haphazard. For repeated application they must be reset. This is most efficiently done inside the procedure by resetting conducting switches that are no longer candidates (this occurs when `probe(i)` yields false). When doing so, the total number of probes for reconstructing f (by invoking `BinarySearch` for each wire) comes at $M \log_2 N$ and the total number of changes at about $\frac{3}{2}MN$ (without resetting, the number of changes for determining a single wire is exactly $N-1$).

A second thing to note is that the possible intervals of switches `L..R` are not arbitrary. For instance, in case of ten switches, the interval 2..3 will never occur (the intervals 1..5, 1..3, and 3..3 can occur). The following table summarizes the intervals that occur when 'binary searching' the switch for some wire i among ten switches:

$N = 10$	switches									
phases	1	2	3	4	5	6	7	8	9	10
1	•	•	•	•	•	○	○	○	○	○
2	○	○	○	•	•	•	•	•	○	○
3	•	•	○	○	•	○	○	•	•	○
4	○	•				•	○			
changes	4	3	2	2	1	3	2	1	1	0

In phase 1, switches 1..5 (marked • in the table) are made conducting and switches 6..10 (marked ○) are kept non-conducting. Next, wire i is probed and we continue with that half of the interval in which its switch is now known to occur. The row labeled 'changes' counts the number of switch changes for each column. Altogether this method requires at most 19 changes and, on average, 3.4 probes per wire.

It is imaginable that the binary searches for all wires can be combined. Observe that when the switch for wire i is known to be, say, in the interval 6..8 then the states of the switches outside this interval are irrelevant to the result of probing i.

We will now describe a recursive procedure that combines all binary searches. For that purpose we introduce array `g` to record for each wire its interval of candidate switches:

```
var g: array [A] of
  record gL, gR: B end;
  { g[i].gL <= f[i] <= g[i].gR }
```

Array `g` generalizes f, but specializes G. Procedure `init_g` initializes the array:

```
procedure init_g;
  var i: A;
  begin
  for i := 1 to M do
    with g[i] do begin
      gL := 1 ; gR := N
      end { with }
  end; { init_g }
```

Procedure `BS1` does the reconstruction:

```
procedure BS1;
  var i: A;
  begin
  init_g ;
  AllBS1(1, N, false) ;
  for i := 1 to M do
    f[i] := g[i].gL
  end; { BS1 }
```

where procedure AllBS1 combines all binary searches:

```
procedure AllBS1(L, R: B; p: boolean);
  { switches L..R are p-conducting }
  var i: A; j, h: B;
  begin
  if L <> R then begin
    h := (L + R) div 2 ;
    for j := h+1 to R do change(j) ;
    { sw. L..h are p-conducting, }
    { h+1..R are non-p-conducting }
    for i := 1 to M do with g[i] do
      if (gL <= h) and (h <= gR) then
        if probe(i) = p then gR := h
        else gL := h+1 ;
    AllBS1(L, h, p) ;
    AllBS1(h+1, R, not p)
    end { if }
  end; { AllBS1 }
```

We have incorporated a small optimization to reduce the number of changes. In procedure `BinarySearch` the for-loop with `j` ranges over `L..h`, whereas now `j` ranges over `h+1..R`. When the range `L..R` contains an odd number of switches, there is some freedom to split it. Our choice in `AllBS1` minimizes the number of changes. The following table shows the states for ten switches:

$N = 10$	switches									
phases	1	2	3	4	5	6	7	8	9	10
1	○	○	○	○	○	●	●	●	●	●
2	○	○	○	●	●	●	●	●	○	○
3	○	○	●	●	○	●	●	○	○	●
4	○	●				●	○			
changes	0	1	1	1	2	1	2	2	2	3

Thus, there are 15 switch changes (instead of 19 in the preceding table). In general, the expected number of probes per wires is $\log_2 N$. The number of changes is approximately $\frac{1}{2} N \log_2 N$, because in each phase about half the switches are changed (this number only depends on N and not on the cable). The *expected* number of changes can be reduced by suppressing superfluous recursive calls. A recursive call is unnecessary when no wires are connected to switches in that interval. Here is the adapted code:

```
procedure AllBS2(L, R: B; p: boolean);
  { switches L..R are p-conducting }
  var i: A; j, h: B; u, v: boolean;
  begin
  if L <> R then begin
    h := (L + R) div 2 ;
    for j := h+1 to R do change(j) ;
    { sw. L..h are p-conducting, }
    { h+1..R are non-p-conducting }
    u := false ; v := false ;
    { u = # wires to sw. L..h > 0 }
    { v = # wires to sw. h+1..R > 0 }
    for i := 1 to M do with g[i] do
      if (gL <= h) and (h <= gR) then
        if probe(i) = p then begin
          gR := h ;
          u := true
          end { then }
        else begin
          gL := h+1 ;
          v := true
          end { else } ;
    if u then AllBS2(L, h, p) ;
    if v then AllBS2(h+1, R, not p)
    end { if }
  end; { AllBS2 }
```

Note that *BS*2 issues the same commands as *BS*1 when none of the recursive calls with $L \neq R$ happen to be suppressed. On average, *BS*2 performs better than *BS*1 because there are many cables for which recursive calls (with $L \neq R$) can be suppressed. For worst-case performance the comparison is more complicated.

The worst case for *BS*1 occurs when the number of probes is maximal, since the number of changes only depends on N. There are only two possibilities for the number of probes per wire: $\log_2 N$ rounded down and rounded up. In the example of ten switches, the worst cases are obtained by connecting wires only to switches 1, 2, 6, or 7 (appearing in all 4 phases and not just in 3). Such a worst case requires $15 + 10 * 4 + 1 = 56$ commands.

Worst-case performance of $BS2$ is more difficult to characterize. To obtain a situation no better than the worst case for $BS1$, all wires should be connected to 'maximal' switches *and* no recursive calls with $L \neq R$ should be suppressed. For $N = 6$ and $M \geq 2$, this is indeed possible (see table below): connect one wire to switch 1 and the others to switch 4.

	$N=5$					$N=6$					
ph.	1	2	3	4	5	1	2	3	4	5	6
1	○	○	○	●	●	○	○	○	●	●	●
2	○	○	●	●	○	○	○	●	●	●	○
3	○	●				○	●		●	○	
ch.	0	1	1	1	2	0	1	1	1	2	2

For $N = 5$, however, it is impossible: *either* no wires are connected to switches in the range 4..5 (and the recursive call `AllBS2(4, 5, ...)` is suppressed, avoiding one change) *or*, otherwise, at least one wire takes only 2 probes instead of 3. It turns out that $BS2$'s worst-case performance is no better than $BS1$'s when $3 * 2^k \leq N \leq 4 * 2^k$.

Another way to arrive at a logarithmic method is based on writing the switch labels (minus 1, for best results) in binary notation and using the bits to decide on the switch state for each phase. The following table shows method BN at work for $N = 10$ and phases done most significant bit (MSB) first:

$N=10$	switches									
phases	1	2	3	4	5	6	7	8	9	10
1	○	○	○	○	○	○	○	○	●	●
2	○	○	○	○	●	●	●	●	○	○
3	○	○	●	●	○	○	●	●	○	○
4	○	●	○	●	○	●	○	●	○	●
changes	0	1	2	1	2	3	2	1	2	3

When the number of switches is a power of two, method BN is equivalent to the preceding one. In general, however, it requires, on average, more changes and probes (for $N = 10$, it takes 17 changes, and 4 probes per wire). Doing the phases in the reverse order, least significant bit (LSB) first, is even slightly worse. BN's code is given below. Parameter `rev` determines the order of the phases. Array `s` keeps track of the switch states to facilitate efficient switch setting. Note that `odd(a div 2^b)` yields the `b`-th bit of `a`.

```
procedure BN(rev: boolean);
  { if rev then LSB first else MSB }
  var
    s: array [B] of boolean;
    { s[j] == sw. j is conducting }
    i: A; j: B; w, k: integer;
  begin
  for i := 1 to M do f[i] := 1 ;
  for j := 1 to N do s[j] := false ;
  w := 1 ; k := 0 ; { w = 2^k }
  while w < N do begin
    w := 2*w ; k := k + 1
    end { while } ;
  { k = # phases }
  if rev then w := 1
  else w := w div 2 ;
  { w = bit weight }
  while k <> 0 do begin
    { adjust switches }
    for j := 1 to N do
      if odd((j-1) div w) <> s[j]
      then begin
        change(j) ;
        s[j] := not s[j]
        end { if } ;
    { adjust f }
    for i := 1 to M do
      if probe(i) then
        f[i] := f[i] + w ;
    if rev then w := 2*w
    else w := w div 2 ;
    k := k - 1
    end { while }
  end; { BN }
```

History and Variations

This task was conceived while moving into my new home. Electricity wires run from each wall outlet and light fixture to a box of switches (and fuses) in the basement (in my case, ten switches). I had to reconstruct the connection pattern because the previous owner had not committed it to paper. A probe command involves walking to a particular room to test an outlet or fixture. A change command involves walking to the basement and flipping a switch. This also causes a power cycle on all equipment connected to the switch. I wanted to minimize both, and started thinking about clever methods.

The task can be generalized in several ways. Consider sets A and B with M and N elements respectively, and a relation $R{:}A \leftrightarrow B$, that is, $R \subseteq A \times B$. The program's task is to reconstruct R by doing measurements.

In the original task, R is restricted to a total function from A to B. The table below gives the number of possible relations R for various constraints on R:

constraint on R	number of Rs
none	2^{MN}
left total	$(2^M - 1)^N$
right total	$(2^N - 1)^M$
partial function	$(N+1)^M$
total function	N^M
injection ($M \le N$)	$\frac{N!}{(N-M)!}$
surjection ($M \ge N$)	$\frac{N(M-1)!}{(M-N)!}$
bijection ($M = N$)	$N!$

On information-theoretical grounds, the base-two logarithm of the number of possible relations is a lower bound on the number of probes required for reconstruction. Only for the cases of 'no constraint' and of a partial or total function is it easy to attain this lower bound. The case of 'no constraint' is not very interesting because you cannot do better than probe every wire-switch pair. The case of a bijection (that is, a wire permutation) is particularly intriguing, but most likely it is as complicated as minimum-comparison sorting.

Also the possibilities for doing measurements can be varied. For instance, one could introduce switches on side A as well. Furthermore, other cost functions can be used: for instance, a probe command is ten times as expensive as a change command. Finally, it can be required that the switches are back in their initial state when the program terminates.

Motivations and Judging

I have chosen for total functions because they are easy to explain, the task is not trivial, and it is free from the complications surrounding permutations. The restriction $M = N$ was imposed to simplify the input (allowing uniform treatment of Pascal, C++, and BASIC) and the complexity analysis.

I have chosen the lower bounds on M (and N) equal to one, because that avoids the (trivial and somewhat confusing) boundary case zero but includes the boundary case one (which needs care in some logarithmic approaches). I have chosen the upper bound approximately 100 because that is big enough to clearly distinguish quadratic and logarithmic methods, yet small enough to impose a relatively small bound on the total number of commands (namely, approximately 1000).

I wanted the best competitors to find $BS2$ (not just $BS1$) and I wanted 'nice unrevealing' numbers for the upper bounds. The reason for choosing upper bound 90, instead of 100, is that (i) $BS2$ outperforms $BS1$ (worst-case) for $65 \le N \le 95$ but not for $96 \le N \le 128$, and (ii) $BS2$ 'accidentally' requires at worst 900 commands for $N = 90$, whereas for other values of N this upper bound is not such a 'nice' number.

The performance bound (on the number of commands) is fixed, that is, independent of the actual values of M and N. The tests will be chosen such that quadratic methods succeed for at least one but not all cases. I also wanted at least one test case for which a 'naive' logarithmic method (such as *BN*) fails (because it does not sufficiently reduce the number of changes).

Some tests will be played against 'Teaser', a program that tries to give away as little information as possible, thereby eliciting worst-case behavior. Teaser does not choose a function at the start, but answers the probes such that $F(G)$ is reduced as little as possible. When both probe answers give equal reductions (as for most logarithmic methods), Teaser tries to maximize the number of changes by a 'voting' scheme. In a sense, Teaser lets the program under test construct its own worst-case function f.

Some input will be chosen randomly to evaluate average-case behavior, avoiding special cases that cause exceptionally good or bad performance.

The following tables summarize the performance of the quadratic methods:

Number of Commands

	average	worst
$W1$	$\frac{3}{2}M^2+1$	$3M^2+1$
$W2$	M^2+M+1	$2M^2+M+1$
$S1$	M^2+2M+1	
$S2$	$\frac{1}{2}M^2+\frac{1}{2}M$	M^2

Maximum feasible M

	average	worst
$W1$	24	17
$W2$	29	20
$S1$	29	
$S2$	44	30

Here is a table summarizing the ten test cases:

nr	*M*	*f*	Failure target
1	1	*C*	incorrect (esp. log.)
2	15	*R*	incorrect
3	21	*T*	very ineff.
4	30	*T*	ineff. S-meth.
5	35	*R*	ineff. avg. W-meth.
6	40	*R*	ineff. avg. S-meth.
7	80	*T*	non-logarithmic
8	86	*T*	ineff. binary notation
9	89	*T*	binary notation
10	90	*T*	non-suppressing

Column '*nr*' gives the sequence number of the test case, column '*M*' gives the number of wires, column '*f*' describes the connectivity function (*C*=wire 1 to switch 1; *R*=random; *T*=teasing), and the rightmost column indicates which methods are intended to fail. The test cases are roughly arranged in order of increasing severity: when a program fails a test it, usually, will also fail all subsequent tests.

Addendum

After IOI'95, several people (some of them participants!) have pointed out to me that it is possible to do better than *BS*2. For example, the case of $M = N = 90$ can be solved in a worst-case of 878 commands, which is well below the 900 commands allowed (and required by *BS*2).

The idea is to do an *asymmetric* binary search (*ABS*), changing fewer than half the switches in such a way that the additional number of probes still gives an optimal result. The optimum degree of asymmetry can be determined by dynamic programming. This takes considerably more time and memory than *BS*2, but could be done as preprocessing. Nobody actually succeeded with this approach at IOI'95.

The smallest case improved by *ABS* is for $M = 2$ and $N = 6$: *BS*2 needs 14 commands instead of the optimum 13. In the first phase it is optimal to change only two (instead of three) of the six switches:

$N = 6$	switches					
phases	1	2	3	4	5	6
1	○	○	○	○	●	●
2	○	○	●	●	●	○
3	○	●	●	○		
changes	0	1	1	2	1	2

Compare this table to the 'symmetric' tables above for $N = 5$ and $N = 6$. The maximum number of commands (changes plus probes plus 'done') now still seems to be $7 + 6 + 1 = 14$. However, this can never occur in the asymmetric case, because either a wire is connected to a switch in the range 5..6 (in which case we save a probe), or no wire is connected to switches in the range 5..6 (in which case we save a change). Note the similarity to the reason given above for why $BS2$ performs no better than $BS1$ with $N = 5$.

Let $BS2[i,j]$ and $ABS[i,j]$ be the worst-case number of changes-plus-probes (not counting 'done') for solving the problem with i wires and j switches, using $BS2$ and an optimized ABS, respectively. Observe that these worst-case numbers can be determined by considering the situation where s of the j switches are changed and all wires are probed, such that k of the i wires turn out to be connected to the changed switches. The numbers $BS2[i,j]$ satisfy the recurrence relation:

$$\begin{aligned} &BS2[i,j] = \\ &\quad s + i + (\mathbf{max}\, k : 0 \le k \le i \\ &\qquad\qquad : BS2[k,s] + BS2[i-k, j-s]) \end{aligned}$$

where $s = j\ \mathbf{div}\ 2$. The numbers $ABS[i,j]$ satisfy the recurrence relation:

$$\begin{aligned} &ABS[i,j] = \\ &\quad (\mathbf{min}\, s : 1 \le s \le j\ \mathbf{div}\ 2 \\ &\qquad : s + i + H(i,j,s)) \end{aligned}$$

where

$$\begin{aligned} &H(i,j,s) = \\ &\quad (\mathbf{max}\, k : 0 \le k \le i \\ &\qquad : ABS[k,s] + ABS[i-k, j-s]) \end{aligned}$$

The program `WORST.PAS` determines $BS2[i,j]$, $ABS[i,j]$, and an optimal number s of switches to change for ABS (a minus sign indicates a deviation from the symmetric split of $BS2$). For $M = N = 90$, the output file `TABLE.TXT` shows that an optimal number of switches to change in the first phase is only 38 (instead of 45 used by $BS2$).

Tom Verhoeff