```
The WoodCutting Problem
=======================

(the translation of "WC2.txt" to a mathematical problem,
see "WC2.txt" for constant and type declarations)

// enumeration/algebraic types for the decisions
type ReorderDecision = { none, forward, swap };
type DiscardDecision = { none, keep, discard };
type AssemblyDecision = { none, accept };
type CutDecision = { none } ∪ { cut(cnum,cuts) | cnum: CutIndex, cuts: Cuts }

The Problem
===========

Given:

  ibnum: InBoardIndex,
  inboards: InBoards,
  obnum: OutBoardIndex,
  outboards: OutBoards, // unconstrained at indices >= obnum
  bempty: Bool,
  buffer: Board,
  ipnum: InPieceIndex,
  inpieces: InPieces, // unconstrained at indices >= ipnum
  opnum: OutPieceIndex,
  outpieces: OutPieces, // unconstrained at indices >= ipnum
  gints: GlobalIntervals

Find:

  rds: Array[RDNUM,ReorderDecision],
  cds: Array[CDNUM,CutDecision],
  dds: Array[DDNUM,DiscardDecision]

Where:

  rds,cds,dds = argmin_(
      rds:Array[RDNUM,ReorderDecision],
      cds: Array[CDNUM,CutDecision],
      dds: Array[DDNUM,DiscardDecision]).
    choose cost: Cost.
    ∃ibnum':InBoardIndex, obnum':OutBoardIndex.
    ∃bempty': Bool, buffer': Board.
    ∃ipnum': InPieceIndex, opnum': OutPieceIndex.
    Main(rds,cds,dds,ibnum,ibnum',inboards,obnum,obnum',outboards,
      bempty,bempty',buffer,buffer',
      ipnum,ipnum',
      inpieces,
      opnum,opnum',
      outpieces, gints,
      0, cost)

The Predicates
```

```
==============

// the production line (consisting of multiple "stages")
Main(
  rds: Array[RDNUM,ReorderDecision],
  cds: Array[CDNUM,CutDecision],
  dds: Array[DDNUM,DiscardDecision],
  ibnum: InBoardIndex, ibnum':InBoardIndex,
  inboards: InBoards,
  obnum: OutBoardIndex, obnum':OutBoardIndex,
  outboards: OutBoards, // unconstrained at indices >= obnum
  bempty: Bool, bempty': Bool,
  buffer: Board, buffer': Board,
  ipnum: InPieceIndex, ipnum': InPieceIndex,
  inpieces: InPieces, // unconstrained at indices >= ipnum
  opnum: OutPieceIndex, opnum': OutPieceIndex,
  outpieces: OutPieces, // unconstrained at indices >= ipnum
  gints: GlobalIntervals,
  cost: Cost, cost':Cost,
) ⇔
(
  // try at most RDNUM reordering decisions (if no action is possible,
  // perform a "dummy" action that leaves the state unchanged)

  ∃ibnums: Array[RDNUM+1,InBoardIndex].
  ∃obnums: Array[RDNUM+1,OutBoardIndex].
  ∃bemptys: Array[RDNUM+1,Bool].
  ∃buffers: Array[RDNUM+1,Board].
  ∃ipnums: Array[CDNUM+1,InPieceIndex].
  ∃opnums: Array[DDNUM+1,OutPieceIndex].
  ∃costs: Array[DDNUM+1,Cost].

  ibnums[0] = inbum ∧ ibnums[RDNUM] = ibnum' ∧
  obnums[0] = obnum ∧ obnums[RDNUM] = obnum' ∧
  bemptys[0] = bempty ∧ bemptys[RDNUM] = bempty' ∧
  buffers[0] = buffer ∧ buffers[RDNUM] = buffer' ∧
  inpnums[0] = ipnum ∧ inpnums[CDNUM] = ipnum' ∧
  outpnums[0] = opnum ∧ outpnums[DDNUM] = opnum' ∧
  costs[0] = cost ∧ costs[DDNUM] = cost' ∧

  ∀i:Int[0,RDNUM-1].
  (
    if (ibnum < IBNUM ∧ obnum < OBNUM) ∨ (ibnum < IBNUM ∧ (bempty ∨ obnum < OBNUM)) then
      Reorder(rds[i],
        ibnums[i],ibnums[i+1],inboards,
        obnums[i],obnums[i+1],outboards,
        bemptys[i],bemptys[i+1],buffers[i],buffers[i+1])
    else
      rds[i] = none ∧
      ibnums[i+1] = ibnums[i] ∧ obnums[i+1] = obnums[i] and
      bemptys[i+1] = bemptys[i] ∧ buffers[i+1] = buffers[i]
  ) ∧

  // try at most CDNUM cutting decisions (each with at most CNUM cut positions)
```

```
  ∀i:Int[0,CDNUM-1].
  (
    if i < obnum then
      Cut(cds[i],obnums[RDNUM],outboards,ipnums[i],ipnums[i+1],sinpieces)
    else
      cds[i] = none ∧
      ipnums[i+1] = ipnums[i]
  ) ∧

  // try at most DDNUM discarding decisions
  ∀i:Int[0,DDNUM-1].
  (
    if (i < ipnum ∧ opnum < OPNUM) ∨ (i < ipnum) then
      Discard(dds[i],i,ipnums[CDNUM],inpieces,opnums[i],opnums[i+1],outpieces,
        costs[i],costs[i+1])
    else
      dds[i] = none ∧
      opnums[i+1] = opnums[i] ∧ costs[i+1] = costs[i]
  ) ∧

  // try at most ADNUM assembly decisions
  ∃blens: BeamLengths.
  ∃blen: Array[ADNUM+1,BeamLength].
  ∃bnum: Array[ADNUM+1,BeamIndex];
  ∃bdepth: Array[ADNUM+1,BeamDepth];
  ∃bnum0: Array[ADNUM+1,BeamIndex];
    blen[0] = 0 ∧ bnum[0] = 0 ∧ bdepth[0] = 0 ∧ bnum0[0] = 0 ∧
  ∀i:Int[0,ADNUM-1].
  (
    if i < opnum then
      Assembly(accept,i,opnum,outpieces,gints,blens,
        blen[i],blen[i+1],bnum[i],bnum[i+1],
        bdepth[i],bdepth[i+1],bnum0[i],bnum0[i+1])
    else
      blen[i+1] = blen[i] ∧ bnum[i+1] = bnum[i] ∧
      bdepth[i+1] = bdepth[i] ∧ bnum0[i+1] = bnum0[i]
  )
);

// the reordering stage
stage Reorder(d: ReorderDecision,
  ibnum: InBoardIndex, ibnum':InBoardIndex,
  inboards: InBoards,
  obnum: OutBoardIndex, obnum':OutBoardIndex,
  outboards: OutBoards, // unconstrained at indices >= obnum
  bempty: Bool, bempty': Bool,
  buffer: Board, buffer': Board
) ⇔
(
  (d = forward ⇒
    ibnum < IBNUM ∧ obnum < OBNUM ∧
    let board: Board = inboards[ibnum] in
    ibnum' = ibnum+1 ∧
    obnum' = obnum+1 ∧
```

```
    outboards[obnum] = board ∧
    bempty' = bempty ∧ buffer' = buffer
  ) ∧
  (d = swap ⇒
    ibnum < IBNUM ∧ (bempty ∨ obnum < OBNUM) ∧
    let board: Board = inboards[ibnum] in
    ibnum' = ibnum+1 ∧
    obnum' = if bempty then obnum else obnum+1 ∧
    (¬bempty ⇒ outboards[obnum] = buffer) ∧
    bempty' = false ∧
    buffer' = board;
  )
);

// the cutting stage
stage Cut(d: CutDecision,
  i: OutBoardIndex,
  obnum: OutBoardIndex,
  outboards: OutBoards,
  ipnum: InPieceIndex, ipnum': InPieceIndex,
  inpieces: InPieces // unconstrained at indices >= ipnum
) ⇔
(
  exists cnum:CutIndex, cuts:Cuts. d = cut(cnum,cuts) ⇒
  (
    i < obnum ∧
    ipnum+cnum ≤ IPNUM ∧
    let board: Board = outboards[i] in
    (∀j: CutIndex. j < board.cnum ⇒
      let cint: InterVal = board.cints[j] in
      ∃k: CutIndex. k < cnum ∧
        let cut:Cut = cuts[k] in
        cint.1 ≤ cut ∧ cut ≤ cint.2) ∧
    ipnum' = ipnum+cnum ∧
    (∀j: CutIndex. j < cnum ⇒
      let start: Length = if j = 0 then 0 else cut[j-1] in
      inpieces[ipnum+j] = cut[j]-start)
  )
);

// the discarding stage
Discard(d: DiscardDecision,
  i: InPieceIndex,
  ipnum: InPieceIndex,
  inpieces: InPieces,
  opnum: OutPieceIndex, opnum': OutPieceIndex
  outpieces: OutPieces; // unconstrained at indices >= opnum
  cost: Cost, cost': Cost
) ⇔
(d = keep ⇒
  (
    i < ipnum ∧ opnum < OPNUM ∧
    let piece: Piece = inpieces[i] in
    opnum' = opnum+1 ∧
```

```
    outpieces[opnum] = piece ∧
    cost' = cost
  )
) ∧
(d = discard ⇒
  (
    i < ipnum ∧
    let piece: Piece = inpieces[i] in
    cost' = cost+piece ∧
    opnum' = opnum;
  )
);

// the assembly stage
Assembly(d: AssemblyDecision,
  i: OutPieceIndex,
  opnum: OutPieceIndex,
  outpieces: OutPieces,
  gints: GlobalIntervals,
  blens: BeamLengths, // unconstrained at indices >= i
  blen: BeamLength, blen': BeamLength,
  bnum: BeamIndex, bnum': BeamIndex,
  bdepth: BeamDepth, bdepth': BeamDepth,
  bnum0: BeamIndex, bnum0': BeamIndex
) ⇔
(d = accept ⇒ (
  i < opnum ∧
  let blen0: BeamLength = blen+outpieces[i] in
  blens[i] = blen0 ∧
  blen0 ≤ BLEN ∧
  (¬∃j:GlobalIndex. j < GNUM ∧
    gints[j].1 <= blen0 ∧ blen0 <= gints[j].2) ∧
  (∀j:BeamIndex. j < bnum0 ⇒
    let diff: BeamLength = blen0-blens[i-bnum-bnum0+j] in
    DIFF ≤ if diff ≥ 0 then diff else -diff) ∧
  if blen0 < BLEN then
    blen' = blen0 ∧
    bnum' = bnum+1 ∧
    bdepth' = bdepth ∧ bnum0' = bnum0
  else
    blen' = 0 ∧
    bnum' = 0 ∧
    if bdepth = BDEPTH then
      bdepth' = 0 ∧
      bnum0' = 0
    else
      bdepth' = bdepth+1 ∧
      bnum0' = bnum
  )
);
```