```
system WoodCutting
{
  const LEN: Int; // maximum length of a board
  type Length = Int[0,LEN];

  const CNUM: Int; // maximum number of cut intervals (and thus cuts) per board
  type CutIndex = Int[0,CNUM];
  type Cuts = Array[CNUM,Length];

  const GNUM: Int; // number of globally prohibited intervals in board
  type GlobalIndex = Int[0,GNUM];
  type GlobalIntervals = Array[GNUM,Interval];

  const DIST: Length; // minimum distance between two cuts

  type IntervalType = { bad, weak };
  type Interval = Record[type:IntervalType,from:Length,to:Length];
  type CutIntervals = Array[CNUM,Interval];
  type Board = Record[length:Length,cnum:CutIndex,cints:CutIntervals];

  const IBNUM: Int; // maximum number of boards before reordering stage
  type InBoardIndex = Int[0,IBNUM];
  type InBoards = Array[IBNUM,Board];

  const OBNUM: Int; // maximum number of boards after reordering stage
  type OutBoardIndex = Int[0,OBNUM];
  type OutBoards = Array[OBNUM,Board];

  type InPiece = Record[good:Bool, length:Length];
  type OutPiece = Length;

  const IPNUM: Int; // maximum number of pieces before discarding
  type InPieceIndex = Int[0,IPNUM];
  type InPieces = Array[IPNUM,InPiece];

  const OPNUM: Int; // maximum number of pieces after discarding
  type OutPieceIndex = Int[0,OPNUM];
  type OutPieces = Array[OPNUM,OutPiece];

  const APNUM: Int; // maximum number of pieces to assemble
  type AssemblyPieceIndex = Int[0,APNUM];
  type AssemblyPieces = Array[APNUM,OutPiece];

  const BLEN: Int;                // desired length of a beam
  type BeamLength = Int[0,BLEN];  // actual length of beam
  const BDEPTH: Int;              // desired number of layers
  type BeamDepth = Int[0,BDEPTH]; // actual number of layers
  const BNUM: Int;                // maximum number of pieces per beam
  type BeamIndex = Int[0,BNUM];   // actual number of pieces
  type BeamLengths = Array[OPNUM,BeamLength];

  // may be used to limit the decision search space
  const RBDNUM = IBNUM; // number of reordering boards decisions (<= IBNUM)
  const CBDNUM = OBNUM; // number of cutting boards decisions (<= OBNUM)
  const DPDNUM = IPNUM; // number of discarding pieces decisions (<= IPNUM)
  const RPDNUM = IPNUM; // number of reordering pieces decisions (<= IPNUM)
  const APDNUM = OPNUM; // number of assembling decisions (<= OPNUM)

  type Cost = Real; // need not be bounded

  // the production line (consisting of multiple "stages")
  pipeline main(
    inout ibnum: InBoardIndex,
    in inboards: InBoards,
    inout obnum: OutBoardIndex,
    in outboards: OutBoards, // unconstrained at indices >= obnum
    inout bempty: Bool,
    inout buffer: Board,
    inout ipnum: InPieceIndex,
    in inpieces: InPieces,   // unconstrained at indices >= ipnum
    inout opnum: InPieceIndex,
    in outpieces: OutPieces, // unconstrained at indices >= ipnum
    inout apnum: AssemblyPieceIndex,
    in apieces: AssemblyPieces, // unconstrained at indices >= apnum
    inout pempty: Bool,
    inout pbuffer: Piece,
    in gints: GlobalIntervals,
    inout cost: Cost
  )
  {
    // try at most RBDNUM reordering decisions (if no action is possible,
    // perform a "dummy" action that leaves the state unchanged)
    for i:Int[0,RBDNUM-1] do
    {
      try ReorderBoards(ibnum,inboards,obnum,outboards,bempty,buffer);
    }

    // try at most CBDNUM cutting decisions (each with at most CNUM cut positions)
    for i:Int[0,CBDNUM-1] do
    {
      try Cut(i,obnum,outboards,ipnum,inpieces);
    }

    // try at most DPDNUM discarding decisions
    for i:Int[0,DPDNUM-1] do
    {
      try Discard(i,ipnum,inpieces,opnum,outpieces,cost);
    }

    // try at most RPDNUM reordering decisions
    for i:Int[0,RPDNUM-1] do
    {
      try AssemblyPieces(opnum,outpieces,apnum,apieces,pempty,pbuffer);
    }

    // try at most APDNUM assembly decisions
    val blens: BeamLengths;
    var blen: BeamLength = 0;
    var bnum: BeamIndex = 0;
    var bdepth: BeamDepth = 0;
    var bnum0: BeamIndex = 0;
    for i:Int[0,APDNUM-1] do
    {
      try Assembly(i,apnum,apieces,gints,blens,blen,bnum,bdepth,bnum0);
    }
  }

  // the first reordering stage
  stage ReorderBoards(
    inout ibnum: InBoardIndex,
    in inboards: InBoards,
    inout obnum: OutBoardIndex,
    in outboards: OutBoards, // unconstrained at indices >= obnum
    inout bempty: Bool,
    inout buffer: Board
  )
  {
    action forward()
    requires ibnum < IBNUM && obnum < OBNUM;
    {
      in board: Board = inboards[ibnum];
      ibnum' = ibnum+1;
      obnum' = obnum+1;
      outboards[obnum] = board; // equality, not assignment!
      unchanged bempty, buffer;
    }
    action swap()
    requires ibnum < IBNUM && (bempty || obnum < OBNUM);
    {
      in board: Board = inboards[ibnum];
      ibnum' = ibnum+1;
      obnum' = if bempty then obnum else obnum+1;
      !bempty => outboards[obnum] = buffer;
      bempty' = false;
      buffer' = board;
    }
  }

  // the cutting stage
  stage Cut(
    in i: OutBoardIndex,
    in obnum: OutBoardIndex,
    in outboards: OutBoards,
    inout ipnum: InPieceIndex,
    in inpieces: InPieces // unconstrained at indices >= ipnum
  )
  {
    action cut(cnum:CutIndex,cuts:Cuts)
      requires i < obnum;
    {
      constraint ipnum+cnum <= IPNUM;
      val board: Board = outboards[i];
      constraint forall j: CutIndex with j < board.cnum.
        cuts[j] <= board.length && (j+1 < board.cnum => cuts[j] < cuts[j+1]);
      constraint forall j: CutIndex with j < board.cnum.
        val cint: Interval = board.cints[j];
        if cint.type = weak then
          exists k: CutIndex with k < cnum.
            cint.from <= cuts[k] && cuts[k] <= cint.to;
        else // cint.type = bad then
          exists k: CutIndex with k < cnum-1.
            cint.from = cut[k] && cint.to = cut[k+1];
      ipnum' = ipnum+cnum;
      constraint forall k: CutIndex with k < cnum.
        var inpiece: Inpiece = inpieces[ipnum+j].
        val start: Length = if k = 0 then 0 else cut[k-1];
        val end: Length = cut[k];
        inpiece.length = end-start;
        inpiece.type =
           exists j: CutIndex with j < board.cnum.
             val cint: Interval = cints[j].
             cint.type = bad && cint.from <= start && end <= cint.to;
    }
  }

  // the discarding stage
  stage Discard(
    in i: InPieceIndex,
    in ipnum: InPieceIndex,
    in inpieces: InPieces,
    inout opnum: OutPieceIndex,
    in outpieces: OutPieces; // unconstrained at indices >= opnum
    inout cost: Cost;
  )
  {
    action keep()
    requires i < ipnum && opnum < OPNUM;
    {
      val piece: Piece = inpieces[i];
      constraint piece.good;
      opnum' = opnum+1;
      outpieces[opnum] = piece.length; // equality, not assignment!
      unchanged cost;
    }
    action discard()
    requires i < ipnum;
    {
      val piece: Piece = inpieces[i];
      cost' = cost+piece;
      unchanged opnum;
    }
  }

    // the second reordering stage
  stage AssemblyPieces(
    inout opnum: OutPieceIndex,
    in outpieces: OutPieces,
    inout apnum: AssemblyPieceIndex,
    in apieces: AssemblyPieces, // unconstrained at indices >= apnum
    inout pempty: Bool,
    inout pbuffer: Piece
  )
  {
    action forward()
    requires opnum < OPNUM && apnum < APNUM;
    {
      in piece: Piece = outpieces[opnum];
      opnum' = opnum+1;
      apnum' = apnum+1;
      apieces[apnum] = piece; // equality, not assignment!
      unchanged pempty, pbuffer;
    }
    action swap()
    requires apnum < APNUM && (pempty || apnum < APNUM);
    {
      in piece: Piece = outpieces[opnum];
      opnum' = opnum+1;
      apnum' = if pempty then apnum else apnum+1;
      !pempty => apieces[apnum] = pbuffer;
      pempty' = false;
      pbuffer' = piece;
    }
  }

  // the assembly stage
  stage Assembly(
    in i: AssemblyPieceIndex,
    in apnum: AssemblyPieceIndex,
    in apieces: AssemblyPieces,
    in gints: GlobalIntervals,
    in blens: BeamLengths, // unconstrained at indices >= i
    inout blen: BeamLength,
    inout bnum: BeamIndex,
    inout bdepth: BeamDepth,
    inout bnum0: BeamIndex
  )
  {
    action accept()
    requires i < apnum;
    {
      val blen0: BeamLength = blen+apieces[i];
      blens[i] = blen0;
      constraint blen0 <= BLEN;
      constraint !exists j:GlobalIndex with j < GNUM.
        gints[j].1 <= blen0 && blen0 <= gints[j].2;
      constraint forall j:BeamIndex with j < bnum0.
        value diff: BeamLength = blen0-blens[i-bnum-bnum0+j];
        DIFF <= if diff >= 0 then diff else -diff;
      if blen0 < BLEN then
        blen' = blen0;
        bnum' = bnum+1;
        unchanged bdepth, bnum0;
      else
        blen' = 0;
        bnum' = 0;
        if bdepth = BDEPTH then
          bdepth' = 0;
          bnum0' = 0;
        else
          bdepth' = bdepth+1;
          bnum0' = bnum;
    }
  }
}
```