PROBLEMA DIFÍCil 2

1 Análise Inicial

Para o último problema, foi pedido para analisar os cheques presentes em tabuleiros de xadrez enviados pelo usuário até que se recebesse um tabuleiro vazio. Primeiramente, pensou-se em criar uma matriz de caracteres para representar o tabuleiro e percorrê-lá verificando se cada peça encontrada teria ”visão”do rei inimigo. Porém, essa abordagem seria muito ineficiente, então decidiu fazer a verificação a partir das coordenadas de cada rei, visto que assim a quantidade de casas que seriam percorridas é muito menor.

2 Estruturas

Como será necessário analisar as posições de peças em uma matriz, criou-se uma estrutura que guarda coordenadas x e y em uma única variável. Isso foi feito para facilitar a manipulação da matriz.

```
typedef struct Coord_t{ //Usou-se a estrutura de coordenadas para facilitar o manejo das
variaveis dentro da matriz
  int x,y;
} coord;
```

Figura 1: Estrutura de Coordenadas

3 Enumeração

Visto que as possibilidades de saídas para esse problema são o caso em que o rei branco está em cheque, o caso em que o rei preto está em cheque, o caso em que nenhum rei está em cheque e o caso em que o tabuleiro está vazio, decidiu-se fazer uma enumeração desses casos para tornar o código mais legível e facilitar sua depuração.

```
enum{ //Usou-se a enumeracao para deixar o codigo mais didatico e legivel
  NO_KINGS = -1,
  NO_CHECKS,
  WHITE_CHECK,
  BLACK_CHECK
};
```

Figura 2: Enumeração de Casos Possíveis

4 Funções

4.1 Preparação da Matriz de Tabuleiro

Como o problema requer a manipulação de uma matriz de maneira precisa, é de extrema importância se certificar que ela é preparada de maneira adequada.

Para isso, primeiramente, foram feitas uma função que inicializa a matriz com pontos e uma que printa todas as linhas da matriz. Quando a inicialização do tabuleiro foi considerada satisfatória, fez-se também uma função especificamente para ler as linhas de entrada do usuário e armazená-las na matriz. Essas funções foram de grande importância para a depuração inicial do código e evitaram problemas futuros.

```
void inicializa_tabuleiro(char tab[8][8]){ //Reseta o tabuleiro
  for(int i = 0; i < 8; i++)
    for(int j = 0; j < 8; j++)
      tab[i][j] = '.';
}

void printa_tabuleiro(char tab[8][8]){ //Printa as linhas do tabuleiro, usada para depuracao
do codigo
  for(int i = 0; i < 8; i++){
    for(int j = 0; j < 8; j++)
      printf("%c",tab[i][j]);
    printf("\n");
  }
}

void preenche_tabuleiro(char tab[8][8]){//Preenche o tabuleiro de acordo com as entradas do
usuario
  for(int i = 0; i < 8; i++){
    scanf("%s", tab[i]); //Le cada linha
    while(getchar() != '\n' && getchar() != EOF); //Limpa o buffer
  }
}
```

Figura 3: Funções de Preparação da Matriz

4.2 Coordenadas dos Reis

Como comentado anteriormente, decidiu-se abordar o problema analisando o tabuleiro a partir da coordenada de cada rei, logo, é importante ter uma função que consegue armazenar essas coordenadas em variáveis de maneira simples e eficiente.

Com isso em mente, a função *get_kings* recebe o tabuleiro e dois endereços de variáveis de coordenadas, uma para o rei branco (”K”) e uma para o rei preto (”k”). Ela inicializa as

coordenadas dos reis na posição (-1,-1) para denunciar os casos em que algum dos reis não esteja no tabuleiro. Após isso, o tabuleiro é percorrido verificando se algum espaço é um rei. Quando um rei é encontrado, suas coordenadas são armazenadas na variável correspondente e conta-se em um contador. O contador é usado para tornar o processo mais eficiente, pois assim é possível parar de percorrer o tabuleiro assim que ambos os reis forem encontrados.

```
void get_kings(char tab[8][8], coord *K, coord *k){ //Guarda as coordenadas dos reis nas
variaveis K e k
  int ctd = 0;
  K->x = -1; //Inicializa as coordenadas dos reis em (-1,-1)
  K->y = -1;
  k->x = -1;
  k->y = -1;

  for(int i = 0; i < 8; i++) //Percorre a matriz ate encontrar os dois reis
    for(int j = 0; j < 8; j++){
      if(tab[i][j] == 'K'){
        K->x = j;
        K->y = i;
        ctd++;
      }
      else if(tab[i][j] == 'k'){
        k->x = j;
        k->y = i;
        ctd++;
      }
      if(ctd >= 2) break;
    }
}
```

Figura 4: Armazenamento das Coordenadas dos Reis

4.3 Verficação de Cheques

Com o intuito de tornar o código mais limpo, fizeram-se funções que verificam cada tipo de cheque que pode acontecer, dependendo de qual peça está sendo observada. Por exemplo, a função *check_black_rook_queen* verifica se a peça observada é uma torre ou rainha pretas e retorna *WHITE_CHECK* caso sim, visto que essas peças podem causar um cheque no rei branco. Se a verificação for falsa, é retornado *NO_CHECKS*, pois a peça observada não é do tipo que pode gerar cheque naquela situação. De maneira análoga, foram feitas funções que verificam cheques no rei preto também. Como a lógica desse grupo de funções é bastante simples, serão apresentadas apenas dois exemplos nesse documento.

```c
//Funcoes que verificam quais pecas para denunciar um cheque
int check_black_rook_queen(char p){
  if(p == 'r' || p == 'q')
    return WHITE_CHECK;
  else
    return NO_CHECKS;
}
```

Figura 5: Função que Verifica um Cheque de Torre ou Rainha Pretas

```c
int check_white_bishop_queen(char p){
  if(p == 'B' || p == 'Q')
    return BLACK_CHECK;
  else
    return NO_CHECKS;
}
```

Figura 6: Função que Verifica um Cheque de Bispo ou Rainha Brancos

4.4 Analise do Tabuleiro

Com todas as funções necessárias prontas, falta apenas analisar o tabuleiro para encontrar que tipo de caso descreve cada entrada. Então, recebe-se o tabuleiro em questão e as coordenadas dos reis dentro dele e imediatamente verifica-se se tais coordenadas estão na posição (-1,-1), usada caso não sejam encontrados reis. Se for verdadeiro, retorna o valor *NO_KINGS*. Visto que uma das condições de entrada é de que ambos os reis estarão presentes sempre que tiver alguma peça no tabuleiro, pode-se usar esse retorno para denunciar um tabuleiro vazio.

```c
int check_checks(char tab[8][8], coord K, coord k){
  int i = 0, j = 0;

  //Se as coordenadas estiverem na situacao de inicializacao, nao haviam reis no tabuleiro
  if(K.x == -1 || K.y == -1 || k.x == -1 || k.y == -1) return NO_KINGS;
```

Figura 7: Caso Onde Não Há Reis no Tabuleiro

Após isso, inicia-se a verificação dos cheques possíveis em cada rei, começando pelo branco. Primeiro, verifica-se se há peões pretos nas casas superior direita e superior esquerda, o que seria um cheque no rei branco. Os peões são verificados primeiro pois, dessa forma, pode-se desconsiderá-los na checagem das diagonais feitas futuramente. Vale lembrar que também é verificado se o rei não está na borda superior ou em alguma bora lateral para evitar acessos a posições inválidas na matriz.

```
  //Rei Branco (K)

  //Peões
  if(K.y > 0){ //Verifica-se os peoes primeiro pois assim a checagem das diagonais fica mais
simples
    if(K.x < 7)
      if(check_black_pawn(tab[K.y-1][K.x+1])) return WHITE_CHECK;

    if(K.x > 0)
      if(check_black_pawn(tab[K.y-1][K.x-1])) return WHITE_CHECK;
  }
```

Figura 8: Verificação dos Peões Pretos

Então, é feita a verificação das retas verticais e horizontais que passam pelo rei branco. Para isso, analisa-se separadamente as casas acima, abaixo, na esquerda e na direita das coordenadas do rei até encontrar um espaço que não esteja vazio. Se algum for encontrado, verifica-se se a peça naquele lugar é uma rainha ou uma torre preta, visto que são essas peças que podem fazer cheque nessas posições. Caso seja verdadeiro, encontra-se um cheque e pode-se retornar o valor *WHITE_KING_CHECK*, já que não é possível ter dois reis em cheque ao mesmo tempo. Se a peça analizada não for uma rainha ou uma torre preta, pode-se quebrar o laço daquela direção, pois peças além daquele ponto não poderão dar cheque no rei.

```
//Torres e Rainhas
//Sao verificadas as posicoes acima, abaixo, na esquerda e na direita das coordenadas do rei recebido

//Para Cima
for(i = K.y-1; i >= 0; i--){
  if(tab[i][K.x] != '.'){
    if(check_black_rook_queen(tab[i][K.x])) return WHITE_CHECK;
    break;
  }
}

//Para Baixo
for(i = K.y+1; i < 8; i++){
  if(tab[i][K.x] != '.'){
    if(check_black_rook_queen(tab[i][K.x])) return WHITE_CHECK;
    break;
  }
}

//Para a Esquerda
for(i = K.x-1; i >= 0; i--){
  if(tab[K.y][i] != '.'){
    if(check_black_rook_queen(tab[K.y][i])) return WHITE_CHECK;
    break;
  }
}

//Para a Direita
for(i = K.x+1; i < 8; i++){
  if(tab[K.y][i] != '.'){
    if(check_black_rook_queen(tab[K.y][i])) return WHITE_CHECK;
    break;
  }
}
```

Figura 9: Verificação das Torres e Rainhas Pretas

Após isso, verifica-se as diagonais a partir da casa do rei. Se for encontrado algum espaço ocupado, verifica-se se a peça naquele local é um bispo ou uma rainha pretos, já que são essas peças que podem fazer cheque. Retornando *WHITE_KING_CHECK* caso sim ou quebrando o laço de verificação daquela direção caso não.

```
//Bispos e Rainhas

//Sao verificadas as diagonais das coordenadas do rei recebido

//Superior Esquerda
i = K.y-1;
j = K.x-1;
while(i >= 0 && j >= 0){
  if(tab[i][j] != '.'){
    if(check_black_bishop_queen(tab[i][j])) return WHITE_CHECK;
    break;
  }
  i--;
  j--;
}

//Superior Direita
i = K.y-1;
j = K.x+1;
while(i >= 0 && j < 8){
  if(tab[i][j] != '.'){
    if(check_black_bishop_queen(tab[i][j])) return WHITE_CHECK;
    break;
  }
  i--;
  j++;
}

//Inferior Esquerda
i = K.y+1;
j = K.x-1;
while(i < 8 && j >= 0){
  if(tab[i][j] != '.'){
    if(check_black_bishop_queen(tab[i][j])) return WHITE_CHECK;
    break;
  }
  i++;
  j--;
}

//Inferior Direita
i = K.y+1;
j = K.x+1;
while(i < 8 && j < 8){
  if(tab[i][j] != '.'){
    if(check_black_bishop_queen(tab[i][j])) return WHITE_CHECK;
    break;
  }
  i++;
  j++;
}
```

Figura 10: Verificação dos Bispos e Rainhas Pretas

Finalmente, faz-se a verificação das casas ao redor do rei que podem ser ameaçadas por um cavalo. Por causa da natureza do movimento do cavalo, foi preciso analisar cada uma das oito casas individualmente, tendo cuidado, novamente, para não acessar posições inválidas na matriz.

```
  //Cavalos
  //Verifica-se cada caso do cavalo separadamente, pois nao ha logica melhor para verificar os
movimentos possiveis sem correr o risco de acessar posicoes invalidas na matriz
  //Dois para baixo e um para a direita
  if(K.x < 7 && K.y < 6)
    if(check_black_knight(tab[K.y+2][K.x+1])) return WHITE_CHECK;

  //Um para baixo e dois para a direita
  if(K.x < 6 && K.y < 7)
    if(check_black_knight(tab[K.y+1][K.x+2])) return WHITE_CHECK;

  //Um para cima e dois para a direita
  if(K.x < 6 && K.y > 0)
    if(check_black_knight(tab[K.y-1][K.x+2])) return WHITE_CHECK;

  //Dois para cima e um para a direita
  if(K.x < 7 && K.y > 1)
    if(check_black_knight(tab[K.y-2][K.x+1])) return WHITE_CHECK;

  //Dois para cima e um para a esquerda
  if(K.x > 0 && K.y > 1)
    if(check_black_knight(tab[K.y-2][K.x-1])) return WHITE_CHECK;

  //Um para cima e dois para a esquerda
  if(K.x > 1 && K.y > 0)
    if(check_black_knight(tab[K.y-1][K.x-2])) return WHITE_CHECK;

  //Um para baixo e dois para a esquerda
  if(K.x > 1 && K.y < 7)
    if(check_black_knight(tab[K.y+1][K.x-2])) return WHITE_CHECK;

  //Dois para baixo e um para a esquerda
  if(K.x > 0 && K.y < 6)
    if(check_black_knight(tab[K.y+2][K.x-1])) return WHITE_CHECK;
```

Figura 11: Verificação dos Cavalos Pretos

Com isso, são verificados todos os cheques possíveis para o rei branco. A lógica feita para o rei preto é análoga a usada anteriormente, porém, verificando se as peças são brancas e retornando o valor *BLACK_KING_CHECK* caso seja encontrado um cheque. Logo, será apresentado apenas um exemplo dessa lógica nesse documento, com o objetivo de mantê-lo mais conciso e objetivo.

```
//Rei Preto (k)

//Peões
if(k.y < 7){
  if(k.x < 7)
    if(check_white_pawn(tab[k.y+1][k.x+1])) return BLACK_CHECK;

  if(k.x > 0)
    if(check_white_pawn(tab[k.y+1][k.x-1])) return BLACK_CHECK;
}

//Torres e Rainhas

//Para Cima
for(i = k.y-1; i >= 0; i--){
  if(tab[i][k.x] != '.'){
    if(check_white_rook_queen(tab[i][k.x])) return BLACK_CHECK;
    break;
  }
}

//Para Baixo
for(i = k.y+1; i < 8; i++){
  if(tab[i][k.x] != '.'){
    if(check_white_rook_queen(tab[i][k.x])) return BLACK_CHECK;
    break;
  }
}
```

Figura 12: Verificações para o Rei Preto

Concluindo, após se fazer a verificação de todas as peças que podem ameaçar o rei preto, caso ainda não tenha sido retornado nenhum valor de cheque, a função retorna o valor *NO_CHECKS*.

```
  //Dois para baixo e um para a esquerda
  if(k.x > 0 && k.y < 6)
    if(check_white_knight(tab[k.y+2][k.x-1])) return BLACK_CHECK;

  return NO_CHECKS;
}
```

Figura 13: Retorno da Função Caso não haja Cheques

4.5 Função Principal

Dessa forma, falta apenas organizar as funções criadas em uma função principal que obedece os requisitos de entrada e saída. Para tal, primeiramente, foi criado um vetor para armazenar os resultados, já que podem ser entrados multiplos jogos. Adotou-se um máximo de 1000 jogos para esse código, pois acredita-se que é um valor suficiente.

```
int main(void) {
  char tab[8][8], buff;
  int *results = NULL, number_of_games = 0;
  coord K, k;

  results = (int*) malloc(MAX_GAMES*sizeof(int)); //Usou-se um numero maximo de 1000 casos para fazer a
alocacao de espaco
```

Figura 14: Vetor de Resultados

Então, cria-se um laço para receber as entradas do usuário, processar os dados e guardar os resultados. A condição do laço é deixada sempre em verdadeiro pois a condição de parada usada depende do resultado do último jogo lido, logo, considerou-se que essa é a melhor solução. Após isso, faz-se a incialização do tabuleiro, o recebimento da entrada do usuário e o armazenamento das coordenadas do rei para aquela entrada.

```
  while(1){ //sempre entra-se no laco, pois nao ha maneira de verificar a condicao de parada antes de
receber os tabuleiros
    inicializa_tabuleiro(tab);

    preenche_tabuleiro(tab); //Recebe o tabuleiro do usuario

    get_kings(tab, &K, &k); //Guarda as posicoes dos reis
```

Figura 15: Preparação do Tabuleiro e Armazenamento das Coordenadas dos Reis

Depois disso, é feita a analise do tabuleiro e o resultado encontrado é guardado em um vetor. Se esse resultado for o tabuleiro vazio, ou seja, *NO_KINGS*, encontra-se a condição de parada e quebra-se o laço. Se for outro resultado, sabe-se que a próxima entrada será dada após uma linha em branco. Então, foi usada uma variável de *char* (”buff”) para guardar essa linha e preparar a função para receber o próximo tabuleiro.

```
    results[number_of_games] = check_checks(tab, K, k); //Guarda o resultado de cada analise de jogo

    if(results[number_of_games++] == NO_KINGS) break; //A condicao de parada do laco ocorre quando eh
recebido um tabuleiro vazio, ou seja, sem reis

    fgets(&buff, 1, stdin); //Elimina-se a linha vazia entre cada entrada do usuário a armazenado numa
variável
  }
```

Figura 16: Armazenamento do Resultado e Checagem da Condição de Parada

Por fim, tem-se o vetor com todos os resultados dos jogos enviados, então, faz-se um laço para percorrê-lo e imprimir na tela as mensagens requisitadas, de acordo com cada resultado. Depois de finalizar o laço, libera-se a memória alocada e o programa termina.

```
  for(int i = 0; i < number_of_games; i++){ //Faz se o print final dependendo do resultado de cada jogo
    if(results[i] == NO_CHECKS) printf("Jogo #%d: nenhum rei esta em cheque.\n", i+1);
    else if(results[i] == WHITE_CHECK) printf("Jogo #%d: rei branco esta em cheque.\n", i+1);
    else if(results[i] == BLACK_CHECK) printf("Jogo #%d: rei preto esta em cheque.\n", i+1);
  }

  free(results); //Libera o espaco alocado

  return 0;
}
```

Figura 17: Mensagens Finais e Liberação da Memória