COMO SERIA A IMPLEMENTAÇÃO

Vamos a um exemplo(que foi dado no artigo):

R = (10,10) ----> R[10][10] = {0} ---> preenchida com zeros:

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Queremos cortar R seguindo as dimensões r1 = (7,2). Como R está vazia inicialmente, o único ponto candidato será o [0][0](ponto de "origem" de R). Assim, cortamos a peça a partir desta coordenada em R. O corte com relação a r1 será representado com "1" na matriz. Logo após o corte, a matriz R ficará desta forma:

```
1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Os pontos candidatos de r1 são aqueles mais a esquerda e abaixo e mais a direta e acima e estes serão adicionados como possíveis pontos de inserção para peças futuras. Desta forma, o destaque dos pontos candidatos em R ficaria assim:

```
1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Pontos candidatos: [0][0](origem),[0][6](+ direita e acima de r1), [1][0](+ esquerda e abaixo de r1)

Agora, vamos cortar a peça r2 = (2,4). Precisamos achar o primeiro ponto candidato que seja o ponto de "origem" desta peça onde a mesma possa ser inserida sem sobreposição. Para isso, verificamos, para cada ponto candidato(indo da esquerda para direita e de cima para baixo, o que conclui pelo exemplo dado no artigo) se as dimensões da peça se ajustam ao espaço necessário a partir deste ponto candidato. Assim, vemos que a peça r2(representado por “2” em R) será cortada a partir do ponto candidato [0][6].

Adicionando os pontos candidato de r2 em R, teremos a seguinte configuração:

```
1 1 1 1 1 1 1 2 2 0
1 1 1 1 1 1 1 2 2 0
0 0 0 0 0 0 0 2 2 0
0 0 0 0 0 0 0 2 2 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Pontos candidatos: [0][0](origem),[0][6],[1][0],[0][8] e [3][7].

Veja:

O ponto [0][6] possui o valor “1” de r1. A análise é baseada na verificação de podermos inserir a partir da coordenada a ***direita****([0][7]) ou* ***abaixo****([1][6]). Sempre esses valores apresentarão um valor “0” ou o número correspondente a peça que contém o ponto candidato ou ambas apresentarão valores diferentes de zero(ponto candidato que não será considerado para inserção). Essa ideia é aplicada em todos os casos. Mas lembre-se: toda vez que tivemos uma retorno posição com esta análise, é imprescindível verificar se, a partir daquele ponto possível, as dimensões da peça não se sobreporão ou caberão naquele espaço.*

<u>O porquê desta ideia?</u>

Suponha que em algum momento você avalie o ponto candidato [3][7] para inserção de uma peça qualquer. Se apenas considerarmos o próximo ponto a direita, diremos que será inviável a inserção, sendo que não será: Se observar o ponto logo abaixo de [3][7], ele tem o valor zero e um espaço totalmente vago para corte. Poderíamos muito bem aproveitar este espaço.

Cortando a peça r3 = (2,2):

```
         ---------------> 1º)Direita(ordem de verificação PC)

        |   1 1 1 1 1 1 1 2 2 0->Faltaria espaço para a peça r3!
        |   1 1 1 1 1 1 1 2 2 0
        |   3 3 0 0 0 0 0 2 2 0
        |   3 3 0 0 0 0 0 2 2 0
        |   0 0 0 0 0 0 0 0 0 0
        |   0 0 0 0 0 0 0 0 0 0
        |   0 0 0 0 0 0 0 0 0 0
        |   0 0 0 0 0 0 0 0 0 0
        |   0 0 0 0 0 0 0 0 0 0
        v   0 0 0 0 0 0 0 0 0 0
     2º)Baixo
(ordem de verificação PC)
```

Pontos candidatos: [0][0](origem),[0][6],[1][0],[0][8], [3][7], [2][1] e [3][0].

Cortando a peça r4 = (2,2):

```
            ---------------> 1º)Direita(ordem de verificação PC)

           |     1  1  1  1  1  1  1  2  2  0->Faltaria espaço para a peça r4!
           |     1  1  1  1  1  1  1  2  2  0
           |     3  3  4  4  0  0  0  2  2  0
           |     3  3  4  4  0  0  0  2  2  0
           |     0  0  0  0  0  0  0  0  0  0
           |     0  0  0  0  0  0  0  0  0  0
           |     0  0  0  0  0  0  0  0  0  0
           |     0  0  0  0  0  0  0  0  0  0
           |     0  0  0  0  0  0  0  0  0  0
           v     0  0  0  0  0  0  0  0  0  0
       2º)Baixo
(ordem de verificação PC)
```

Pontos candidatos: [0][0](origem),[0][6],[1][0],[0][8], [3][7], [2][1], [3][0], [3][2] e [2][3].

Estado final do exemplo:

```
1  1  1  1  1  1  1  2  2  0
1  1  1  1  1  1  1  2  2  0
3  3  4  4  0  0  0  2  2  0
3  3  4  4  0  0  0  2  2  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
```

Vermelho = r1
Azul = r2
Amarelo = r3
Verde = r4

Algumas observações:

1. Podemos representar uma peça como uma *struct* desta forma:

```
typedef struct PECA_R
{
     int ponto_candidato_direita_x;
     int ponto_candidato_direita_y;
     int ponto_candidato_esquerda_x;
     int ponto_candidato_esquerda_y;
     int ponto_origem_em_R; //Origem onde foi inserido em R
}PECA_R
```

2. E também para o tipo da peça:

```
typedef struct TIPO_PECA
{
     int quantidade_disponivel;
     int largura;
     int comprimento;

}TIPO_PECA
```

E assim, finalizamos o exemplo. Foi uma das formas que encontrei para tratar essa construção e o problema em si.

Agora, falar sobre o deslocamento das peças na fase de desconstrução:

As peças restantes em R serão deslocadas inicialmente a esquerda e depois para cima(foco no ponto de origem de R--->[0][0])

Considere o seguinte caso:

0	0	0	0	0	0	0	4	4	4
0	0	0	0	0	0	0	4	4	4
0	1	1	1	0	0	0	4	4	4
0	1	1	1	0	0	0	4	4	4
0	1	1	1	0	0	0	4	4	4
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	2	2	0	0
0	5	5	5	0	0	2	2	0	0
0	5	5	5	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Precisamos deslocar, primeiramente, todos para a esquerda.
Para isso, faremos:

1. Ordenar todas as peças presentes em R(considerando repetidas) em ordem crescente de acordo com a distância euclidianda(fórmula abaixo) com relação ao ponto de origem da peça e o ponto de origem de R([0][0]);

$$\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

2. Deslocar, seguindo a ordem da lista, cada peça para a esquerda.

Resultado:

0	0	0	4	4	4	0	0	0	0
0	0	0	4	4	4	0	0	0	0
1	1	1	4	4	4	0	0	0	0
1	1	1	4	4	4	0	0	0	0
1	1	1	4	4	4	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	2	2	0	0	0	0	0
5	5	5	2	2	0	0	0	0	0
5	5	5	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Agora, deslocar para cima. Segue a mesma ordem de deslocamento utilizado anteriormente:

1	1	1	4	4	4	0	0	0	0
1	1	1	4	4	4	0	0	0	0
1	1	1	4	4	4	0	0	0	0
5	5	5	4	4	4	0	0	0	0
5	5	5	4	4	4	0	0	0	0
0	0	0	2	2	0	0	0	0	0
0	0	0	2	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Finaliza-se.

Observação: Sim, você se perguntará como algoritcamente o deslocamento das peças se dá por R. Bem, não pensei em uma forma otimizada ainda, mas a princípio você pegaria o ponto candidato a esquerda e o ponto de origem da peça e, pra cada índice e ao mesmo tempo, percorreria a esquerda a linha até chegar em um valor diferente de zero. Se no fim da iteração os índices estiverem em uma mesma coluna, você desloca a peça até lá.(desculpa, tentei resumir aqui, mas estou com sono demais pra pensar mais a fundo rsrs)