Segment Tree e *Fenwick Tree*

Parte A

1. **Complete a sentença:** Um árvore de segmentos (*Segment Tree*) armazena, em cada um de seus _____, informações relativa um _____ de índices de um *array* de tamanho N.
2. Uma *Segment Tree* é uma árvore binária? Se sim, é completa? Se não, que tipo de árvore seria?
3. Desenhe uma *Segment Tree*, indicando os intervalos associados a cada nó, para um *array* de tamanho $N = 12$.
4. Defina:
   (a) *range minimum query*
   (b) *range maximum query*
   (c) *range sum query*
5. Quais são as duas operações fundamentais de uma *Fenwick Tree*?
6. Uma *Fenwick Tree* é uma árvore balanceada? Justifique sua resposta.
7. Seja k um inteiro positivo e defina $p(k)$ como a maior potência de 2 que divide k, isto é, $p(k) = j$ se 2^j divide k e 2^{j+1} não divide k.
   (a) Compute os valores de $p(1), p(4), p(6), p(12)$ e $p(15)$.
   (b) Qual é a relação entre $p(k)$ e a representação de k na base binária?
   (c) Escreva $p(k)$ em função de k, utilizando operações lógicas e aritméticas.
8. Desenhe uma *Fenwick Tree*, indicando os intervalos associados a cada nó, para um *array* de tamanho $N = 12$. ***Dica*:** A raiz é o nó 1, e cada nó está associado ao intervalo $[k - p(k) + 1, k]$.
9. Qual é o tamanho do intervalo associado ao nó k de uma *Fenwick Tree*?

Parte B

10. Qual é a complexidade, por *query*, em uma implementação *naive* de uma *range minimum query (RMQ)*?
11. Dê uma cota superior para o número de nós em uma *Segment Tree* para um *array* de tamanho N.
12. Caso o vetor seja estático, sem operações de atualização, o problema da RMQ pode ser resolvido de forma mais eficiente, usando-se programação dinâmica. Escreva o pseudocódigo desta solução e determine sua complexidade assintótica.
13. Em quais situações *range sum queries* podem respondidas de forma mais eficiente com uma *Fenwick Tree* do que com um vetor de prefixo pré-processado?
14. O pseudocódigo abaixo implementa uma operações básicas de uma *Segment Tree*. Implemente estas operações em C/C++

Algoritmo 1 *Segment Tree*

Entrada: Um vetor *array* v com N elementos

```
1: procedure INIT(v)
2:     for i ← 1, 4(v.N + 1) do
3:         node[i] ← 0                          ▷ O índice zero é um sentinela
4:     end for
5:     BUILD(v, 1, 0, v.N − 1)                  ▷ i = 1 é o índice da raiz da árvore
6: end procedure
7:
```

```
8: function RMQ(v, a, b)                          ▷ Retorna o índice do elemento de menor valor em [a, b]
9:     return RMQ(v, 1, 0, v.N − 1, a, b)
10: end function
11:
12: function UPDATE(v, a, b, x)                   ▷ Faz v[j] = x para todos j ∈ [a, b] e atualiza os demais intervalos
13:     for j ← 0, v.N − 1 do
14:         v[j] ← x
15:     end for
16:     return UPDATE(v, 1, 0, v.N − 1, a, b)
17: end function
18:
19: // Funções auxiliares
20:
21: function LEFT(i)                                ▷ Retorna o índice do filho à esquerda do nó
22:     return 2i
23: end function
24:
25: function RIGHT(i)                               ▷ Retorna o índice do filho à direita do nó
26:     return 2i + 1
27: end function
28:
29: function BUILD(v, i, a, b)                      ▷ i é o índice o nó ([a, b]) a ser preenchido
30:     if a = b then
31:         node[i] ← a
32:         return node[i]
33:     end if
34:     x ← BUILD(LEFT(i), a, (a + b)/2)
35:     y ← BUILD(RIGHT(i), (a + b)/2 + 1, b)
36:     if v[x] ≤ v[y] then
37:         node[i] ← x
38:     else
39:         node[i] ← y
40:     end if
41:     return node[i]
42: end function
43:
44: function RMQ(v, i, L, R, a, b)                  ▷ [L, R] representa o nó onde a busca está sendo feita
45:     if a > R  or b < L then                     ▷ O intervalo [a, b] não se intersecta com o nó
46:         return ∞                                ▷ ∞ é um valor sentinela que represente o infinito positivo
47:     end if
48:
49:     if a ≤ L  and R ≤ b then                    ▷ [L, R] ∈ [a, b]
50:         return node[i]
51:     end if
52:
53:     x ← RMQ(v, LEFT(i), L, (L + R)/2, a, b)
54:     y ← RMQ(v, RIGHT(i), (L + R)/2 + 1, R, a, b)
55:
56:     if v[x] ≤ v[y] then
57:         return x
58:     else
59:         return y
60:     end if
61: end function
62:
63: function UPDATE(v, i, L, R, a, b)                ▷ [L, R] representa o nó onde a busca está sendo feita
64:     if a > R  or b < L then                     ▷ O intervalo [a, b] não se intersecta com o nó
65:         return ∞                                ▷ ∞ é um valor sentinela que represente o infinito positivo
66:     end if
67:
```

```
68:     if L = R then
69:         return node[i]
70:     end if
71:
72:     x ← UPDATE(v, LEFT(i), L, (L + R)/2, a, b)
73:     y ← UPDATE(v, RIGHT(i), (L + R)/2 + 1, R, a, b)
74:
75:     if v[x] ≤ v[y] then
76:         node[i] ← x
77:     else
78:         node[i] ← y
79:     end if
80:
81:     return node[i]
82: end function
```

15. Escreva, em pseudocódigo, as operações de *range sum query* (RSQ) e de atualização de um ponto em uma *Fenwick Tree*.

16. Compare as árvores de segmentos e as árvores de Fenwick, quando utilizadas no problema das RSQ, e em outros problemas, apontando as vantagens e desvantagens de cada uma.

17. Use uma árvore de Fenwick para implementar, de forma eficiente, RSQ com *point queries* (determinar o valor de $A[k]$) e *range updates* (somar uma constante c a todos os valores de $A[i]$, com $i \in [a, b]$).

18. Implemente uma árvore de segmentos:

(a) com *lazy propagation*

(b) com *polinomial updates*

(c) com nós dinamicamente alocados

(d) persistente

(e) com nós que armazenam estruturas de dados

(f) bidimensional

19. Use uma árvore de Fenwick para implementar, de forma eficiente, RSQ com *range queries* e *range updates*.