Tarea 2 - Programación y algoritmos 2

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En la presente tarea se analizará la solución implementada al problema: **11402 - Ahoy, Pirates!** de UVA, así como la complejidad de la misma.

1. Descripción del problema

En este problema se da la descripción de un conjunto de piratas los cuales pertenecen a un bando: Piratas bucaneros y piratas bárbaros. Una vez dada dicha descripción se da un conjunto de consultas las cuales pueden ser de distintos tipos: Cambiar todos los piratas a un bando dentro de un rango, invertir el bando de todos los piratas en un rango, preguntar cuántos piratas bucaneros hay en cierto rango.

Las restricciones del problema son:

1. Tiempo límite: 5s
2. $1 \leq n \leq 1024000$ (número de piratas)
3. $1 \leq q \leq 1000$ (número de consultas)
4. $1 \leq i \leq j \leq n$ (rango de los intervalos de consulta)

2. Solución implementada

Para este problema se usó el árbol de segmentos en conjunto con la técnica de **lazy propagation** la cual se encarga de hacer las actualizaciones sólo cuando son necesarias y además permite hacer actualizaciones en rango, diferente a lo que permitía hacer la versión más simple del árbol. La idea para este problema es en cada nodo llevar el conteo de cuántos piratas bucaneros hay en el rango correspondiente, esto permitirá responder cada pregunta en a lo más $O(logn)$, y para cada actualización usar una versión *lazy* del árbol la cual permitirá llevar el control de las mismas. La construcción del árbol se hace de manera habitual, a continuación se muestra el psudocódigo de la misma.

Algorithm 1 Construcción del árbol.

```
1: procedure BuildTree(Tree, Arr, node, lo, hi)
2:     if lo == hi then
3:         Tree[node] ← Arr[lo]
4:     else
5:         mid ← (lo + hi) / 2
6:         lc ← 2 * node
7:         rc ← 2 * node + 1
8:         BuildTree(Tree, Arr, lc, lo, mid)
9:         BuildTree(Tree, Arr, rc, mid + 1, hi)
10:        Tree[node] ← Tree[lc] + Tree[rc]
```

Una vez que se tiene construido el árbol lo siguiente es hacer los métodos para consultar y actualizar el mismo y ahí es donde entra el lazy tree. En cada uno de esos métodos primero hay que fijarse si existen actualizaciones pendientes de ser así se realizan en el nodo y se propagan a los hijos (si es que existen) posteriormente los métodos proceden igual.
Primero se realiza el método de actualización, el cuál es el más critico de todo el programa ya que si un nodo tiene una actualización pendiente y recibe una nueva hay que ver cómo combinar dichas actualizaciones para generar la nueva, no es difícil ver que todos los tipos de actualizaciones se pueden combinar para generar nuevas actualizaciones que se encuentran en el mismo conjunto, a continuación se muestra las posibles combinaciones:

1. Si la nueva actualización es del tipo F (todos los piratas se convierten en bucaneros). Esta actualización sobreescribre a todas las anteriores ya que sin importar qué bando tengan los piratas en el rango al final todos se convertirán en bucaneros.
2. Si la nueva actualización es del tipo E (todos los piratas se convierten en bárbaros). Igual que en el caso anterior esta actualización sobreescribe a las anteriores por la misma lógica.
3. Si la nueva actualización es del tipo I (todos los piratas invierten su bando). Esta actualización se procesa por casos. Si hay pendiente una tipo F al final esta la convertirá en tipo E. Si hay pendiente una tipo E al final esta la convertirá en tipo F. Si hay pendiente una tipo I las dos se anularán y en caso de no haber actualizaciones pendientes se procesa normal.
4. Si es una consulta de tipo S sólamente se regresa la suma del valor de los nodos que contengan el rango de interés.

A continuación se mostrará el método de actualización del árbol.

Algorithm 2 Actualización del árbol.

```
1: procedure UPDATE(Tree, Lazy, Arr, node, lo, hi, i, j, updateType)
2:     mid ← (lo + hi) / 2
3:     lc ← 2 * node
4:     rc ← 2 * node + 1
5:     if Lazy[node] ≠ 0 then
6:         if Lazy[node] = 1 then
7:             Tree[node] ← hi - lo
8:         else if Lazy[node] = 2 then
9:             Tree[node] ← 0
10:        else
11:            Tree[node] ← hi - lo - Tree[node]
12:        if lo ≠ hi then
13:            CompoundUpdate(lc, Lazy, Lazy[node])
14:            CompoundUpdate(rc, Lazy, Lazy[node])
15:        Lazy[node] ← 0
16:    if i > hi or j < lo then
17:        Return
18:    if i ≤ lo and j ≥ hi then
19:        if updateType = 1 then
20:            Tree[node] ← hi - lo
21:        else if updateType = 2 then
22:            Tree[node] ← 0
23:        else
24:            Tree[node] ← hi - lo - Tree[node]
25:        if lo ≠ hi then
26:            CompoundUpdate(lc, Lazy, updateType)
27:            CompoundUpdate(rc, Lazy, updateType)
28:    else
29:        Update(Tree, Lazy, Arr, lc, lo, mid, i, j, updateType)
30:        Update(Tree, Lazy, Arr, rc, mid + 1, hi, i, j, updateType)
31:        Tree[node] ← Tree[lc] + Tree[rc]
```

Algorithm 3 Actualización compuesta.

```
1: procedure CompoundUpdate(node, Lazy, updateType)
2:     if updateType = 1 or updateType = 2 then
3:         Lazy[node] ← updateType
4:     else if updateType = 3 then
5:         if Lazy[node] = 0 then
6:             Lazy[node] ← updateType
7:         else if Lazy[node] = 1 then
8:             Lazy[node] ← 2
9:         else if Lazy[node] = 2 then
10:            Lazy[node] ← 1
11:        else
12:            Lazy[node] ← 0
```

En el método de actualización anterior primero se verifica si hay actualizaciones pendientes, de ser así se ejecutan en el árbol y posteriormente se propaga la actualización a los hijos en caso de existir, esto usando el método *compoundUpdate* el cual combina las actualizaciones nuevas con las pendientes usando los casos mencionados anteriormente. Posteriormente la actualización se hace parecido al método de consulta, se verifica si los rangos no coinciden entonces no se hace nada, luego en caso de que el rango del nodo esté dentro del solicitado se actualiza el mismo, y en caso de coincidir parcialmente se actualiza a los hijos y se combina.

El método de consulta se hace como correspondería en la versión sencilla del árbol sólamente que se añade la verificación de actualizaciones pendientes tal como en el método anterior.

Finalmente se puede ver que la versión lazy del árbol no añade más complejidad en las consultas ni en las actualizaciones del árbol ya que las actualizaciones sólo se hacen cuando son necesarias, en otro caso se dejan indicadas en el lazy tree, por lo que se tiene que por query la complejidad es de $O(logn)$. La complejidad final del algoritmo para este problema es de $O(n)$ para el preprocesamiento del árbol y de $O(mlogn)$ para responder las m querys.

3. Resultados

La implementación del algoritmo se envió al sistema UVA para su evaluación y obtuvo el resultado de Aceptado con un tiempo de: 0.98 s

#	Problem	Verdict	Language	Run Time	Submission Date
21929142	11402 Ahoy, Pirates!	Accepted	C++	0.980	2018-09-08 23:52:12

(a) Figura 1. Resultado del envío.