(algunas) Estructuras de datos útiles

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Campamento de Programación: Cochabamba 2016

“For the fashion of Minas Tirith was such that it was built on seven levels, each delved into a hill, and about each was set a wall, and in each wall was a gate.”

J.R.R. Tolkien, “The Return of the King”

Contenidos

Contenidos

Visión del usuario de una tabla aditiva

Tabla aditiva

Dado un arreglo de r dimensiones, digamos de $n_1 \times n_2 \times \cdots \times n_r$, una tabla aditiva es una estructura que permite averiguar rápidamente la suma de los elementos de cualquier subarreglo contiguo (intervalo, rectángulo, paralelepípedo, etc). En este caso **NO** nos interesará realizar modificaciones a la matriz durante el proceso de consultas.

La estructura consta de dos fases de uso:

- Primero la estructura es inicializada con los datos del arreglo.
- Y luego se realizan una serie de consultas a la misma, sin modificar el arreglo.

Complejidad pretendida

La implementación que propondremos tendrá una complejidad:

- *Lineal* para la inicialización o preproceso (es decir, proporcional a la **cantidad de elementos** del arreglo).
- *Constante* para las queries (Es decir, responderemos cada consulta en $\mathbf{O}(\mathbf{1})$).

Contenidos

Arreglo acumulado

Definición

Dado un arreglo unidimensional v de n elementos $v_0, v_1, \cdots, v_{n-1}$, definimos el **arreglo acumulado** de v como el arreglo unidimensional V de $n+1$ elementos $V_0, \cdots, V_n$ y tal que:

$$V_i = \sum_{j=0}^{i-1} v_j$$

Notar que V es muy fácil de calcular en tiempo lineal utilizando programación dinámica:

- $V_0 = 0$
- $V_{i+1} = v_i + V_i, \forall\ 0 \leq i < n$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por intervalos $[i, j)$ con $0 \leq i \leq j \leq n$.
- La respuesta al query $[i, j)$ sera $Q(i, j) = \sum_{k=i}^{j-1} v_k$
- Pero:

$$Q(i, j) = \sum_{k=i}^{j-1} v_k = \sum_{k=0}^{j-1} v_k - \sum_{k=0}^{i-1} v_k = V_j - V_i$$

- Luego podemos responder cada query en tiempo constante computando una sola resta entre dos valores del arreglo acumulado.

Contenidos

Arreglo acumulado

Definición

Dado un arreglo bidimensional v de $n \times m$ elementos $v_{i,j}$ con $0 \leq i < n, 0 \leq j < m$, definimos el **arreglo acumulado** de v como el arreglo bidimensional V de $(n+1) \times (m+1)$ elementos tal que:

$$V_{i,j} = \sum_{a=0}^{i-1} \sum_{b=0}^{j-1} v_{a,b}$$

¿Podremos calcular fácilmente V en tiempo lineal como en el caso unidimensional? **¡Sí!** Utilizando programación dinámica.

- $V_{0,j} = V_{i,0} = 0 \ \ \forall\, 0 \leq i \leq n, 0 \leq j \leq m$
- $V_{i+1,j+1} = v_{i,j} + V_{i,j+1} + V_{i+1,j} - V_{i,j} \ \ \forall\, 0 \leq i < n, 0 \leq j < m$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por rectángulos $[i_1, i_2) \times [j_1, j_2)$ con $0 \leq i_1 \leq i_2 \leq n, 0 \leq j_1 \leq j_2 \leq m$.
- La respuesta al query $[i_1, i_2) \times [j_1, j_2)$ sera
$$Q(i_1, i_2, j_1, j_2) = \sum_{a=i_1}^{i_2-1} \sum_{b=j_1}^{j_2-1} v_{a,b}$$
- Pero de manera similar a como hicimos para calcular el arreglo acumulado, resulta que:
$$Q(i_1, i_2, j_1, j_2) = V_{i_2,j_2} - V_{i_1,j_2} - V_{i_2,j_1} + V_{i_1,j_1}$$
- Luego podemos responder cada query en tiempo constante computando sumas y restas de cuatro valores del arreglo acumulado.

Contenidos

Arreglo acumulado

El caso general es completamente análogo... Dejamos las cuentas escritas rápidamente.

Definición

Dado un arreglo v de $n_1 \times \cdots \times n_r$ elementos $v_{i_1,\ldots,i_r}$ con $0 \leq i_k < n_k, 1 \leq k \leq r$, definimos el **arreglo acumulado** de v como el arreglo V de $(n_1 + 1) \times \cdots \times (n_r + 1)$ elementos tal que:

$$V_{i_1,\cdots,i_r} = \sum_{k_1=0}^{i_1-1} \cdots \sum_{k_r=0}^{i_r-1} v_{k_1,\cdots,k_r}$$

Arreglo acumulado (Cálculo usando programación dinámica)

El cálculo de V mediante programación dinámica viene dado por:

- $$V_{i_1,\cdots,i_r} = 0 \quad \text{si para algún } k \text{ resulta } i_k = 0$$
- $$V_{i_1+1,\cdots,i_r+1} = v_{i_1,\cdots,i_r} + \sum_{d_1=0}^{1} \cdots \sum_{d_r=0}^{1} (-1)^{r+1+\sum_{k=1}^{r} d_k} V_{i_1+d_1,\cdots,i_r+d_r}$$

  Donde en las sumatorias anidadas se excluye el caso en que $d_k = 1 \ \forall \ k$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por subarreglos $[i_{10}, i_{11}) \times \cdots \times [i_{r0}, i_{r1})$
- La respuesta al query $[i_{10}, i_{11}) \times \cdots \times [i_{r0}, i_{r1})$ será:
- $$Q(i_{10}, i_{11}, \cdots, i_{r0}, i_{r1}) = \sum_{d_1=0}^{1} \cdots \sum_{d_r=0}^{1} (-1)^{r+\sum_{k=1}^{r} d_k} V_{i_{1\,d_1}, \cdots, i_{r\,d_r}}$$
- Luego podemos responder cada query en tiempo constante computando sumas y restas de 2^r valores del arreglo acumulado.

Contenidos

Tarea

- http://www.spoj.pl/problems/KPMATRIX/
- http://www.spoj.pl/problems/TEM/
- http://www.spoj.pl/problems/MATRIX/

Contenidos

Problema a resolver

- Se quiere saber la suma (o máximo, o cualquier operación asociativa) en un **prefijo** del arreglo.
- Además, para esta estructura es necesario que las modificaciones se realicen sumándole (o tomando máximo) una constante (negativa o positiva), a un valor en una posición dada (Operación que notaremos $add(i, x)$). Es decir, las operaciones de modificación son **acumulativas**.
- Notar que si la operación tiene inverso como la suma (pero **no si usamos la estructura para máximo / mínimo**), podemos computar cualquier subrango: si notamos $accum(i)$ a la suma de los primeros i, la suma del intervalo $[i, j)$ se calcula simplemente como $accum(j) - accum(i)$ (análogo a tablas aditivas).

Complejidad

- En esta charla nos limitaremos al caso unidimensional. El caso multidimensional es análogo, utilizando “tablas aditivas sobre tablas aditivas”.
- La implementación que propondremos tendrá una complejidad logarítmica para sus operaciones.

Contenidos

Estructura : Descripción

- Utilizaremos un arreglo que en cada posición guardará la suma de ciertos intervalos.
- Más precisamente, utilizaremos un arreglo V de n elementos indexados desde 1. Notar que los elementos de v, el vector original son $v_0, \cdots, v_{n-1}$; mientras que los de V son $V_1, \cdots, V_n$.
- El arreglo V estará definido por: $V_i = \sum_{j=i-k}^{i-1} v_j$, siendo k la potencia de 2 más grande que divide a i.
- La estructura comienza con todos los valores del arreglo en 0, y cada vez que se hace un *add* a una posición del vector original, se suma la cantidad correspondiente a todos los intervalos que lo contengan.
- Para el cálculo de $accum(i)$, se suman todos los intervalos necesarios.

Accum

Código fuente C++ para el *accum*

```
1  int accum(int i)
2  {
3      int res = 0;
4      while (i != 0)
5      {
6          res += v[i];
7          i = (i-1)&i;
8      }
9      return res;
10 }
```

- La idea es simplemente ir acumulando en *res* todas las sumas relevantes.
- El invariante de ciclo es: “La respuesta al problema viene dada por $res + \sum_{j=0}^{i-1} v_j$”

Add

Código fuente C++ para el *add*

```
1  void add(int i, int x)
2  {
3      int ceros = ~i;
4      for (int bit = ceros & (-ceros); ; ceros -= bit, bit = ceros & (-ceros))
5      {
6          int index = (i &~(bit-1)) | bit;
7          if (index > N) break;
8          v[index] += x;
9      }
10 }
```

- La idea es simplemente sumar x a todos los intervalos guardados que contengan a v_i

Contenidos

Tarea

- http://www.spoj.pl/problems/MATSUM/
- http://www.topcoder.com/stat?c=problem_statement&pm=6551&rd=9990

Contenidos

Visión del usuario de LCA

LCA

Dado un árbol con raíz de n nodos, una estructura de LCA permite responder rápidamente consultas por el **ancestro común más bajo** entre dos nodos (es decir, el nodo más alejado de la raíz que es ancestro de ambos).

- Sorprendentemente se puede reducir una estructura de LCA a una de RMQ!
- Luego no daremos explícitamente una estructura para LCA, sino que mostraremos como reducirla a RMQ para poder aplicar cualquiera de las técnicas ya vistas.

Contenidos

Recorrido de DFS

- Utilizando un recorrido de DFS, podemos computar un arreglo v de $2n - 1$ posiciones que indica el orden en que fueron visitados los nodos por el DFS (cada nodo puede aparecer múltiples veces).
- Aprovechando este recorrido podemos computar también la **profundidad** (distancia a la raíz) de cada nodo i, que notaremos P_i.
- También guardaremos el índice de la primer aparición de cada nodo i en v, que notaremos L_i (Cualquier posición servirá, así que es razonable tomar la primera).

Utilizacion del RMQ

- La observación clave consiste en notar que para dos nodos i, j con $i \neq j$:

$$LCA(i, j) = RMQ(\min(L_i, L_j), \max(L_i, L_j))$$

- Tomamos mínimo y máximo simplemente para asegurar que en la llamada a *RMQ* se especifica un rango válido ($i < j$).
- Notar que en la igualdad anterior, $RMQ(i, j)$ compara los elementos de v por sus valores de profundidad dados por P.
- La complejidad de la transformación del LCA al RMQ es $O(n)$, con lo cual la complejidad final de las operaciones será la del *RMQ* utilizado.

Contenidos

Distancias en árboles

- Queremos utilizar consultas de LCA para obtener una estructura capaz de computar rápidamente distancias entre nodos en un árbol.
- Notar que computar todas las distancias en un árbol utilizando un algoritmo como DFS o BFS n veces toma tiempo $O(n^2)$, que es lineal en la cantidad de distancias existentes.
- El algoritmo que propondremos por lo tanto solo constituirá una ventaja importante cuando se quieran consultar muchas menos que las n^2 distancias (pero suficiente cantidad como para que resolver cada una en forma independiente no sea práctico)

Distancias en árboles (Algoritmo)

- Tomamos un elemento cualquiera como raíz.
- Utilizamos DFS o BFS para recorrer el árbol computando las distancias desde la raíz hasta cada uno de los vértices.
- A partir de ahora, para resolver la distancia entre dos nodos cualesquiera i y j, utilizamos la identidad:

$$D(i,j) = D(r,i) + D(r,j) - 2D(r, LCA(i,j))$$

- El algoritmo propuesto responde una distancia con una complejidad de $O(1)$ más una consulta de LCA. La inicialización más allá del LCA es un único DFS o BFS, por lo que es $O(n)$.

Contenidos

Tarea

- http://acm.uva.es/p/v109/10938.html
- http://www.spoj.pl/problems/QTREE2/
- http://poj.org/problem?id=2763
- http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=2045
- http://poj.org/problem?id=1986

Contenidos

Visión del usuario

Ascenso rápido en árbol

Dado un árbol con raíz de n nodos, queremos responder rápidamente la consulta de dado un nodo v y un entero no negativo k, cuál es el k-ésimo ancestro en el camino desde v hasta la raíz.

- Se podría pedir también una query de operación acumulada (por ejemplo máximo) entre todos los nodos del camino al ancestro.
- Sumado a un LCA y profundidades, sirve para encontrar el k-ésimo nodo en un camino en el árbol

Contenidos

Sparse Table en árbol

También conocida como “saltitos potencia de dos en el árbol”

- Si el árbol es un “palito”, queda algo análogo a Sparse Table
- Por cada nodo, se guardan quiénes son sus ancestros a distancias potencia de 2 (eventualmente se podrían guardar también valores acumulados)
- Mediante $O(\lg k)$ saltos, es posible llegar al k-ésimo ancestro
- Guardando la pila de ancestros durante un recorrido de DFS, es posible llenar la lista de “saltos” de cada nodo al visitarlo

Contenidos

Estructura : Descripción

- Para trabajar con el segment tree asumiremos que n es potencia de 2. De no serlo, basta extender el arreglo v con a lo sumo n elementos adicionales para que sea potencia de 2.
- Utilizaremos un árbol binario completo codificado en un arreglo:
- A_1 guardará la raíz del árbol.
- Para cada i, los dos hijos del elemento A_i serán A_{2i} y A_{2i+1}.
- El padre de un elemento i será `i/2`, salvo en el caso de la raíz.
- Las hojas tendran índices en $[n, 2n)$.
- La idea será que cada nodo interno del árbol (guardado en un elemento del arreglo) almacene el mínimo entre sus dos hijos.
- Las hojas contendrán en todo momento los elementos del arreglo v.

Estructura : Inicializacion

- Llenamos $A_n, \cdots, A_{2n-1}$ con los elementos del arreglo v.
- Usando programación dinámica hacemos simplemente:

$$A_i = \min(A_{2i}, A_{2i+1})$$

- Notar que se debe recorrer i en forma descendente para no utilizar valores aún no calculados.
- El proceso de inicialización toma tiempo $O(n)$, y la estructura utiliza $O(n)$ memoria.

Estructura : Modificaciones

- Para modificar el arreglo, utilizaremos nuevamente la fórmula:

$$A_i = \min(A_{2i}, A_{2i+1})$$

- Notemos que si cambiamos el valor de v_i por x, debemos modificar A_{n+i} haciéndolo valer x, y recalcular los nodos internos del árbol...
- Pero sólo se ven afectados los ancestros de A_{n+i}.
- Luego es posible modificar un valor de v en tiempo $O(\lg n)$, recalculando sucesivamente los padres desde A_{n+i} hasta llegar a la raíz.

Estructura : Queries

Consideremos el siguiente algoritmo recursivo:

- $$f(k,l,r,i,j) = \begin{cases} A_k & \text{si } i \leq l < r \leq j; \\ +\infty & \text{si } r \leq i \text{ o } l \geq j \\ \min(f(2k,l,\frac{l+r}{2},i,j), & \\ ,f(2k+1,\frac{l+r}{2},r,i,j)) & \textit{sino.} \end{cases}$$
- La respuesta vendrá dada por $RMQ(i,j) = f(1,0,n,i,j)$
- La complejidad temporal de un query es $O(\lg n)$

Contenidos

Tarea

- http://www.spoj.pl/problems/KGSS/
- http://poj.org/problem?id=2374
- http://acm.sgu.ru/problem.php?contest=0&problem=155

Contenidos

Motivación

- A veces, en los problemas tenemos modificaciones de **rangos** completos, y no de un único elemento.
- En ese caso, modificar los elementos uno por uno sería demasiado caro. Queremos actualizar **de a nodos grandes** completos, al igual que hacemos con las queries.
- La forma de hacerlo es ser “lazy”, y tener en cada nodo información de operaciones “que todavía falta propagar”.

Cambios en la implementación

- Cada nodo guardará ahora dos datos distintos: su **valor**, que es esencialmente lo mismo de antes, y una **información lazy**, que indica lo que falta propagar hacia abajo.
- Cuando se realiza una actualización de rango:
  - Se buscan los nodos que forman el rango de manera idéntica a la query de rangos del Segment Tree normal.
  - En cada uno de dichos nodos, se debe **actualizar la información lazy**, y **recalcular el valor del nodo** acorde a la operación que estamos realizando.
  - El valor de los ancestros de los nodos modificados, se recalcula de la manera normal a la vuelta de la recursión.
- Siempre que se va a bajar de un nodo a sus dos hijos, se debe primero **propagar** la información lazy hacia los hijos.

Ejemplo de uso

- Dada una cadena de paréntesis que abren y cierran, sobre la cual se hacen modificaciones de caracteres, saber en todo momento si está “bien parenteseada”.
- Algoritmos geométricos, como “área de unión de rectángulos” (Se mencionará en la clase de geometría).

Contenidos

Motivación

- Normalmente, cuando operamos con Segment Tree, cada update "destruye" la versión anterior de la estructura (no la tenemos más).
- Decimos que una estructura es **persistente**, cuando esto no ocurre: Siempre tenemos acceso a todas las versiones previas de la estructura, y un update lo que hace es simplemente darnos una nueva versión disponible.

Implementación

- Una forma de implementar persistencia eficientemente es con lo que se llama **Path Copying**.
- Para esto guardamos el Segment Tree explícitamente como árbol, **con punteros**: Cada nodo tiene punteros a sus dos hijos.
- Los nodos existentes **jamás se modifican**: Cuando se hace un cambio, **se crean nodos nuevos**.
- Los nodos nuevos apuntan a los nodos viejos no modificados.
- Como solamente se actualizan $O(\lg n)$ nodos en una query normal, aquí solamente se crean esa cantidad de nodos nuevos por operación.

Implementación con Lazy Creation

- Curiosamente, la persistencia nos simplifica la tarea de implementar **Lazy Creation** (Es decir, crear los nodos del Segment Tree solamente cuando se usan).
- Para esto podemos iniciar con un nodo cuyos dos hijos sean sí mismo, inicializado “en cero”. Esto representa un árbol infinito de ceros, mediante un único nodo en memoria.
- Como el Segment Tree es persistente, este nodo nunca se modifica, así que la estructura “no se rompe”, y solo se crean nodos nuevos ante updates.

Ejemplo de uso

- Contar cantidad de puntos de un conjunto que caen en un rectángulo cualquiera (“Tabla aditiva rala”)
- Tener disponible para cada nodo de un árbol con raíz, un Segment Tree con todos los ancestros insertados.