Estructuras de datos útiles (AKA piolas)

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2011

“Estructuras piolas. Yeah, mi tipo favorito de estructuras.”

Javier Corti

“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

¿Qué es una estructura de datos útil?

Estructura: Visión del usuario.

Una estructura de datos adecuada es un tipo abstracto de datos que nos sirve para responder “queries” sobre un conjunto de datos, posiblemente sujetos a modificaciones durante el proceso de query.

- Se le llama “queries” simplemente a preguntas que nos interesa hacerle a la estructura. Notar que esta forma de ver lo que es una estructura es extremadamente general y virtualmente cualquier cosa se puede pensar como una estructura. No obstante, esta forma de pensar resulta útil.

Ejemplos de visión del usuario de las estructuras

Ejemplo 1

Si tenemos un arreglo de enteros, nos puede interesar una estructura que sea capaz de dar la suma de cualquier intervalo del arreglo rápidamente. Además, puede interesarnos o no según el problema modificar el arreglo entre medio de las queries.

Ejemplo 2

En otro ejemplo, podríamos tener un conjunto de enteros al cual agregamos y quitamos números, y nos interesa ser capaces de saber en todo momento cuál es el más chico del conjunto. O quizá cuál es la mediana del conjunto.

¿Qué es una estructura de datos útil? (Continuado)

Estructura: Visión de la implementación interna.

Una estructura de datos adecuada es una forma concreta de organizar los datos en la memoria, junto a los algoritmos necesarios para manipular dichos datos organizados.

- Al igual que antes, la definición es extremadamente general y casi cualquier porción de software con una función específica puede llegar a verse como una estructura... Pero de vuelta es una forma de pensar que resulta útil.

Ejemplos de visión de la implementación de las estructuras

Ejemplo 1 (Continuado)

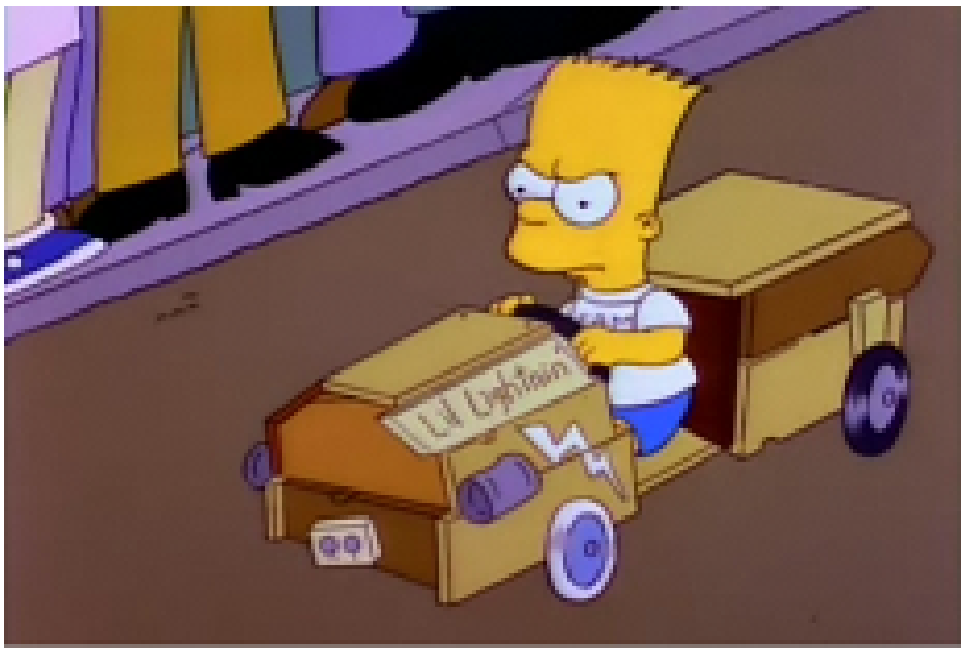
Mantenemos un arreglo con todos los valores. Para modificar un valor, simplemente lo modificamos en el arreglo. Cuando nos piden la suma de un intervalo, iteramos sobre todo el intervalo acumulando la suma.

Ejemplo 2 (Continuado)

Mantenemos una lista ordenada con todos los números de nuestro conjunto. Cuando nos piden quitar o agregar uno, lo quitamos o agregamos y luego reordenamos toda la lista resultante. El menor elemento siempre lo obtenemos como el primero de la lista, y la mediana como el elemento en la posición del medio de la misma.

¡Falta la nitro!

- Ya vimos dos ejemplos de estructuras, pero las implementaciones que propusimos son muy ineficientes (Aunque pueden resultar más que suficiente en muchos contextos, dando una solución fácil y efectiva).
- En lo que resta de la presentación mostraremos varias estructuras eficientes para utilizar en diversas situaciones.



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 $v_{i_1,\cdots,i_r}$ 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

Visión del usuario del union find

Union find

Dado un conjunto de n elementos que identificaremos con enteros de 0 a $n-1$, que inicialmente se hallan separados en n "bolsas" , "clases" o "componentes" distintas con un elemento cada una; una estructura de union find permite averiguar rápidamente la componente de cualquier elemento (**find**) y también permite unir las componentes de dos elementos dados en una sola (**union** o **join**).

Contenidos

Estructura

Utilizaremos dos arreglos para la estructura:

- Mantenemos un arreglo `P` de n elementos con la componente del elemento i en la posición i.
- Mantenemos un arreglo `C` de n conjuntos, donde el i-ésimo conjunto contiene todos los elementos de la i-ésima componente.
- La forma habitual de representar estos conjuntos es con listas (Por ejemplo se puede usar simplemente un `vector<int>` en C++).
- Inicialmente, `P[i] = i , C[i] = {i}`

Implementando las operaciones

- **find(i)**: Simplemente devolvemos `P[i]`. Es $O(1)$
- **join(i,j)**: Elegimos **el más chico** (menor cantidad de elementos) entre `C[i]` y `C[j]` (supongamos que sea `C[i]`), y volcamos todos sus elementos en el otro.
  Ademas para cada uno de los elementos `x` en `C[i]`, actualizamos su valor en `P` de forma tal que sea `P[x] = j`. Esta operación toma $O(size(\texttt{C[i]}))$
- El costo promedio por operación es $O(\lg n)$
- Una ventaja de esta implementación es que es muy fácil iterar una componente particular, ya que tenemos sus elementos en `C[i]`. Esta posibilidad es muy práctica y a veces puede resultar necesaria si se necesita mantener información adicional además de la dada por el union find.

Contenidos

Estructura

Mantendremos un bosque (conjunto de árboles), en el que cada árbol representará una componente particular, identificada por su raíz. Utilizaremos un solo arreglo para la estructura:

- Mantenemos un arreglo `P` de n elementos con el padre del elemento i en la posición i (si es la raíz de una componente, lo marcaremos con -1).
- Inicialmente, `P[i] = -1` para todos los `i`

Implementando las operaciones

- **find(i)**: Si `P[i] == -1`, `find(i) = i`, sino
  `find(i) = find(P[i])`
- **join(i,j)**: Si `find(i) != find(j)`, hacemos
  `P[find(i)] = find(j)`
- Esta implementación tiene un problema: El find es muy lento si el árbol es profundo.
- Solución: "Aplanar" el árbol durante los find.
- **find(i)**: Si `P[i] == -1`, `find(i) = i`, sino hacemos
  `find(i) = P[i] = find(P[i])`
- La complejidad promedio por operación termina siendo $O(\lg n)$
- Una desventaja de la representación con árboles es que no hay una forma fácil de iterar los elementos de una componente determinada.

Codigo fuente C++

```
1 int P[MAXN];
2 int find(int i) { return (P[i] == -1 ? i : P[i] = find(P[i])); }
3 void join(int i, int j) { if(find(i)!=find(j)) P[find(i)] = find(j); }
4 void init(int n) {fill(P,P+n,-1);}
```

Contenidos

Tarea

- http://livearchive.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=360&page=show_problem&problem=2628 (“Monkey Island”, regional Polaca 2009)
- http://www.topcoder.com/stat?c=problem_statement&pm=2932
- http://www.topcoder.com/stat?c=problem_statement&pm=7921

Contenidos

Visión del usuario de RMQ

RMQ

Dado un arreglo v de n elementos $v_0, \cdots, v_{n-1}$, una estructura de RMQ permite responder rápidamente consultas por el valor

$$RMQ(i,j) = \min_{k=i}^{j-1} v_k, 0 \leq i < j \leq n$$

.

Visión del usuario (Continuada)

- Hablamos del mínimo pero según el problema puede interesar el máximo. Todo lo que hagamos para mínimo vale igual para máximo, intercambiando las nociones de máximo/mínimo.
- Opcionalmente, la estructura puede soportar modificaciones al arreglo v o no. Veremos una estructura que lo soporta y una que no.
- Hemos dicho que las queries de RMQ devuelven el mínimo valor en el intervalo, pero a veces puede ser útil devolver el **índice** de un mínimo valor. Todo lo que hagamos sirve igual para este caso, con sólo tener cuidado de guardar en la estructura índices en lugar de valores.

Contenidos

Estructura : Inicialización

- Utilizaremos programación dinámica para llenar el arreglo M, definido como:
$$M(i,k) = RMQ(i, i+2^k), 0 \leq i \leq n-2^k$$
- Para esto basta notar:
- $$M(i,0) = RMQ(i, i+1) = v_i$$
- $$M(i,k+1) = RMQ(i, i+2^{k+1}) =$$
$$= \min(RMQ(i, i+2^k), RMQ(i+2^k, i+2^{k+1})) = \min(M(i,k), M(i+2^k, k))$$
- La tabla M contiene $\Theta(n \lg n)$ elementos, luego la inicialización requiere $\Theta(n \lg n)$ tiempo y memoria.

Estructura : Queries

- Una vez computado el arreglo M, para responder un query basta notar que:

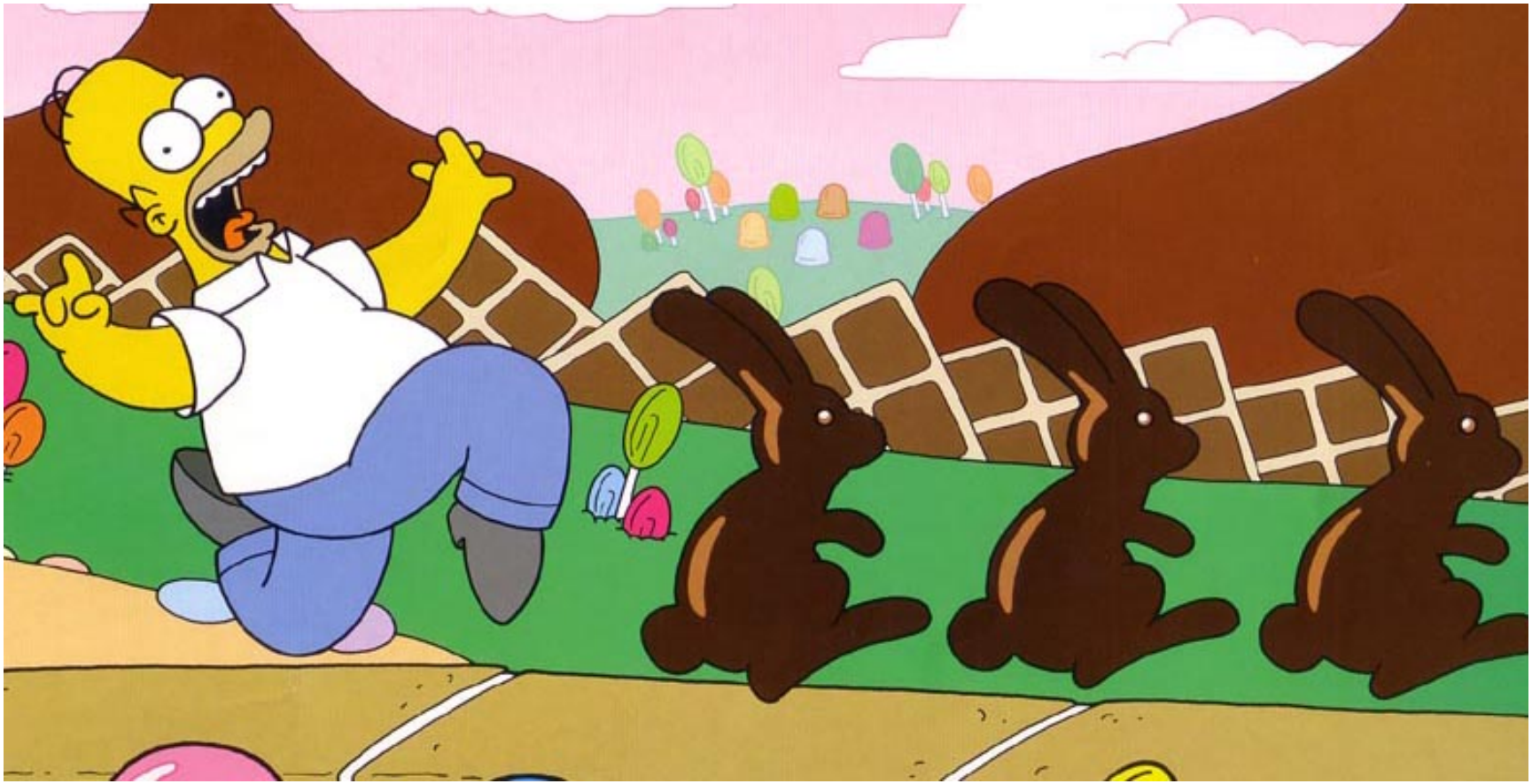
$$RMQ(i,j) = \min(RMQ(i, i+2^k), RMQ(j-2^k, j)) =$$

$$= \min(M(i,k), M(j-2^k, k)), 2^k \leq j-i < 2^{k+1}$$

- Luego podemos responder queries en $O(1)$, siempre y cuando podamos computar k en $O(1)$.
- Esto se puede lograr al trabajar con enteros de 32 bits haciendo:
- `k = 31 - __builtin_clz(j-i)` en C++,
- `k = 31 - Integer.numberOfLeadingZeros(j-i)` en Java.
- Notar que esta estructura **no** permite modificar el arreglo durante el proceso de consulta.

¡Intervalo!

A despejarse el bocho. En un rato continuamos.



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

Problema:

Aplicación de RMQ:
http://acm.sgu.ru/problem.php?contest=0&problem=155

- ¿Qué pasa con la raíz?
- Esto nos sugiere un algoritmo de divide and conquer...
- ¡Pero la complejidad en peor caso resulta cuadrática!

Buscando ideas...

Hay dos problemas que resolver en el paso del divide and conquer:

- Encontrar el mínimo elemento en la heap (raíz del arbol). La solución trivial es $O(n)$
- Dividir a los nodos en los menores que la raíz y los mayores que la raíz. La solución trivial es $O(n)$
- Idea 1: Para dividir los nodos rápidamente, podemos ordenarlos una sola vez al comienzo. $O(n \lg n)$
- Luego, en todo momento los elementos de nuestro problema constituyen un intervalo, y podemos separar en $O(1)$

Pero necesitamos más ideas...

- Idea 2: Para encontrar el mínimo, podemos usar una estructura de RMQ sobre el arreglo de elementos ordenados, ya que siempre se consultará por un intervalo del mismo.
- Como no interesa modificar el arreglo, lo mejor es utilizar sparse table ya que es más simple que un segment tree. Luego de un primer preproceso $O(n \lg n)$, podremos encontrar mínimos en $O(1)$
- La complejidad final resulta $O(n \lg n)$

Nosotros ya estamos pero...

- Para el problema de competencia en cuestión, una solución $O(n \lg n)$ es más que suficiente.
- Pero es posible resolverlo en $O(n)$, utilizando una pila adecuadamente.
- Ver por ejemplo:
  `http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor`

Contenidos

Tarea

- http://www.spoj.pl/problems/KGSS/
- http://poj.org/problem?id=2374

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

- Ya hemos mencionado la visión del usuario de esta estructura.
- La diferencia consiste en este caso en que además de consultar suma de subarreglos consecutivos, nos interesara modificar los elementos almacenados en la estructura. Por simplicidad, supondremos que lo haremos sumándole una constante (negativa o positiva), a un valor en una posición dada (Operación que notaremos $add(i, x)$).
- Tambien por simplicidad estudiaremos el cálculo de la suma de los elementos con índice menor que un cierto valor i dado, operación que notaremos $accum(i)$. Notar que la suma del intervalo $[i, j)$ se calcula simplemente como $accum(j) - accum(i)$.
- 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);
5           bit <= N;
6           ceros -= bit, bit = ceros &(-ceros))
7               v[(i &~(bit-1)) | bit] += x;
8  }
```

- 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

Extras

Otras estructuras interesantes.

- Dado un árbol con raíz, dar una estructura capaz de responder, para cualquier nodo v, cuál es el k-ésimo nodo en el camino desde v hasta la raíz, en tiempo $O(\lg k)$, con tiempo de preproceso $O(n \lg n)$. En varios problemas se combina esta estructura con LCA.
- Sumar una constante a todos los elementos de un sub-arreglo de un arreglo en $O(1)$, con la limitación de que solo se puede mirar el estado del arreglo al final de todas estas sumas, luego de hacer un cómputo de complejidad lineal. Está fuertemente relacionado con las tablas aditivas.

Contenidos

Chiao

¡¡¡¡¡¡¡¡Éxito a todos en todo!!!!!!!!