Tipos Abstractos de Datos
Técnicas algorítmicas de Divide y vencerás y Vuelta atrás

Isabel Pita.

Facultad de Informática - UCM

12 de septiembre de 2023

Tipos de Datos

- Seleccionar correctamente los tipos de datos utilizados para resolver el problema.
- El uso adecuado del tipo de datos puede ser la diferencia entre obtener un veredicto AC o TLE.
- Es necesario conocer las características de cada tipo (sus puntos fuertes, debilidades y la complejidad en tiempo y espacio) y saber usarlo.

TADs lineales I

- Array estático (*tipo* `v[MAX]`)/dinámico (`vector`), es el tipo que más se utiliza.
  Operaciones de la librería (`<algorithm>`):
  - Ordenar: `sort`($\mathcal{O}(n\log(n))$, `partial_sort`, `stable_sort`.
  - Búsqueda: `lower_bound`, `upper_bound`, `binary_search`
    Complejidad $\mathcal{O}(\log(n))$.
    Siendo n el número de elementos del vector.
- Listas enlazadas (`list`). No suelen utilizarse porque el acceso a los elementos es muy ineficiente. Sólo resultan interesantes si se pide hacer inserciones en cualquier lugar de la lista de forma dinámica

TADs lineales II

- Pilas (`stack`). Operaciones:
  - `pop()`, `push()`, `top()`, `empty()`. Complejidad $\mathcal{O}(1)$
- Colas (`queue`). Operaciones:
  - `pop()`, `push()`, `front()`, `empty()`. Complejidad $\mathcal{O}(1)$
- Colas dobles (`deque`). Operaciones:
  - `front()`, `back()`,`pop_front()`, `pop_back()`, `push_front()`, `push_back()`, `empty()`. Complejidad $\mathcal{O}(1)$

TADs no lineales

- Diccionarios ordenados: `map`/`set`.
  Operaciones (siendo n el número de elementos de la colección)
  - `insert`, `erase`, `operator[]`, `find`, `count`.
    Complejidad: $\mathcal{O}(\log(n))$
- Diccionarios no ordenados
  `unordered_map`/`unordered_set`.
  Operaciones
  - `insert`, `erase`, `operator[]`, `find`, `count`.
    Complejidad: $\mathcal{O}(1)$
- Colas de prioridad `priority_queue`. Librería `queue`.
  Operaciones: (siendo n el número de elementos de la colección)
  - `top` ($\mathcal{O}(1)$), `push` ($\mathcal{O}(\log(n))$), `pop` ($\mathcal{O}(\log(n))$), `empty` $\mathcal{O}(1)$)

Cola de prioridad

- Una cola de prioridad obtiene el objeto más prioritario de la colección. Pueden ser de máximos o de mínimos.

Cola prioridad	Obj. prioritario	$a_1 < \ldots < a_n$	$b_1 > \ldots > b_n$
máximos	derecha	a_n	b_n
mínimos	izquierda	a_1	b_1

- La cola de prioridad implementada en la STL es una cola de máximos.

```
priority_queue<int,vector<int>> cm; //  less
priority_queue<int,vector<int>,greater<int>> cm;
```

Los objetos función `greater` y `less` se encuentran en la librería `<functional>`

Definir un orden en TADs y funciones

Se define el orden en un objeto función:

```
class ord{
public:
 ord(){}
 bool operator() (const type& a, const type& b) const
  {
      // Aqui la comparacion
  }
};
```

Ejemplos de uso:

```
sort(v.begin(), v.end(), ord());

map<string, int, ord> mp;
```

Divide y vencerás

- El problema se hace *más sencillo* dividiéndolo en partes más pequeñas.
- Son soluciones recursivas.
- Las soluciones más usadas se basan en la búsqueda binaria (sobre vectores ordenados):
  - Comprobar si existe un elemento en un vector.
  - Encontrar los valores en que una función se hace cero (Método de bisección). Se aplica sobre un rango de valores $[a..b]$ tal que $f(a) < 0 \wedge f(b) > 0$ o $f(a) > 0 \wedge f(b) < 0$. Se calcula el valor de la función en el punto medio del intervalo y se selecciona uno de los dos subintervalos como solución.
  - Búsqueda en un espacio de soluciones ordenado de forma lineal.

Vuelta atrás (Backtraking)

- Método que explora todo el árbol de soluciones de un problema.
- La complejidad de los algoritmos de vuelta atrás es exponencial o factorial. Por lo tanto solo debe utilizarse cuando no existe otro método conocido para resolver el problema (o es muy costoso de implementar) y los datos de entrada son pequeños.

Complejidad en tiempo vs tamaño de los datos

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that your CPU can compute $100M$ items in 3s.

[1]Competitive programming 3. S. Halim, F. Halim.

Límites

- Familiarity with these bounds:
  - $2^{10} = 1,024 \approx 10^3$, $2^{20} = 1,048,576 \approx 10^6$.
  - 32-bit signed integers (`int`) and 64-bit signed integers (`long long`) have upper limits of $2^{31}-1 \approx 2 \times 10^9$ (safe for up to ≈ 9 decimal digits) and $2^{63}-1 \approx 9 \times 10^{18}$ (safe for up to ≈ 18 decimal digits) respectively.
  - Unsigned integers can be used if only non-negative numbers are required. 32-bit unsigned integers (`unsigned int`) and 64-bit unsigned integers (`unsigned long long`) have upper limits of $2^{32}-1 \approx 4 \times 10^9$ and $2^{64}-1 \approx 1.8 \times 10^{19}$ respectively.
  - If you need to store integers $\geq 2^{64}$, use the Big Integer technique (Section 5.3).
  - There are $n!$ permutations and 2^n subsets (or combinations) of n elements.
  - The best time complexity of a comparison-based sorting algorithm is $\Omega(n \log_2 n)$.
  - Usually, $O(n \log_2 n)$ algorithms are sufficient to solve most contest problems.
  - The largest input size for typical programming contest problems must be $< 1M$. Beyond that, the time needed to read the input (the Input/Output routine) will be the bottleneck.
  - A typical year 2013 CPU can process $100M = 10^8$ operations in a few seconds. [2]

[2] Competitive programming 3. S. Halim, F. Halim.

Ejercicios y cuestiones

Describe como implementarías los siguientes algoritmos e indica que coste tendrían.

1. Dado un array de tamaño $n-1$ que contiene los números [1..n] entre los que falta uno. Determina que número falta.
2. Determina si un array contiene uno o más pares de elementos duplicados. Estudia diferentes posibilidades para los valores del vector y da la mejor solución para cada caso.
3. Dado un vector de valores enteros ordenados indica si existen dos valores en el vector cuya suma sea un valor x.
4. Dado un vector de valores enteros, no ordenados, mostrar los valores que estén en un rango $[a..b]$ en orden creciente.
5. Describe cómo implementarías dos pilas con un array.