More on Arrays and Structures

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 15–16, 2011

What We Have Done

- Why Correctness Proof?
- Sequential and Binary Search
- Space Complexity
- Time Complexity
- Asymptotic Notations
- Practical Complexity
- Arrays
- Dense Array versus Sparse Array
- Concrete versus Abstract Data Type
- Reading Assignment:
  Some Examples of Space/Time Complexity, Dynamic 1-D Array

Arrays: from Implementation to Abstraction

C Implementation View

(One-dimensional) array is a block of consecutive memory that

- holds a list of N elements
- allows users to retrieve the k-th element
- allows users to store to the k-th location

An Abstract View

Abstract (one-dimensional) array

- holds a list of N elements
- allows users to retrieve the k-th element
- allows users to store to the k-th location

different implementations:
different space/time complexity

Dense Array versus Sparse Array

one abstract array, two possible implementations

```
int dense[10] = {1, 3, 0, 0, 0, 0, 0, 0, 0, 2};
int sparse[3][2] = {{0, 1}, {1, 3}, {9, 2}};
```

- dense array: store everything (consecutively), needs 10 positions
  - space: $O(N)$ for a length-N array
  - retrieving: $O(1)$
  - storing: $O(1)$
  - creating: $O(1)$
- sparse array: store only non-zero (index, element) pairs, needs 3 pairs
  - space: $O(E)$ for E elements, better than $O(N)$ if E small
  - retrieving: $O(\log E)$ if index ordered (HOW?)
  - storing: ???
  - creating: ???

note: often use **array** to mean dense array only

Concrete Data Type (Sec. 1.4)

array consists of ...

- objects: a set of (*index*, *element*) pairs (== a list of elements)
- actions: retrieve, store, create which sets/gets the objects

- concrete data type: the **actual outcome** of the type
  - object representation + action implementation
  - for actual coding, per-platform optimization, etc.

(dense 1-D) array in C

- object representation: a block of consecutive memory, with a chunk representing each *element* element for each *index*
- action implementation: $[\cdot]$ for retrieving and storing, `malloc` for creating, etc.

Abstract Data Type (Sec. 1.4)

array consists of ...

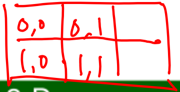
- objects: a set of (*index*, *element*) pairs (== a list of elements)
- actions: retrieve, store, create which sets/gets the objects

- abstract data type: the **pseudo essence (contract)** of the type
  - object specification + action specification
  - for illustration, high-level analysis, etc.

abstract 1-D array

- object specification: (*index*, *element*) pairs with *index* $\in \{0, \cdots, N-1\}$
- action specification:
  retrieve(*index*) returns the *element* associated with *index*;
  store(*index*, *element*) sets *element* to be associated with *index*;
  create(N) creates the objects, etc.
  (sometimes with time/space constraints)

will usually look at abstract data type first before going concrete



abstract rectangular 2-D array

- object specification: (*index*, *element*) pairs with $index \in \{(0,0),(0,1),\cdots,(N-1,M-1)\}$
- action specification: retrieve(*index*); store(*index*, *element*); create(N, M), etc.

2-D array by 1-D array in C

- object representation: a block of consecutive memory of size $N * M$, with a chunk representing each *element* for each *index*
- action implementation:

2-D Array (Subsec. 2.2.2): by 1-D Array

```
#define N (100)
#define M (200)
int* twodim = (int*)malloc(sizeof(int)*N*M);

int get(int* arr, int n, int m)
  { return arr[n*M + m]; }
```

2-D Array: by 1-D Array with Constant Folding

abstract rectangular 2-D array

- object specification: (*index*, *element*) pairs with $index \in \{(0,0),(0,1),\cdots,(N-1,M-1)\}$
- action specification: retrieve(*index*); store(*index*, *element*); create(N, M), etc.

2-D array by 1-D array with constant folding in C

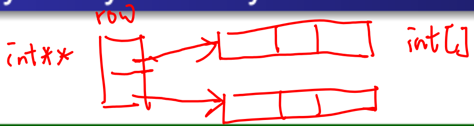
- object representation: a block of consecutive memory of size $N * M$, with a chunk representing each *element* for each *index*
- action implementation:

2-D Array: by 1-D Array with Constant Folding

```
#define N (100)
#define M (200)
int twodim[N][M];

int get(int arr[][M], int n, int m)
  { return arr[n][m];}
```

2-D Array: by Array of Arrays



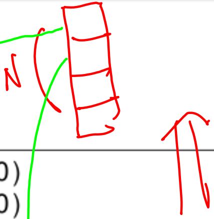
abstract rectangular 2-D array

- object specification: (*index*, *element*) pairs with $index \in \{(0,0),(0,1),\cdots,(N-1,M-1)\}$
- action specification:
  retrieve(*index*); store(*index*, *element*); create(N, M), etc.

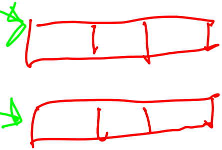
2-D array by array of arrays in C

- object representation: N blocks of consecutive memory of size M
- action implementation:

2-D Array: by Array of Arrays




```
#define N (100)
#define M (200)
int** twodim = (int**)malloc(sizeof(int*)*N);
for(int n=0;n<N;n++)
  twodim[n] = (int*)malloc(sizeof(int)*M);
int get(int** arr, int n, int m)
  { return arr[n][m];}
```



Comparison of Three Implementations

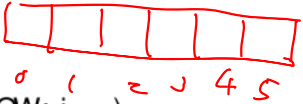
```
int* twodim = (int*)malloc(sizeof(int)*N*M);
int twodim[N][M];
int** twodim = (int**)malloc(sizeof(int*)*N);
```

	1	2	3
space	$N * M$ integers	$N * M$ int.	$N * M$ int. + N pointers
type	`int*`	`int*[M]`	`int**`
create	constant	constant	$O(N)$
retrieve	arithmetic+dereference	arith.+deref.	deref.+deref.

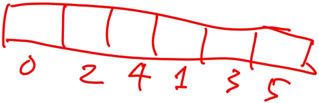
method 2 for static allocating; method 1 or 3 for dynamic allocating (your choice)

A Tale between Two Programs

```
int rowsum(){
  int i, j;
  int res = 0;
  for(i=0;i<MAXROW; i++)
    for(j=0;j<MAXCOL; j++)
      res += array[i][j];
}
```




```
int colsum(){
  int i, j;
  int res = 0;
  for(j=0;j<MAXCOL; j++)
    for(i=0;i<MAXROW; i++)
      res += array[i][j];
}
```



Reading Assignment

be sure to go ask the TAs or me if you are still confused

Polynomials (Sec. 2.4)

```c
typedef struct{
  int degree;
  double* coef;
} densepoly;
typedef struct{
  int nTerms;
  int* expo; /* expo[i] ordered */
  double* coef;
} sparsepoly;
```

- `densepoly` versus `sparsepoly`: like dense array versus sparse
- a simple polynomial adding algorithm
  - allocate the resulting poly
  - fill in the values
- trivial for `densepoly` (HW1), slightly harder for `sparsepoly`

Adding Sparse Polynomials

```c
typedef struct{
   int nTerms;
   int* expo; /* expo[i] ordered */
   double* coef;
} sparsepoly;
```

add $x^{100} + 2x^3 + 3$ and $4x^4 + 5x^3 + 6x + 7$

- allocate the resulting poly
- fill in the values

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Sparse Matrix (Sec. 2.5)

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Specialty

a rectangular 2-D array that contains many common elements (0) that we may not want to repeatedly store

Matrix Abstract Data Type

matrix consists of ...

- objects: a set of (*row*, *column*, *value*) triples where *value* is numerical
- actions: create, transpose, add, multiply, (retrieve, store)

- dense implementation: as 2D dense arrays
- array of array implementation:
  - "(dense 1D) of (sparse 1D)"
  - "(sparse 1D) of (sparse 1D)"
- ordered triples implementation: our next topic

Ordered Triples Implementation

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Ordered(-by-row-then-by-col) Triples

6	6	8
row	*col*	*value*
0	0	15
0	3	22

- space complexity ?
- time complexity for retrieve?

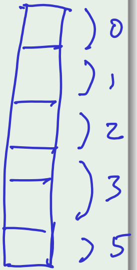
simple exercise: compare to unordered triple implementation

Transposing A Sparse Matrix

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \end{bmatrix}^T = \begin{bmatrix} 15 & 0 & 0 \\ 0 & 11 & 0 \\ 0 & 3 & 0 \\ 22 & 0 & -6 \\ 0 & 0 & 0 \\ -15 & 0 & 0 \end{bmatrix}$$

Ordered(-by-row-then-by-col) Triples

6	3	6
row	*col*	*value*
0	0	15
0	3	22
0	5	-15
1	1	11
1	2	3
2	3	-6



Transpose by Scanning Each *col* (Prog. 2.8)

set up a sparse matrix *res* of attributes (*col*, *row*, *elements*)
for $j \leftarrow 0$ to $col - 1$ **do**
  for $e \leftarrow 0$ to *elements* **do**
    if $e.col == j$
      append ($e.col$, $e.row$, $e.value$) to *res*
    end if
  end for
end for

- space complexity: $\Theta(elements)$ for *res*, constant for others
- time complexity: $\Theta(col * elements)$ (so $O(col * elements)$)

Netflix competition:
$row = 17700$, $col = 480189$, $elements = 100480507$
col * elements $\approx 5 \cdot 10^{13}$ ($>$ 5 hr on 2.8 GHz CPU)

Transpose by Scanning Once (Prog. 2.9) I

- to save time on transposing, we want to scan only once

set up a sparse matrix *res* of attributes (*col*, *row*, *elements*)
for ~~$j \leftarrow 0$ to $col - 1$~~ **do**
    for $e \leftarrow 0$ to *elements* **do**
        if ~~$e.col == j$~~
            append (*e.col*, *e.row*, *e.value*) to the (*e.col*)-th pile
        end if
    end for
end for

Transpose by Scanning Once (Prog. 2.9) II

- where's the j-th pile? pre-compute pile size and starting locations

set up two arrays *pilesize* and *pilestart*, each of length *col*
for $e \leftarrow 0$ to *elements* **do**
  if $e.col == j$
    $pilesize[j] \leftarrow pilesize[j] + 1$
  end if
end for
for $j \leftarrow 0$ to $col - 1$ **do**
  $pilestart[j] \leftarrow pilestart[j-1] + pilesize[j]$
end for

Transpose by Scanning Once (Prog. 2.9) III

- space complexity: $\Theta(elements)$ for *res*, $\Theta(col)$ for the helping arrays
- time complexity: $\Theta(col)$ for pre-computing, $\Theta(elements)$ for scanning

Netflix competition:
$row = 17700$, $col = 480189$, $elements = 100480507$
col * 2 + elements 10^8 ($0.04 * constant$ sec. on 2.8 GHz CPU)

Recap: Trade-off

Concrete Implementation of the SparseMatrix Data Structure

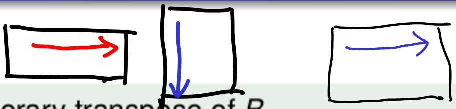
- unordered triples: simpler transpose $O(elements)$, time-consuming retrieval $O(elements)$
- ordered triples: harder transpose $O(col + elements)$, efficient retrieval $O(logrow * logcol)$

The Transpose Algorithm

- scanning each column: smaller space $O(elements)$, time-consuming algorithm $O(col * elements)$
- scanning once: bigger space $O(col + elements)$, efficient algorithm $O(col + elements)$

Good Programmer (a.k.a. you):
understand the trade-off clearly and make wise choices!

Matrix Multiplication (Subsec. 2.5.4)



do a temporary transpose of B
for $i \leftarrow 0$ to $rowA - 1$ **do**
  for $j \leftarrow 0$ to $colB - 1$ **do**
    compute $C[i, j]$ by multiplying $A[i, :]$ and $B[:, j]^T$
  end for
end for

- multiplying $A[i, :]$ and $B[:, j]^T$: similar to (sparse) polynomial adding —keep it in your toolbox
- time complexity:
  - each multiplying takes $O(\#A_i + \#B_j)$
  - total (careful counting): $O(colB * \#A + rowA * \#B)$

~~Structures and Unions~~ (Sec. 2.6)

Multi Dim   Matrix

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Reading Assignment

be sure to go ask the TAs or me if you are still confused

Strings: Pattern Matching (Subsec. 2.7.3)

find the position that *pat* (first) shows up in *string*

for $i \leftarrow 0$ to $len(string) - 1$ **do**
  if *pat* matches $strings[i, i + len(pat) - 1]$
    matching found
  end if
end for
matching not found

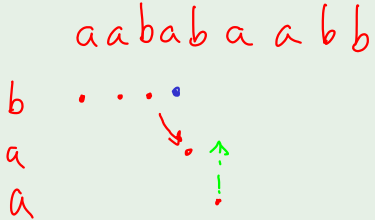
string aababaabb

pat baa

- the IF takes $O(m)$ for $m = len(pat)$
  —can use heuristic on comparing *begin* and *end* first, but still $O(m)$ in the worst case
- so total $O(n * m)$ for $n = len(string)$

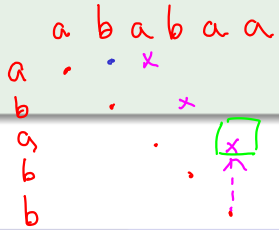
Slow Pattern Matching

$i, j \leftarrow 0$
while $i < len(string)$ and $j < len(pat)$ **do**
  if $pat[j] == string[i]$
    $i \leftarrow i + 1, j \leftarrow j + 1$ (continue matching)
  else
    $i \leftarrow i - j + 1$
    $j \leftarrow 0$
    (fail and totally go back)
  end if
end while
check matching status



Fast Pattern Matching

$i, j \leftarrow 0$
while $i < len(string)$ and $j < len(pat)$ **do**
  if $pat[j] == string[i]$
    $i \leftarrow i + 1, j \leftarrow j + 1$ (continue matching)
  else
    $i \leftarrow i$
    decrease j such that $pat[0, j-1]$ matches $string[i-j, i-1]$
    (fail but continue partially)
  end if
end while
check matching status



Fast Pattern Matching

$i, j \leftarrow 0$
while $i < len(string)$ and $j < len(pat)$ **do**
  if $pat[j] == string[i]$
    $i \leftarrow i + 1, j \leftarrow j + 1$ (continue matching)
  else
    $i \leftarrow i$
    decrease j to t such that $pat[0, t-1]$ matches $string[i-t, i-1]$
    (fail but continue partially)
  end if
end while
check matching status

- number of $i \leftarrow i + 1$: $O(len(string))$
- number of decrease j: $O($ number of $j \leftarrow j + 1$ $)$ because j is always > 0
- total: $O(len(string))$ IF the decrease step is $O(1)$

How To Make the Decrease Step $O(1)$

- main tool: pre-compute and store (like fast transpose)
- how to decrease j to t such that $pat[0, t-1]$ matches $string[i-t, i-1]$
  - originally, $pat[0, j-1]$ matches $string[i-j, i-1]$
  - want, $pat[0, t-1]$ matches $string[i-t, i-1]$
  - so, $pat[0, t-1]$ matches $pat[j-t, j-1]$

  —want: largest t such that $pat[0, t-1]$ matches $pat[j-t, j-1]$
- a trivial algorithm of $O(len(pat)^2)$ (how?)
- faster algorithm: see textbook