ECSE-322

21 January 2008

Lecture 8

Multidimensional Arrays

Searching and sorting

unordered $O(N)$

ordered $O(\log_2 N)$ O

exchange sort $O(N^2)$

S retrievals $O(SN)$ $O(N^2) + O(S\log N)$

partition sort - Quicksort $O(N\log_2 N)$

The Quick Sort

- Take the original array and set index i to the left element, j to the right element
- Compare element i with element j. If j is greater than i, decrement j and repeat.
- If i is greater than j, exchange elements and increment i.
- Keep going until all elements have been considered -- the original left hand element will now be in its correct place and the array will be partitioned...

The Quick Sort

An example:

42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
36	23	74	11	65	58	94	42	99	87
36	23	74	11	65	58	94	42	99	87
36	23	42	11	65	58	94	74	99	87
36	23	42	11	65	58	94	74	99	87
36	23	42	11	65	58	94	74	99	87
36	23	42	11	65	58	94	74	99	87
36	23	11	42	65	58	94	74	99	87

42 in its correct place

All these are > 42

The Quick Sort

- We now have two lists
    - 36 23 11
    - 65 58 94 74 99 87
  - So repeat on each of these.
- Complexity issues:
    - This works well if the partition is approximately in half.
    - If the partition is one in one piece and the rest, then the performance degenerates to $O(N^2)$
    - The *best* performance is *O(NlogN)*

Multidimensional Arrays

- In the real world, most objects in engineering are multi-dimensional..
  - In describing them to a computer we need to be able to represent dimensions greater than one..
    - e.g. A building exists in 3d space - each point has three co-ordinates.
    - In electrical circuits each component may have several inputs and outputs, each input and output may connect several components - a multi-dimensional structure.

Multidimensional Arrays

- The real world is multidimensional
- Computer memory is one dimensional
- A mapping is needed between the two…
  - How can this be done?
  - What is the cost of doing it?

Multidimensional Arrays

- Consider:

1	5	7
3	8	4
10	12	21

This could be stored as:

1 5 7 **3** **8** **4** ***10*** ***12*** ***21***

Or as:

1 3 10 **5** **8** **12** ***7*** ***4*** ***21***

Multidimensional Arrays

- To understand the mapping the following information is needed:
  - What the linear array represents (e.g. a two dimensional array).
  - Whether it is stored by rows or columns.
  - How many columns (rows) are represented.
  - How long each column (row) is.
  - The type of each element.

Should all be same

Multidimensional Arrays

- In a high level language, this information is provided by a *Declaration Statement*…
  - e.g. in C:

r c

Float A[3][3];

- The two methods of storage are:
  - *storage by columns* (First index varying fastest)
  - *storage by rows* (Last index varying fastest)

Multidimensional Arrays

- Mapping a 2D array onto linear storage with row-wise storage generates:

  1,1 corresponds to 1

  1,2 corresponds to 2

  1,3 corresponds to 3

  2,1 corresponds to 4

  …

  3,3 corresponds to 9

For a 3 by 3 array

1,1 1,2 1,3 ?

Multidimensional Arrays

- In general, the mapping rule can be written as:

  s(i,j) = (i-1)J + j

  s(i,j,k) = (i-1)JK + (j-1)K + k

  s(i,j,k,l) = (i-1)JKL + (j-1)KL + (k-1)L + l

  ....

This can be expressed as a nested polynomial

s(i,j,k,l) = (((i-1)J + (j-1))K + (k-1))L + l

3

1,1 1,2 1,3

0 i,j

A(2,1,7,9)

Multidimensional Arrays

- Complexity issues:
  - A D dimensional array requires D-1 multiplications, decrements and additions.
  - Thus

    Accessing a multidimensional array through a high level language is computationally expensive…

    e.g. Copying all the elements of a 3 dimensional array requires considerable address calculation.

FOR I = 1, N
J = 1, M
K =

Vectored Arrays

- How can we speed up address calculation?
  - Look at the algorithm and analyse it..
  - Consider the array s(I,J)

    $s(i,j) = (i-1)J + j$

    Let $b(i) = (i-1)J$

    Then $s(i,j) = b(i) + j$

    b(i) does not depend on j!
  - b(i) is known as the *base address* of row i

Vectored Arrays

- One base address is needed per row *and can be computed when the array is set up*.
- Indexing then only requires additions.
- The price?
  - An extra vector of memory locations to store the base addresses..
  - In an array of 12 rows, 10 columns there are 120 elements but only 12 base addresses

Vectored Arrays

- If the elements are real numbers (4 bytes) and the base addresses are integers (2 bytes) only 5% more memory is needed to store the base addresses.
- This is *Array Vectoring*
- The process can be extended to higher dimensions…

Vectored Arrays

– e.g. in 4D:

$$s(i,j,k,l) = b_1(i) + b_2(j) + b_3(k) + l$$

– For an N dimensional array:

- The extra memory required is O(N)
- The time for index computation is O(N)

Other Storage Schemes

- Many arrays have special properties
  - e.g. symmetry.
  - Using structural information can reduce storage requirements.
  - *But a price is always incurred!* (in this case, the base address vector may not be usable)

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 5 \end{bmatrix}$$

Other Storage Schemes

- Consider:

$$\begin{array}{cccc} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 10 \end{array}$$

This matrix is *symmetric* (a_{ij} = aji)

Only store

$$\begin{array}{cccc} 1 & 2 & 4 & 7 \\ & 3 & 5 & 8 \\ & & 6 & 9 \\ & & & 10 \end{array}$$

Other Storage Schemes

- So the matrix to be stored is:

$$
\begin{array}{cccc}
(1,1) & (1,2) & (1,3) & (1,4) \\
 & (2,2) & (2,3) & (2,4) \\
 & & (3,3) & (3,4) \\
 & & & (4,4) \\
1 & 2 & &
\end{array}
$$

Note that the column index is always greater than or equal to the row index:

r = min(i,j) (r is the row number)

c = max(i,j) (c is the column number)

store the elements in column wise order in a linear array.

Finding an Element

- Where is s(r,c)?
  - At location a(k) (this is a_{rc})…
  - The number of complete columns to the left of a_{rc} is c-1.
  - In a triangular matrix, c-1 columns contain n=c(c-1)/2 elements
  - In the c^{th} column the element is in the r^{th} position so

$$s(r,c) = r + c(c-1)/2$$

This can be vectored

Sparse Multidimensional Arrays

- These occur everywhere!
- How do we store an array with mostly non-zero entries so that storage is minimized?
- In 2-D (basically an extension of 1-D):
  - Store the non-zero values in a linear array
  - Use a linear array to indicate the rows of the non-zeros
  - Use a linear array to indicate the columns of non-zeros

$a(r, c)$

Sparse Multidimensional Arrays

Consider:

11	0	0	0	0	12
0	22	0	15	0	6
0	0	0	5	7	0
15	0	0	2	0	13
14	9	17	0	0	0
0	10	1	31	0	0

Array of Non-zeros: 11, 15, 14, 22, 9, 10, 17, 1, 15, 5, 2, 31, 7, 12, 6, 13

Array of row indices: 1, 4, 5, 2, 5, 6, 5, 6, 2, 3, 4, 6, 3, 1, 2, 4

Array of column indices: 1, 1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 5, 6, 6, 6

This needs 3 linear arrays each of size equal to the number of non-zeros

OK - but can we do better?