Arrays

Ajit Rajwade

Arrays (continued)

- Bubble sort
- Time complexity analysis
- Insertion sort
- Character strings
- Auxiliary Space Complexity
- Merge sort
- Quick sort
- Multi-dimensional arrays: matrices
- Dynamic Memory Allocation

Bubble Sort

Bubble Sort

- Another sorting technique, with some similarities to selection sort.
- In the first iteration of bubble sort, you compare each element of the unsorted array with the one immediately after it.
- That is for every `i`, if `A[i] > A[i+1]`, then you swap the values of `A[i]` and `A[i+1]`.
- At the end of this iteration, the **largest** element of `A` is in the **last** place.
- This step is repeated for `n-1` iterations.
- At the end of the `i`-th iteration, the **last `i` elements** are in their correct place.
- Thus in the beginning of iteration `i = 1`, the last element is in the correct place.
- This is similar to selection sort, but bubble sort achieves this by **many more swap operations**.
- Compare the codes for bubble and selection sort: the latter just found the maximum element and swapped `A[i]` and `A[maxindex]`.

```
void bubble_sort(float *A, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        // Last i elements will be in place after
this for loop
        for (j = 0; j < n - i - 1; j++)
        {
            if (A[j] > A[j + 1])
                swap(&A[j], &A[j + 1]);
        }
    }
}
```

Call from main as: `bubble_sort(A,n);`

How long does bubble sort take?

- When `i = 0`, the inner loop runs for `n-1` iterations (requires at most `n-1` comparison operations)
- When `i = 1`, the inner loop runs for `n-2` iterations (requires at most `n-2` comparison operations)
- …when `i = n-2`, the inner loop runs for 1 iteration.
- The total time = `n-1 + n-2 + … + 2 + 1 = (n-1)(n-1+1)/2 = n(n-1)/2`.

Notion of Time Complexity

The notion of time complexity of an algorithm

- The time complexity of an algorithm tells you the amount of time an algorithm takes.
- Estimated by counting the number of basic operations performed by the algorithm (assuming each operation takes an equal amount of time).
- We saw that bubble sort takes `(n-2)(n-1)/2` operations in the worst case. The dominating term here is n^2 (multiplied by a constant) the moment `n` exceeds some (small) value.
- Similarly, for selection sort the time taken is also n^2.
- The time taken for linear search is (dominantly) `n` operations.
- The time taken for binary search is (dominantly) `log n` operations.
- In all of these, `n` is the number of elements in the array.

Big O notation

- Commonly used to represent the time complexity of an algorithm.
- Definition:
  - Let f and g be two functions defined on the set of positive integers to reals.
  - We say f(n) is O(g(n)) if there exist positive integers n0 and C such that |f(n)| <= C g(n) for all values of n >= n0.
- Example:
  - `(n-2)(n-1)/2` is $O(n^2)$.
  - `3n` is `O(n)`.
  - `0.4 log(n)` is `O(log n)`.
  - Bubble sort and selection sort are $O(n^2)$ in the worst case.
  - Linear search is `O(n)` in the worst case.
  - Binary search is `O(log n)` in the worst case.

Insertion Sort (also called Insert Sort)

Insertion Sort

- Another sorting algorithm which is `O(n`^{2}`)` in time just like bubble sort and selection sort.
- It is based on the idea of maintaining a sorted sub-array at each iteration, and inserting the next element in such a way that the resultant sub-array remains sorted.
- A one-element array is trivially sorted.
- The second element is compared to the first element: if it is smaller than the first element, it is swapped with the first element.
- This leads to a sorted subarray with 2 elements.
- The third element is inserted into the correct place leading to a sorted subarray with 3 elements.
- This is repeated till the `n`-th element is inserted in the correct place.
- At the end of the `i`-th iteration, the first `i` elements of the array are in sorted order, and the remaining may not be.

Insertion sort: routine

```
void insertion_sort (int *A, int n) {
    int i, j;
    for (i=1;i < n;i++){ // for each element of the array
        j = i;
        while (j > 0 && A[j-1] > A[j]) { // find the location to insert it to
       // maintain a sorted subarray of i elements
            swap (&A[j],&A[j-1]);
            j = j - 1;
        }
    }
}
```

Demo of insertion sort from wikipedia:
https://en.wikipedia.org/wiki/Insertion_sort#/media/File:Insertion-sort-example-300px.gif
There is a slight difference between this code and the dry-run shown in the above gif. I will leave it to you to figure it out and modify the code above to reflect that change.

Time complexity

- The outer loop runs for `n-1` iterations.
- For each of these, the inner (while loops) runs for `1 + 2 + 3 + … + n-1` iterations which is $O(n^2)$.

Character Strings

Character Strings

- A character array is used to store a string of characters like what appears in everyday text.
- A character array should be assigned enough number of characters like arrays of any other type:
- Example declarations
  - `char institute_name [20];`
  - `char xyz[] = “something”;`
  - `char pqr[20] = “something”;`
  - `char *name = “Ramanujan”;`
- If you wanted to store “IITB” into a character array (say `A`), then `A` must have a size of at least 5 - four for the characters of the string, and the fifth place for the `NULL` character `‘\0’` (ASCII value 0).
- Every valid string must end in the `NULL` character. The `NULL` character unambiguously marks the end of a character string.
- The `NULL` character is not printable.

String input and output operations

- A string can be printed on the screen using `cout`, for example from the previous slide: `cout << A; cout << “hello world”;`
- While printing a string, all characters from the zeroth index are printed one by one (including spaces) until the NULL character is encountered.
- A `char*` can be used to point to a string. For example:

```
char institute_name[] = “IITB”; char *xyz =
institute_name;
```

String input and output operations

- To read from the keyboard into a string, we use cin.
- For example: `cin >> institute_name;` will cause the input from the keyboard to be entered into `institute_name`, until the user hits enter, but there are some caveats.
- An initial set of spaces are ignored, i.e. not entered into `institute_name`.
- If you enter “IIT Bombay”, everything after the space is ignored and only “IIT” is stored in institute_name followed by `‘\0’` (NULL). This method is not useful to take in strings with spaces.
- This method is potentially dangerous, if the number of characters entered by the user exceeds the size of the array.

String input and output operations

- To deal with spaces and the unsafe situation regarding array size, the following function should be used:

```
char x[100]; cin.getline(x,n); // n = number of
characters to be entered by a user
```

- `getline` allows all characters entered by the user, including whitespaces, to be entered into `x`, until one of the following occurs:
  - The user hits enter (‘\n’)
  - The user enters n-1 characters without a newline. In this case, the n-th character is set to ‘\0’ and no further input is taken. It remains the responsibility of the programmer to ensure that n is not greater than the size of the character array.

String operations: string length

- The length of a string is the number of characters excluding the ‘\0’ character.
- A function which accomplishes this:

```
int string_length (char *x)
{
    int L = 0;
    while (x[L] != ‘\0’) L++;
    return L;
}
```

Comment: Note that we did not need to pass the length of `x` to this function. There is a potential infinite loop if x did not end with ‘\0’.

String operations: string copy

- The aim here is to copy the contents of one string into the other, including the NULL character.

```
void string_copy (char *dest, const char* source) {
int i;
for(i=0;source[i]!= '\0';i++){
    dest[i] = source[i];
}
dest[i] = source[i];
}
```

Comment: Note that we did not need to pass the length of `source` to this function. There is a potential infinite loop if `source` did not end with ‘\0’. You can also use 0 instead of ‘\0’.

String operations: concatenation

- Concatenation is the process of appending the characters of one string to the other. The latter string becomes longer as a result.
- For example when you concatenate “IIT” and “Bombay”, you get “IITBombay”.

```
void string_concat (char *first, char *second) {

    int i = j = 0;

    while (first[i] != 0) i++; // at the end of this loop, i = length of first

    while (second[j] != 0) {

        first[i++] = second[j++]; // copying the characters of second into first

    }

    first[i] = 0; // put a null character

}
```

String operations: String comparison

- You know that a dictionary organizes strings in alphabetical order.
- Given two strings, suppose you want to determine which one of them comes before the other in alphabetical order.
- This is called string comparison:
  - If both strings are equal, our string compare function should return 0
  - If the first string comes before the other, the function should return -1
  - Otherwise the function should return +1

```
int string_compare (char *first, char * second){

int i = 0;

while (true){

      if (first[i] == 0 && second[i] == 0) return 0;

      if (first[i] == 0) return -1;

      if (second[i] == 0) return +1;

      if (first[i] < second[i]) return -1;

      if (first[i] > second[i]) return +1;

      i++;

      }

      return 0;

}
```

This function does not require string lengths to be specified.
But this function will treat the strings “abc” and “Abc” to be unequal.
What changes will you make to this function to ignore case differences, i.e. treat “abc”, “ABC”, “Abc”, “abC” as all equal?

What does the following function do? Exercise in pointers!

```
void something_string (char *s, char *t){
    while (*s++ = *t++);
    // note this contains = and not == deliberately
}
```

String operations: String reverse

- Write a function to create a copy of a string with contents reversed (the NULL character must remain at the end of the reversed string).
- For example, the reverse of “Pune” is “enuP”.
- If a string and its reverse are the same, then the string is called a **palindrome** (eg: malayalam, redivider, talat, 10001)

```
void string_reverse(char* s, char* rev_s){

    int n = string_length(s);

    for(int i=0;i<n;i++) rev_s[n-1-i] = s[i];

    rev_s[n] = 0; // what will happen if you did not include this statement?

}
```

This function requires the length of string `s` to be computed. It fills in the reversed string inside `rev_s`.The programmer needs to ensure that adequate memory is contained in `rev_s`. **Can you modify this routine to reverse the contents of s in-place without requiring a separate array? That is called in-place string reversal.**

String operations: String reverse (in place)

```
void string_reverse_in_place (char* s){
int n = string_length(s);
for(int i=0;i<n/2;i++) swap(&s[i],&s[n-1-i]);
s[n] = 0; // what will happen if you did not include this statement?
}
```

We call this function in the form: `string_reverse_in_place(s);`

In-built string functions

- The library `cstring` contains several in-built functions, a few of which we wrote from scratch in previous slides.
- Some of these functions are (you must write `#include <cstring>` to use them):
  - `void strcpy (char* s1, const char* s2)  // copying s2 into s1`
  - `int strlen (const char* s)  // string length`
  - `void strcat (char* s, const char* t)  // string concatenation`
  - `int strcmp (const char* s, const char* t)  // compare s with t and returns 0,-1,1 if s is respectively equal to, lexicographically less than, lexicographically greater than t`

String tokenizer

- A very interesting operation on strings: imagine you had a **sentence** given to you in the form of a character string.
- And you wanted to automatically **extract all words** ignoring spaces and punctuation marks.
- The characters to be ignored are called **delimiters**.
- The words, i.e. sub-strings of interest, are then called **tokens**.
- `cstring` has an in-built function called `strtok` which extracts tokens from a parent string, given a string of delimiters (i.e., punctuation marks and spaces in our present example, but could be *anything* else as per your application).
- An example program is on the next slide: using only spaces as delimiters.

```
#include <cstring>
#include <iostream>
using namespace std;
int main(){ // test-driver for the strtok() function:
char s[] = "We are eagerly awaiting Mood Indigo, and then New
Year’s Party";
char* p;
cout << "The string is: [" << s << "]\nIts tokens are:\n";
p = strtok(s, " "); // the delimiter is the space char.
while (p){
    cout << "\t[" << p << "]\n";
    p = strtok(NULL, " ");
}
cout << "Now the string is: [" << s << "]\n";
}
```

OUTPUT:

```
The string is: [We are eagerly awaiting
Mood Indigo, and then New Year’s Party]

Its tokens are:

        [We]

        [are]

        [eagerly]

        [awaiting]

        [Mood]

        [Indigo,]

        [and]

        [then]

        [New]

        [Year’s]

        [Party]

Now the string is: [We]
```

String tokenizer: explanation

- Multiple calls to `strtok` are generally needed for tokenization (assuming the original string contains more than one token).
- The very first call `p = strtok(s, " ");` sets the pointer to the very first token inside `s`, i.e. “We”, and sets the space character following “We” to ‘\0’.
- The second argument indicates that the delimiters contain a space character.
- Thus both `s` and `p` contain “We”.
- Every successive call has the form **`p = strtok(NULL, " ");`**
- The NULL indicates that subsequent calls to `strtok` should continue tokenizing from the location in `s` saved from the previous `strtok` call.
- You can change the delimiter in each subsequent call, though we haven’t done so.
- It passes the pointer `p` to the next non-blank character that follows the new `\0’, and each space character following it is set to `\0’;
- Note `strtok` changes the string (`s` here) that it tokenizes. If you needed the string later, you should keep a copy of it.

String tokenizer: applications

- In actual applications for **word extraction from sentences**, the delimiters will be spaces, tabs (`\t’), newlines (`\n’), punctuation marks (,.;:!?).
- Tokenizers are also used in **compilers**: the first step of a compiler is to read the code from the source code file and divide it into tokens.
- In such cases, the delimiters are usually spaces, tabs and newlines.
- In a second pass, special processing needs to be done to remove comments and classify the tokens as variable, literal, keyword, and so on.

Notion of (Auxiliary) Space Complexity

(Auxiliary) Space Complexity of an Algorithm

- This measures the amount of **extra** space required by the algorithm, NOT INCLUDING the basic input to the algorithm.
- It is also measured using the O notation.
- For example, `string_reverse` requires another array of `n` elements, apart from scalar variables `n`, `i`.
- On the other hand, `string_reverse_in_place` requires only scalar variables `n`, `i` (no other array is required).
- Therefore the space complexity of `string_reverse` is `O(n)`, whereas that of `string_reverse_in_place` is `O(1)`.
- The time complexity of both algorithms is `O(n)`.

Algorithm	Time complexity	Space complexity
Selection sort	`O(n`^{2}`)`	`O(1)`
Bubble sort	`O(n`^{2}`)`	`O(1)`
Linear search	`O(n)`	`O(1)`
Binary search	`O(log n)`	`O(1)` for non-recursive, `O(log n)` for recursive
`string_length`	`O(n)`	`O(1)`
`string_compare`	`O(min(n1,n2))`	`O(1)`
`string_reverse`	`O(n)`	`O(1)` for in-place version, otherwise `O(n)`

Convince yourself that these are correct!

Merge Sort

Merge Sort

- Selection and bubble sort take $O(n^2)$ time. Likewise, insertion sort, which we will study later, also takes $O(n^2)$ time.
- Merge-sort takes only $O(n \log n)$ time, which is much faster.
- In fact, merge-sort is the fastest possible sorting algorithm based on comparisons.
- It has been proved that no comparison based sorting algorithm can run faster than $O(n \log n)$ time. The proof is beyond the scope of this course.

Merge sort

- In its most widely used form, merge-sort is recursive (though it can be implemented non-recursively).
- To sort an array A, we divide it into two parts B1 and B2 which have nearly equal size.
- We sort B1 and B2 individually, and then merge them in such a way that the resultant array is sorted.
- The division of A is easy; the merge step is the crux, and we will see how to do this.

```
void mergesort (int *A, int n){

     if (n > 1){

          int B1[n/2], B2[n-n/2];

          for (int i=0;i<n/2;i++) { B1[i] = A[i]; }

          for (int i=n/2;i<n;i++) { B2[i-n/2] = A[i]; }

          mergesort(B1,n/2); mergesort (B2,n-n/2);

          merge(A,B1,B2,n/2,n-n/2);

     }

}
```

Merge-sort demo example

https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif

One more example: https://www.geeksforgeeks.org/merge-sort/

Merge routine

```
void merge(int *A, int *B1, int *B2, int n1, int n2){

int cB1, cB2, cA; cB1 = cB2 = cA = 0;// counters for B1,B2,A respectively

while (cB1 < n1 && cB2 < n2){

     // copy an element from B1 to A as B1 has the smaller element

     if (B1[cB1] < B2[cB2]) A[cA++] = B1[cB1++];

     // copy an element from B2 to A as B2 has the smaller element

     else if (B1[cB1] > B2[cB2]) A[cA++] = B2[cB2++];

     // both have equal-valued elements, copy from both to A

     else { A[cA++] = B1[cB1++]; A[cA++] = B2[cB2++];}

}

// at most one of the following two while loops will be entered (why?)

// copy the remaining elements from  either B1 or B2 to A

while(cB1 < n1) A[cA++] = B1[cB1++];

while(cB2 < n2) A[cA++] = B2[cB2++];

}
```

Merge routine

- It maintains three counters: one each for arrays `B1`, `B2` and `A` (`cB1`, `cB2`, `cA`) respectively.
- If `B1` has the smaller element at its current counter, copy it into `A` and increment `cB1`. Increment `cA`.
- Otherwise if `B2` has the smaller element at its current counter, copy it into `A` and increment `cB2`. Increment `cA`.
- Otherwise if both have `B1` and `B2` have equal values at their counter, copy both into `A`. Increase `cA` by 2. Increment `cB1` and `cB2`.
- Keeping doing these steps until either `cB1` equals `n1` or `cB2` equals `n2`.
- If `cB1` equals `n1`, copy remaining elements of `B2` into `A`; otherwise if `cB2` equals `n2`, then copy remaining elements of `B1` into `A`.

Merge Sort: Time and space complexity

- Time complexity of merge operation = O(n1+n2) = O(n).
- Let T(n) = time taken to sort n numbers.
- Then due to the recursion, T(n) = T(n/2) + T(n-n/2) + time to merge = T(n/2) + T(n-n/2)+O(n) = 2T(n/2) + O(n) assuming n is even.
- Carrying out this recursive relationship further
  - $T(n) = 2T(n/2) + O(n) = 2(2T(n/4) + O(n/2)) + O(n)$
  - $= 2^2\, T(n/2^2) + 2O(n)$
  - $=2^k\, T(n/2^k) + k\, O(n)$ – upon carrying the recursive relation further for k steps
  - $= n\, T(1) + \log n\, O(n)$ where $k = \log(n)$
  - $= n + O(n \log n)$ since T(1) is a constant $= O(n \log n)$
- Space complexity (auxiliary): O(n) due to the extra space needed for arrays `B1` and `B2` at each level of recursion, for a total of `k = log n` levels. Space needed = n + n/2 + n/4 + … + n/2^k.

Quick Sort

Quick sort

- Also an efficient sorting algorithm which typically requires O(n log n) time.
- Does not require the extra space of merge-sort.
- Let A be an n-element array to sort.
- Let r be any element of A. We will re-arrange A such that all elements less than r are to the left of r, and all elements greater than r are to its right.
- The element is called the **pivot** or **splitter**.
- At this point, r is the only element that is necessarily in the right place.
- Continue this procedure recursively on sub-arrays A[1:r-1] and A[r+1:n] (*this notation is for illustration only, it is not valid C++ syntax*).

```
void quicksort (int *A, int low, int high) {

    if (high - low < 1) return;  // base case: nothing to sort in an array of 
1 element

    int pivotindex;

    pivotindex = partition(A,low,high);  // partition the array across the 
pivot

    quicksort(A,low,pivotindex-1);  // quicksort on left part (low to pivot-1)

    quicksort(A,pivotindex+1,high);  // quicksort on right part (pivot+1 to 
high)

}
```

- The partition function rearranges the array such that all elements less than the pivot are to the left of the array, followed by the pivot, followed by elements that are larger than the pivot.
- The pivot is taken to be the last element: see next slide, though other choices will also work well.

```
int partition (int *A, int low, int high)
{
    int i,j;
    int pivot = A[high]; // last element is the pivot

    i = low-1;
    for(j=low;j<=high-1;j++)
    {
        if (A[j] < pivot)
        {
            i++;
            swap(&A[i],&A[j]);
        }
    }
    swap(&A[high],&A[i+1]);
    return i+1;
}
```

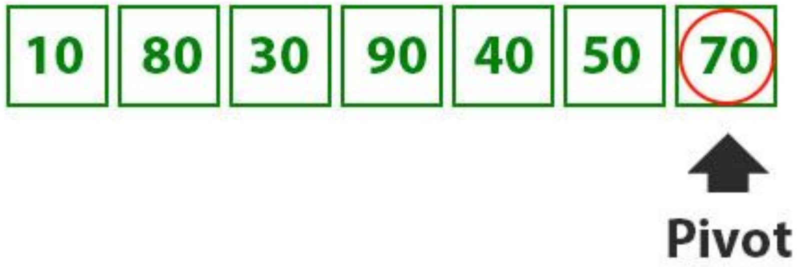
- This method maintains index `i` as it scans through the array using another index `j` such that all elements at indices from `low` to `i-1` (both inclusive) are less than the pivot, and all elements at indices from `i` to `high-1` (both inclusive) are greater than or equal to the pivot.
- This method is called Lomuto's partition scheme and is commonly taught to students. There exist better partitioning methods that we will not go into (eg: Hoare's method - see the wiki article on quicksort)

Source of image: geeksforgeeks

Partition

Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}

- *Indexes: 0 1 2 3 4 5 6*
- *low = 0, high = 6, pivot = arr[h] = 70*
- *Initialize index of smaller element, **i = -1***



Counter variables

I: Index of smaller element

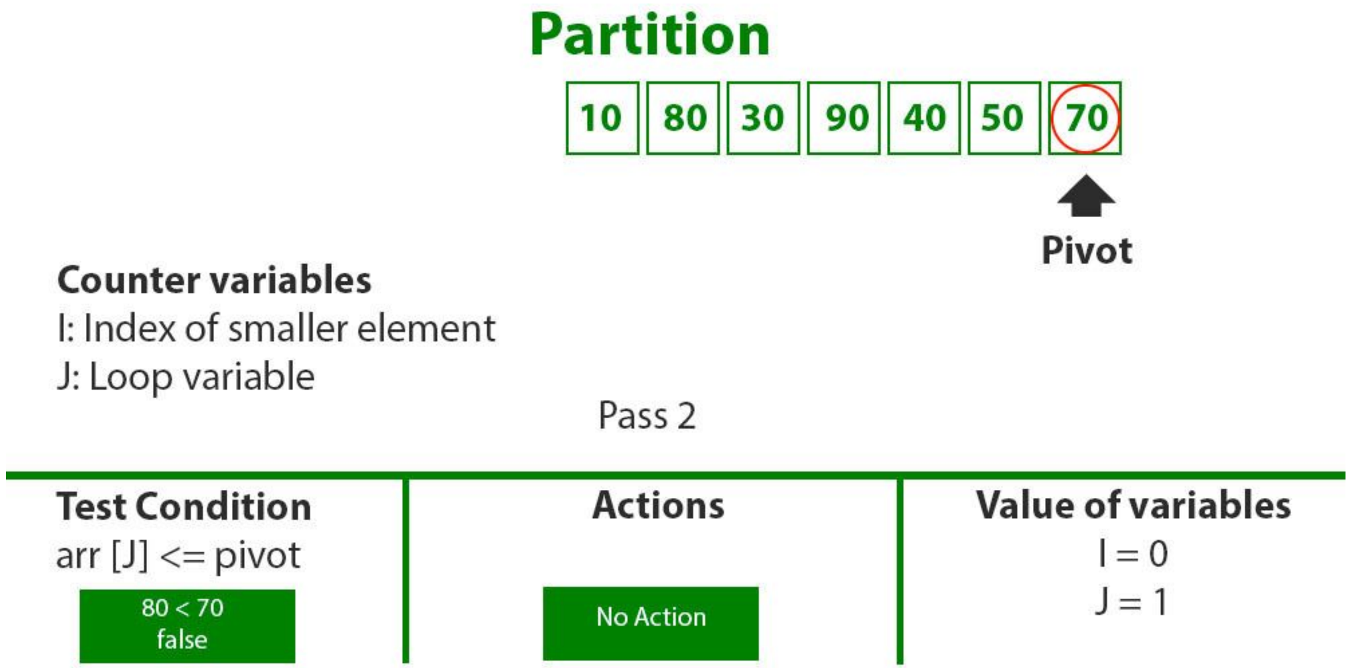
J: Loop variable

We start the loop with initial values

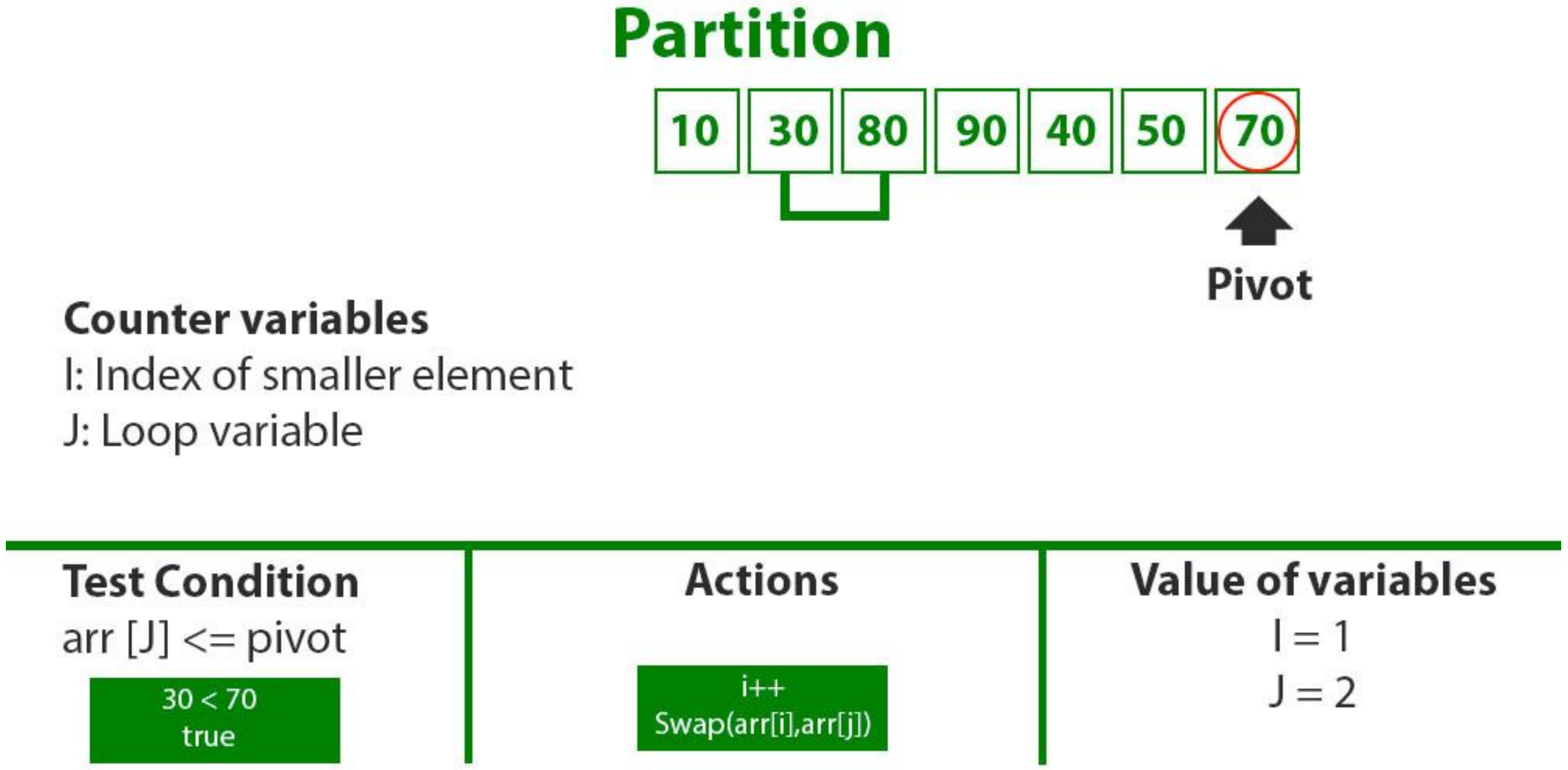
Test Condition	**Actions**	**Value of variables**

arr [J] <= pivot		I = -1
		J = 0

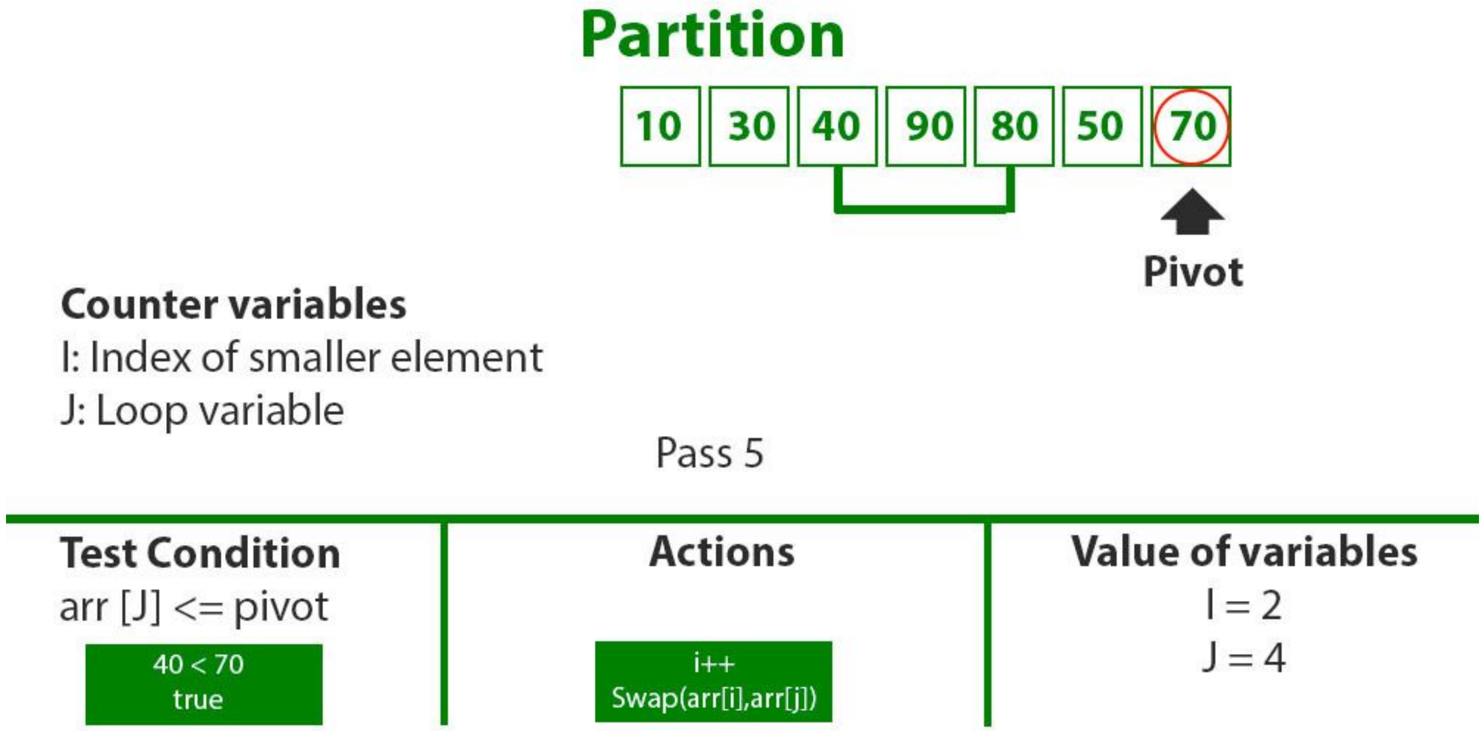
- *Traverse elements from j = low to high-1*
    - ***j = 0**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
    - ***i = 0***
- *arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same*
- ***j = 1**: Since arr[j] > pivot, do nothing*



- ***j = 2*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i = 1***
- *arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30*



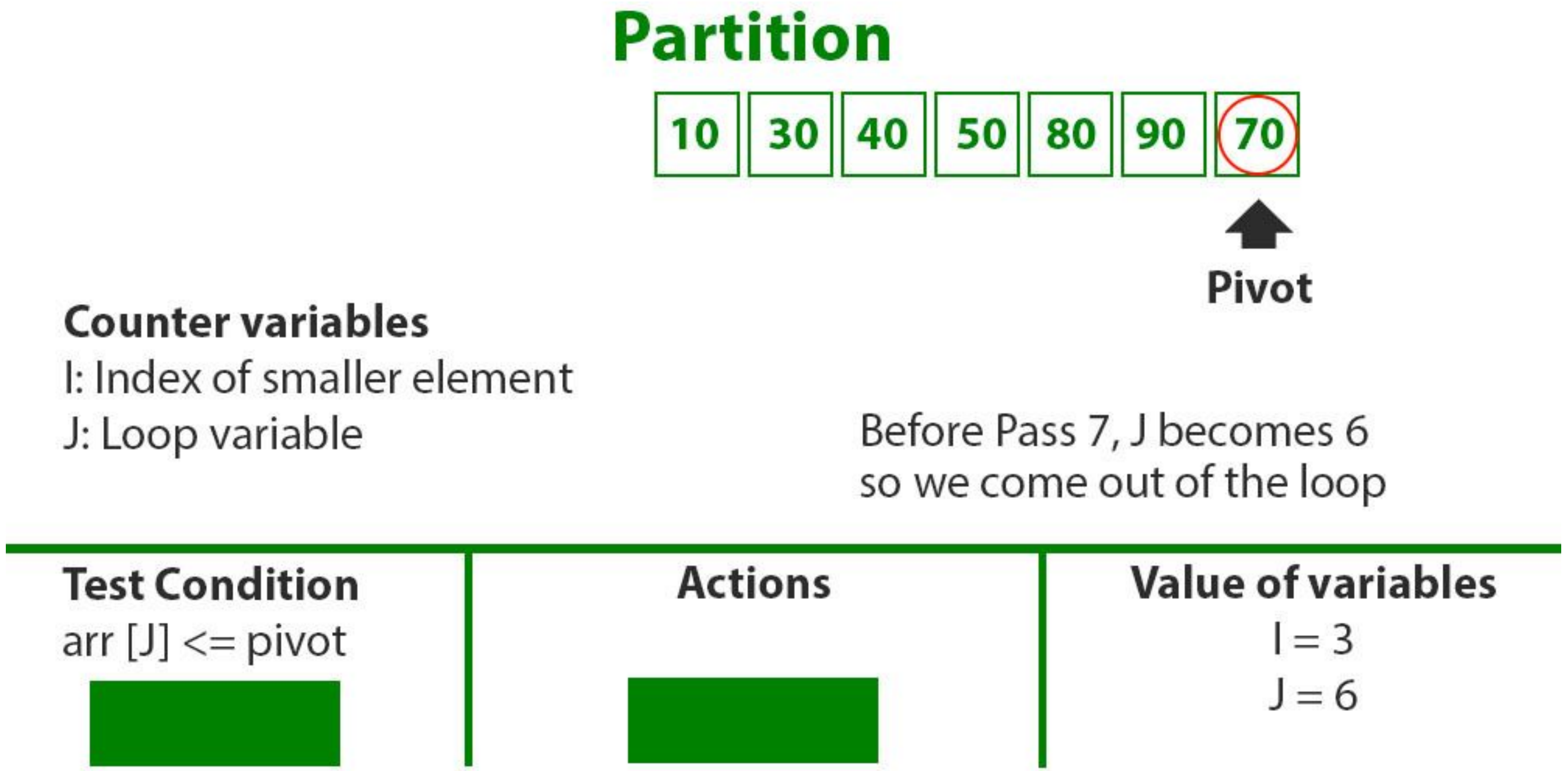
- ***j* = 3** : *Since arr[j] > pivot, do nothing // No change in i and arr[]*
- ***j* = 4** : *Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i* = 2**
- *arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped*



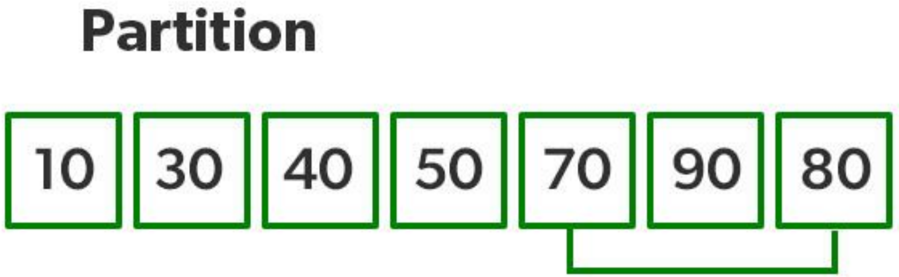
Test Condition	Actions	Value of variables

arr [J] <= pivot
40 < 70
true
	i++
	Swap(arr[i],arr[j])
		I = 2
		J = 4

- ***j = 5*** *: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]*
- ***i = 3***
- *arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped*



- *We come out of loop because j is now equal to high-1.*
- ***Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)***
- *arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped*



Counter Variable

I : Index of smaller element
J : Loop variable

We know swap arr[i+1] and pivot

I = 3

- *Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.*
- *Since quick sort is a recursive function, we call the partition function again at left and right partitions*

How long does quick sort take?

- Quick-sort partitions the array of n elements into two parts - one less than the pivot and the other after the pivot.
- For simplicity, we will consider the latter half to be the pivot and the elements after the pivot.
- Let the pivot be at index p, then $T(n) = T(p) + T(n-p) + O(n)$ since partitioning takes O(n) time.
- Let us assume that the pivot always falls in the centre of the array after partitioning.
- Then $p = n/2$. Then $T(n) = 2T(n/2) + n$.
- Continuing further, $T(n) = 2(2T(n/4) + n/2) + n = 2^2T(n/2^2) + 2n = 2^kT(n/2^k) + kn$.
- This goes on till $2^k = n$, i.e. $k = \log2(n)$, and then $T(n) = n\,T(1) + n \log2(n) = O(n \log n)$ since T(1) is a constant.
- Space: O(log n) due to recursion.

How long does quick sort take?

- However if the pivot fell to always the last element of the array or subarray after rearranging, then the picture is different.
- Then `T(n) = T(1) + T(n-1) + O(n) = T(1) + (T(1) + T(n-2)) + n-1 + n`

  `= 2T(1) + T(n-2) + n-1+n`

  `= 3T(1) + T(n-3) + n-2+n-1+n`

  `=n + T(1) + 1 + 2 + 3 + … + n-1 + n`

  $= O(n^2)$
- In such a case, quicksort has a space complexity of `O(n)`.

How long does quick sort take?

- Luckily, we can prevent such a thing from happening by choosing the pivot to be the median of the unsorted array (though the Lomuto's partition scheme does not work with this strategy).
- There exists a O(n) time algorithm for computing the median of an unsorted array.
- That algorithm is complicated and beyond the scope of this course.
- The advantage of quick sort over merge sort is that it does not need another array to store elements temporarily. So it has a better space complexity.

Multi-dimensional Arrays

Multi-dimensional arrays

- For storing objects like matrices, the arrays studied so far are inadequate.
- We need two-dimensional arrays, which are declared in the following manner:

  ```
  float arr[n1][n2];
  ```

  which is an array of `n1*n2` floats

- These floats are accessed in the form `arr[i][j]` where `i` lies from 0 to `n1-1` (both inclusive) and `j` lies from 0 to `n2-1` (both inclusive).
- `n1` and `n2` are said to be the numbers of rows and columns of `arr` respectively.
- `i` = row index, `j` = column index

Multi-dimensional array declaration

Examples:

```
double a[3][2] = {{1,2},{3,4},{5,6}};

double c[2][4] = {{1,2,3,4},{5,6,7,8}};
```

Passing two-D arrays to a function

- 2D Arrays can be passed to a function in the following manner. Notice that the second dimension of the array must be given to the function

```
void matrix_add (int A[][100], int B[][100], int C[][100], int
n1, int n2){

    for (int i=0;i<n1;i++){

        for(int j=0;j<n2;j++){

            C[i][j] = A[i][j] + B[i][j];

        }

    }

}
```

Time complexity: O(n1 x n2)
Space complexity: O(n1 x n2)

This function adds two matrices and creates a third matrix. Just like with 1D arrays, 2D (or higher-D) arrays are always **passed by reference, not by value**.

Passing 2D arrays to a function

- We will now write a function to multiply two matrices of size `nr1 x nc1` (matrix `A`) and `nr2 x nc2` (matrix `B`) respectively to create a third matrix of size `nr1 x nc2` (matrix `C`).
- The routine should check that the number of columns of matrix 1 = number of rows of matrix 2, failing which an error message should be printed and the routine should just return.
- We have `C[i][j]` = dot product of i-th row of `A` and j-th column of `B`

  = summation over `k` of `A[i][k]*B[k][j]`

- We will now write a routine for this.

```
void matrix_multiply (int A[][MAX], int B[][MAX], int C[][MAX], int
nr1, int nc1, int nr2, int nc2){

    if (nc1 != nr2) { cout << "dimension mismatch"; return;}

    int i,j,k;

    for(i=0;i<nr1;i++) {

        for(j=0;j<nc2;j++){

            C[i][j] = 0;

            for(k=0;k<nc1;k++){

            C[i][j] += A[i][k]*B[k][j]; // dot product of i-th row of
A and j-th column of B

            } // close innermost for loop (on k)

        } // close middle for loop (on j)

    } // close outer for loop (on i)

}
```

Time complexity: O(nr1 x nc2 x nc1)
Space complexity: O(nr1 x nc2)

You can also use int A[MAX][MAX] as the parameter instead of int A[][MAX]

Input and Output of 2D arrays

- We write a function to obtain the entries of a 2D array from the user via keyboard:

```
void get2Darray (int A[][MAX], int n1, int n2){
    int i,j;
    for(i=0;i<n1;i++){
        for(j=0;j<n2;j++){
            cout << “enter the value of A at row: “ << i << “, 
column: ” << j << endl;
            cin >> A[i][j];
        }
    }
}
```

Input and Output of 2D arrays

- We now write a function to print the entries of a 2D array on the screen, each row on a new line:

```
void print2Darray (int A[][MAX], int n1, int n2){
    int i,j;
    for(i=0;i<n1;i++){
        for(j=0;j<n2;j++){
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
}
```

Input and Output of 2D arrays

```
int main() {
    int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
    int n1, n2;
    cin >> n1 >> n2 >> n3;
    get2Darray(A,n1,n2);
    get2Darray(B,n2,n3);
    matrix_multiply(A,B,C,n1,n2,n2,n3);
}
```

Use of 2D Arrays

- Storage of **image** matrices in image processing or graphics: an image acquired by a camera is inherently a 2D array
- All kinds of **matrix** operations: matrix addition, subtraction, multiplication, determinant, inverse, transpose, etc.
- An array of **characters strings** is essentially a 2D character array, each row of which is a string of characters.
- Example: `char xyz[10][20];`
- These strings are accessed in the form `xyz[0]`, `xyz[1]` and so on.
- The next slide has a sample program to read in multiple character strings and print them on the screen.

```
int main(){

char name[5][20];

int count=0;

cout << "Enter at most 5 names with at most 19 characters
each:\n";

while (cin.getline(name[count++], 20)); // terminated by ctrl-Z

cout << "The names are:\n";

for (int i=0; i<count; i++)

cout << "\t" << i << ". [" << name[i] << "]" << endl;

}
```

Output:

```
Enter at most 4 names with at most 19 characters each:
George Washington
John Adams
Thomas Jefferson
^Z (or ^D — control Z or control D, which signals the end of input)
The names are:
0. [George Washington]
1. [John Adams]
2. [Thomas Jefferson]
```

3D Arrays

- These are used to store video data, which has x, y and time as well.
- They are also used to store structures called “tensors” in linear algebra.
- Example declaration:
  - `int a[n1][n2][n3];`
  - Element access is of the form `a[i][j][k]` where `i` lies from 0 to `n1-1` (both inclusive), `j` lies from 0 to `n2-1` (both inclusive) and `k` lies from 0 to `n3-1` (inclusive)

Dynamic Memory Allocation

Dynamic Memory Allocation

- A declaration of the form `int *p;` allocates memory to store an address (of an integer datatype) inside p.
- But the memory at that address is not necessarily allocated to the program under consideration.
- Such a pointer is called a **dangling pointer**.
- Attempting to change the contents of that memory location is dangerous, i.e. a statement of the form `*p = 3;` is dangerous
- One way out is as follows: `int q=10; int *p; p = &q;`
- Another way out is to explicitly allocate memory for the pointer itself in the following manner:

```
int *p = new int; // allocates memory for a single integer
pointed to by p

*p = 3; // this is fine, as p now has a proper memory location
```

Dynamic Memory Allocation

- The `new` operator returns the address of `s` bytes of memory where `s` is the size (in bytes) of the datatype (`int` in this example, but could be easily replaced by `float` or `double` or any other datatype).
- The value inside the location can also be initialized right away at the time of memory allocation using syntax of the following form:
  - `int *p = new int (3);`
- In rare cases, there may be insufficient memory available in the system for various reasons. In such cases, `p` will get a NULL address.
- For robustness of the program, it is best to check that `p` is NOT NULL before proceeding to use it.

```
int *p = new int (3);

if (p == NULL) { cout << “Insufficient memory available”; exit(0);}

*p = 4;
```

Dynamic Memory Allocation of Arrays

- The `new` operator can also be used to allocate an array of any datatype, using syntax of the following form:
  - `int *p = new int [20];` OR `int *p; p = new int[20];`
- This is called dynamic memory allocation, and it creates an array `p` of 20 integers at run time (note the square brackets), and the memory is allocated only when this particular statement is executed.
- Note that the original address (if any) inside `p` is **over-written** by the `new` operator.
- On the other hand, a statement of the form `int q[20];` allocates an array of 20 integers at the time of compilation, and is called static memory allocation.
- Dynamic memory allocation is very useful if the size of the array depends on the input size.

Memory de-allocation

- The `delete` operator reverses the action of `new`.
- That is `delete` returns the memory allocated previously via the `new` operator back to the operating system.
- `delete` should be applied only to pointers which have been allocated memory via `new`, and it is the programmer’s responsibility to adhere to this rule.
- Example:

```
float *q = new float (3.14);

delete q;

*q = 2.0; // this is dangerous
```

Memory de-allocation

- The `delete` operator can also be applied to arrays in the following manner:

```
float *p = new float [20];

delete [] p;
```

- The `[]` indicates that `p` was an entire array of floats.
- Note: the following is dangerous:

```
float x = 3.0;

float *y = &x;

delete y; // dangerous as y was not allocated memory via new
```

The `get()` function here creates a dynamic array:

```
void myget(double*& a, int& n) {
     cout << "Enter number of items: "; cin >> n;
     a = new double[n]; // array of n doubles, allocated dynamically
     cout << "Enter " << n << " items, one per line:\n";
     for (int i = 0; i < n; i++) { cout << "\t" << i+1 << ": ";
     cin >> a[i];
     }
}

void print(double* a, int n) {
     for (int i = 0; i < n; i++)
     cout << a[i] << " ";
     cout << endl;
}

int main() {
     double *a; int n;
     myget (a,n); // create an array of n doubles
     print (a,n); // print the array
     delete [] a; // deallocate memory
}
```

Note: in the `myget` function, the pointer `a` is passed **by reference**! This is needed because the address in `a` (that is the value of `a`) is changed inside `myget` due to the `new` operator. We want this change to reflect in the calling function, so we need to pass `a` by reference! Another way to accomplish this is to just rewrite `myget` as `double* myget (int&n);`

Dynamic Memory Allocation of 2D arrays

- We have seen DMA of 1D arrays.
- We need to work harder for DMA of 2D arrays.
- There are two different methods for this: consider DMA of a 2D array of size `n1` x `n2` integers.
- For this, you first create a `int**` variable called `A`, and then allocate a 1D array of `n1` pointers to integer. Then, to each of these pointers, you allocate a 1D array of `n2` integers each.
- This variable `A` is now your 2D array.
- During deallocation, you first deallocate all the `n1` different integer arrays (each containing `n2` integers) and then deallocate the 1D array of pointers to integer.
- Upon deallocation, `A` will now be a dangling pointer.
- **Side note:** an `int**` variable can contain the base address of a 2D array of integers.

```
int **allocate_2d (int n1, int n2){
    int **A = new int*[n1]; // array of n1 int* variables
    for(int i=0;i<n1;i++){
        A[i] =  new int [n2]; // array of n2 int variables
    }
    return A;
}
void deallocate_2d (int **A, int n1, int n2){
for(int i=0;i<n1;i++) delete [] A[i];
delete [] A;
}
int main(){
    int **A = allocate_2d(4,5); deallocate_2d(A,4,5);
}
```

Array of pointers

- In the previous slide, A is an array of pointers to integers, i.e. it is an array whose each element is a pointer to another variable of some data-type (say T).
- The array of pointers on the previous slide was dynamically allocated.
- However it can also be statically allocated via declarations such as `int *A[n1];`
- The elements of A can be allocated memory using the operator `new`.

Memory leaks

- Dynamic memory allocation allows for a lot of flexibility.
- However it can lead to **memory leaks**, i.e. a situation where memory is allocated in a loop dynamically and no memory is deallocated.
- This can burden the overall RAM of a computer system and hamper its performance.
- For example: the following function called multiple times from within a loop will cause a memory leak.

```
int* myfunction (int n){

     int *a = new int [n];

     for(int i=0;i<n;i++) a[i] = i*n;

     return a;

}

int main() {

     int n = 1000,*a;

     for(int i=0;i<n;i++) a = myfunction(n);

}
```

Memory leaks

- To avoid the memory leak, the following modification involving `delete` needs to be brought in:

```
int* myfunction (int n){
      int *a = new int [n];
      for(int i=0;i<n;i++) a[i] = i*n;
      return a;
}
int main() {
      int n = 1000,*a;
      for(int i=0;i<n;i++) { a = myfunction(n); delete [] a;}
}
```

Difference between static and dynamic memory allocation

- A declaration of the form int A[10]; is an example of static allocation. The memory for A is allocated at compile time.
- A statement of the form int* A = new int[10]; is an example of dynamic memory allocation. The memory for A is allocated during run time.
- In static allocation, the memory is allocated from the system stack. The memory is released after the function call is completed. No explicit deallocation is required.
- In dynamic memory allocation, the memory is not released automatically when the function call is completed, and needs to be explicitly deallocated using the delete operator.
- See example on next slide.

Difference between static and dynamic memory allocation

```
void myfunction (int n) {

    int A[n]; // or int A[100];
    // some code here
    return;
}
```

static

```
void myfunction2 (int n) {

    int *A = new int[n]; // or new int[100];
    // some code here
    delete [] A; // required expressly to avoid
memory leaks
    return
}
```

dynamic