Identify the lines that cause errors

A) Dereference of uninitialized or otherwise invalid pointer
B) Insufficient (or none) allocated storage for operation
C) Storage used after free
D) Allocation freed repeatedly
E) Free of unallocated or potentially storage
F) Return, directly or via argument, of pointer to local variable
G) Dereference of wrong type
H) Assignment of incompatible types
I) Program logic confuses pointer and referenced type
J) Incorrect use of pointer arithmetic
K) Array index out of bounds
Identify the lines that cause errors

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int low, high;
    int A[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
    free(A);
    return 0;
}
```

Identify the lines that cause errors

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int low, high;
    int A[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
    free(A);
    return 0;
}
```

Binary Search

- The idea of the binary search is that given a sorted array, one can efficiently search for a target in O(log n) time.

Now to algorithms

The binary search algorithm

<table>
<thead>
<tr>
<th>Low</th>
<th>mid</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If (A[mid] == target) done
If (A[mid] < target) search A[mid+1, high]
If (low > high) target not found

Only 10% of the programmers can correctly implement binary search

Idea of the insertion sort

- If (A[mid] < target) search A[mid+1, high]
- If (low > high) target not found
Insertion Sort example

Making the insertion sort efficient

Find the location to insert 35

How many bytes of memory was moved based on code logic?

Cost of moving memory

How efficient is insertion sort? (hint: count operations)

for (int i=n-1; i>0; i--)
A[i] = A[i-1];

• How many bytes of memory was moved based on code logic?

• What if we can copy the entire block at once. How would we do that.
• Computers perform bit shifting very efficiently

memcpy

memmove

Passing arguments to functions

#include <stdio.h>
#include <stdlib.h>

int main()
int A[] = {1,2,3,4,5,6};
printf("%d\n", sum(A, 4));
return 0;

Computers perform bit shifting very efficiently

memcpy

Great way to move things around in an array

Great way to move things around in an array
How arguments are passed to functions

- Arguments to functions are passed by value.
- That is a copy of the value of the variable is given to the function.
- If the copy is just a value, function cannot change the original variable.
- If the copy is an address of a variable, the function can change the value of the calling variable.
- Arrays are always passed by "reference". That is, the address of A is given to the function.

```c
#include <stdio.h>
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    printf("%d, ", a[0]);
    return 0;
}
```

Understanding **

- char** is an address of a variable of type char*
- char** reads
  - Pointer to a char*

- Recall : char* argv[]
  - Command line arguments are saved as an array of char*'s (or char**)

Passing an array of strings to a function

```c
/* This function inserts the word to array[index].
 * Must allocate memory to hold the word.
 */
int insert(char **array, char *word, int narg, int index)
{
    array[index] = word;
    return EXIT_SUCCESS;
}
```

Next lecture is on memory management

Go to recitation Wednesday

SL4 is optional, but very helpful

Quiz 2 will be available shortly