Searching

Objectives

- Call stack
- Recursion
- Analyze searching algorithms
  - Linear search
  - Binary search

Call Stack

Stacks

- A stack is an *ordered* collection of items
  - Items can be inserted at the top
    - Referred to as *pushing*
  - And removed from the top
    - Referred to as *popping*
- Stacks are last-in-first-out (LIFO)
  - Like a stack of weight plates
  - Or books

i.e. the order is retained, it does not mean that it is sorted

delicious but not the same sort of stack …

Function Calls

- Function calls are processed in LIFO order
  - When a function completes, control returns to the function that called it
    - Referred to as the *calling function*
- Function calls are characterized by four things
  - Parameters
  - Local variables
  - Return value
  - Return address
- These four things are maintained on a *call stack*
  - Each function call is pushed onto or popped from a *stack frame*

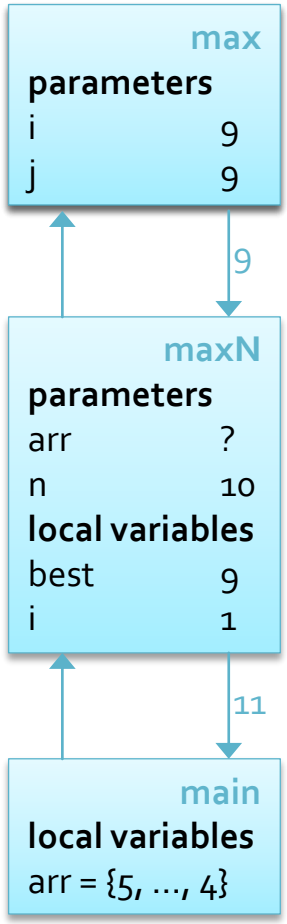
Functions Calling Functions

```
int max(int i, int j) {
   if (i < j) {
         i = j;
   }
   return i;
}

int maxN(int arr[], int n) {
   int highest = arr[0];
   for (int i = 1; i < n; i++) {
         highest = max(highest, arr[i]);
   }
   return highest;
}

int main () {
   int arr[10] = {5, 9, 4, 2, 3, 11, 4, 1, 0, 4};
   printf("Highest value: %d\n", maxN(arr, 10));
   return 0;
}
```

called repeatedly with each element in *arr* from 1 to *n*-1



More Repetition

- We've seen while loops and for loops – two methods of performing repetition
  - There is another way to repeat a process
    - That uses function calling instead of loops
- Consider computing factorials
  - Note that the factorial of 5 = 5 *4!
    - Let`s state this more generally
  - The factorial of *x* = *x* * (*x* - 1)!
  - And the factorial of 1 = 1

Recursive Factorial

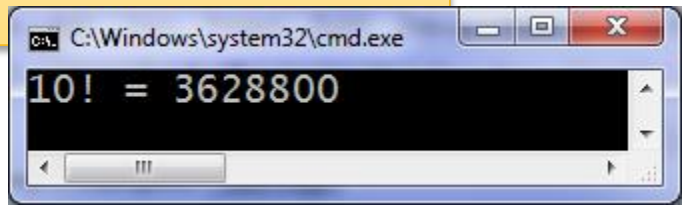
- Let's write a C function to compute factorials using the ideas presented previously
  - $x! = x * (x - 1)!$ and $1! = 1$

```
// PRE: x must be a +ve integer
// Function that returns the factorial of x
long long factorial(int x){
        if(x <= 1)
        {
                return 1;
        }else{
                return x * factorial(x-1);
        }
}
```

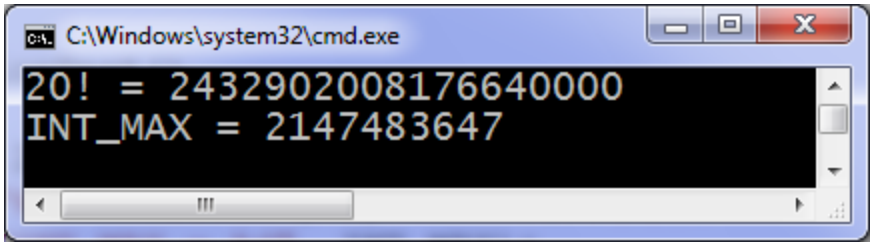
does this work?

Testing Factorial

```
void recursionTest()
{
        int x = 10;
        printf("%d! = %lld", x, fact4(x));
}
```



incidentally, in case you were wondering why factorial returned a *long long*, here is 20!



Recursive Functions

- The *factorial* function is *recursive*
  - A recursive function calls itself
  - Each call to a recursive function results in a *separate* call to the function, with its own input
- Recursive functions are just like other functions
  - The invocation is pushed onto the call stack
  - And removed from the call stack when the end of the function or a return statement is reached
  - Execution returns to the previous function call

Main Function

- *main* is also a function
  - Running a C program is the same thing as making a function call to main(…)
  - The command shell calls main and
  - The return value is sent back to the command shell
- Main can take arguments
  - `int main(int argc, char* argv[]) { … }`
  - *argv* is an array of strings of size *argc*
    - Any string arguments typed after the program invocation are stored in *argv*

Stack Variables

- Stack memory is sequential
- Stack memory is released when a function terminates
  - Pointers to local variables in a released function should not be returned
- Memory assigned to variables on the stack cannot grow or shrink
  - Since everything above them on the stack would have to be moved to make room for them
  - Use dynamic memory instead

Linear Search

Searching

- It is often useful to find out whether or not a list contains a particular item
  - What's Bob's phone number?
  - What grade did Kate get in assignment 1?
- Two possible specifications of return values
  - True or false
  - Or the position of the item in the list
    - -1 for failure

Input Organization

- The organization of the input can make a big difference to the efficiency of a search
- Is the input sorted?
  - Use binary search
- Is the data stored in a data structure that makes searching efficient?
  - Binary search tree
  - Hash table
- If none of the above use *linear search*

Linear Search

- Start with the first item
  - Iterate through the array one element at a time
  - Until a match is found
    - Return true or the index of the match
  - Or all elements have been checked
    - Return false or -1

Linear Search

```
int linearSearch(int arr[], int n, int target){
    repeat for i = 0 to n-1
        check the next element, arr[i]
        Algorithm:
            if equal to target return true or index

    target not found so return false or -1
}
```

Linear Search

```
int linearSearch(int arr[], int n, int target){
    for (int i=0; i < n; i++){
        if(arr[i] == target){
            return i;
        }
    } //for
    return -1; //target not found
}
```

The function returns as soon as the target item is found

return -1 to indicate that the target has not been found

Barometer Instruction

- Search an array of *n* items
- The barometer instruction is equality checking (or *comparisons* for short)
  - `arr[i] == target;`
  - There are actually two other barometer instructions
    - What are they?
- How many comparisons does linear search perform?

```
int linearSearch(int arr[], int n, int target){
     for (int i=0; i < n; i++){
          if(arr[i] == target){
               return i;
          }
     } //for
     return -1; //target not found
}
```

Linear Search Comparisons

- Best case
  - The target is the first element of the array
  - Make 1 comparison
- Worst case
  - The target is not in the array or
  - The target is at the last position in the array
  - Make *n* comparisons in either case
- Average case
  - Is it (*best case* + *worst case*) / 2, i.e. (*n* + 1) / 2?

Improving Linear Search

- Comparisons are relatively expensive elementary operations
  - Use a *sentinel* value to cut the number of comparisons in half
- The O notation running time is unchanged
  - Still O(*n*)
  - But the leading constant is halved

Improved Linear Search

```
int linearSearch(int arr[], int n, int target){
    arr[n] = target;
    int i = 0;
    while(arr[i] != target){
        i++;
    } //while
    if (i != n){
        return i;
    }
    return -1;
}
```

this is an error but is conceptually correct

Improved Linear Search Discussion

- Remember that leading constants don't matter for Big O comparisons
  - They don't matter when comparing two algorithms with different Big O running times
- But they do matter when two algorithms have the same Big O growth rate
  - Optimized vs. un-optimized algorithm
  - Fast vs. slow machine running the same algorithm

Searching Sorted Arrays

- If we sort the target array first we can make the linear search average cost around *n* / 2
  - Once a value equal to or greater than the target is found the search can end
    - So, if a sequence contains 8 items, on average, linear search compares 4 of them,
    - If a sequence contains 1,000,000 items, linear search compares 500,000 of them, etc.
- However, if the array is sorted, it is possible to do *much better* than this by using binary search

Binary Search

Divide and Conquer

- Searching can be performed much more efficiently if the array is sorted
  - For unsorted arrays we must use linear search
  - For sorted arrays we can use binary search
- Binary search is a *divide and conquer* algorithm
- Divide
  - Cut the array into 2 (or more) roughly equal sized pieces
- Conquer
  - Use what you know about the pieces to solve the problem

Binary Search

- Binary search examines the central element of the array
  - If this value is greater than the target then the target must be in the lower half of the array
  - If it is less than the target then the target must be in the upper half of the array
  - If it is equal to the target then return true
- Repeat the process with the central element of the candidate sub-array
  - Until the target is found or no candidates are left

Binary Search Notes

- Pre-condition
  - Array must be sorted
- It is necessary to keep track of which sub-array is to be searched
  - Use integer variables for indexes
  - Identify the candidate sub-array with *first* and *last* indexes
  - The midpoint is (*first* + *last*) / 2
    - Note that integer division deals with sub-arrays of even size

Binary Search Algorithm

```
int binSearch(int arr[], int n, int target){
    search sub-array arr[first … last]
    while not empty

        compare middle element to target
        Algorithm:
            return true if middle element equal to target
            exclude last half if target < middle element
            exclude first half if target > middle element

    no candidates so return false
}
```

Best Case

- In the best case the target is the midpoint element of the array
  - Requiring one iteration of the while loop

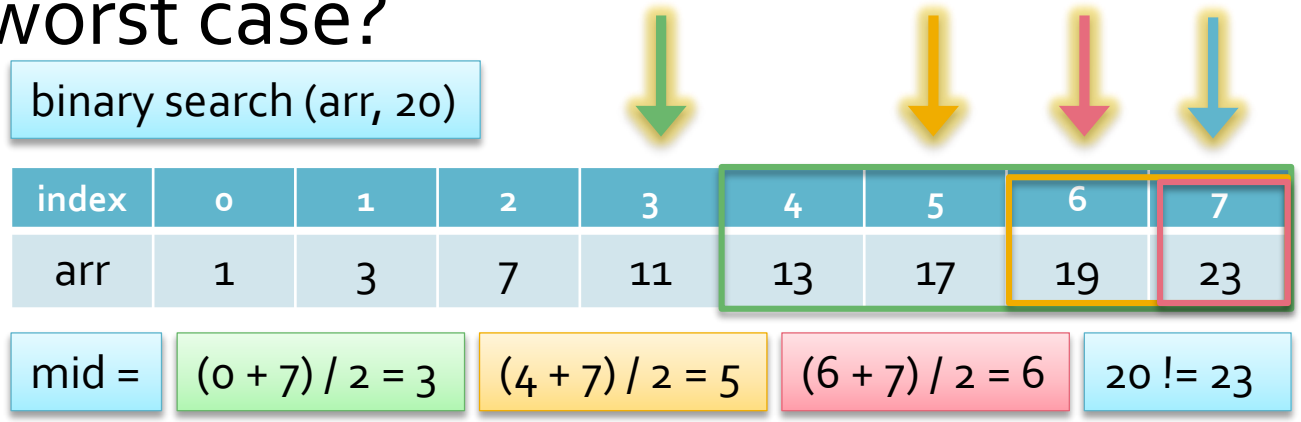
binary search (arr, 11)

index	0	1	2	3	4	5	6	7
arr	1	3	7	11	13	17	19	23

mid = (0 + 7) / 2 = 3

Worst Case

- What is the worst case for binary search?
  - Either the target is not in the array, or
  - It is found when the search space consists of one element
- How many times does the while loop iterate in the worst case?



Binary Search in C

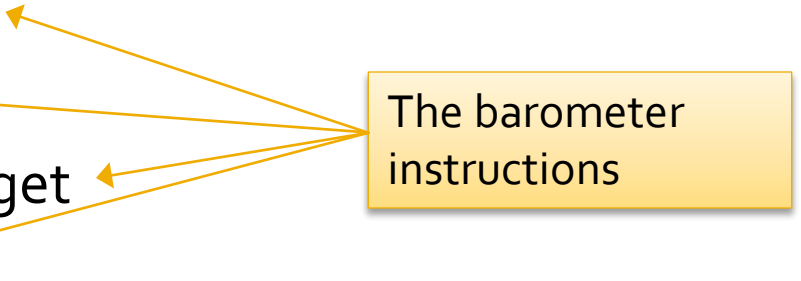
```
int binarySearch(int arr[], int n, int target){
    int first = 0;
    int last = n - 1;
    int mid = 0;
    while (first <= last){
        mid = (first + last) / 2;
        if(target == arr[mid]){
            return mid;
        } else if(target > arr[mid]){
            first = mid + 1;
        } else { //target < arr[mid]
            last = mid - 1;
        }
    } //while
    return -1; //target not found
}
```

Analyzing Binary Search

- The algorithm consists of three parts
  - Initialization (setting first and last)
  - While loop including a return statement on success
  - Return statement which executes when on failure
- Initialization and return on failure require the same amount of work regardless of input size
- The number of times that the while loop iterates depends on the size of the input

Binary Search Iteration

- The while loop contains an *if*, *else if*, *else* statement
- The first if condition is met when the target is found
  - And is therefore performed at most once each time the algorithm is run
- The algorithm usually performs 5 operations for each iteration of the while loop
  - Checking the while condition
  - Assignment to mid
  - Equality comparison with target
  - Inequality comparison
  - One other operation (setting either first or last)



Analyzing the Worst Case

- Each iteration of the while loop halves the search space
  - For simplicity assume that n is a power of 2
    - So $n = 2^k$ (e.g. if $n = 128$, $k = 7$)
- How large is the search space?
  - After the first iteration the search space is halved to $n/2$
  - After the second iteration the search space is $n/4$
  - After the k^{th} iteration the search space consists of just one element, since $n/2^k = n/n = 1$
    - Because $n = 2^k$, $k = \log_2 n$
  - Therefore at most $\log_2 n + 1$ iterations of the while loop are made in the worst case

O(log n)

Improving Binary Search

- Observation
  - The target value is only equal to an array element at most once
    - As the algorithm then returns true
  - This means that the first comparison in the if statement is usually false
    - Necessitating a second comparison to determine which sub-array search
- Solution
  - Re-order the if statement to do less work in the loop

Improved Binary Search

```
int binarySearch(int arr[], int n, int target){
    int first = 0;
    int last = size - 1;
    int mid = 0;
    while (first <= last){
        mid = (first + last) / 2;
        if(target > arr[mid]){
            first = mid + 1;
        } else if(target < arr[mid]){
            last = mid - 1;
        } else { //target == arr[mid]
            return mid;
        }
    } //while
    return -1; //target not found
}
```

Recursive Binary Search

```
int binarySearch(arr[], int n, int target){
    if(n <= 0){
        return 0;
    }
    int mid = n / 2;
    if(arr[mid] == target){
        return 1;
    } else if(target < arr[mid]){
        return binarySearch(arr, mid, target);
    } else { //target > arr[mid]
        return binarySearch(arr+mid+1, n-mid-1, target);
    }
}
```

Binary Search vs Linear Search

n	3 + 4n	$4 + 8\log_2(n)$
3	15	17
10	43	31
100	403	57
1,000	4,003	84
10,000	40,003	111
100,000	400,003	137
1,000,000	4,000,003	164
10,000,000	40,000,003	191

Linear Search vs Binary Search

- Binary search is much faster than linear search but
  - It is harder to code
  - The array has to be sorted
- Keeping an array sorted can be expensive
  - If there is a lot more searching than updating
    - Keep the array sorted (slow) and use binary search (fast)
  - If there is a lot more updating than searching
    - Don't sort the array (fast) and use linear search (slow)
  - Or … don't use an array