Examination 2 (also known as E2) is tomorrow at 6:30pm CST in Jesse Hall auditorium.

Remember to bring #2 pencils.

No pencil, no score!

Complexity Question

Suppose you are given a number N. What is the best complexity possible for determining if N is greater than some other given number M?

(1) O(1) (2) O(N) (3) O(M)

(4) O(N+M) (5) O(NM)

Complexity Question

Suppose you are given a number N. What is the best complexity possible for determining if N is greater than some other given number M?

(1) O(1) (2) O(N) (3) O(M)

(4) O(N+M) (5) O(NM)

Note that the time to check whether "N > M" doesn't depend on the actual values of M and N.

Complexity Question

Suppose you are required to read N lines of data from a file, and each line of data contains 72 alphabetic letters. What is the complexity to print all permutations of the 72 letters in each line?

(1) $O(N)$ (2) $O(N^2)$ (3) $O(N^{72})$ (4) $O(2^N)$ (5) $O(N!)$

Complexity Question

Suppose you are required to read N lines of data from a file, and each line of data contains 72 alphabetic letters. What is the complexity to print all permutations of the 72 letters in each line?

(1) O(N) (2) $O(N^2)$ (3) $O(N^{72})$ (4) $O(2^N)$ (5) O(N!)

Printing (72!)*N characters requires time proportional to N. Another way to think about it is to consider what happens if we double the size of N: the running time is doubled. That shows that the running time is proportional to N.

It may take a trillion trillion trillion trillion years to complete even for N=1, but it's still an O(N) algorithm.

Efficient Search & Recursion

Dataset Membership

Problem: Store a collection of values in a data structure so that a *membership* function can be efficiently supported.

Specifically, we want to implement a function that can determine whether a given value is or is not in the dataset without having to examine all or most of the elements of the dataset.

Dataset Membership - Simple Solution

```
/* Return true if query value is in the unordered array, else return false.
   "first" is the index of the beginning of the array.
   "last" is the number of elements in the array.
   "query" is the value to be located in the array.
   Complexity:  O(last-first)      */

boolean  member(int unorderedArray[ ], int first, int last, int query)
{
    int i;
    for (i = first; i <= last; i++) {
        if (unorderedArray[i] == query) {
            return true;
        }
    }
    return false;
}
```

Binary Search

The simple algorithm for determining membership assumes only that the data is stored in an unordered array, and it simply examines each element until it finds the query value or determines that it isn't in the dataset.

Binary search assumes that the data values are stored in an ordered array, e.g., sorted in ascending order.

The sorted-order data structure provides information that can be exploited to reduce the number of values that have to be examined in the process of satisfying a membership query.

Binary Search Algorithm

1. If the dataset is empty, return FALSE.

2. Examine the median value in the dataset:
   If it is less than the query value, then disregard
   all elements of the dataset that are less than
   the median from future consideration.

   Else if it is greater than the query value, then
   disregard all elements of the dataset that are
   greater than the median from future consideration.

   Else return TRUE - it equals the query value.

3. Repeat process starting at Step 1.

Membership - Binary Search

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    int mid;
    while (first <= last) {
        mid = (first+last)/2;
        if (query > orderedArray[mid]) {    // query value is in the 2nd half of array
            first = mid+1;
        }
        else if (query < orderedArray[mid]) {    // query value is in the 1st half of array
            last = mid-1;
        }
        else {          // query value is found
            return true;
        }
    }
    return false;
}
```

Dataset Membership - Binary Search

Find the number 5:

1	2	3	4	5	6	7	8	9	10	11	12	13	15	16

Dataset Membership - Binary Search

Find the number 5:

$5 < 8$

1	2	3	4	5	6	7	8	9	10	11	12	13	15	16

Dataset Membership - Binary Search

Find the number 5:

$5 < 8$

Dataset Membership - Binary Search

Find the number 5:

5 > 4

Dataset Membership - Binary Search

Find the number 5:

5 > 4

Dataset Membership - Binary Search

Find the number 5:

$5 < 6$

Dataset Membership - Binary Search

Find the number 5:

$5 < 6$

Dataset Membership - Binary Search

Find the number 5:

Number found.

Membership - Binary Search

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    int mid;
    while (first <= last) {
        mid = (first+last)/2;
        if (query > orderedArray[mid]) {    // query value is in the 2nd half of array
            first = mid+1;
        }
        else if (query < orderedArray[mid]) {    // query value is in the 1st half of array
            last = mid-1;
        }
        else {          // query value is found
            return true;
        }
    }
    return false;
}
```

Binary Search – Different (recursive) implementation

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    if  (first > last)  {
        return false;
    }
    int mid = (first+last)/2;
    if (query > orderedArray[mid]) {
            return member(orderedArray, mid+1, last, query);
    }
    else if (query < orderedArray[mid]) {
            return member(orderedArray, first, mid-1, query);
    }
    return true;
}
```

RECURSION

A recursive function/method is one that calls itself.

Recursion is just another type of program flow control like *if-then-else* statements and loops.

Here's the way to think about recursion: When you start designing a complicated program, you typically begin by writing a high-level routine with calls to other routines that haven't been written yet. Your assumption is that if those other routines are implemented properly, then your high-level routine will work properly. You can make *recursive* calls to your high-level routine itself in the same way.

The main thing to keep in mind is that each recursive call should reduce the problem so that at some point it can be solved without any further recursive calls.

Terminating Recursion

Recursive algorithms are nice because they reduce large problems to easier ones. Of course there has to be some limit to the number of recursive calls. This limit is called a **base case**.

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    if  (first > last)  {
         return false;
    }
    int mid = (first+last)/2;
    if (query > orderedArray[mid])
            return member(orderedArray, mid+1, last, query);
    else if (query < orderedArray[mid])
            return member(orderedArray, first, mid-1, query);
    return true;
}
```

Base case

Recursion Example

// Compute sum of elements in array of size n.

```
int  sumOfArray(int array[ ], int n)
{
     if  (n <= 0)  {    // Base case
          return  0;
     }
     return array[n-1]  + sumOfArray(array, n-1);
}
```

Recursion Example

```
// Reverse elements (first-last) of an array.
void  reverseArray(int array[ ], int first, int last)
{
    if  (first >= last)  {    // Base case
        return;
    }
    int temp = array[first];        // Swap first and
    array[first] = array[last];     // last elements
    array[last] = temp;
    // Recursively reverse middle of the array
    reverseArray(array, first+1, last-1);
}
```

Recursion Example

```
// Determine if query value is in the array.
boolean  member(int array[ ], int first, int last, int query)
{
    if  (first > last)  return false;
    if (array[first] == query) return true;
    return member(array, first+1, last, query);
}
```

Recursion Example

```
// Determine maximum value in a non-empty array.
int  findMax(int array[ ], int n)
{
    if  (n == 1)  return array[0];     // base case
    int temp = findMax(array, n-1);
    if (array[n-1] >= temp) return array[n-1];
    else return temp;
}
```

Recursion and Efficiency

A recursive call at the end of a function or subroutine is called *tail recursion*. Tail recursion can always be replaced with simple iteration (a loop).

Compilers should automatically replace tail-recursive calls with loops, so there usually isn't any loss in efficiency.

Tail-recursion can be used in place of a simple loop, but you should only do so if it makes the logic of the code clearer.

It is best to use recursion whenever it is natural to do so, and compare it with an iterative implementation if efficiency is a critical issue.

Tail Recursion Example

```
// Compute sum of elements in array of size n.
int  sumOfArray(int array[ ], int n)
{
     if  (n <= 0)  {    // Base case
          return  0;
     }
     return array[n-1]  + sumOfArray(array, n-1);
}
```

Tail-recursive call

Tail Recursion Example

```
// Reverse elements (first-last) of an array.
void  reverseArray(int array[ ], int first, int last)
{
    if  (first >= last)  {    // Base case
        return;
    }
    int temp = array[first];        // Swap first and
    array[first] = array[last];     // last elements
    array[last] = temp;
    // Recursively reverse middle of the array
    reverseArray(array, first+1, last-1);
}
```


Tail-recursive call

Tail Recursion Example

```
// Determine if query value is in the array.
boolean  member(int array[ ], int first, int last, int query)
{
    if  (first > last)  return false;
    if (array[first] == query) return true;
    return member(array, first+1, last, query);
}
```


Tail-recursive call

Tail Recursion Example

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    if  (first > last)  {
        return false;
    }
    int mid = (first+last)/2;
    if (orderedArray[mid] == query) return true;
    if (query > orderedArray[mid])
            return member(orderedArray, mid+1, last, query);
    else return member(orderedArray, first, mid-1, query);
}
```

Tail-recursive call -- only one of the two recursive calls gets executed

Tail Recursion Example

```
boolean  member(int orderedArray[ ], int first, int last, int query)
{
    if  (first > last)  return false;

    int mid = (first+last)/2;

    if (orderedArray[mid] == query) return true;

    if (query > orderedArray[mid])  first = mid + 1;
    else last = mid - 1;

    return member(orderedArray, first, last, query);
}
```


Tail-recursive call

(compare this to the earlier iterative versions of binary search)

How to Develop Recursive Algorithms

If you are developing a routine to perform a particular operation, then you can assume it exists and works correctly if you make recursive calls to it.

If the logic of your routine is correct with this assumption, then the assumption will also be correct. Yes, it's almost like magic.

```
// Compute sum of elements in array of size n.
int  sumOfArray(int array[ ], int n)
{
     if  (n <= 0)  {    // Base case
          return  0;
     }
     return array[n-1]  + sumOfArray(array, n-1);
}
```

How to Develop Recursive Algorithms

If you know what a recursively called routine is supposed to do, then assume that it does it correctly. There's no need to think about *how* it does it for the same reason there's no need to worry about how a library function works.

```
// Compute sum of elements in array of size n.
int  sumOfArray(int array[ ], int n)
{
     if  (n <= 0)  {    // Base case
          return  0;
     }
     return array[n-1]  + sumOfArray(array, n-1);
}
```

How to Develop Recursive Algorithms

Feeling compelled to think about (or work out on paper) how levels of recursive calls may or may not work is a sure indication that you aren't thinking about recursion in the right way.

```
// Compute sum of elements in array of size n.
int  sumOfArray(int array[ ], int n)
{
     if  (n <= 0)  {    // Base case
          return  0;
     }
     return array[n-1]  + sumOfArray(array, n-1);
}
```

Recursive Binary Search Algorithm

```
Boolean binarySearch(orderedArray, query)

  If orderedArray is empty, return FALSE

  Let mid = middle element of orderedArray

  If (query is less than mid)
      Return binarySearch(first half of orderedArray, query)

  Else If (query is greater than mid)
      Return binarySearch(last half of orderedArray, query)

  Else Return TRUE
```

Why Does Binary Search Take O(log N) Time?

Each comparison eliminates half of the elements in the dataset. How many times can a number N be divided in half before the result is less than 1?

Think about it from the opposite direction, how many times can a number be doubled, starting at 1, before the number is greater than or equals N?

$$2*2*2*...*2 >= N$$
$$2^k >= N$$
$$k >= \log_2(N)$$

So N can be divided in half k -- which equals $\log_2(N)$ -- times.

Why Binary Search is of Theoretical Importance

It turns out that any algorithm that uses only comparison operations to guide its search of elements in a dataset of size N must use at least $\log_2(N)$ comparisons for some queries, so no such search algorithm can have a worst-case complexity better than O(log N).

This implies that binary search is optimal in terms of worst-case complexity.

The proof that binary search is optimal means that there's no point trying to find a comparison-based algorithm with better complexity. Analyzing and developing provably-optimal algorithms is a major part of research in computer science.

What are the Implications of Optimality?

Suppose someone says that they have a search algorithm with worst-case complexity O(1). What can be concluded?

What are the Implications of Optimality?

Suppose someone says that they have a membership-search algorithm with worst-case complexity O(1). What can be concluded?

Either the algorithm performs its search using something other than comparison information, or it does not have O(1) worst-case complexity.

Optimality results are always based on various assumptions, so evading their implications requires techniques that do not require the assumptions.

Summary

- Complexity analysis is important for understanding how an algorithm will scale with its key parameters.
- Binary search can be shown to scale exponentially faster than naive linear search. In fact, its O(log(N)) complexity is almost indistinguishable from O(1) complexity until N gets very large.