Slides

1. Table of Contents
2. Definitions
3. Simple Recursion
4. Recursive Execution Trace
5. Recursion Attributes
6. Recursive Array Summation
7. Recursive Array Summation Trace
8. Coding Recursively
9. Recursive Design
10. Avoiding Pitfalls
11. Middle Decomposition
12. Call Tree Traces
13. Edges & Center Decomposition
14. Recursive Sorting
15. Backtracking
16. Knapsack Solution
17. Runtime Stack
18. Knap Runtime Trace Snapshot
19. Storage Organization
Definitions

Recursion

- see Recursion
- a process in which the result of each repetition is dependent upon the result of the next repetition.
- Simplifies program structure at a cost of function calls

Hofstadter's Law

- “It always takes longer than you expect, even when you take into account Hofstadter's Law.”

Sesquipedalian

- a person who uses words like sesquipedalian.

Yogi Berra

- “It's déjà vu all over again.”
A procedure or function which calls itself is a recursive routine.

Consider the following function, which computes \( N! = 1 \times 2 \times \ldots \times N \)

```c
int Factorial(int n) {
    int Product = 1,
        Scan    = 2;

    while ( Scan <= n ) {
        Product  =  Product  *  Scan ;
        Scan = Scan + 1 ;
    }
    return (Product) ;
}
```

Now consider a recursive version of `Factorial`:

```c
int Factorial(int n ) {
    if ( n > 1 )
        return( n * Factorial (n-1) );
    else
        return(1);
}
```
First the “recursive descent” . . .

Factorial (5)

5 * Factorial (4)

return 120

4 * Factorial (3)

return 24

3 * Factorial (2)

return 6

2 * Factorial (1)

return 2

1

return 1

. . . and then the return sequence
Recursion Attributes

- Every recursive algorithm can be implemented non-recursively.

  \[ \text{recursion} \iff \text{iteration} \]

- Eventually, the routine must not call itself, allowing the code to "back out".

- Recursive routines that call themselves continuously are termed:

  \[ \text{infinite recursion} \iff \text{infinite loop} \]

- Problem with this recursive factorial implementation?

  **Negative numbers!**

- Recursion is inefficient at runtime.
Here is a recursive function that takes an array of integers and computes the sum of the elements:

```cpp
// X[] array of integers to be summed
// Start start summing at this index . . .
// Stop . . . and stop summing at this index
//
int SumArray(const int X[], int Start, int Stop) {
    // error check
    if (Start > Stop || Start < 0 || Stop < 0)
        return 0;
    else if (Start == Stop)              // base case
        return X[Stop];
    else // recursion
        return (X[Start] + SumArray(X, Start + 1, Stop));
}
```
The call:

```cpp
const int Size = 5;
int X[Size] = {37, 14, 22, 42, 19};
SumArray(X, 0, Size-1); // note Stop is last valid index
```

would result in the recursive trace:

```
// return values:
SumArray(X, 0, 4)                        // == 134
   return(X[0]+SumArray(X,1,4))          // == 37 + 97
      return(X[1]+SumArray(X,2,4))       // == 14 + 83
         return(X[2]+SumArray(X,3,4) )    // == 22 + 61
            return(X[3]+SumArray(X,4,4)) // == 42 + 19
```
Mathematical Induction Model

- Solve the trivial "base" case(s).
- Restate general case in 'simpler' or 'smaller' terms of itself.

Array Sum Example

- Determine the size of a single linked list.

Base Case: array size = 1, sum = the element
General Case: first element + Sum(Rest of Array)

// X[] array of integers to be summed
// Start start summing at this index . . .
// Stop . . . and stop summing at this index

int SumArray(const int X[], int Start, int Stop) {

    // error check
    if (Start > Stop || Start < 0 || Stop < 0)
        return 0;
    else if (Start == Stop)               // base case
        return X[Stop];
    else // recursion
        return (X[Start] + SumArray(X, Start + 1, Stop));
}

Example of “tail recursion” (going up recursion).
Tail recursive functions are characterized by the recursive call being the last statement in the function, (can easily be replaced by a loop).
Problem: Implement a recursive selection sort.

Solution: Code a function to find the smallest element in an array and swap it to the *beginning* of the array. Define the *beginning* of the array to be the left most cell of the array range passed to the function. Call the function recursively on the unsorted cells of the array.

```cpp
// X[] array of integers to be sorted
// Start start selection pass here . . .
// Stop . . . and stop the search pass here
void SelSort(int X[], int Start, int Stop) {

    int Begin = Start,
        Check,
        SmallSoFar = Begin;
    int tmpInt;

    if (Start < Stop) {
        for(Check = Begin + 1; Check<=Stop; Check++)
        {
            if (X[Check] < X[SmallSoFar])
                SmallSoFar = Check;
        }
        tmpInt = X[Begin];
        X[Begin] = X[SmallSoFar];
        X[SmallSoFar] = tmpInt;

        SelSort(X, Start + 1, Stop);
    }
}
```
The call:

```cpp
const int Size = 5;
int X[Size] = {37, 14, 22, 42, 19};
SelSort(X, 0, Size - 1);
// note Stop is last valid index
```

would result in the recursive trace:

```
// Contents of X when call occurs:
SelSort(X, 0, 4)       // {37, 14, 22, 42, 19}
SelSort(X, 1, 4)      // {14, 37, 22, 42, 19}
SelSort(X, 2, 4)     // {14, 19, 22, 42, 37}
SelSort(X, 3, 4)    // {14, 19, 22, 37, 42}
SelSort(X, 4, 4)   // {14, 19, 22, 37, 42}
```
Problem:

- Given an array of integers of n+1 elements code a function to return the index of the maximum value in the array.

Solution:

- Check if the middle element is the largest if so return its index otherwise return the index of either the largest element in the lower half or the largest element in the upper half, whichever is the larger of the two.

```cpp
int rMax(const int ray[], int start, int end ) {
    const int Unknown = -1;
    int mid, h1max, h2max;
    if (end < start) return Unknown;
    mid = (start + end) / 2;
    h1max = rMax(ray, start, mid-1); //left half
    if (h1max == Unknown) h1max = start;
    h2max = rMax(ray, mid+1, end); //right half
    if (h2max == Unknown) h2max = end;
    if ( (ray[mid] >= ray[h1max]) &&
         (ray[mid] >= ray[h2max]) )
        return mid;
    else
        return( (ray[h1max] > ray[h2max]) ?
                 h1max : h2max );
}
```

“Unknown” checks ensure that indices are within array subscript range
Given:

Call Tree Trace of

\[ \text{rmax}(\text{ray}, 0, 4); \quad \text{-- -- -- - 4} \]

Middle decomposition (splitting problem into halves), recursive functions are best traced with tree diagrams.
Problem:

- sort a subset, \((m:n)\), of an array of integers (ascending order)

Solution:

- Find the smallest and largest values in the subset of the array \((m:n)\) and swap the smallest with the \(m^{th}\) element and swap the largest with the \(n^{th}\) element, (i.e. order the edges).
- Sort the center of the array \((m+1: n-1)\).

Solution Trace

<table>
<thead>
<tr>
<th>m</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>56</td>
<td>23</td>
<td>66</td>
<td>44</td>
<td>78</td>
<td>99</td>
<td>30</td>
<td>82</td>
<td>17</td>
<td>36</td>
</tr>
</tbody>
</table>

unsorted array

after call#1

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>23</td>
<td>66</td>
<td>44</td>
<td>78</td>
<td>36</td>
<td>30</td>
<td>82</td>
<td>56</td>
<td>99</td>
</tr>
</tbody>
</table>

 Variation of the “selection” sort algorithm

after call#3

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>23</td>
<td>30</td>
<td>44</td>
<td>56</td>
<td>36</td>
<td>66</td>
<td>78</td>
<td>82</td>
<td>99</td>
</tr>
</tbody>
</table>
Recursive Sorting

void duplexSelection(int ray[], int start, int end)
{
    int mini = start, maxi = end;

    if (start < end) { //start==end => 1 element to sort

        findMiniMaxi(ray, start, end, mini, maxi);
        swapEdges(ray, start, end, mini, maxi);
        duplexSelection(ray, start+1, end-1);
    }
}

void findMiniMaxi(const int ray[], int start, int end, int& mini, int& maxi)
{
    if (start < end) { //subset to search exists

        if (ray[start] < ray[mini]) mini = start;
        else if (ray[start] > ray[maxi]) maxi = start;
        findMiniMaxi(ray, start+1, end, mini, maxi);
    }
}

void swapEdges(int ray[], int start, int end, int mini, int maxi)
{
    //check for swap interference
    if ( (mini == end) && (maxi == start) ) {
        swap(ray[start], ray[end]);
    } //check for low 1/2 interference
    else if (maxi == start) {
        swap(ray[maxi], ray[end]);
        swap(ray[mini], ray[start]);
    } // (mini == end) || no interference
    else {
        swap(ray[mini], ray[start]);
        swap(ray[maxi], ray[end]);
    }
}

void swap(int& x, int& y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
Knapsack Problem (*very weak form*)

- Given an integer total, and an integer array, determine if any collection of array elements within a subset of the array sum up to total.
- Assume the array contains only positive integers.

Special Base Cases

- total = 0 :
  † solution: the collection of no elements adds up to 0.
- total < 0 :
  † solution: no collection adds to sum.
- start of subset index > end of subset index :
  † solution: no such collection can exist.

Inductive Step

- Check if a collection exists containing the first subset element.
  † Does a collection exist for total - ray[ subset start ] from subset start + 1 to end of subset?
- If no collection exists containing ray[ subset start ] check for a collection for total from subset start + 1 to the end of the subset.

Backtracking step. Function searches for alternative solution “undoing” previous possible solution search work.
**Knap backtracking function**

```cpp
bool Knap (const int ray[], int total, int start, int end)
{
    if (total == 0) // empty collection adds up to 0
        return true;
    if ( (total < 0) || (start > end) ) // no such
        return false; // collection exists

    // check for collection containing ray[start]
    if (Knap(ray, total-ray[start], start+1, end))
        return true;

    // check for collection w/o ray[start]
    return (Knap(ray, total, start+1, end));
}
```

**Trace**

```
| TRUE |

Knap(ray, 50, 1, 4)  TRUE |

Knap(ray, 30, 2, 4)  FALSE |

| TRUE |

Knap(ray, -10, 3, 4)  FALSE |

| TRUE |

Knap(ray, -30, 4, 4)  TRUE |

| TRUE |

Knap(ray, 0, 5, 4)  TRUE |
```

**Backtracking steps**
Recursion Underpinnings

- Every instance of a function execution (call) creates an **Activation Record, (frame)** for the function.
- Activation records hold required execution information for functions:
  † Return value for the function
  † Pointer to activation record of calling function
  † Return memory address, (calling instruction address)
  † Parameter storage
  † Local variable storage

Runtime Stack

- Activation records are created and stored in an area of memory termed the **“runtime stack”***.

![Runtime Stack Diagram]

---

**Fnx() activation record**

- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fnx return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fnx return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fnx return value

**Fny() activation record**

- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fny return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fny return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fny return value

**main() activation record**

- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fn return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fn return value
- Local storage
- Parameter storage
- Return address
- Pointer previous act. rec.
- Fn return value
First backtrack step (during fourth call)

- Let first recursive call in knap be at address $\alpha$
- Let second recursive call in knap be at address $\beta$

Activation record destroyed as third recursive call completes execution
### Storage Organization

#### Typical C++ program execution memory model

<table>
<thead>
<tr>
<th>System privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not accessible to the C program)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(text segment)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>(data segment)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function activation record management</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic memory structure management</td>
</tr>
</tbody>
</table>

#### Storage Corruption

- Infinite regression results in a collision between the “run-time” stack & heap termed a “run-time” stack overflow error.
- Illegal pointer de-references (garbage, dangling-references) often result in memory references outside the operating system allocated partition, (segment) for the C program resulting in a “segmentation error” (GPF - access violation) and core dump.