Recursion

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. Comparison Problem
16. Iterative vs Recursive Solution
17. Backtracking
18. Knapsack Solution
19. Runtime Stack
20. Knap Runtime Trace Snapshot
21. 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
\[ N! = 1 \times 2 \times \ldots \times N \]

```cpp
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`:

```cpp
int Factorial(int n) {
    if (n > 1)
        return (n * Factorial(n-1));
    else
        return (1);
}
```
9. Recursion

Recursive Execution Trace

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

First the “recursive descent” . . .

. . . and then the return sequence
Recursion Attributes

- Every recursive algorithm can be implemented non-recursively.

  recursion $\iff$ iteration

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

- Recursive routines that call themselves continuously are termed:

  infinite recursion $\iff$ 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:

```c
// 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.

List Example

- Determine the size of a single linked list.

  Base Case : Empty List, size = 0
  General Case : 1 + Size(Rest of List)

```
int LinkList::SizeList ()
{
    return (listSize(Head));
}

int LinkList::listSize (LinkNode* list)
{
    if ( list == NULL )
        return (0);
    else
        return ( 1 + listSize(list->getNext()) );
}
```

Trace listSize(list)

- listSize(list=(6, 28, 120, 496))
  = (1 + listSize(list=(28, 120, 496)))
  = (1 + (1 + listSize(list=(120, 496)))))
  = (1 + (1 + (1 + listSize(list=(496)))))
  = (1 + (1 + (1 + (1 + listSize(list=(•))))))
  = (1 + (1 + (1 + (1 + 0))))
  = (1 + (1 + 1))
  = (1 + 2)
  = (1 + 3)
  = 4

"Tail recursive" functions are characterized by the recursive call being the last statement in the function, (can easily be replaced by a loop).
Problem:
- Code a function void intComma( long ) that outputs the int comma separated:
- e.g.,
  - the call: intComma( 123456789 );
  - displays: 123,456,789

Top-Down Design

```cpp
void intComma ( long num ) {
    if (num is less than 1000)
        display num
    else
        display comma separated digits above 1000
        display comma
        display digits below 1000
}
```

Code

```cpp
void intComma ( long num ) {
    if (num < 1000)
        cout << setw(3) << num;
    else {
        intComma(num / 1000);
        cout << ',' << setw(3) << num % 1000;
    }
}
```

Consider:
- intComma( 123456789 );
- intComma( 1001 );
9. Recursion

General Solution

```cpp
void intComma ( long num ) {  
    if (num < 0) { // display sign for negatives
        cout << '-';
        num = -num;
    }
    if (num < 1000)
        cout << setw(3) << num;
    else {
        intComma(num / 1000);
        cout << ','; // display digits
        num = num % 1000; // separately
        cout << (num / 100); // for zeroes
        num = num % 100;
        cout << (num / 10) << (num % 10);
    }
}
```

Example of “going down” (head) recursion

Trace intComma(9087605430);

```cpp
intComma(9087605430)
    = intComma(9087605) and • • •
    = intComma(9087) and • • •
    = intComma(9) and • • •
    = intComma(9)
```

string prefix =
    (num < 10) ? "00" :
    (num < 100) ? "0" : "";
cout << prefix << num;

Output

Example of escape recursion

```
9
9,087
9,087,605
9,087,605,430
```
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.

```c
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:

\[
\text{ray} = \begin{bmatrix}
[0] & 56 \\
[1] & 23 \\
[2] & 66 \\
[3] & 44 \\
[4] & 78 \\
\end{bmatrix}
\]

Call Tree Trace of

\[
r\text{max}(\text{ray}, 0, 4);
\]

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\textsuperscript{th} element and swap the largest with the n\textsuperscript{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>n</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>

after call#1

<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>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>

... 

after call#3

<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>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
9. Recursion

```c
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) ) { //check for low 1/2 interference
        swap(ray[start], ray[end]);
    } else if (maxi == start) {
        swap(ray[maxi], ray[end]);
        swap(ray[mini], ray[start]);
    } else {
        swap(ray[mini], ray[start]);
        swap(ray[maxi], ray[end]);
    }
}

void swap(int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```
Comparison Problem

9. Recursion

Given: Link List & Item classes

```cpp
#include "LinkList.h"
#include "Item.h"
```

Problem:

- Given two ordered single linked-lists code a Boolean function, subList, that determines if the first list is a sublist of the second list. List, L1, is a sublist of another list, L2, if all of the elements in list L1 are also elements in list L2.

- The following assumptions for the lists hold:
  † There are no duplicate elements in the lists.
  † The elements in the lists are in ascending order.

e.g.

```
#include "LinkList.h"
#include "Item.h"

LinkList L1, L2;
L1.gotoHead(); L2.gotoHead();
subList(L1, L2); // returns true
subList(L2, L1); // returns false
```
Iterative Solution:

```cpp
bool subList (LinkList L1, LinkList L2) {
    L1.gotoHead();
    L2.gotoHead();
    bool stillSublist = true;
    while ((stillSublist) && (L1.inList())) {
        while ((L2.inList()) &&
            (L2.getCurrentData() < L1.getCurrentData()))
            L2.Advance();
        stillSublist = (!L2.inList()) ? (false) :
            (L2.getCurrentData() == L1.getCurrentData());
        L1.Advance();
    }
    return stillSublist;
}
```

Recursive Solution:

```cpp
bool subList (LinkList L1, LinkList L2) {
    if (L1.isEmpty()) return true;
    if (L2.isEmpty()) return false;
    if (L1.getCurrentData() < L2.getCurrentData())
        return false; //miss
    if (L1.getCurrentData() == L2.getCurrentData()) { //hit
        L1.Advance(); L2.Advance();
        return (subList(L1, L2)); //for next
    }
    //else (L2.getCurrentData() < L1.getCurrentData())
    L2.Advance();
    return (subList2(L1, L2));
    }
```
Backtracking

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

**Knap(ray, 100, 0, 4)**

1. **TRUE**
2. **Knap(ray, 50, 1, 4)**
3. **TRUE**
4. **FALSE**
   - **Knap(ray, -10, 3, 4)**
   - **FALSE**
   - **Knap(ray, -30, 4, 4)**
   - **TRUE**
   - **Knap(ray, 30, 4, 4)**
   - **TRUE**
5. **Knap(ray, 0, 5, 4)**

---

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”**.
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 $\beta$
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.