Table of Contents

- Definitions
- Coding Recursively
- Recursive Design
- Avoiding Pitfalls
- Middle Decomposition
- Call Tree Traces
- Edges & Center Decomposition
- Recursive Sorting
- Comparison Problem
- Iterative vs Recursive Solution
- Backtracking
- Knapsack Solution
- Knap Runtime Trace Snapshot
- 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.”
Mathematical Induction Model

- Solve the trivial "base" case(s).
- Restate the 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 listSize ( nodePtr list )
{
    if (list == NULL)
        return 0 ;
    else
        return (1 + listsize(list->link));
}
```

Example of “tail recursion” (going up recursion)

Example of “tail recursive” functions are characterized by the recursive call being the last statement in the function, (can easily be replaced by a loop).

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)))
  = (1 + (1 + 2))
  = (1 + 3)
  = 4
Problem:
- Code a function void intComma( int ) that outputs the int comma separated:
- e.g.,
  - the call: intComma( 123456789 );
  - displays: 123,456,789

Top-Down Design

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

Consider:
```
intComma( 123456789 );
intComma( 1001 );
```
Avoiding Pitfalls

✿ General Solution

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

✿ Example of “going down” recursion

✿ Example of "going down" recursion

✿ Trace intComma(9087605430);

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

✿ output

```
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 it’s 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);
    h1max = (h1max != Unknown) ? h1max : start;
    h2max = rMax(ray, mid+1, end);
    h2max = (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:

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

Call Tree Trace of $\max(\text{ray}, 0, 4)$:

- $\text{mid}=2$
- $h1\text{max}=0$
- $h2\text{max}=1$
- $\max(\text{ray}, 0, 1)$
- $\text{mid}=0$
- $h1\text{max}=1$
- $h2\text{max}=1$
- $\max(\text{ray}, 0, -1)$
- $\text{unknown}$
- $\max(\text{ray}, 1, 1)$
- $\text{mid}=1$
- $\max(\text{ray}, 1, 0)$
- $\text{unknown}$
- $\max(\text{ray}, 2, 1)$
- $\text{unknown}$
- $\max(\text{ray}, 3, 4)$
- $\text{mid}=3$
- $h1\text{max}=4$
- $h2\text{max}=4$
- $\max(\text{ray}, 3, 2)$
- $\text{unknown}$
- $\max(\text{ray}, 4, 4)$
- $\text{mid}=4$
- $\max(\text{ray}, 4, 3)$
- $\text{unknown}$
- $\max(\text{ray}, 5, 4)$
- $\text{unknown}$

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>Unsorted Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>56 23 66 44 78 99 30 82 17 36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After Call #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 23 66 44 78 36 30 82 56 99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After Call #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 23 30 44 56 36 66 78 82 99</td>
</tr>
</tbody>
</table>
void duplexSelection(const 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(const 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;
}
Recursion

Comparison Problem

Given:

```c
typedef struct node *listType;
typedef struct {
    infoType element;
    listType link;
} node;

listType list = NULL;
```

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. A list, A, is a sublist of another list, B, if all of the elements in list A are also elements in list B.

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

Example:

```
A

B
```

<table>
<thead>
<tr>
<th></th>
<th>28</th>
<th>496</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>496</td>
</tr>
<tr>
<td></td>
<td></td>
<td>600</td>
</tr>
</tbody>
</table>

- subList(A, B) would return true
- subList(B, A) would return false
Iterative Solution

```c
bool subList ( listType x, listType y ) {
    listType  p = x;
    listType  q = y;
    bool stillSublist = true;

    while ( (stillSublist) && ( p ) ) {
        while ( q && (lessThan(q->element, p->element) )
            q = q-> link ;

        stillSublist = ( (!q) ? (false) :
            (equalTo(q->element, p->element)) ) ;
        p = p->link;
    }
    return stillSublist;
}
```

Recursive Solution

```c
bool subList ( listType x, listType y ) {
    if (!x) return true;
    if (!y) return false;

    if (greaterThan(y->element, x->element))
        return false;

    if (equalTo (y->element, x->element))
        return ( sublist(x->link, y->link) );

    return ( sublist(x, y->link) );
}
```
Knapsack Problem (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.

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
Knap(ray, 30, 3, 4)   // TRUE
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 $\beta$
Typical C program execution memory model

<table>
<thead>
<tr>
<th>System privileges (not accessible to the C program)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Code</td>
</tr>
<tr>
<td>Static Data</td>
</tr>
<tr>
<td><strong>Runtime Stack</strong></td>
</tr>
<tr>
<td>- Function activation record management</td>
</tr>
<tr>
<td>- Dynamic memory structure management</td>
</tr>
<tr>
<td>Heap</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.