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 \( N! = 1 \times 2 \times \ldots \times N \):

```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);
}
```

**Head Recursion**: working from the end towards the front.
Recursion

Problem Solving


Factorial (5)

5 * Factorial (4) → return 24

4 * Factorial (3) → return 6

3 * Factorial (2) → return 2

2 * Factorial (1) → return 1

First the “recursive descent” . . .

. . . 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?

  \[
  \text{Negative numbers!}
  \]

• Recursion is inefficient at runtime.
Recursion

Recursive Solution Strategies

- Tail recursion
  - Work from the beginning to the end
  - Solve for first part of data set & call recursively

- Head recursion
  - Work from the end to the beginning
  - Solve for last part of data set & call recursively

- Middle decomposition
  - Solve for middle part of data set & call recursively
  - Split the data into halves and call recursively on both

- Edges & center decomposition
  - Solve for first and last parts of data set & call recursively working towards the center of the data

- Backtracking (greedy algorithms)
  - Enumerate to try every possible solution
  - Back up to possible previous solutions
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));
}
```

**Tail Recursion**: working from the beginning towards the end.
The call:

```c
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:

```c
SumArray(X, 0, 4) // return values:
  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)

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 + (1 +0)))))
  = (1 + (1 + (1 + (1 + 1))))
  = (1 + (1 + 2))
  = (1 + 3)
  = 4

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:
- 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:


Call Tree Trace of

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

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

- Select an item in the array as the pivot key.
- Divide the array into two partitions: a left partition containing elements < the pivot key and a right partition containing elements ≥ the pivot key.

Trace

Start with i and j pointing to the first & last elements, respectively.
Select the pivot (3): [3 1 4 1 5 9 2 6 5 8]

L                                           R

Swap the end elements, then move L, R inwards.
[8 1 4 1 5 9 2 6 5 3]

L                           R

Swap, and repeat:
[2 1 4 1 5 9 8 6 5 3]

L   R

Swap, and repeat:
[2 1 1 | 4 5 9 8 6 5 3]

R    L

Partition Function:

```c
int Partition(Item A[], int start, int end, const Item& pivot ){
  int L = start, R = end;
  do {
    swap( A[L] , A[R] );
    while (A[L] < pivot )    L++;
    while (!(A[R] < pivot))  R--; //assumes < overload
  } while (R > L);
  return (L);
}
```
Quicksort: find pivot

Pivoting

- Partitioning test requires at least 1 key with a value < that of the pivot, and 1 value ≥ to that of the pivot, to execute correctly.
- Therefore, pick the greater of the first two distinct values (if any).

```c
const int MISSING = -1;

int FindPivot(const Item A[], int start, int end) {
    Item firstkey;  // value of first key found
    int pivot;      // pivot index
    int k;          // run right looking for other key

    firstkey = A[start];
    // return -1 if different keys are not found
    pivot = MISSING;
    k = start + 1;
    // scan for different key
    while ((k <= end) && (pivot == MISSING)) {
        if (firstkey < A[k])   // select key
            pivot = k;
        else if (A[k] < firstkey)
            pivot = start;
        else
            k++;
    }
    return pivot;
}
```

Improving FindPivot

- Try and pick a pivot such that the list is split into equal size sublists, (a speedup that should cut the number of partition steps to about 2/3 that of picking the first element for the pivot).
  † Choose the middle (median) of the first 3 elements.
  † Pick k elements at random from the list, sort them & use the median.
- There is a trade-off between reduced number of partitions & time to pick the pivot as k grows.
QuickSort Function (recursive)

```c
const int MISSING = -1;
void QuickSort( Item A[], int start, int end ) {
  // sort the array from start ... end
  Item pivotKey;
  int pivotIndex;
  int k;  //index of partition >= pivot

  pivotIndex = FindPivot( A, start, end );
  if (pivotIndex != MISSING) {
    pivotKey = A[pivotIndex];
    k = Partition( A, start, end, pivotKey );
    QuickSort( A, start, k-1 );
    QuickSort( A, k, end );
  }
}
```

Average Case = $O(N\log N)$

- quicksort is based upon the intuition that swaps, (moves), should be performed over large distances to be most effective.
- quicksort's average running time is faster than any currently known $O(n\log_2 n)$ internal sorting algorithms (by a constant factor).
- For very small $n$ (e.g., $n \leq 16$) a simple $O(n^2)$ algorithm is actually faster than Quicksort.
- When the sublist is small, use another sorting algorithm, (selection).

Worst Case = $O(N^2)$

- In the worst case, every partition might split the list of size $j - i + 1$ into a list with 1 element, and a list with $j - i$ elements.
- A partition is split into sublists of size 1 & $j-i$ when one of the first two items in the sublist is the largest item in the sublist which is chosen by findpivot.
- When will this worst case partitioning always occur?
Iterative QuickSort

Iterative Conversion

- Iterative implementation requires using a stack to store the partition bounds remaining to be sorted.
- Assume a stack implementation of elements consisting of two integers:

```
struct StackItem {
    int low, hi;
};
```

Partitioning

- At the end of any given partition, only one subpartition need be stacked.
- The second subpartition (equated to the second recursive call), need not be stacked since it is immediately used for the next subpartitioning.

Stacking

- The order of the recursive calls, (i.e., the sorting of the subpartitions) may be made in any order.
- Stacking the larger subpartition assures that the size of the stack is minimized, since the smaller subpartition will be further divided less times than the larger subpartition.
Quicksort Function (iterative)

```c
void QuickSort( Item A[], int start, int end ) {
    // sort the array from start ... end
    Item pivotKey;
    int pivotIndex, tmpBnd;
    int k;                       //index of partition >= pivot
    StackItem parts;
    Stack subParts;

    parts.low = start;   parts.hi = end;
    subParts.Push( parts ); //prime stack

    while ( ! subParts.Empty() ) { //while partitions exist
        parts = subParts.Pop();

        while ( parts.hi > parts.low ) {
            pivotIndex = FindPivot( A, parts.low, parts.hi );

            if (pivotIndex != MISSING) {
                pivotKey = A[pivotIndex];
                k = Partition( A, parts.low , parts.hi , pivotKey );
                // push the larger subpartition

                if ( (k-parts.low) > (parts.hi-k) ) { //stk low part
                    tmpBnd = parts.hi;
                    parts.hi = k-1;
                    subParts.Push( parts );
                    parts.low = k; //set current part to upper part
                    parts.hi = tmpBnd;
                } //end if
                else { // stack upper (larger) part
                    tmpBnd = parts.low;
                    parts.low = k;
                    subParts.Push( parts );
                    parts.low = tmpBnd; //set current part to low part
                    parts.hi = k-1;
                } // end else
            } // end if
            else // halt inner loop when all elements equal
                parts.hi = parts.low;
        } // end while
    } // end while
}
```
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>(m)</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>after call#1</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0])</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>after call#3</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0])</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variation of the “selection” sort algorithm</th>
</tr>
</thead>
</table>

- \(m\) and \(n\) are the indices of the subset.
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;
}
Comparison Problem

Given: Link List & Item classes

```c
#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.**

![Diagram of two linked lists](attachment:image.png)

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

```c
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:

```c
bool subList (LinkList L1, LinkList L2) {
    if (L1.inList()) return true;
    if (L2.inList()) 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));
}
```
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.
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)  Knap(ray, 50, 1, 4)  Knap(ray, 30, 2, 4)  Knap(ray, -10, 3, 4)  Knap(ray, 30, 3, 4)  Knap(ray, 0, 5, 4)
   ↓       ↑            ↓       ↑            ↓       ↑            ↓       ↑            ↓       ↑
   TRUE    TRUE         FALSE   TRUE         FALSE   TRUE         TRUE   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”.

![Diagram of runtime stack with activation records for main(), Fnx(), and Fny() functions, showing local storage, parameter storage, return address, pointer to previous activation record, and 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 $\beta$
Typical C++ program execution memory model

- **System privileges**
  (not accessible to the C program)

- **Binary Code**
  (text segment)

- **Static Data**
  (data segment)

- **Runtime Stack**
  Function activation record management

- **Dynamic memory structure management**

- **Heap**

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.