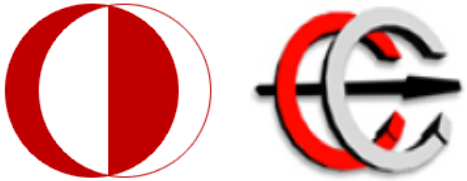


EE 441 Data Structures

Lecture 3: Functions and Recursion

Functions: Explanation

- Definition
  - A function is a block of code that performs a specific task
  - Use of functions promotes modularity and code reuse
- Basic Structure

```
return_type function_name(parameters) {
    // function body
    return value;   // optional (depends on return_type)
}
```

  - Return type: Type of value returned (int, double, void, …)
  - Name: Identifier for the function
  - Parameters: Input values (optional)
  - Body: Statements to execute

Argument Passing

- Write a function that
  - Takes two integers as input
  - Swaps the integer values

```
swap(int a, int b);
```

- Desired operation

```
int main(){
   int i=10;
   int j=20;
   cout << "Before swap(): i:" << i << "j: " << j << "\n";
   swap(i,j);
   cout << "After swap(): i: "<< i << "j: " << j << "\n";
}
```

```
Before swap(): i: 10 j: 20
After swap(): i: 20 j: 10
```

Argument Passing

- C++ offers three ways for argument passing
  - Pass by value

```
swap(int a, int b);
```

  - Pass by address

```
swap(int *a, int *b);
```

  - Pass by reference

```
swap(int &a, int &b);
```

Argument Passing: Pass by Value

- Pass by Value
  - Default argument passing mechanism
  - Function makes a local copy of the original argument when called
  - The copy remains in scope until the function returns
  - The copy is destroyed immediately afterwards

→ A function that takes value-arguments cannot change them

→ Changes will apply only to local copies, not the actual caller's variables!

- Remark
  - We need to override the default passing mechanism if the (function) callee should modify its arguments

```
#include <iostream>
using namespace std;
void swap(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main(){
  int i = 10;
  int j = 20;
  cout << "Before swap(): i:
    "<<i<<" j: "<<j<<endl;
  swap(i,j);
  cout << "After swap(): i:
    "<<i<<" j: "<<j<<endl;
}
```

scope

Local copies of the arguments

```
Before swap(): i: 10 j: 20
After swap(): i: 10 j: 20
```

Code: Lecture3_examples Block 1

Argument Passing: Pass by Address

- Passing the arguments' address to the function
- The function makes a local copy of the address

→ Changes are made on the data stored in the address location

- Problem: this technique is tedious and error-prone

```
#include <iostream>

using namespace std;
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
  int i = 10;
  int j = 20;
  cout << "Before swap(): i:
    "<<i<<" j: "<<j<<endl;
  swap(&i,&j);
  cout << "After swap(): i:
    "<<i<<" j: "<<j<<endl;
}
```

```
Before swap(): i: 10 j: 20
After swap(): i: 20 j: 10
```

Code: Lecture3_examples Block 2

Argument Passing: Pass by Reference

- References

  `int &a`

- Syntactically behave like ordinary variables
- Used as pointers from a compiler's point of view

→ Enable a function to alter its arguments without forcing programmers to use the difficult *, & and -> notation

```
#include <iostream>
using namespace std;
void swap(int &a, int &b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main(){
  int i = 10;
  int j = 20;
  cout << "Before swap(): i:
    "<<i<<" j: "<<j<<endl;
  swap(i,j);
  cout << "After swap(): i:
    "<<i<<" j: "<<j<<endl;
}
```

Referenced global arguments

```
Before swap(): i: 10 j: 20
After swap(): i: 20 j: 10
```

Code: Lecture3_examples Block 3

References: Details

- A reference is "an alias for an existing object"

```
int m = 0;
int &ref = m;
```

- The reference `ref` serves as an alias for the variable `m`
  - → `ref` and `m` behave as distinct names of the same object
  - → Any change applied to `m` is reflected in `ref` and vice versa
- There can be an infinite number of references to the same object

```
int &ref2 = ref;
int &ref3 = m;
```

- Here `ref2` and `ref3` are aliases of `m`, too
- Note: there is no concept as a "reference to a reference"
  - → The variable `ref2` is an alias of `m`, not `ref`

Advantages of Pass by Reference

- Combines benefits of passing by address and passing by value
  - Efficiency: Like passing by address because the callee doesn't get a copy of the original value but rather an alias thereof (under the hood, compilers substitute reference arguments with ordinary pointers)
  - Intuitiveness: It offers a more intuitive syntax and requires less keystrokes from the programmer
- References are usually safer than ordinary pointers because they are always bound to a valid object
- C++ doesn't have null references → no need to check whether a reference argument is null before examining its value or assigning to it
- Passing objects by reference is usually more efficient than passing them by value because no large chunks of memory are being copied

Example 1

```
#include <iostream>
using namespace std;

int main (){
    int i, *p;
    int numbers[6];
    p = numbers;
    for (i = 0; i < 6; i++)
        *(p+i)=i;

    printArray(numbers,6);
    p = &numbers[2];  *p = 29;
    printArray(numbers,6);
    p = numbers + 3;  *p = 17;
    printArray(numbers,6);
    p = numbers; *p = 21;
    printArray(numbers,6);
    p++;  *p = 32;
    printArray(numbers,6);
    p = numbers;  *(p+4) = 16;
    printArray(numbers,6);
}
```

```
// Helper function
void printArray(int* start, unsigned int
size){
 for(unsigned int i = 0; i < size; i++)
    cout << *(start+i) << " ";
 cout << endl << endl;
}
```

```
0 1 2 3 4 5

0 1 29 3 4 5

0 1 29 17 4 5

21 1 29 17 4 5

21 32 29 17 4 5

21 32 29 17 16 5
```

Code: Lecture3_examples Block 4

Example 2

```
#include <iostream>

using namespace std;


int main(){
    int *a, c, d = 5;
    int *b = &d;
    c=2;
    a = &c;
    *a = d + 12;
    printValues(a,b,c,d);
    a = b;
    *b = *a + 5;
    printValues(a,b,c,d);
}
```

```
// Helper function to print values
void printValues(int *a, int *b, int c,
int d){
  cout << "*a: " << *a << " *b: " << *b
  << " c: " << c << " d: " << d << endl;
  cout << "a: " << a << " b: " << b <<
  endl << endl;
}
```

```
*a: 17 *b: 5 c: 17 d: 5
a: 0x61fe0c b: 0x61fe08

*a: 10 *b: 10 c: 17 d: 10
a: 0x61fe08 b: 0x61fe08
```

Code: Lecture3_examples Block 5

Example 3

```
#include <iostream>

using namespace std;

void triple(double &num){
        num = 3*num;
}

int main(){
   double d = 10.0;
   triple(d);
   cout << "d: " << d << endl;
   return 0;
}
```

```
d: 30
```

Code: Lecture3_examples Block 6

Example 4

```cpp
#include <iostream>

using namespace std;

void triple(double &num){
        num = 3*num;
}

int main(){
        double d = 10.0;
        triple(&d);
        cout << "d: " << d << endl;
        return 0;
}
```

Code: Lecture3_examples Block 7

```
src/main.cpp: In function 'int main()':
src/main.cpp:153:16: error: invalid initialization of non-const reference of type 'double&' from an rvalue of type 'double*'
  153 |         triple(&d);
      |                ^~
src/main.cpp:147:21: note: in passing argument 1 of 'void triple(double&)'
  147 | void triple(double &num){
      |             ~~~~~~~~~~~^~~
```

Example 5

```
#include <iostream>

using namespace std;

void triple(double *num){
        *num = 3**num;
}

int main(){
        double d = 10.0;
        triple(&d);
        cout << "d: " << d << endl;
        return 0;
}
```

```
d: 30
```

Code: Lecture3_examples Block 8

Example 6

```
#include <iostream>

using namespace std;

void triple(double num){
        num = 3*num;
}

int main(){
        double d = 10.0;
        triple(d);
        cout << "d: " << d << endl;
        return 0;
}
```

```
d: 10
```

Code: Lecture3_examples Block 9

Function Pointers: Explanation and Example

- Function pointer definition
  - Stores address of a function
  - Allows calling functions indirectly or passing them as arguments
- Function pointer declaration
  return_type (*ptr)(param_types);
- Usage of function pointers
  - Callbacks (event handlers, interrupts)
  - Configurable algorithms (sorting, applying operations)
  - Runtime flexibility (decide behavior during program execution)

```
#include <iostream>
using namespace std;

// Two functions with same arguments
int add(int a, int b) { return a + b; }
int multiply(int a, int b) { return a * b; }

// Function that accepts a function pointer
void apply(int x, int y, int (*op)(int,int)) {
    cout << "Result: " << op(x, y) << endl;
}

int main() {
    apply(5, 3, add);       // Result: 8
    apply(5, 3, multiply);  // Result: 15
    return 0;
}
```

Code: Lecture3_function_pointer Block 1

Lambda Functions: Explanation and Example

- Lambda function definition
  - Anonymous inline function
  - Compact alternative to defining separate functions
- Syntax

  [capture](parameters) ->
  return_type {body};
- Capture List [ ]
  - [=] capture by value
  - [&] capture by reference
  - [ ] no external variables
- Advantage of Lambdas
  - Short, local, flexible
  - Commonly used for one-off computations or callbacks

```
#include <iostream>
using namespace std;

int main() {
    // Simple lambda without capture
    auto square = [](int x) -> int { return x * x; };
    cout << square(5) << endl;   // 25
    // Lambda with capture
    int factor = 10;
    auto scale = [=](int x) { return x * factor; };
    cout << scale(3) << endl;    // 30
    // Passing lambda as argument
    auto apply = [](int a, int b, auto f) {
        cout << f(a, b) << endl;
    };
    apply(5, 3, [](int x, int y) {return x + y;}); // 8
}
```

Code: Lecture3_function_pointer Block 2

Function Pointers vs Lambda Functions

```
void apply(int a, int b, int (*f)(int,int)) {
    cout << f(a, b) << endl;
}

int multiply(int x, int y) { return x * y; }

int main() {
    // pass normal function pointer
    apply(3, 4, multiply);
}
```

Code: Lecture3_function_pointer Block 3

```
template<typename F>
void apply(int a, int b, F f) {
    cout << f(a, b) << endl;
}

int main() {
    apply(3, 4, [](int x, int y){ return x * y; });
// pass inline lambda
}
```

Code: Lecture3_function_pointer Block 4

```
// Starting from C++20
void apply(int a, int b, auto f) {
    cout << f(a, b) << endl;
}
```

Code: Lecture3_function_pointer Block 5

Template Functions

- Function Template Definition
  - Lets you write one function that works with many data types
  - The compiler generates the correct version when you call it
- Syntax

```
template <typename T>
T myMax(T a, T b) {
    return (a > b) ? a : b;
}
```

Usage

```
int i = myMax(3, 7); // Compile for int
double d = myMax(3.5, 2.1); // double
char c = myMax('a', 'z');   // char
```

- Remarks
  - T is a placeholder for a type
  - Compiler replaces T automatically depending on the given arguments
  - Advantage: avoids code duplication
  - Operations used (such as “>”) must be defined for the used types

Iteration: Idea

- Definition
  - Repetition of code using loops (for, while, do-while)
  - Each repetition = one iteration of the loop
- General Procedure
  - A loop control variable is initialized
  - A condition determines whether to continue
  - The body of the loop executes
  - The control variable is updated

    → step repeats until condition fails
- Advantages
  - Efficient when the number of steps is known or easily controlled
  - Avoids the overhead of repeated function calls (as in recursion)
  - Useful for accumulation, searching, array processing, mathematical series

Iteration: x^n Example

- Idea
  - Start from $x^0 = 1$
  - Compute $x^{i+1} = x^i \cdot x$


```
#include <iostream>
using namespace std;

// Iterative power function
int powerIter(int x, int n) {
    int result = 1;
    cout << result << " ";
    for (int i = 0; i < n; i++) {
        result *= x; // multiply x n times
        cout << result << " ";
    }
    return result;
}
```

```
int main() {
    int x = 2, n = 8;
    cout << x << "^" << n << endl;
    // Compute result
    cout << "Result: " << powerIter(x, n) << endl;
    return 0;
}
```

- Properties
  - Requires ≈ n iterations
  - Requires 2 variables

Code: Lecture3_iteration Block 1

Iteration: Fibonacci Example

- Idea
  - $F_0 = 1$, $F_1 = 1$
  - $F_{i+1} = F_i + F_{i-1}$

```
#include <iostream>
using namespace std;
// Fibonacci number by iteration
int fibonacciIter(int n, bool print) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int a = 0, b = 1, next;
    if (print == 1) cout << b << " ";
    for (int i = 2; i <= n; i++) {
        next = a + b;
        a = b;
        b = next;
        if (print == 1) cout << b << " ";
    }
    return b;
}
```

```
int main() {
    int n = 10;
    cout << "Fibonacci(" << n << ") = "
              << fibonacciIter(n,true) << endl;

    cout << "First 10 Fibonacci numbers: ";
    for (int i = 0; i < n; i++) {
        cout << fibonacciIter(i,false) << " ";
    }
    cout << endl;
    return 0;
}
```

- Properties
  - Requires ≈ n iterations
  - Requires 4 variables

Code: Lecture3_iteration Block 2

Recursion: Idea

- Same Question: How to compute the power function: x^n
- Reminder - Iteration: multiply x by itself n times
  - $2^3 = 2 \cdot 2 \cdot 2 = 8$
  - Now compute 2^4, repeat all the previous multiplications
  - $2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$
- Recursion: We observe that the result for n (x^n) can be computed from the result for n-1 (x^{n-1})
  - $x^n = 1$ if $n = 0$
  - $x^n = x \cdot x^{n-1}$ if $n > 0$
  - Once we have a value for the initial computation for the powers of 2 ($2^0 = 1$) the successive values are twice the previous value
  - We use a smaller power to compute another

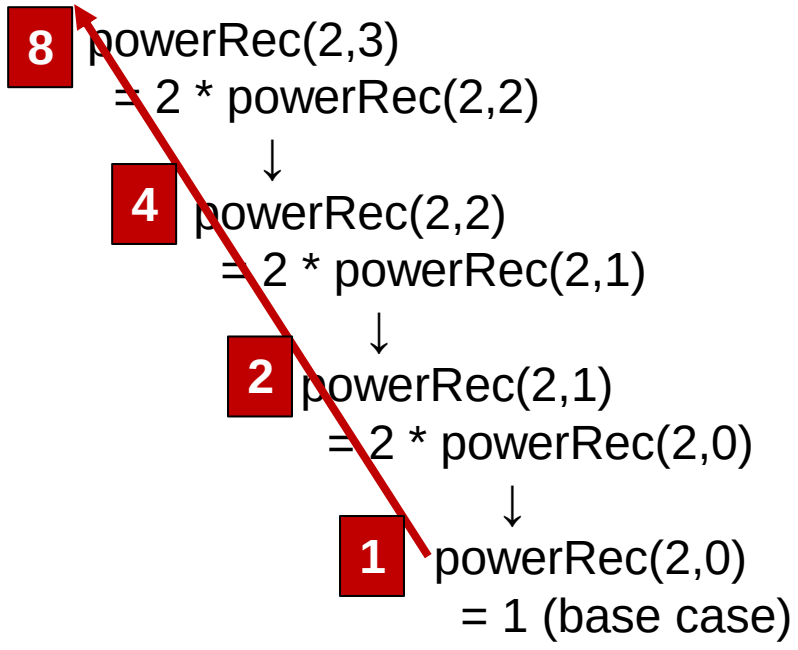
    $\rightarrow$ Recursive definition of the function

Recursion: Example

```
#include <iostream>
using namespace std;

// Recursive power function
int powerRec(int x, int n) {
    cout << n << " ";
    if (n == 0) return 1;          // base case
    return x * powerRec(x, n-1);  // recursive step
}

int main() {
    int x = 2, n = 8, result;
    result = powerRec(x, n);
    cout << endl;
    cout << x << "^" << n << " = " << result << endl;
    return 0;
}
```

Call of another instance of the same function

- Recursion Tree: 2^3



Code: Lecture3_recursive Block 1

Recursion: Discussion

- Divide and conquer design
  - Recursive step: Partition the problem into smaller sub-problems that are solved by using the same algorithm
  - Stopping condition: Partitioning terminates when we reach simpler sub-problems that cannot be solved with that algorithm
- An algorithm is defined recursively if it has
  - One or more stopping conditions: evaluated for certain parameters
  - A recursive step: current value of the algorithm can be defined in terms of a previous value

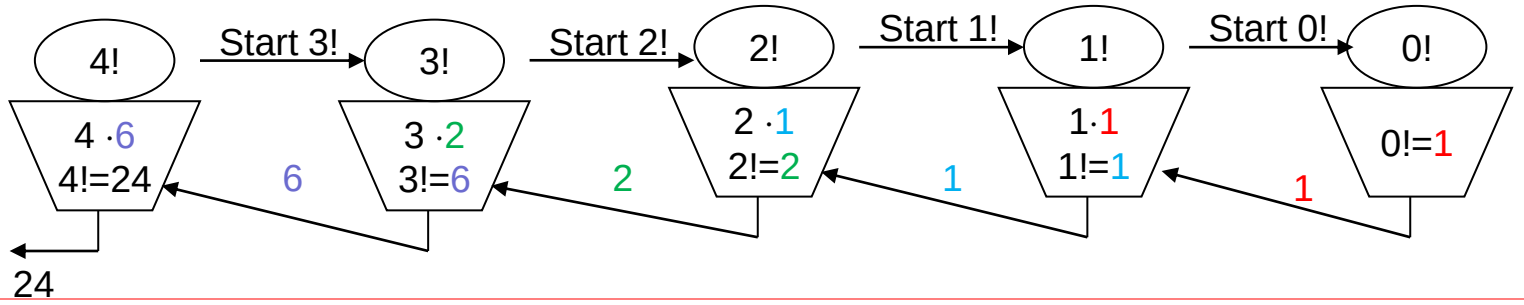
Recursive Function: Factorial

- $\text{Factorial}(n)$: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$
  - 0!=1: special definition
- Iterative version: use a loop
- Recursive
  - $n! = 1$ if $n = 0$: stopping condition
  - $n! = n \cdot (n-1)!$ if $n > 0$: recursive step

Recursive Function: Factorial Illustration

- factorial(n): n-machine that computes the result using $n \cdot (n-1)!$,
  $\rightarrow$ needs result from $(n-1)$-machine and it will output the final result
- factorial($n-1$): $(n-1)$-machine that computes the result using $(n-1) \cdot (n-2)!$,
  $\rightarrow$ needs result from $(n-2)$-machine and it will output to n-machine
- Overall: A sequence of machines that pass information back and forth
- Each needs the result of the previous machine, gives the result to the following machine

- Machine n starts machine n-1
- Machine 1 starts machine 0
- Machine 0 can work independently and produce result
- Result passed back to machine 1
- Machine n produces final result



Recursive Function: Factorial Implementation

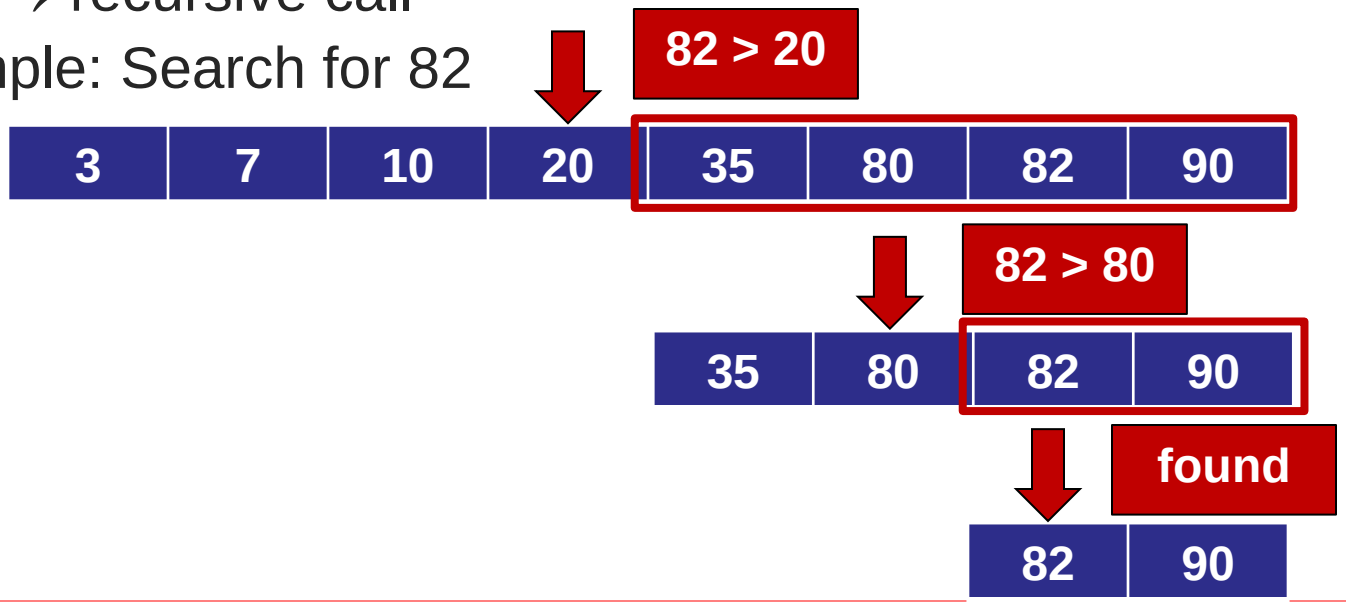
```
int Factorial (int n){
        if (n==0)//stopping condition is 0!
                return 1;
        else //recursive step
                return n*Factorial(n-1);
}
```

Code: Lecture3_recursive Block 2

Recursion Example: Binary Search

- Sorted list of items
- Keep dividing the list into two
- Conduct binary search the half-list that can contain the key
- Keep dividing the list and conduct binary search

  → recursive call
- Example: Search for 82



Recursion Example: Binary Search

```
// recursive version of the binary
// search to locate a key in an
// ordered array A
int BinSearch(int A[], int low, int
high, int key){
  int mid;
  int midvalue;

  // key not found is a stopping
  // condition
  if (low > high)
      return(-1);

  // compare against list midpoint
  // and subdivide if a match does
  // not occur. Apply binary search
  // to the appropriate sublist
```

```
  else{
    mid = (low+high)/2;
    midvalue = A[mid];
    // stopping condition if key matched
    if (key == midvalue)
      return(mid); // key found at index mid

    // look left if key < midvalue;
    // otherwise, look right
    else if (key < midvalue)
        // recursive step
        return BinSearch(A,low,mid-1,key);
    else
       // recursive step
        return BinSearch(A,mid+1,high,key);
  }
}
```

Code: Lecture3_recursive Block 2

Recursion Example: Fibonacci Numbers

- Fibonacci numbers
  - $F_n = 0, \text{if } n = 0$
  - $F_n = 1, \text{if } n = 1$
  - $F_n = F_{n-1} + F_{n-2}, \text{if } n \geq 2$
- Non-recursive (iterative) program

```
int Fibonacci(int n){
  int sum;
  int prev=-1;
  int result=1;
  for(int i=0;i<=n;i++){
    sum=result+prev;
    prev=result;
    result=sum;
  }
  return result;
}
```

- Recursive program

```
int Fibonacci (int n){
  if(n==0||n==1)
    return n;
  else
    return Fibonacci(n-1) +
                Fibonacci(n-2);
}
```

Code: Lecture3_recursive Block 2

Recursion: Memoization

- Definition
  - Technique of storing results of expensive function calls
  - Reuse the stored result when the same inputs occur again
- General Operation
  - Check if the result for input *n* is already stored
  - If yes → return stored value
  - If no → compute it, store it, then return it
- Benefits
  - Reduces exponential recursion to linear time
  - Keeps the recursive style of code, but much more efficient

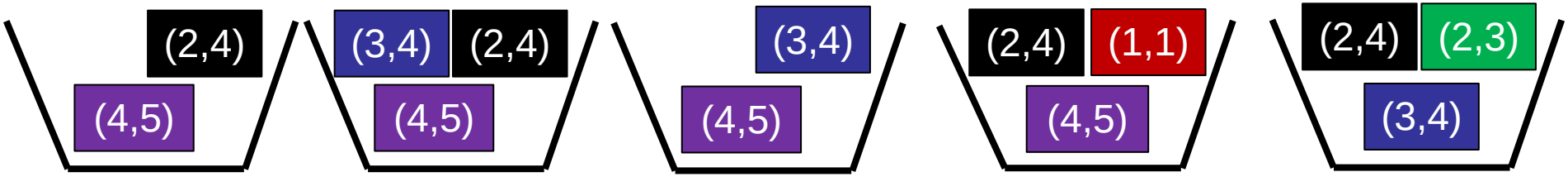
```
#include <iostream>
#include <vector>
using namespace std;
vector<int> memo(100, -1);   // cache

int fibMemo(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) // already computed?
        return memo[n];
    // store result in memo before returning
    return memo[n] = fibMemo(n-1) +
fibMemo(n-2);
}

int main() {
    cout << "fib(40) = "
            << fibMemo(40) << endl;
}
```

Knapsack Problem: Definition

- Problem statement
  - Given n items, each item i with
    - weight w[i]
    - value v[i]
  - Knapsack (bag) has capacity W
  - Each item can be chosen at most once (0 or 1 times)
- Goal
  - Maximize the total value without exceeding capacity
- Example: Assume W = 7

Item	w[i]	v[i]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

Optimal?



W = 6, V = 9 | W = 8, V = 13 | W = 7, V = 9 | W = 7, V = 10 | W = 7, V = 11

Knapsack Problem: Brute Force Solution

- Naive Approach
  - Try all possible subsets of items
  - For each subset: Compute total weight
  - If ≤ W, compute total value
  - Track the maximum
- Required Computation
  - There are 2^n subsets
  - For 5 items: 32 subsets
  - For 20 items: over 1 million subsets!
    → infeasible for large n
- Example with Capacity W = 7
  - Items 1, 2, and 5: w[1] + w[2] + w[5] = 6; v[1] + v[2] + v[5] = 9

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

Knapsack Problem: Brute Force Solution

- Naive Approach
  - Try all possible subsets of items
  - For each subset: Compute total weight
  - If ≤ W, compute total value
  - Track the maximum
- Required Computation
  - There are 2^n subsets
  - For 5 items: 32 subsets
  - For 20 items: over 1 million subsets!
    → infeasible for large n
- Example with Capacity W = 7
  - Items 1, 2, and 5: w[1] + w[2] + w[5] = 6; v[1] + v[2] + v[5] = 9
  - Items 2 and 3: w[2] + w[3] = 7, v[2] + v[3] = 9

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

Knapsack Problem: Brute Force Solution

- Naive Approach
  - Try all possible subsets of items
  - For each subset: Compute total weight
  - If ≤ W, compute total value
  - Track the maximum
- Required Computation
  - There are 2^n subsets
  - For 5 items: 32 subsets
  - For 20 items: over 1 million subsets!
    → infeasible for large n
- Example with Capacity W = 7
  - Items 1, 2, and 5: w[1] + w[2] + w[5] = 6; v[1] + v[2] + v[5] = 9
  - Items 2 and 3: w[2] + w[3] = 7, v[2] + v[3] = 9
  - Items 1, 3, and 5: w[1] + w[3] + w[5] = 7; v[1] + v[3] + v[5] = 10

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

Knapsack Problem: Brute Force Solution

- Naive Approach
  - Try all possible subsets of items
  - For each subset: Compute total weight
  - If ≤ W, compute total value
  - Track the maximum
- Required Computation
  - There are 2^n subsets
  - For 5 items: 32 subsets
  - For 20 items: over 1 million subsets!
    → infeasible for large n
- Example with Capacity W = 7
  - Items 1, 2, and 5: w[1] + w[2] + w[5] = 6; v[1] + v[2] + v[5] = 9
  - Items 2 and 3: w[2] + w[3] = 7, v[2] + v[3] = 9
  - Items 1, 3, and 5: w[1] + w[3] + w[5] = 7; v[1] + v[3] + v[5] = 10
  - Items 2, 4, and 5: w[2] + w[4] + w[5] = 7; v[2] + v[4] + v[5] = 11 — **optimal**

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

Knapsack Problem: Dynamic Programming

- Optimal Substructure
  - Consider optimal solution for capacity W
  - Condition if item i is part of optimal solution
    - We must use capacity W – w[i] for other items
    - We determine the optimal solution for W – w[i]
- Dynamic Programming Solution
  - Construct table: Capacity vs Selected Items

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

		Capacity							
		0	1	2	3	4	5	6	7
Items	0								
	1								
	2								
	3								
	4								
	5								

cell content: maximum value with items up to this row

Check item i = 4
Capacity of other items must be W – w[i] = 7 – 2 = 5
We need the optimal solution for capacity 5

Knapsack Problem: Recurrence Relation

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0								
	1								
	2								
	3								
	4								
	5								

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(dp[i-1][w], v[i]+dp[i-1][w-w[i]]) & otherwise \end{cases}$$

- Base cases
  - dp[0][w] = 0 (0 items)
  - dp[i][0] = 0 (0 capacity)

Knapsack Problem: Example – Initialization

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0	0	0	0	0	0	0	0	0
	1	0							
	2	0							
	3	0							
	4	0							
	5	0							

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(dp[i-1][w], v[i]+dp[i-1][w-w[i]]) & otherwise \end{cases}$$

- Base cases
  - dp[0][w] = 0 (0 items)
  - dp[i][0] = 0 (0 capacity)

Knapsack Problem: Example – i = 1

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0							
	3	0							
	4	0							
	5	0							

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

v[1] + dp[0][w−w[1]] =
1 + dp[0][w - 1]

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(\mathbf{dp[i-1][w]}, \mathbf{v[i]+dp[i-1][w-w[i]]}) & otherwise \end{cases}$$

Knapsack Problem: Example – i = 2

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0	1	1	4	5	5	5	5
	3	0							
	4	0							
	5	0							

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

v[2] + dp[1][w−w[2]] =
4 + dp[1][w - 3]

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(\mathbf{dp[i-1][w]}, \mathbf{v[i]+dp[i-1][w-w[i]]}) & otherwise \end{cases}$$

Knapsack Problem: Example – i = 3

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0	1	1	4	5	5	5	5
	3	0	1	1	4	5	6	6	9
	4	0							
	5	0							

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

v[3] + dp[2][w−w[3]] =
5 + dp[2][w - 4]

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(\mathbf{dp[i-1][w]}, \mathbf{v[i]+dp[i-1][w-w[i]]}) & otherwise \end{cases}$$

Knapsack Problem: Example – i = 4

- Recurrence Relation

		Capacity							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Items	**0**	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0	1	1	4	5	5	5	5
		0	1	1	4	5	6	6	9
	4	**0**	**1**	**3**	**4**	**5**	**7**	**8**	**9**
	5	0							

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

v[4] + dp[3][w−w[4]] =
3 + dp[3][w - 2]

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(\mathbf{dp[i-1][w]}, \mathbf{v[i]+dp[i-1][w-w[i]]}) & otherwise \end{cases}$$

Knapsack Problem: Example – i = 5

- Recurrence Relation

		Capacity							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Items	**0**	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0	1	1	4	5	5	5	5
	3	0	1	1	4	5	6	6	9
	4	0	1	3	4	5	7	8	9
	5	**0**	**1**	**4**	**5**	**7**	**8**	**9**	**11**

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

v[5] + dp[4][w−w[5]] =
4 + dp[4][w - 2]

- Define: dp[i][w] = maximum value using first i items with capacity w

$$dp[i][w] = \begin{cases} dp[i-1][w] & if\ w[i] > W \\ \max(\mathbf{dp[i-1][w]}, \mathbf{v[i]+dp[i-1][w-w[i]]}) & otherwise \end{cases}$$

Knapsack Problem: Example – i = 5

- Recurrence Relation

		Capacity							
		0	1	2	3	4	5	6	7
Items	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1
	2	0	1	1	4	5	5	5	5
	3	0	1	1	4	5	6	6	9
	4	0	1	3	4	5	7	8	9
	5	0	1	4	5	7	8	9	11

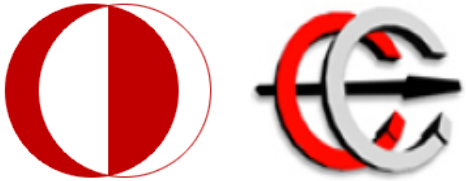
optimal

Item	w[i-1]	v[i-1]
1	1	1
2	3	4
3	4	5
4	2	3
5	2	4

- Result
  - dp[n][W] = the maximum achievable value with all n items and capacity W
  - Backtracking from optimal value using item weight
  - In each column ofbacktracking: Choose the item where the value changes

Generalization

- Knapsack Problem Summary
  - Problem: Choose a subset of items (weight,value) to maximize total value without exceeding knapsack capacity W.
  - Observation: overlapping subproblems and an optimal substructure
  - Solution: Use DP table (items × capacity) with recurrence
- General Idea of Dynamic Programming
  - Brake problems into overlapping subproblems, store and reuse results
  - Core requirements
    - Optimal substructure: Optimal solution to the whole problem can be built from optimal solutions of its subproblems
    - Overlapping subproblems: Same subproblems are solved many times in the naive recursive or iterative approach
- General Approaches
  - Top-Down (Memoization): recursion + cache
  - Bottom-Up (Tabulation): build solution iteratively in a table



EE 441 Data Structures

Lecture 3: Functions and Recursion