Slot 09 - Recursion (Part 2)

Presenter:

Dr. LE Thanh Tung

Content

1. Type of Recursion
2. Dynamic Programming
3. Knapsack Problem

Type of Recursion

- Direct Recursion & Indirect Recursion
- Linear Recursion, Binary Recursion & Multiple Recursion
- Tail Recursion vs Non-tail Recursion
- Nested Recursion

Type of Recursion

```
// direct recursion
int fact(int x){
    if (x == 0)
        return 1;
    else
        return x*fact(x-1);
}
```

```
// indirect recursion
bool isOdd(int x){
    return !isEven(x);
}

bool isEven(int x){
    if (x == 0)
        return true;
    else
        return isOdd(x - 1);
}
```

Type of Recursion

```c
// linear recursion
int fact(int x){
    if (x == 0)
        return 1;
    else
        return x*fact(x-1);
}
```

```c
// binary recursion
int fibo(int x){
    if (x < 2)
        return x;
    else
        return fibo(x - 1) + fibo(x - 2);
}
```

Type of Recursion

```
// Tail Recursion
void printNum_tail(int n){
    cout << n << endl;
    if(n > 0)
        printNum_tail(n-1);
}
```

```
// Non-tail Recursion
void printNum_nontail(int n){
    if(n > 0){
        printNum_nontail(n-1);
        cout << n << endl;
    }
}
```

Type of Recursion

- Nested Recursion

$$h(n) = \begin{cases} 0 & if\ n = 0 \\ n & if\ n > 4 \\ h(2 + h(2n)) & if\ n \leq 4 \end{cases}$$

```
// nested recursion
int func(int n){
    if(n ==0) return 0;
    if(n > 4) return n;
    return
        func(2+ func(2*n));
}
```

Examples

- Write a program in C to count the digits of a given number using recursion
- E.g: n = 5413 → output: 4

Examples

- Write a program in C to count the digits of a given number using recursion

```
int countDigits(int n) {
    if (n == 0) {
        return 0;
    } else {
        return 1 + countDigits(n / 10);
    }
}
```

Examples

- Write a program in C to convert a decimal number to binary using recursion

Examples

- Write a program in C to convert a decimal number to binary using recursion

```
void decimalToBinary(int decimal) {
    if (decimal == 0) {
        return;
    } else {
        decimalToBinary(decimal / 2);
        cout << decimal % 2;
    }
}
```

Examples

- Count the number of items in a linear linked list recursively

Examples

- Count the number of items in a linear linked list

```
int countItems(Node* head) {
    if (head == NULL) {
        return 0;
    } else {
        return 1 + countItems(head->next);
    }
}
```

Examples

- Delete all nodes in a linear linked list by recursion

Examples

- Delete all nodes in a linear linked list by recursion

```
void removeAll(Node*& head) {
    if (head == NULL) {
        return;
    }
    Node* nextNode = head->next;
    delete head;
    head = NULL;
    removeAll(nextNode);
}
```

Examples

- What is the output of the following program as calling **`watch(-7)`**

```
int watch(int n) {
    if (n > 0)
        return n;
    cout << n <<endl;
    return watch(n+2)*2;
}
```

Removing Recursion

- “Never hire a developer who computes the factorial using Recursion”
- Complex Recursion that is hard to understand should probably be considered a "bad smell" in the code and a good candidate to be replaced with Iteration
- There are 2 techniques to remove recursion
  - **Iteration**
  - Stacking

Replacing Recursion with Iteration

- The Simple Method:
  - Convert all recursive calls into tail calls
  - Introduce a loop around the function body
  - Convert tail calls into continue statements
  - Tidy up
- Mechanics:
  - Determine the base case of the Recursion
  - Implement a loop that will iterate until the base case is reached
  - Make a progress towards the base case

Replacing Recursion with Iteration

```
int Fact(int n){
  if(n < 2)
     return 1;
  return n * Fact(n-1);
}
```

Non-tail recursion

```
int Fact(int n, int acc=1){
  if(n < 2)
     return 1 * acc;
  return Fact(n-1, acc * n);
}
```

Convert to Tail recursion

Replacing Recursion with Iteration

```
int Fact(int n, int acc=1){
  while(true)
  {
     if(n < 2)
        return 1 * acc;
     (n, acc) = (n-1, acc * n);
     continue;
  }
}
```

Introduce a loop

Replace recursive call by the original function's agurment list

Replacing Recursion with Iteration

```
int Fact(int n, int acc=1){
  while(n > 1)
  {
     n = n - 1;
     acc = acc * n;
  }
  return acc;
}
```

Tidy up!

Example

- Merge two sorted linear linked lists, keeping the result sorted
  - With recursion
  - Without recursion

Example

- Merge two sorted linear linked lists, keeping the result sorted, recursively

```
Node* mergeTwoLists(Node* head1, Node* head2) {
    if (head1 == NULL) {
        return head2;
    }
    if (head2 == NULL) {
        return head1;
    }

    Node* merged;
    if (head1->data < head2->data) {
        merged = head1;
        merged->next = mergeTwoLists(head1->next, head2);
    } else {
        merged = head2;
        merged->next = mergeTwoLists(head1, head2->next);
    }
    return merged;
}
```

Example

- Merge two sorted linear linked lists, keeping the result sorted without recursion

```
Node* mergedHead = NULL;
if (head1->data <= head2->data) {
    mergedHead = head1;
    head1 = head1->next;
} else {
    mergedHead = head2;
    head2 = head2->next;
}
```

```
Node* current = mergedHead;
while (head1 != NULL && head2 != NULL) {
    if (head1->data <= head2->data) {
        current->next = head1;
        head1 = head1->next;
    } else {
        current->next = head2;
        head2 = head2->next;
    }
    current = current->next;
}

if (head1 != NULL) {
    current->next = head1;
} else {
    current->next = head2;
}
```

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems (minimize/maximize)
- DP can be used when the solution to a problem may be viewed as the result of a **sequence of decisions**
- DP reduces computation by
  - Solving subproblems in a **bottom-up** fashion.
  - **Storing** solution to a subproblem the first time it is solved.
  - **Looking up** the solution when subproblem is encountered again
- Key: determine structure of optimal solutions

- Define the problem and identify its optimal structure
  - Optimal structure: the optimal solution to the problem can be obtained by combining the optimal solutions to its subproblems
- Formulate a recursive solution (top-down)
- Compute the value of an optimal solution in a bottom-up fashion
  - Memorize the recursive solutions by storing the results of previous computation in a table.
  - Convert the recursive solution to an iterative one

Example

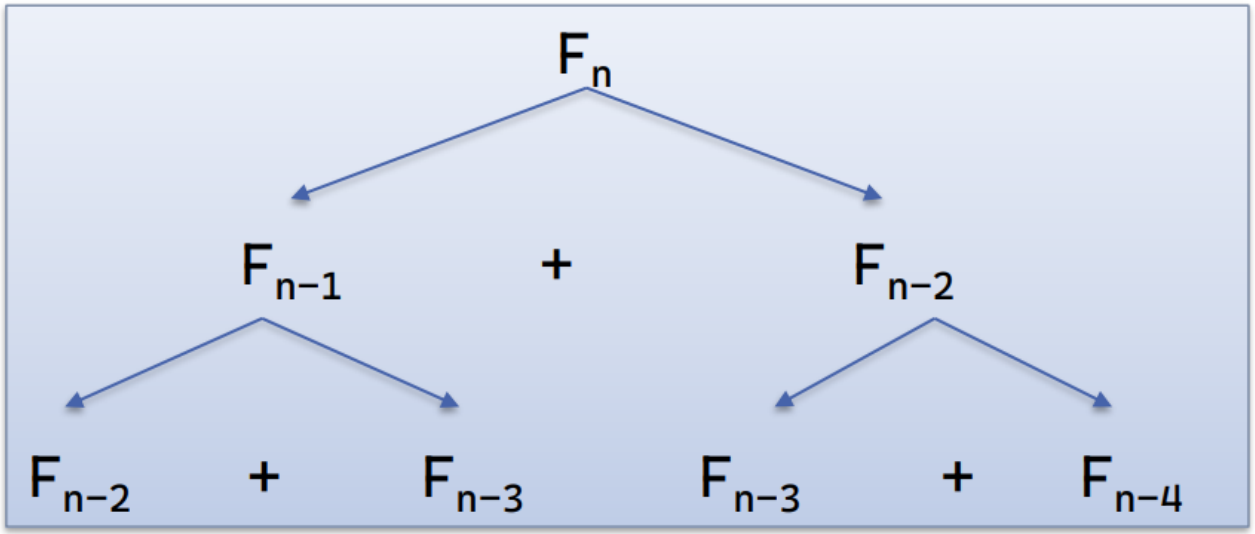
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

$$F_i = i \qquad \text{if } i \leq 1$$

$$F_i = F_{i-1} + F_{i-2} \qquad \text{if } i > 1$$

Solved by a recursive program:

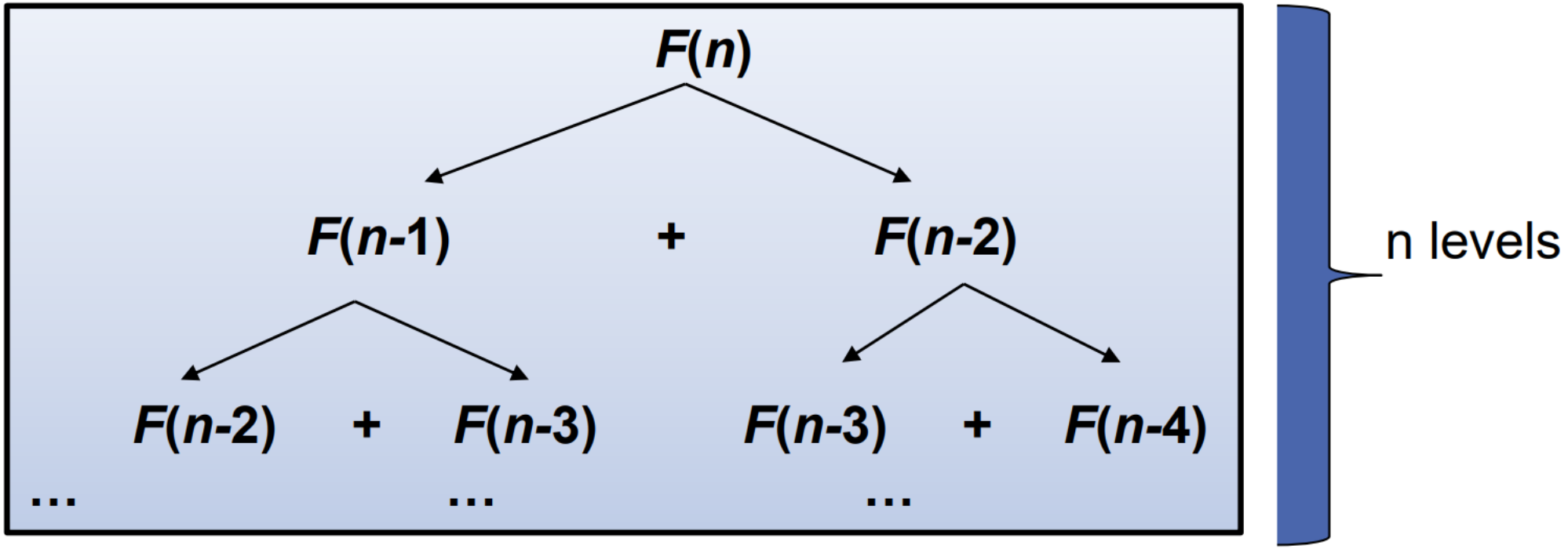
```
int Fib(int n)
{
  if (n <= 1)
    return n;
  else
    return Fib(n - 1)
         + Fib(n - 2);
}
```



→ *This is a **top-down** approach*

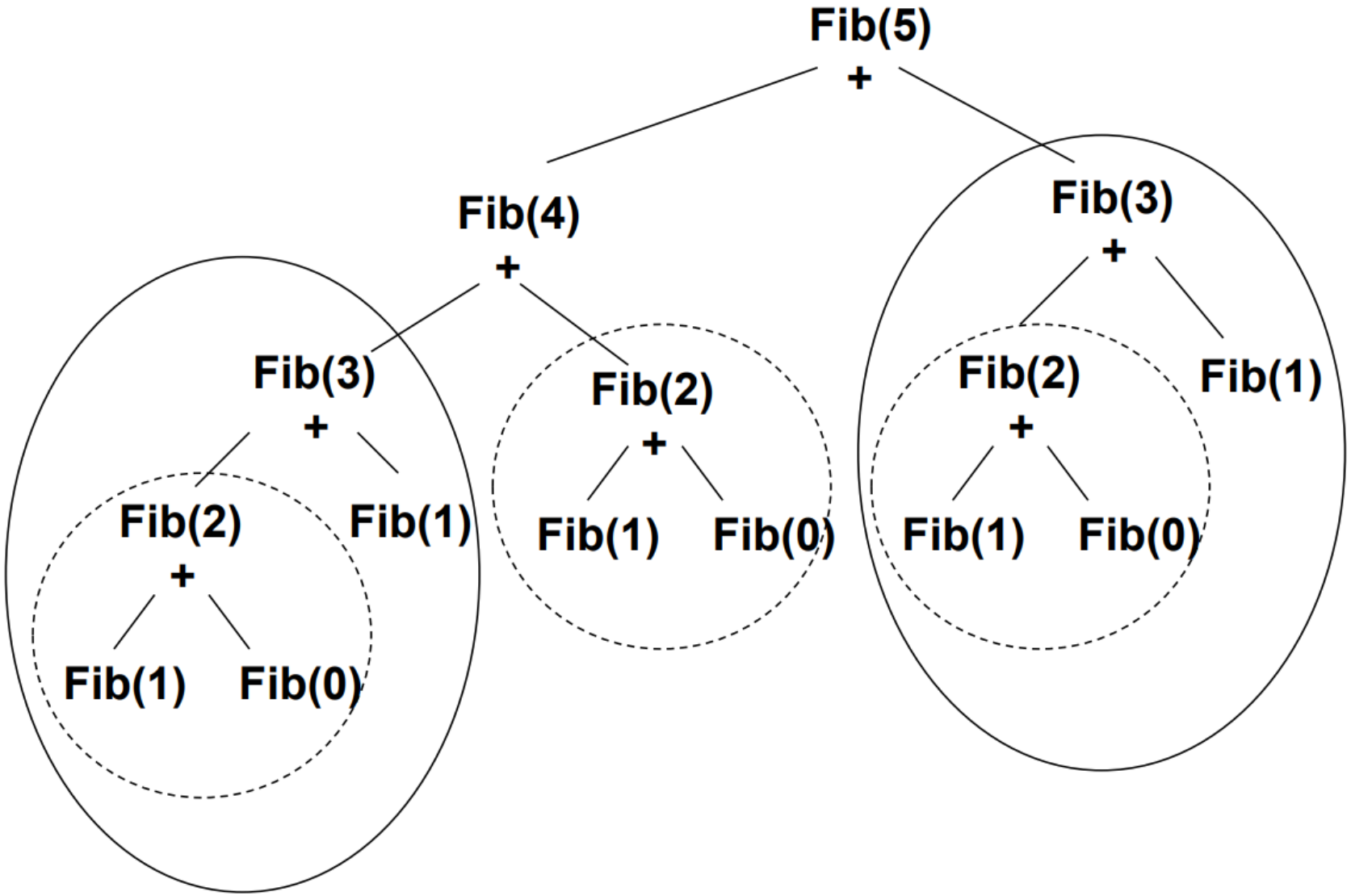
Example

- Why is the top-down so inefficient?
  - Recomputes many sub-problems.
    - How many times is `F(n-5)` computed?



- We can enhance this problem by storing solution to the sub-problem

Example


Fib(5)
+
Fib(4)
+
Fib(3)
+
Fib(2)
+
Fib(1)
Fib(0)
Fib(1)
Fib(2)
+
Fib(1)
Fib(0)
Fib(2)
+
Fib(1)
Fib(0)
Fib(3)
+
Fib(2)
+
Fib(1)
Fib(1)
Fib(0)

Example

- Using a ***bottom-up*** approach can solve this problem!
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

0	1	1	…	F(n-2)	F(n-1)	F(n)

Example

```
#include <iostream>
int Fib_DP(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int *f = new int[n+1];//1 extra to handle case, n
  int i;
  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)
     f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

Example

- Implement the Fibonacci number problem without using the external space

Example

- Write Fibonacci number without using the external space

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int a = 0, b = 1, sum = 0;
    for (int i = 2; i <= n; ++i) {
        sum = a + b;
        a = b;
        b = sum;
    }
    return sum;
}
```

The Knapsack problem

- Problem statement:
  - A thief is robbing a museum and he only has a single knapsack to carry all the items he steals.
  - The knapsack has a capacity for the amount of weight it can hold. Each item in the museum has a weight and a value associated with it

The Knapsack problem – variantion

- 0/1 Knapsack problem
  - Each item is chosen at most once.
  - Decision variable for each item is a binary value (0 or 1)
- Multiple-choice Knapsack problem
  - Each item can be put to the knapsack multiple times.
  - Decision variable for each item is an integer value.
- Bounded Knapsack problem
  - Same with multiple-choice but each item has the max number of times it can be chosen.
- Knapsack problem with fractional items
- Knapsack problem with multiple constraint
- …

0/1 Knapsack problem

- Knapsack's capacity: 10kg
- 5 items can be chosen:
  - Item 1: \$6 (2 kg)
  - Item 2: \$10 (2 kg)
  - Item 3: \$12 (3 kg)
  - Item 4: \$16 (4kg)
  - Item 5: \$20 (5kg)

- Optimal function: $f(n, W)$ $(n = 5, W = 10)$

0/1 Knapsack problem

- Optimal structure: to find $f(n, W)$:
  1. **Case 1:** including the n^{th} item

     → find $f(n-1, W-w_n) + x_n$ with x_n is the value of item n^{th}
  2. **Case 2:** not include the n^{th} item

     → find $f(n-1, W)$
- Hence, optimal f is calculated by:

$$\boldsymbol{f(n, W) = \max\left(f(n-1, W-w_n) + x_n, f(n-1, W)\right)}$$

→ *This can be solved using* ***recursion*** *which is a* ***top-down*** *strategy.*

0/1 Knapsack problem

```
//Returns the maximum value that can be put in a knapsack of
//capacity W
int KnapSack(int n, int wt[], int val[], int W) {
  if (n == 0 || W == 0) // Base Case
     return 0;
 // If weight of the nth item is more than Knapsack capacity W,
 //then this item cannot be included in the optimal solution
  if (wt[n - 1] > W)
     return KnapSack(n - 1, wt, val, W);

  // else: Return the maximum of two cases:
  // (1) nth item included    // (2) not included
  return max( val[n-1] + KnapSack(n-1, wt, val, W-wt[n-1]) ,
                         KnapSack(n-1, wt, val, W) );
}
```

$f(n-1, W-w_n) + x_n$

$f(n-1, W)$

0/1 Knapsack problem

- We can also use a ***bottom-up*** approach and memorize the solutions to subproblems to a table → *Dynamic Programming*
  - Row: items
  - Column: remaining weight capacity of the knapsack
  - We fill the table using the following recurrence relation:

$$f(W,i) = \max\left(f(i-1, W-w_i) + x_i, f(i-1, W)\right)$$

	0kg	1kg	2kg	3kg	4kg	5kg	6kg	7kg	8kg	9kg	10kg
1											
2											
3											
4											
5											

0/1 Knapsack problem

- Use only item 1:

→ $f(1,1) = 0, f(1,2) = 6, f(1,3) = 6, \dots, f(1,10) = 6$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10										
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1 & 2:

→ $f(2,2) = \max(f(1,0) + 10, f(1,2)) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10								
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1 & 2:

→ $f(2,3) = \max(f(1,1) + 10, f(1,3)) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10							
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1 & 2:

→ $f(2,4) = \max(f(1,2) + 10, f(1,4)) = 16, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16						
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1 & 2:

→ $f(2,i) = \max(f(1,W-2)+10, f(1,W))$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,3) = \max(f(2,0) + 12, f(2,3)) = 12$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12							
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,4) = \max(f(2,1) + 12, f(2,4)) = 16$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16						
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,5) = \max(f(2,2) + 12, f(2,5)) = 22$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22					
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3:

→ $f(3,i) = \max(f(2,W-3)+12, f(2,W))$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3, 4:

→ $f(4,i) = \max(f(3,W-4)+16, f(3,W))$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20										

0/1 Knapsack problem

- Use item 1, 2, 3, 4, 5:

→ $f(5,10) = \max(f(4,5) + 20, f(4,10)) = 42$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

0/1 Knapsack problem

- Solution:
  - item 5 + item 3 + item 2 → $42 – 10kg

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

0/1 Knapsack problem

```
int KnapSack(int n, int wt[], int val[], int W)
{
  int i, w;
  //Create a table K to store solutions of subproblems
  int** K = new int*[n + 1];
  for(i = 0; i <= n; i++)
    K[i] = new int[W + 1];
  for (i = 0; i <= n; i++) //Build table K[][] in bottom up manner
    for (w = 0; w <= W; w++) {
      if (i==0 || w==0)
          K[i][w] = 0;
      else if (wt[i-1] <= w)
          K[i][w] = K[i-1][w];
      else
          K[i][w] = max(K[i-1][w-wt[i-1]] + val[i-1], K[i-1][w] );
    }
  return K[n][W];
}
```

$f(i-1, W-w_i) + x_i$

$f(i-1, W)$

Multi-choice Knapsack problem

- Knapsack's capacity: 10kg
- 3 items can be chosen:
  - Item 1: \$5 (3 kg)
  - Item 2: \$7 (4 kg)
  - Item 3: \$8 (5 kg)

→ 1 item can be picked many times

- Optimal function: $f(n, W)$ (n=3, w=10)

Multi-choice Knapsack problem

- Optimal $f(i, w)$ is calculated by:

$$f(i, W) = \max\left(f(i-1, W - kw_i) + kx_i, f(i-1, W)\right)$$

k item *i* + optimum combination of weight $w - kw_i$

NO Item *i* + optimum combination items 1 to $i - 1$

- k is the number of times item i appears in the knapsack. $k = 1,2, \ldots$ so that $kw_i \leq W$

Multi-choice Knapsack problem

- Use only item 1:

→ $f(1,3) = 5, f(1,6) = 10, f(1,9) = 15$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7											
3	5kg	$8											

$1 \times x_1$ $2 \times x_1$ $3 \times x_1$

Multi-choice Knapsack problem

- Use only item 1 & 2:

→ $f(2, W) = \max\big(f(1, W - 4k) + 7k, f(1, W)\big)$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	$8											

Multi-choice Knapsack problem

- Use item 1, 2, and 3:

→ $f(3, W) = \max\big(f(2, W - 5k) + 8k, f(2, W)\big)$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	$8	0	0	0	5	7	8	10	12	14	15	17

Multi-choice Knapsack problem

- Solution:
  - item 2 + 2 x item 1 → $17 – 10kg

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	$8	0	0	0	5	7	8	10	12	14	15	17

Multi-choice Knapsack problem

- Solution:
  - item 2 + 2 x item 1 → $17 – 10kg

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	$8	0	0	0	5	7	8	10	12	14	15	17

Exercise

- Write a C++ program to find the length of Longest Increasing Subsequence by dynamic programming

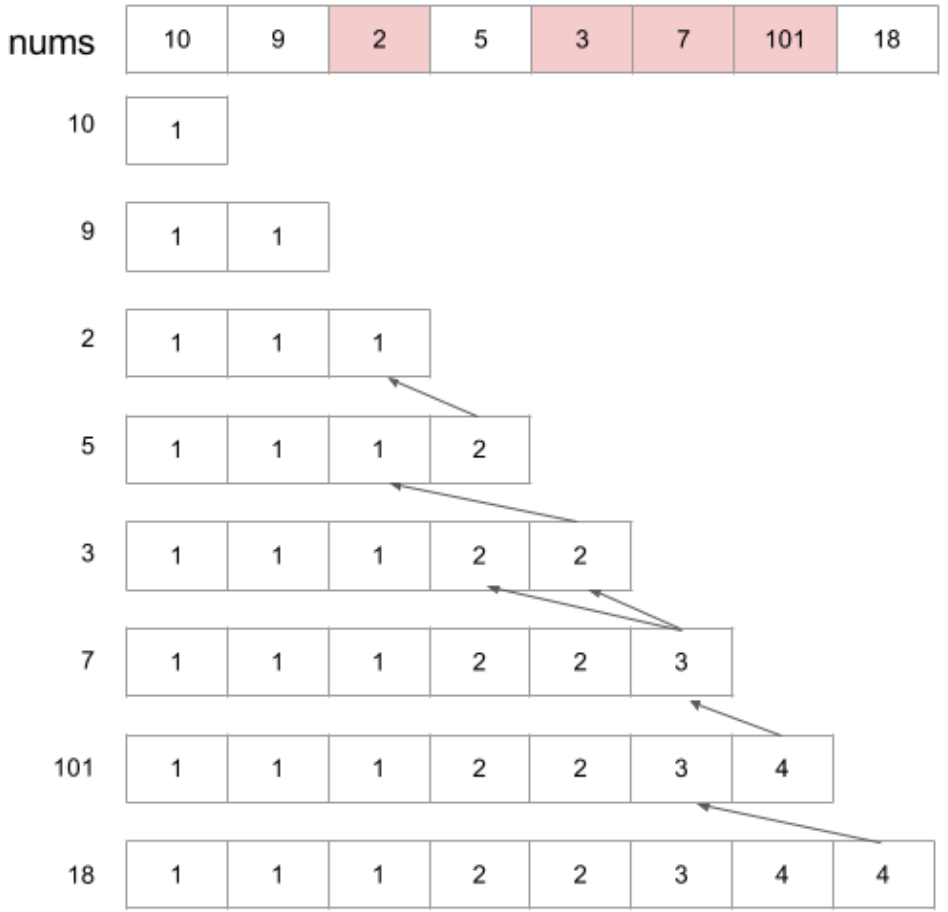
Exercise

- Write a C++ program to find the length of Longest Increasing Subsequence by dynamic programming

```
def: f[i] = length of LIS ends with nums[i]
(nums[i] must be used)

for i = 0 to n-1
  for j = 0 to i
    if nums[i] > nums[j]
      f[i] = max(f[i], f[j] + 1)

ans = max(f)
```



THANK YOU
for YOUR ATTENTION