


## SCC0602 - Algoritmos e Estruturas de Dados I

---

### Design Technique and Dynamic Programming



Professor: André C. P. L. F. de Carvalho, ICMC-USP  
 PAE: Rafael Martins D'Addio  
 Monitor: Joao Pedro Rodrigues Mattos

## Today

- Dynamic programming
  - Fibonacci numbers example
  - Optimization problems
  - Weighted Interval Scheduling Problem (WISP)
  - Principles of dynamic programming
  - Dynamic Time Warping (DTW)

© André de Carvalho - ICMC/USP 2

## Algorithm design techniques

- Algorithm design techniques so far:
  - Iterative (brute-force) algorithms
    - For example, insertion sort
  - Algorithms that use other Abstract Data Types (implemented using efficient data structures)
    - For example, heap sort
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort

3

## Divide and Conquer

- *Divide and conquer* method for algorithm design:
  - **Divide:** If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - **Conquer:** Use divide and conquer recursively to solve the subproblems
  - **Combine:** Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

4

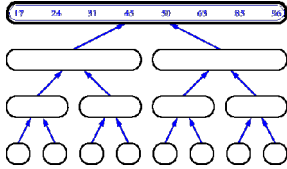
## Divide and Conquer

- For example, **MergeSort**

```

Merge-Sort(A, p, r)
  if p < r then
    q ← (p+r)/2
    Merge-Sort(A, p, q)
    Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
    
```
















- The subproblems are independent and non-overlapping



5

## Fibonacci Numbers

- *Leonardo Fibonacci (1202):*
  - A rabbit starts producing offspring on the second generation after its birth and produces one child each generation
  - How many rabbits will there be after  $n$  generations?

F(1)=1	F(2)=1	F(3)=2	F(4)=3	F(5)=5	F(6)=8
					
					
					
					

6

### Fibonacci Numbers

- $F(n) = F(n-1) + F(n-2)$
- $F(0) = 0, F(1) = 1$ 
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

```

FibonacciR(n)
01 if n ≤ 1 then return n
02 else return FibonacciR(n-1) + FibonacciR(n-2)
    
```

- Straightforward recursive procedure is slow!
- Why? How slow?
- Let's draw the recursion tree

7

### Fibonacci Numbers

- We keep calculating the same value over and over!
  - Subproblems are overlapping – they share sub-subproblems

8

### Fibonacci Numbers

- How many summations are there  $\mathcal{S}(n)$ ?
  - $\mathcal{S}(n) = \mathcal{S}(n-1) + \mathcal{S}(n-2) + 1$
  - $\mathcal{S}(n) \geq 2\mathcal{S}(n-2) + 1$  and  $\mathcal{S}(1) = \mathcal{S}(0) = 0$
  - Solving the recurrence we get  $\mathcal{S}(n) \geq 2^{n/2} - 1 \approx 1.4^n$
- Running time is *exponential!*

9

### Fibonacci Numbers

- We can calculate  $F(n)$  in *linear* time by remembering solutions to the solved subproblems – *dynamic programming*
- Trade space for time!

```

Init_vector(F[], -1)

FibonacciR(n)
if n ≤ 1 then return n
else if F[n] != -1 then return F[n]
else
    F[n] = FibonacciR(n-1) + FibonacciR(n-2)
    return F[n]
    
```

10

### Fibonacci Numbers

- Iterative alternative
  - Compute solution in a bottom-up fashion

```

Fibonacci(n)
F[0] ← 0
F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
return F[n]
    
```

11

### Fibonacci Numbers

- In fact, only two values need to be remembered at any time!

```

FibonacciImproved(n)
if n ≤ 1 then return n
Fim2 ← 0
Fim1 ← 1
for i ← 2 to n do
    Fi ← Fim1 + Fim2
    Fim2 ← Fim1
    Fim1 ← Fi
return Fi
    
```

12

## History

- Dynamic programming
  - Invented in the 1950s by *Richard Bellman* as a general method for optimizing multistage decision processes
  - "Programming" stands for "planning" (not computer programming)

13

## Optimization Problems

- We have to choose one solution out of many – one with the optimal (minimum or maximum) value.
- A solution exhibits a structure
  - It consists of a string of choices that were made – what choices have to be made to arrive at an optimal solution?
- An algorithm should compute the optimal value plus, if needed, an optimal solution

14

## Weighted Interval Scheduling Problem (WISP)

- Weighted Interval Scheduling Problem:
  - Select a subset of intervals with the highest weight sum possible without them overlapping

15

## Weighted Interval Scheduling Problem (WISP)

- Suppose we have the intervals ordered by finishing time.

16

## Weighted Interval Scheduling Problem (WISP)

- Suppose we have the intervals ordered by finishing time.
- And we have defined  $\rho(j)$  as the highest index  $i < j$  such as  $i$  and  $j$  are **disjoint**.

Index	1	2	3	4	5	6
$\rho(j)$	0	0	1	0	3	3

17

## Weighted Interval Scheduling Problem (WISP)

- Formally:
  - We can label the intervals as  $1, \dots, n$
  - We are looking for a subset  $S \subseteq \{1, \dots, n\}$  that maximizes  $\sum_{i \in S} V_i$

Index	1	2	3	4	5	6
$\rho(j)$	0	0	1	0	3	3

18

### Weighted Interval Scheduling Problem (WISP)

- We can say some things about  $S$ :
  - The last interval ( $n$ ) can or cannot belong to  $S$
  - If  $n \in S$ , then any interval between  $p(n)+1$  and  $n-1 \notin S$ 
    - $S$  has an optimal solution with intervals  $\{1, \dots, p(n)\}$

### Weighted Interval Scheduling Problem (WISP)

- We can say some things about  $S$ :
  - The last interval ( $n$ ) can or cannot belong to  $S$
  - If  $n \notin S$ , then there is an optimal solution with intervals  $\{1, \dots, n-1\}$

### Weighted Interval Scheduling Problem (WISP)

- Finding an optimal solution within a interval  $\{1, 2, \dots, n\}$  involves finding optimal solutions in a smaller interval  $\{1, 2, \dots, j\}$ .
- Let  $OPT(j)$  be the optimal sum of intervals for  $\{1, 2, \dots, j\}$ . Then:
  - If  $j \in S$ ,  $OPT(j) = v_j + OPT(p(j))$
  - If  $j \notin S$ ,  $OPT(j) = OPT(j-1)$

21

### Weighted Interval Scheduling Problem (WISP)

- Finding an optimal solution within a interval  $\{1, 2, \dots, n\}$  involves finding optimal solutions in a smaller interval  $\{1, 2, \dots, j\}$ .
- Let  $OPT(j)$  be the optimal sum of intervals for  $\{1, 2, \dots, j\}$ . Then:
  - If  $j \in S$ ,  $OPT(j) = v_j + OPT(p(j))$
  - If  $j \notin S$ ,  $OPT(j) = OPT(j-1)$

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j-1))$$

22

### Weighted Interval Scheduling Problem (WISP)

```

Compute-Opt(j)
if j = 0 then return 0
else
return max(v[j] + Compute-Opt(p(j)), Compute-Opt(j-1))
    
```

- What the recursion tree will look like without using dynamic programming?

23

### Weighted Interval Scheduling Problem (WISP)

24

### Weighted Interval Scheduling Problem (WISP)

- The procedure complexity is similar to the Fibonacci example.
  - EXPONENTIAL!
- A solution for this problem is, again, **dynamic programming**
  - Use of *memoization*: storing partial solutions on a global structure

25

### Weighted Interval Scheduling Problem (WISP)

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
            
```

26

Index	$w_i$	$p(i)$
1	2	0
2	4	0
3	4	1
4	7	0
5	2	3
6	1	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
            
```

27

Index	$w_i$	$p(i)$
1	2	0
2	4	0
3	4	1
4	7	0
5	2	3
6	1	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
            
```

28

Index	$w_i$	$p(i)$
1	2	0
2	4	0
3	4	1
4	7	0
5	2	3
6	1	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
            
```

29

Index	$w_i$	$p(i)$
1	2	0
2	4	0
3	4	1
4	7	0
5	2	3
6	1	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
            
```

30

2	4+OPT(0) OPT(1)	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

31

2	4+OPT(0) OPT(1)	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

32

2	4	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

33

2	4	4+OPT(1) OPT(2)			1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

34

2	4	6			1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

35

2	4	6		2+OPT(3) OPT(4)	1+OPT(3) OPT(5)
1	2	3	4	5	6

Index	1	2	3	4	5	6
$K_i$	2	4	4	7	2	1
$p(i)$	0	0	1	0	3	3

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

36

2	4	6	7+OPT(0)	2+OPT(3)	1+OPT(3)
			OPT(3)	OPT(4)	OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

37

2	4	6	7+OPT(0)	2+OPT(3)	1+OPT(3)
			OPT(3)	OPT(4)	OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

38

2	4	6	7	2+OPT(3)	1+OPT(3)
				OPT(4)	OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

39

2	4	6	7	2+OPT(3)	1+OPT(3)
				OPT(4)	OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

40

2	4	6	7	8	1+OPT(3)
					OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

41

2	4	6	7	8	1+OPT(3)
					OPT(5)

Index	$K_i=2$	$K_i=4$	$K_i=7$	$K_i=1$
1				
2				
3				
4				
5				
6				

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

42

Index

1	$v_1 = 2$	$p(1) = 0$
2	$v_2 = 4$	$p(2) = 0$
3	$v_3 = 4$	$p(3) = 1$
4	$v_4 = 7$	$p(4) = 0$
5	$v_5 = 2$	$p(5) = 3$
6	$v_6 = 1$	$p(6) = 3$

```

M-Compute-Opt(j)
if j = 0 then return 0
else if M[j] is not empty then return M[j]
else
  M[j] = max(v[j] + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
  return M[j]
    
```

43

### Weighted Interval Scheduling Problem (WISP)

- The solution with memorization is efficient and requires only  $O(n)$  steps as long as the intervals are sorted.
- The global vector  $M$  not only helps with calculating the solution, but it also can be used to find the intervals within the solution.
- An interval belongs to the solution if and only if
 
$$v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$$

44

### Weighted Interval Scheduling Problem (WISP)

Index

1	$v_1 = 2$	$p(1) = 0$
2	$v_2 = 4$	$p(2) = 0$
3	$v_3 = 4$	$p(3) = 1$
4	$v_4 = 7$	$p(4) = 0$
5	$v_5 = 2$	$p(5) = 3$
6	$v_6 = 1$	$p(6) = 3$

45

### Weighted Interval Scheduling Problem (WISP)

Index

1	$v_1 = 2$	$p(1) = 0$
2	$v_2 = 4$	$p(2) = 0$
3	$v_3 = 4$	$p(3) = 1$
4	$v_4 = 7$	$p(4) = 0$
5	$v_5 = 2$	$p(5) = 3$
6	$v_6 = 1$	$p(6) = 3$

46

### Weighted Interval Scheduling Problem (WISP)

Index

1	$v_1 = 2$	$p(1) = 0$
2	$v_2 = 4$	$p(2) = 0$
3	$v_3 = 4$	$p(3) = 1$
4	$v_4 = 7$	$p(4) = 0$
5	$v_5 = 2$	$p(5) = 3$
6	$v_6 = 1$	$p(6) = 3$

47

### Weighted Interval Scheduling Problem (WISP)

Index

1	$v_1 = 2$	$p(1) = 0$
2	$v_2 = 4$	$p(2) = 0$
3	$v_3 = 4$	$p(3) = 1$
4	$v_4 = 7$	$p(4) = 0$
5	$v_5 = 2$	$p(5) = 3$
6	$v_6 = 1$	$p(6) = 3$

48



### Weighted Interval Scheduling Problem (WISP)

- We can re-write the recursive algorithm with an iterative version
  - In this case: more efficient. Why?

```

Iterative-Compute-Opt()
M[0] = 0
for j = 1, ..., n
    M[j] = max(v[j] + M[p(j)], M[j-1])
    
```

49

### Memoization

- Solve the problem in a top-down fashion, but record the solutions to subproblems in a table.
- Pros and cons:
  - ☹ Recursion is usually slower than loops and uses stack space
  - ☺ Easier to understand
  - ☺ If not all subproblems need to be solved, you are sure that only the necessary ones are solved

50

### Dynamic Programming

- In general, to apply dynamic programming, we have to address a number of issues:
  1. Show **optimal substructure** – an optimal solution to the problem contains within it optimal solutions to sub-problems
    - Solution to a problem:
      - Making a choice out of a number of possibilities (look what possible choices there can be)
      - Solving one or more sub-problems that are the result of a choice (characterize the space of sub-problems)
    - Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal (use "cut-and-paste" argument)

51

### Dynamic Programming

- 2. Write a recurrence for the value of an optimal solution
  - $M_{opt} = \text{Min}_{\text{over all choices } k} \{(\text{Combination (e.g., sum) of } M_{opt} \text{ of all sub-problems, resulting from choice } k) + (\text{the cost associated with making the choice } k)\}$
  - Show that the number of different instances of sub-problems is bounded by a polynomial

October 23, 2003 52

### Dynamic Programming

- 3. Compute the value of an optimal solution in a bottom-up fashion, so that you always have the necessary sub-results pre-computed (or use memoization)
  - See if it is possible to reduce the space requirements, by "forgetting" solutions to sub-problems that will not be used any more
- 4. Construct an optimal solution from computed information (which records a sequence of choices made that lead to an optimal solution)

October 23, 2003 53

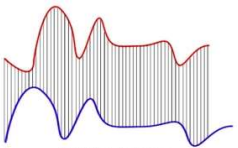
### Dynamic Time Warping

- Given two distinct time series, how can we compare them?
- Using a traditional distance metric?
  - Euclidean?

© André de Carvalho - ICMC/USP 54

### Dynamic Time Warping

- Given two distinct time series, how can we compare them?
- Using a traditional distance metric?
  - Euclidean?

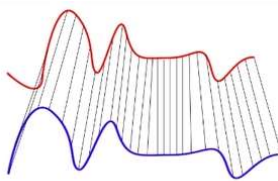


Euclidean Matching

© André de Carvalho - ICMC/USP 55

### Dynamic Time Warping

- Dynamic Time Warping!



Dynamic Time Warping Matching

© André de Carvalho - ICMC/USP 56

### Dynamic Time Warping

- Match every possible point within two series and select the best solution possible
  - Warp one of the series so it can match the other
  - The best result is the one that yields the lowest "score" or "distance"

© André de Carvalho - ICMC/USP 57

### Dynamic Time Warping

- Recurrence function
 
$$DTW(x_i, y_j) = c(x_i, y_j) + \min \begin{cases} DTW(x_{i-1}, y_{j-1}) \\ DTW(x_i, y_{j-1}) \\ DTW(x_{i-1}, y_j) \end{cases}$$
- The cost  $c$  refers to a distance metric between two points
  - Such as Euclidean:
  - $c(x_i, y_i) = \sqrt{(x_i - y_i)^2} = |x_i - y_i|$

© André de Carvalho - ICMC/USP 58

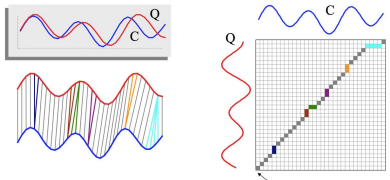
### Dynamic Time Warping

- Memoization
  - Fills a  $x$  vs  $y$  matrix
  - The final DTW distance is the  $n$ th position in both row and column
    - Which represents the end of both series

© André de Carvalho - ICMC/USP 59

### Dynamic Time Warping

- Warping path
  - The path obtained by greedily going through the matrix from  $c(x_n, y_n)$  to  $c(x_1, y_1)$  selecting the smallest distance among the possible
  - Represent the matching between the two time series



Warping path  $w$

© André de Carvalho - ICMC/USP 60

## Next Lecture

- Hashing
- Graphs:
  - Representation in memory
  - Breadth-first search
  - Depth-first search
  - Topological sort


October 23, 2003 61

## Acknowledgement

- A large part of this material were adapted from
  - Simonas Šaltenis, Algorithms and Data Structures, Aalborg University, Denmark
  - Mary Wootters, Design and Analysis of Algorithms, Stanford University, USA
  - George Bebis, Analysis of Algorithms CS 477/677, University of Nevada, Reno
  - David A. Plaisted, Information Comp 550-001, University of North Carolina at Chapel Hill
  - Gustavo E. A. P. A. Batista, Slides on Dynamic Programming, University of São Paulo, Brazil

© André de Carvalho - ICMC/USP 62

## Questions



© André de Carvalho - ICMC/USP 63