ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

M.Sc. Program in Computer Science
Department of Informatics

Design and Analysis of Algorithms

Basic Algorithmic Techniques

Vangelis Markakis

markakis@gmail.com

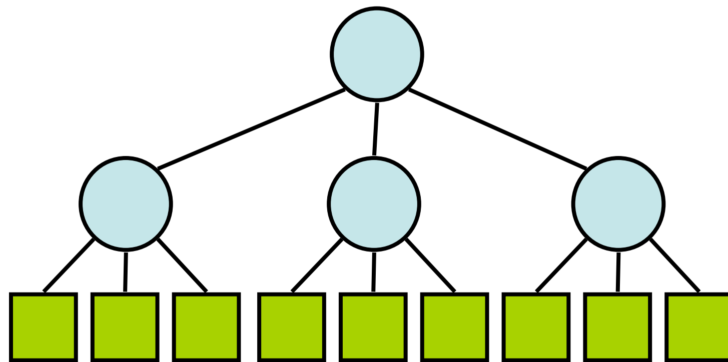
Basic Algorithmic Techniques Content

- Divide and Conquer algorithms
  - Multiplication of numbers, Mergesort, Quicksort
  - Recurrence relations and the Master theorem
  - Variations: Decrease and Conquer
- Greedy Algorithms
  - The general approach
  - Greedy algorithms for Interval Scheduling
  - Gas station problems
- Dynamic Programming
  - Weighted Interval Scheduling
  - Maximum Sub-array

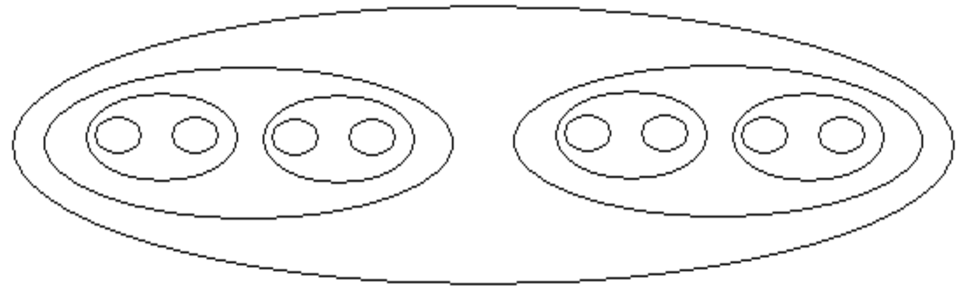
Divide and Conquer algorithms, Recurrence relations, and the Master theorem

Divide & Conquer

- Divide (recursively) the problem into smaller subproblems, of about the same size
- End of recursion: Solve the (appropriately small) subproblems usually in constant time
- Combine (recursively) the solutions to the subproblems until we reach the initial problem



Recursion tree



Analysis: Through recurrence relations

Integer Multiplication

First a useful insight!

- Let a, b, c, d be 4 real numbers
- Suppose we want to compute the product of two complex numbers (a + bi) (c + di)
- We trivially have: (a + bi) (c + di) = ac - bd + (ad +bc)i
- 4 multiplications suffice

A (not so useful at first sight) observation:

Gauss equation : $ad + bc = (a+b)(c+d) - ac - bd$

- 3 multiplications suffice!

Integer Multiplication

INTEGER MULTIPLICATION

I: 2 n-bit numbers, x and y

Q: Compute their product

Back to elementary school:

```
          **********
x         **********
          ----------
          **********
         **********
        **********
       **********
      **********
     **********
    **********
   **********
  *********
 *********
---------------------
*****************
```

Integer Multiplication

Complexity?

```
                **********
x               **********
               -----------
                **********
               **********
              **********
             **********
            **********
           **********
          **********
         **********
         *********
        *********
-----------------------------
     *******************
```

- O(n) to compute each of the n terms that are to be added
- Adding 2 n-bit numbers takes O(n)

Complexity = $\Theta(n^2)$

- Can we do better?

Integer Multiplication

- Divide and Conquer approach
- For simplicity, suppose that n is a power of 2
- Minor modifications if not true

	n/2 bits	n/2 bits	
x =	a	b	$= a\,2^{n/2} + b$
y =	c	d	$= c\,2^{n/2} + d$

For example, if x = 10011010, then x = $(1001)\,2^4 + 1010$

$$x\,y = (a\,2^{n/2} + b)\,(c\,2^{n/2} + d) = a\,c\,2^n + (a\,d + b\,c)\,2^{n/2} + b\,d$$

Integer Multiplication

```
Algorithm IntMult1(x,y)
if n=1 return xy
a = n/2 leftmost bits of x, b = n/2 rightmost bits of x
c = n/2 leftmost bits of y, d = n/2 rightmost bits of y
P1 = IntMult1(a,c), P2 = IntMult1(a,d)
P3 = IntMult1(b,c), P4 = IntMult1(b,d)
return P1 2^n + (P2 + P3) 2^(n/2) + P4
```

Complexity:

4 recursive calls for multiplying numbers with n/2 bits $T(n/2)$

2 multiplications with powers of 2 $O(n)$

3 additions of n-bit numbers $O(n)$

Integer Multiplication

Complexity:

4 recursive calls for multiplying numbers with n/2 bits T(n/2)

2 multiplications with powers of 2 O(n)

3 additions of n-bit numbers O(n)

$$T(n) = \begin{cases} 1, & \text{if n} = 1 \\ 4T(n/2) + O(n), & \text{if n} > 1 \end{cases} \Rightarrow \mathrm{T}(n) = O(n^2)$$

No progress at all!!

Integer Multiplication

	n/2 bits	n/2 bits	
x =	a	b	$= a\,2^{n/2} + b$
y =	c	d	$= c\,2^{n/2} + d$

$$x\,y = (a\,2^{n/2} + b)\,(c\,2^{n/2} + d) = a\,c\,2^{n} + (a\,d + b\,c)\,2^{n/2} + b\,d$$

Recall the observation of Gauss:

$$ad + bc = (a+b)(c+d) - ac - bd$$

$$
\begin{aligned}
x\,y &= ac\,2^{n} + (ad + bc)\,2^{n/2} + bd \\
&= ac\,2^{n} + [(a+b)(c+d) - ac - bd]\,2^{n/2} + bd \\
&= P_1\,2^{n} + [P_3 - P_1 - P_2]\,2^{n/2} + P_2
\end{aligned}
$$

$P_1 = ac$

$P_2 = bd$

$P_3 = (a+b)(c+d)$

Integer Multiplication

```
Algorithm IntMult2(x,y) [Karatsuba 1962]
if n=1  return xy
a = n/2 leftmost bits of x, b = n/2 rightmost bits of x
c = n/2 leftmost bits of y, d = n/2 rightmost bits of y
P1 = IntMult2(a,c), P2 = IntMult2(b,d)
P3 = IntMult2((a+b),(c+d))
return P1 2^n + (P3 - P1 - P2)2^(n/2)  + P2
```

Complexity:

3 recursive calls for multiplying numbers with n/2 bits $T(n/2)$

2 multiplications with powers of 2 $O(n)$

6 additions of n-bit numbers $O(n)$

Integer Multiplication

Complexity:

3 recursive calls for multiplying numbers with n/2 bits T(n/2)

2 multiplications with powers of 2 O(n)

6 additions of n-bit numbers O(n)

$$T(n) = \begin{cases} 1, & \text{if n} = 1 \\ 3T(n/2) + O(n), & \text{if n} > 1 \end{cases} \Rightarrow \mathrm{T}(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Much better than before!!

The Master Theorem

- How do we analyze recurrence relations?
- There are various methods
- The substitution method:
  - Keep substituting until you guess the solution
  - Use induction to prove it formally

Example: $T(n) = T(n-1) + n$, $T(1) = 1$

- $T(n) = T(n-1) + n$
- $= (T(n-2) + n-1) + n$
- $= T(n-2) + n + n-1$
- $= (T(n-3) + n-2) + n + n-1$
- $= \dots$
- $= n + n-1 + n-2 + \dots + 2 + 1 = O(n^2)$

Is there a general result that could be applicable to the recurrence relations we will encounter?

The Master Theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \quad (b^d > a) \\ O(n^d \log_b n), & \text{if } d = \log_b a \quad (b^d = a) \\ O(n^{\log_b a}), & \text{if } d < \log_b a \quad (b^d < a) \end{cases}$$

- Usually convenient to think of n as a power of b, so that n/b is an integer.
- In many cases of interest, b = 2
- More general versions of this theorem are available as well

The Master Theorem - Examples

- Naive integer multiplication
  - T(n) = 4T(n/2) + O(n)
  - a = 4, b = 2, $\log_b a = \log_2 4 = 2$
  - d = 1 < 2 = $\log_b a$
  - Case (iii) applies: $T(n) = O\left(n^{\log_b a}\right) = O(n^2)$
- Karatsuba's algorithm for integer multiplication
  - T(n) = 3T(n/2) + O(n)
  - a = 3, b = 2, $\log_b a = \log_2 3 = 1.59$
  - d = 1 < $\log_b a$
  - Case (iii) applies again: $T(n) = O\left(n^{\log_b a}\right) = O(n^{1.59})$

The Master Theorem - Examples

- $T(n) = 5T(n/25) + O(n^2)$
  - $a = 5$, $b = 25$, $\log_b a = \log_{25} 5 = 0.5$
  - $d = 2 > 0.5 = \log_b a$
  - case (i) applies: $T(n) = O\left(n^d\right) = \mathrm{O}(\mathrm{n}^2)$
- $T(n) = T(2n/3) + O(1)$
  - $a = 1$, $b = 3/2$, $\log_b a = \log_{3/2} 1 = 0$
  - $d = 0 = \log_b a$
  - case (ii) applies: $T(n) = O\left(n^0 \log_{3/2} n\right) = \mathrm{O}(\log \mathrm{n})$
- $T(n) = 9T(n/3) + O(n)$
  - $a = 9$, $b = 3$, $\log_b a = \log_3 9 = 2$
  - $d = 1 < 2 = \log_b a$
  - case (iii) applies: $T(n) = O\left(n^{\log_b a}\right) = \mathrm{O}(\mathrm{n}^2)$

Sorting Problems

SORTING

I: An array with n numbers, A[1..n]

Q: The array sorted in increasing order

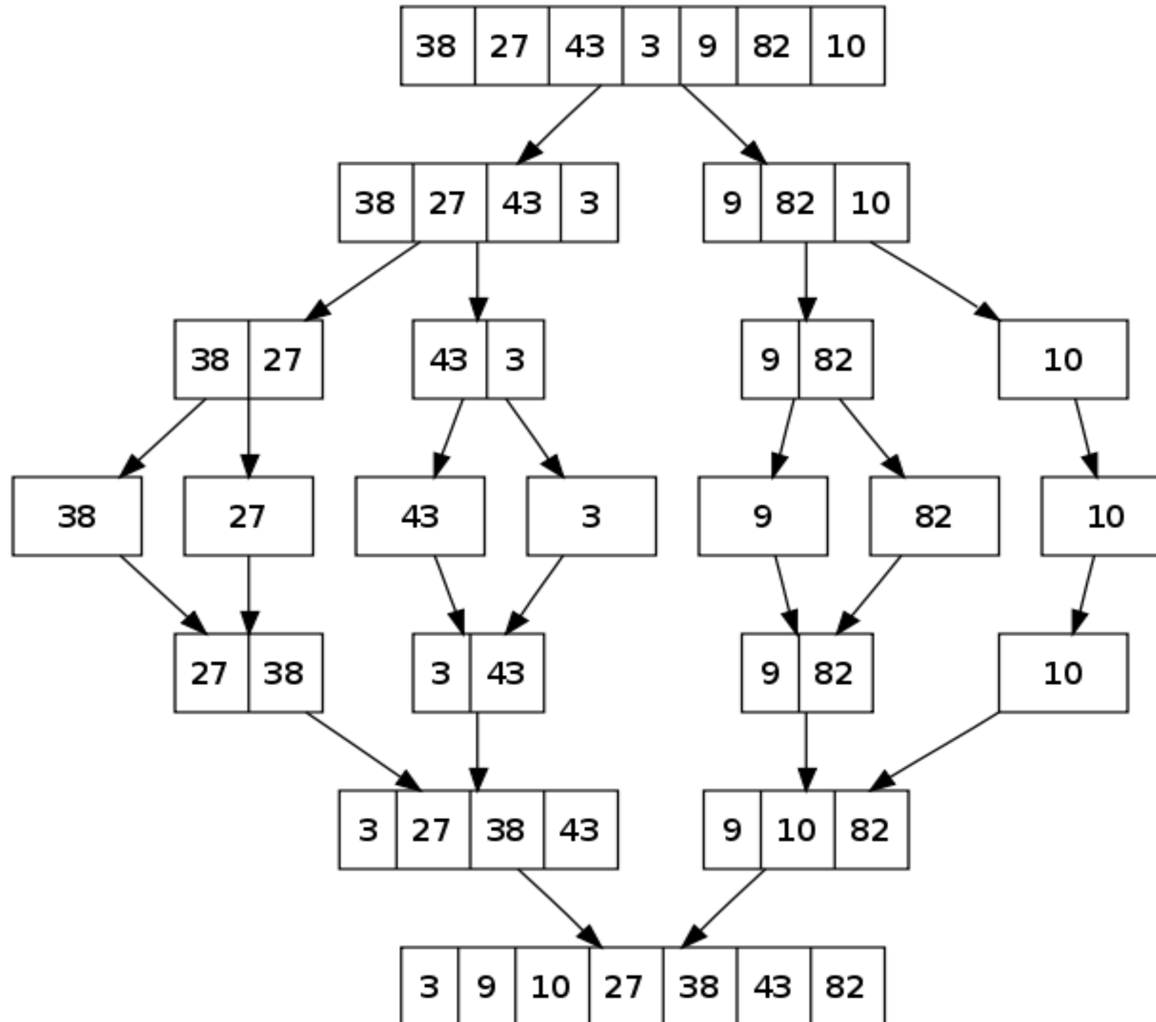
- The input may also be a segment of the array A[p..r]
- One of the most basic problems in Computer Science
- Hundreds of articles mainly in the 60s and 70s on sorting
- Most commong algorithms: Bubblesort, Insertionsort, Selectionsort, Shellsort, Mergesort, Quicksort, Heapsort

The Mergesort Algorithm

- The most natural idea for a recursive sorting algorithm
- Divide the problem into 2 subproblems
- Recursively sort the subproblems
- Merge the 2 sorted solutions into one sorted array

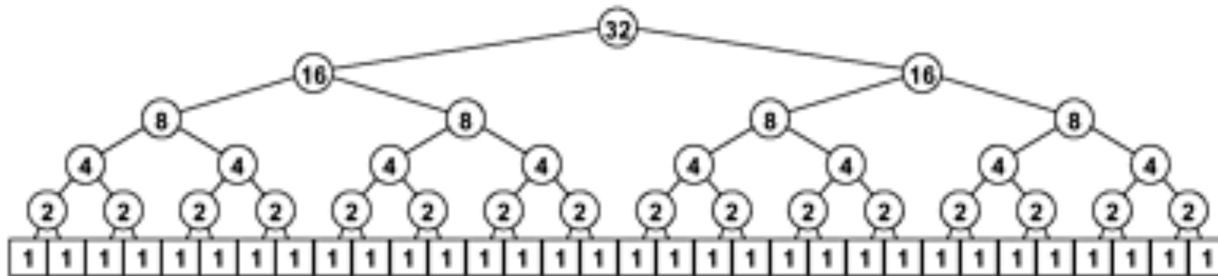
```
Algorithm Mergesort(A, p, r)
// Sorts array A from p to r
if r≤p return;
int m = (p+r)/2;
Mergesort(A, p, m);
Mergesort(A, m+1, r);
Merge(A, p, m, r);
//merge the sorted subarrays A[p..m] and A[m+1..r]
```

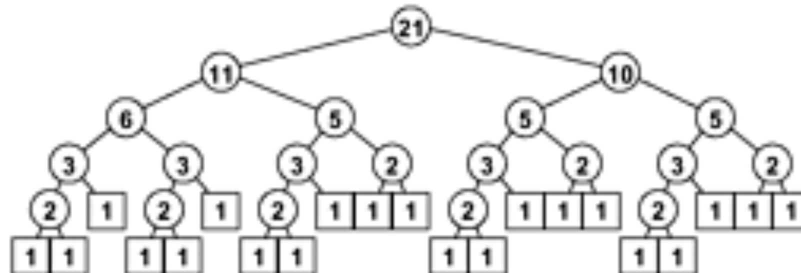
The Mergesort Algorithm



The Mergesort Algorithm

- Recursion trees
- They show how the problem breaks into smaller subproblems
- In Mergesort, the trees are independent of the input





The Mergesort Algorithm

Complexity:

- Let n = r – p + 1 (how many numbers we want to sort)
- Claim: The merge of 2 sorted subarrays of size L and L' respectively can be done in time O(L+L')

Hence:

2 recursive calls for sorting arrays with n/2 numbers T(n/2)

1 merge of subarrays of size n/2 each O(n)

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(n/2) + O(n), & \text{if } n \geq 2 \end{cases} \overset{MasterTheorem}{\Longrightarrow} T(n) = O(n \log n)$$

Space complexity: 2n

- merge needs an auxiliary matrix
- Main drawback of Mergesort when sorting large arrays

Quicksort

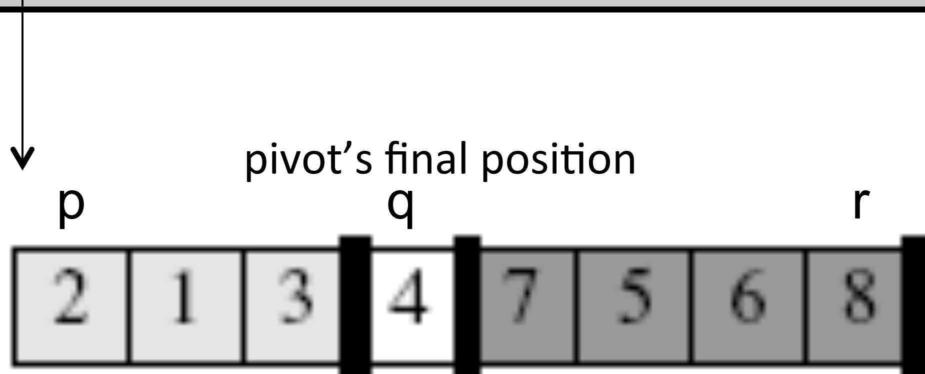
- Idea: Recursion tree dependent on the input
- Use a pivot element x, and partition A so that
  - Elements smaller than x come to the left of x
  - Elements larger than x come to the right of x
- We can now recursively sort the 2 subarrays, left and right of x
- Recurrence relation now depends on how the pivot partitions A

T. Hoare, 1960

R. Sedgewick
Ph.D. thesis, 1975

Quicksort

```
QuickSort (A, p, r)
if p < r:
    select pivot x;
    Partition (A,p,r);
    //it now holds that:
    // A[i] ≤ x, for p ≤ i ≤ q-1
    // x ≤ A[i], for q+1 ≤ i ≤ r
    // q is the final position of x
    QuickSort (A, p, q-1);
    QuickSort (A, q+1, r);
```



Quicksort

- How is Partition implemented?
  - In tutorial
- Worst case: $O(n^2)$ (why?)
- BUT: average case = O(nlogn)
- In practice better than any other method
- Most built-in sorting methods in various systems are based on Quicksort

Decrease and Conquer

- Sometimes we do not need to combine the solutions of different subproblems
- Instead, it is enough to solve recursively one subproblem
- Recurrences of the form $T(n) = aT(n/b) + f(n)$ but with $a = 1$
- Usually problems with low complexity

Decrease and Conquer

Examples

- Binary search in a sorted array
  - Only need to decide which half of the array to look at
  - $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$
- Search in a binary search tree
- Median and selection problems
  - Find the k-th smallest element in a set
  - In tutorial!

Greedy Algorithms

Greedy Algorithms

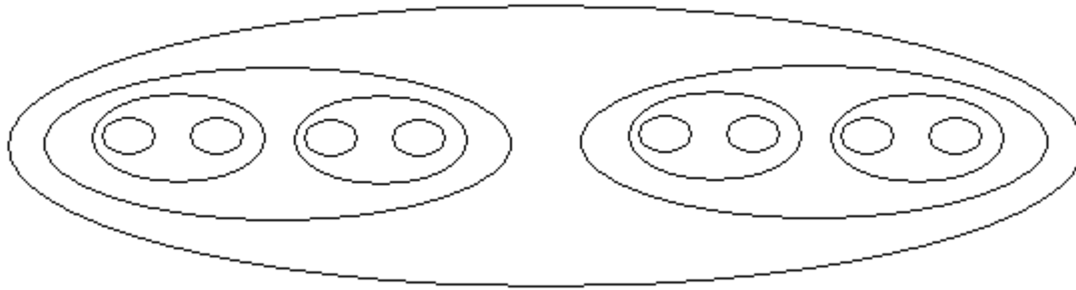
The basic idea:

- Simple algorithms, that evolve in rounds, easy to implement
- Start from the empty solution
- Repeatedly, build up a solution (evolving in rounds)
- At every round, make the choice that looks best at the moment (according to some criterion)
    - What the algorithm chooses in each round
        - Can depend on previous choices
        - Cannot change in the future (myopic moves)
        - Reduces the size of the remaining problem

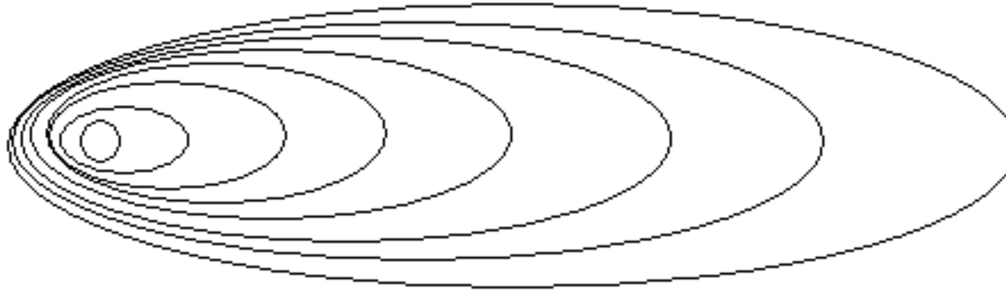
- Proof of correctness:
    - Not as obvious as for divide and conquer algorithms
    - Need to prove that locally optimal choices lead to a globally optimal solution

Algorithm design methods

DIVIDE AND CONQUER



GREEDY



Activity Selection (Interval Scheduling)

- Suppose we want to schedule some tasks (e.g., courses) that need to use a common resource (e.g., a classroom)
- No 2 tasks can be scheduled at the same time

Interval Scheduling

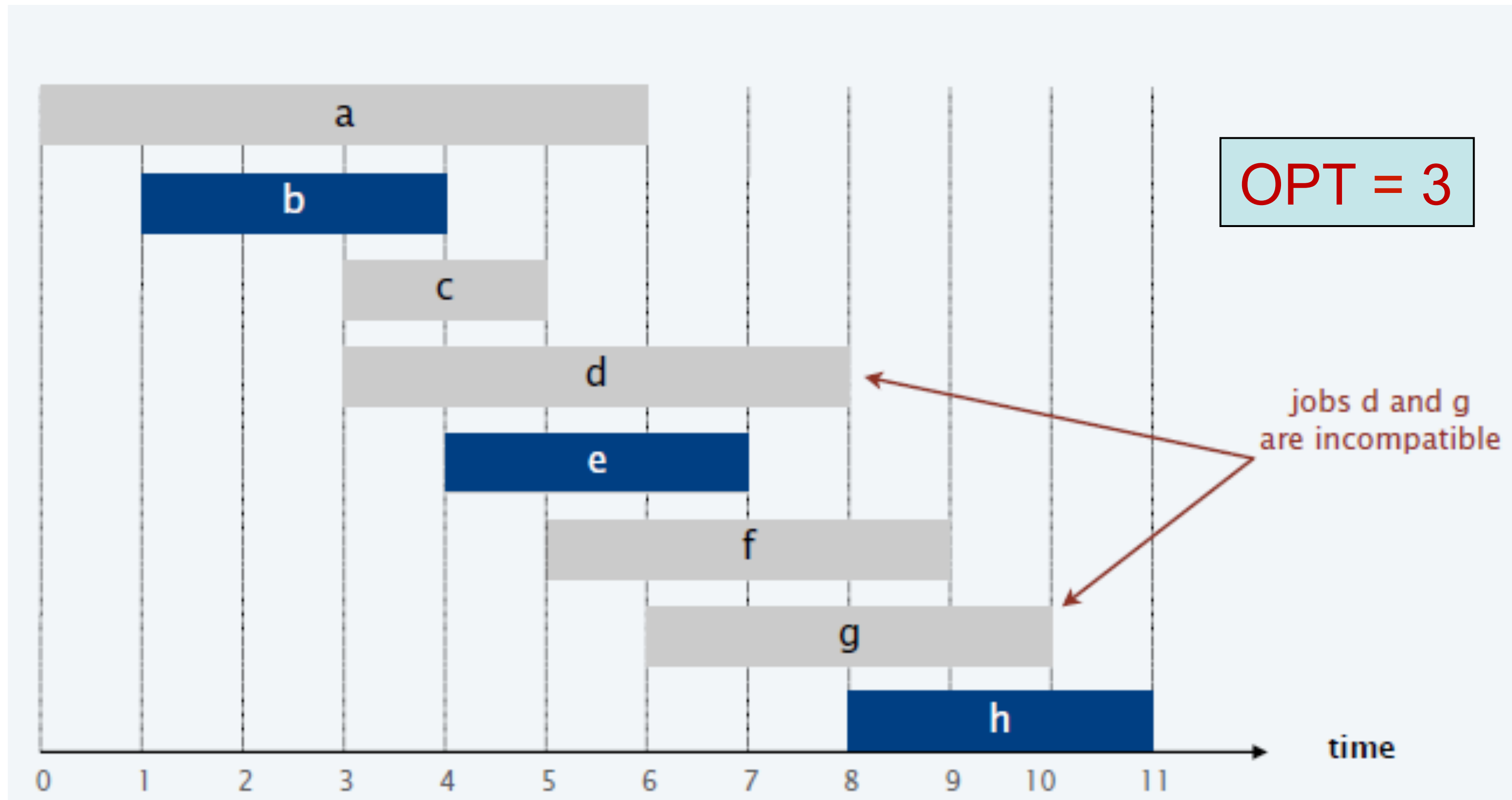
I: A set A, of n jobs, each with a start time s_i and a finish time f_i

Q: Find a feasible schedule with the maximum possible number of tasks (maximum throughput)

Comment: There can be many optimal solutions, scheduling different tasks each. We do not care here which optimal solution we find

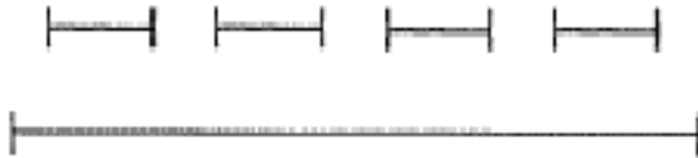
For an instance I, we let OPT denote the optimal schedule

Activity Selection (Interval Scheduling)



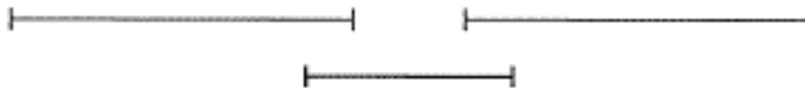
Activity Selection – Some ideas

1) Choose in each round the task that starts at the earliest possible feasible time



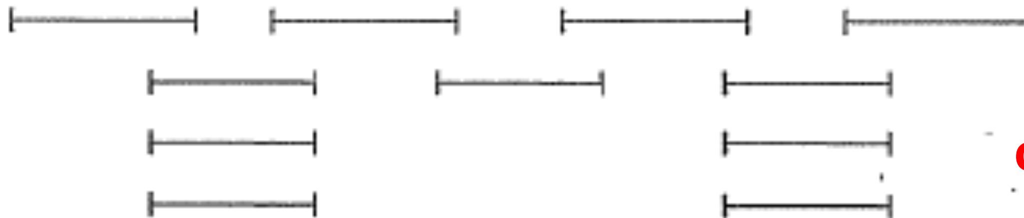
c=1, OPT=4 !

2) Choose in each round the shortest feasible task among the remaining ones



c=1, OPT=2 !

3) Choose in each round the task with the least number of overlaps

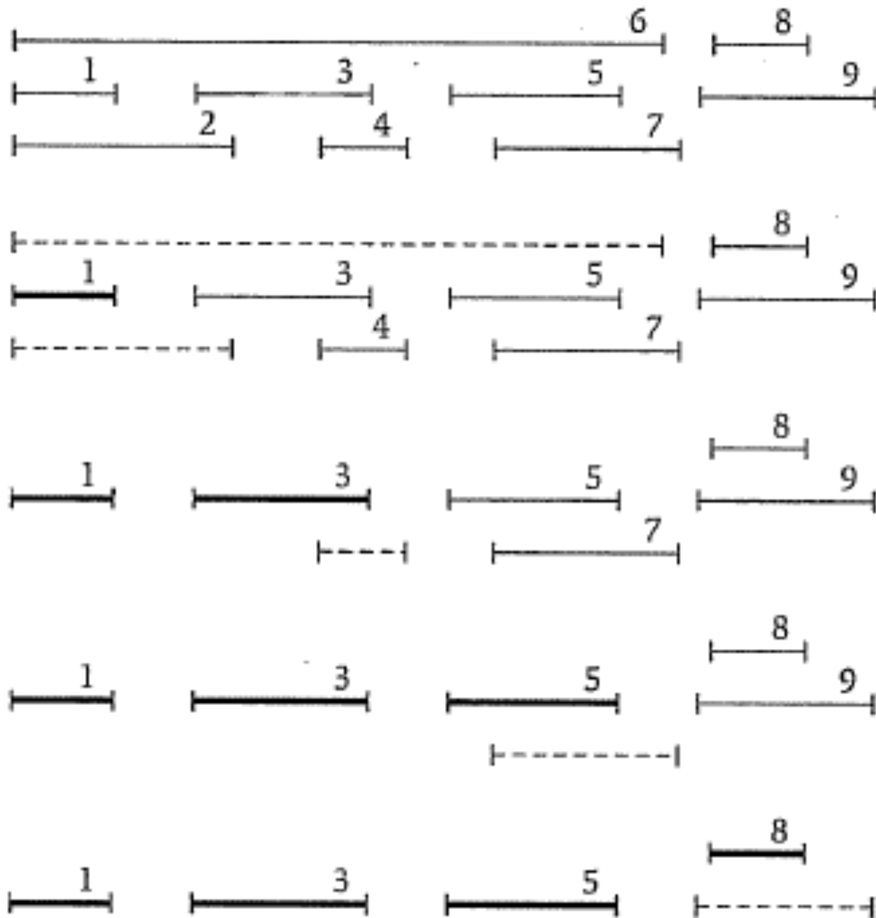


c=3, OPT=4 !

Other ideas?

Activity Selection – The algorithm

Rename the jobs so that **$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$**



- Choose first the job with the earliest finish time, f_1
- Remove those that overlap with job 1
- Continue in the same manner, choosing the earliest finish time among the remaining ones

Clearly a polynomial time algorithm

Is this optimal?

Activity Selection – Proof of correctness

- How do we prove that a greedy algorithm is optimal?
- Usually proof by contradiction or by induction
- But, the crucial property for a greedy algorithm to be optimal, is that it should satisfy the "optimal substructure" property

Optimal substructure in general:
A problem satisfies *optimal substructure* if an optimal solution to a problem contains within it optimal solutions to subproblems

Optimal Substructure for Activity Selection:
An optimal solution (that contains job 1), contains the optimal solution for the jobs
$A' = \{ i \in A : s_i \geq f_1 \}$

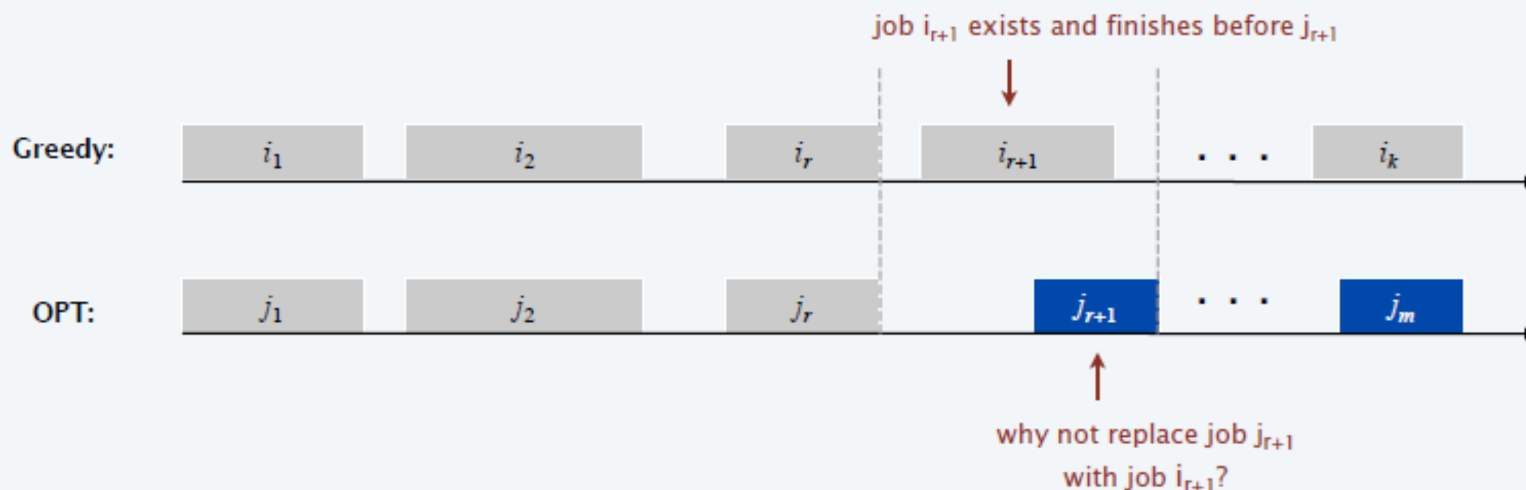
(why?)

Activity Selection – Proof of correctness

Theorem: Choosing in every round the job with the earliest finish time produces an optimal solution for the Activity Selection problem

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, ... i_k denote set of jobs selected by greedy.
- Let j_1, j_2, ... j_m denote set of jobs in an optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.



Activity Selection – Proof of correctness

Theorem: Choosing in every round the job with the earliest finish time produces an optimal solution for the Activity Selection problem

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

Activity Selection – Proof of correctness

In other words:

- What we showed is that the algorithm always “stays ahead” of the optimal solution
- This is how optimality of greedy algorithms is established for many other problems
- In each problem, we need to find the sense in which the algorithm “stays ahead”, i.e., does at least as good as the optimal solution

The gas station problem

- Suppose you want to drive along Route 66 from Chicago to Los Angeles.
- Your car with a full tank can drive up to x miles
- Your map shows all the points on the highway where gas stations are located, along with distances from one to the next
- Can you minimize the number of stops that you make?

Assume Route 66 is a straight line

The gas station problem

I: The coordinates of each available gas station on the line, the parameter x

Q: Find where to stop for gas so as to minimize the number of stops

The gas station problem

Greedy selection of gas stations:

- Do not do now what you can do later!
- Always go to the very last gas station that you can go before you run out of gas

The algorithm:

- Starting from one end of the line (say the left one)
- Move to the right and find the last gas station available in the interval [0, x]. Let p_1 be its location
- From p_1, move to the right and find the right-most gas station in the interval $[p_1, p_1 + x]$
- Continue in the same manner till you reach the end of the line.

The gas station problem

Optimal substructure

- Let p_1 be the first gas station in the optimal solution
- The optimal solution contains within it an optimal solution for the subproblem to the right of p_1
- Otherwise, we could replace it with an optimal solution to the subproblem and obtain a better global solution

Theorem: The algorithm described minimizes the number of stops required on Route 66

Proof: very similar with the proof in the Interval Scheduling problem

Dynamic Programming

Dynamic Programming

Richard Bellman (1953)

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables.

Etymology
(at that time; Bellman was studying multi-stage decision processes)

- Dynamic: relating to time
- Programming : what to do and when to do it
- Dynamic Programming: planning over time

Bellman gave an impressive name to be accepted by the Secretary of Defense (Wilson) who didn't like math research…

Dynamic Programming

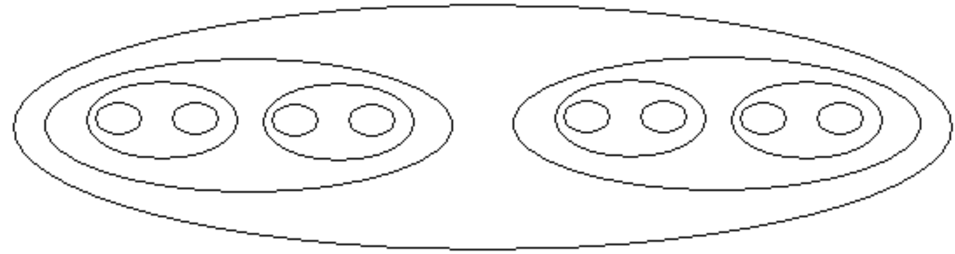
- Define sub-problems of the same structure with the original
  - Overlapping sub-problems
- Optimal substructure
  - the optimal solution includes/can be constructed from the optimal solution to its sub-problems
- Solve the (sub)problem(s) recursively starting from trivial ones
  - write a recursive formula for the optimal solution

  This gives an order of subproblems such that one can be solved given the answers of "smaller" ones (appearing earlier in this order)

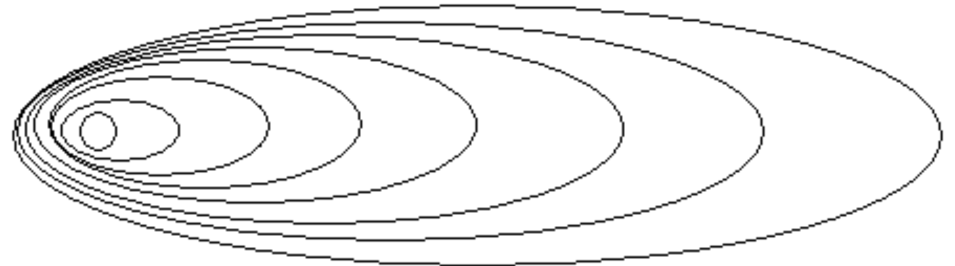
  - Attention: We are not going to solve the problem via recursion
- Translate the recursive formula into an iterative algorithm
  - Use a table to save intermediate results for later use
- Get the value of the optimal solution
- Find the solution itself

Algorithm design methods

DIVIDE AND CONQUER
Non overlapping sub-problems
Recursion can be used

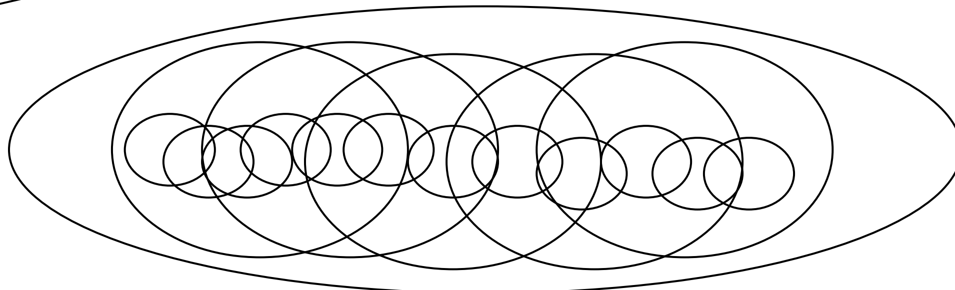


GREEDY
A sub-problem defines the next one
A single (greedy) choice



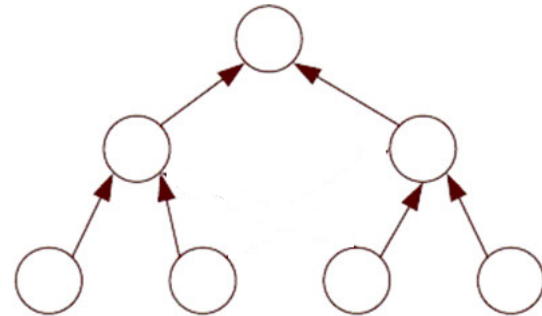
OPTIMAL SUB-STRUCTURE

DYNAMIC PROGRAMMING
Overlapping sub-problems
Recursion is forbidden
Many choices for a sub-problem



Algorithm design methods

DIVIDE AND CONQUER
Non overlapping sub-problems
Recursion can be used



Tree

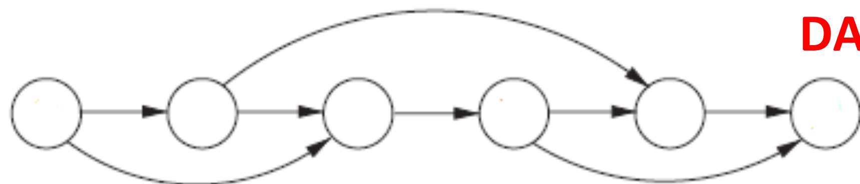
GREEDY
A sub-problem defines the next one
A single (greedy) choice

Chain

DYNAMIC PROGRAMMING
Overlapping sub-problems
Recursion is forbidden
Many choices for a sub-problem

OPTIMAL SUB-STRUCTURE

DAG



Fibonacci numbers

Recall the Fibonacci sequence:

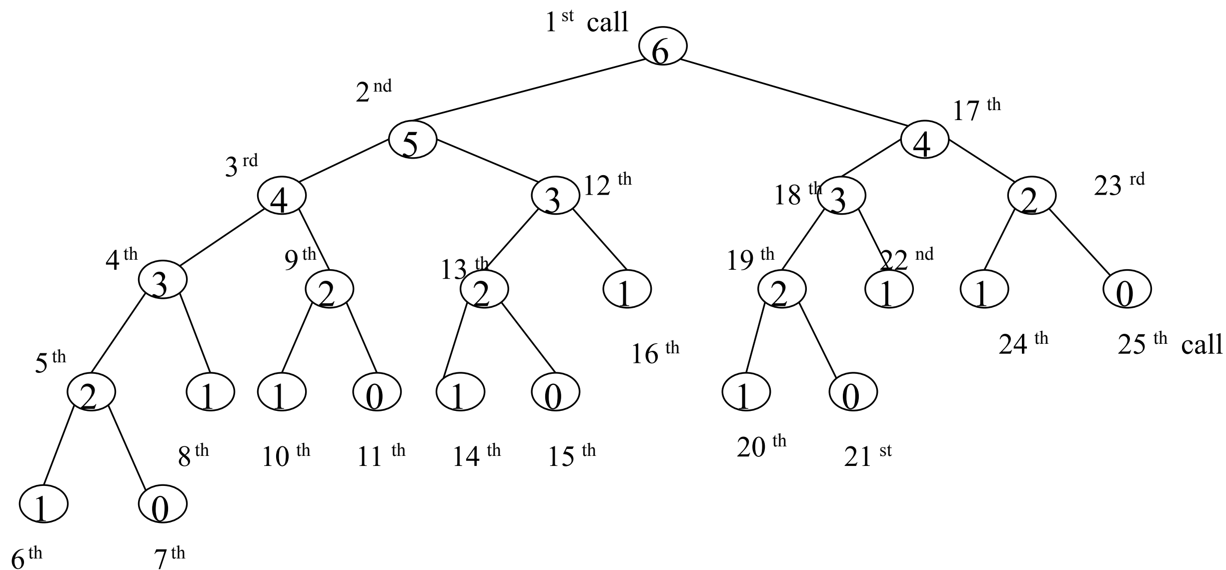
$F_0 = 0; F_1 = 1; \quad F_n = F_{n-1} + F_{n-2}, \; n \geq 2$

Direct implementation of recursion:

```
Algorithm fib1(n) // Direct implementation of recursion
   if n<2 then return n
          else return fib1(n-1)+ fib1(n-2)
```

Fibonacci numbers

Recursion tree:



Recursion?

No, thanks !

$T(n) > 2^{\lfloor n/2 \rfloor}$

Fibonacci numbers

Iterative version (non-recursive): Use a table to store intermediate values

```
Algorithm fib2(n)  //remember already computed values
  f[0]:=0; f[1]:=1;
   for i:=2 to n do f[i]:= f[i-1] + f[i-2]
```

Note however:
Time Complexity: O(n) NOT polynomial in |I| = O(logn)
Space complexity: Also O(n) (but we could do it with 3 memory cells = O(1)

Dynamic programming does not always yield polynomial time algorithms

Weighted Interval Scheduling

- Recall the Interval Scheduling problem
- We want to schedule some tasks (e.g., courses) that need to use a common resource (e.g., a classroom)
- No 2 tasks can be scheduled at the same time
- Weighted version: each task has a weight, may correspond to value or profit that we derive from the execution of each task

Weighted Interval Scheduling

I: A set A, of n jobs, each with a start time s_i, a finish time f_i, and a value v_i

Q: Find a feasible schedule with the maximum possible total value

Weighted Interval Scheduling

- The greedy algorithm we saw before does not work any more
- It may be beneficial to select just one job of high value than maximize the number of non-overlapping jobs
- Actually, no other greedy approach is known for this problem

Dynamic Programming approach

We need to identify an optimal substructure property

Warmup:

- Reorder the jobs so that **$f_1 \le f_2 \le f_3 \le \ldots \le f_n$**
- Let O_j = optimal schedule if we had only the requests {1, 2, ..., j}
- Let OPT(j) = total value of the optimal solution O_j

$$OPT(j) = \sum_{i \in O_j} v_i$$

Weighted Interval Scheduling

- Idea: try to find a recursive formula
- We need to relate OPT(j) with the optimal values for smaller instances

Definition: Let p(j) = largest index i, with i < j such that jobs i and j do not overlap

i.e., the jobs p(j)+1, p(j)+2,...up to j-1 overlap with j

Why is this useful?

Consider an optimal solution O_j for {1, 2,..., j}.

2 observations:

- If j is included in O_j, then O_j also contains an optimal solution for {1, 2,..., p(j)}
  - Since there is no overlap with such jobs
- If j is not included in O_j, then OPT(j) = OPT(j-1)

Hence:

$$OPT(j) = \max\{ v_j + OPT(p(j)), OPT(j-1) \}, \text{ for every } j \geq 1$$

Weighted Interval Scheduling

This directly yields a recursive algorithm:

```
Algorithm WIS1(n)
  // suppose we have pre-computed the values p(j) for every j
  if n=0 return 0;
  else return max(v_n + WIS1(p(n)), WIS1(n-1))
```

To be more precise:

- The input to the algorithm consists of the vectors
  - **s** = (s_1, s_2,..., s_n), the start times
  - **f** = (f_1, f_2,..., f_n), the finish times (assume we have ordered them)
  - **v** = (v_1, v_2,..., v_n), the values
- **`WIS1(j)`** means the execution of the algorithm on the first j jobs

Complexity:

- Recursion tree grows exponentially
- Same problem as with recursive algorithm for Fibonacci

Weighted Interval Scheduling

Memoization:

- Use an array to remember already computed values
- Loop through the array to compute the optimal values to all subproblems

```
Algorithm WIS2(n)
  Set M[0]=0;
  Compute the values p(j) for every j
  for j = 1 to n do
    M[j] = max(v_j + M[p(j)], M[j-1])
  return M[n]
```

Complexity:

- Within each iteration, we need only O(1)
- Hence O(n) total

Weighted Interval Scheduling

- The algorithm only computes the value of the optimal solution
- What if we want to find the schedule as well
- We could use a different array S, so that S[i] maintains the optimal solution up to {1,...,i}
- But this causes some blowup
- We can instead recover the solution from M (why?)

Summarizing:

Theorem: We can solve the Weighted Interval Scheduling problem in time O(n)

- O(nlogn) if the finish times of the jobs are not sorted

Maximum Sub-array (MSA)

Maximum Sub-Array (MSA):

I: Array of numbers A[1..n]

Q: Find a sub-array A[p..q] with a maximum sum of its elements

Hence, we are looking for indices p, q, so that the sub-array A[p..q] maximizes the quantity

$$V(p,q) = \sum_{i=p}^{q} A(i)$$

Example: Profit history

Year	1	2	3	4	5	6	7	8	9
Profit	-3	2	1	-4	5	2	-1	3	-1

We want the period of years with the greatest profit: V(5,8)=9

Maximum Sub-array (MSA)

We need to find a recurrence

Let E(i) be the value of the maximum sequence ending in position i

Observation:

- The MSA is one of the E(i)'s, that is $Vmax = max_i \{ E(i) \}$
- The problem is then reduced to the calculation of the E(i)'s

DP: Find the E(i) based on E(i-1) (exploit optimal substructure)

- E(i) has to contain A(i)
- Two cases for E(i):
  - Either it contains only A[i]: E(i) = A[i]
  - Or it contains the optimal solution E(i-1): E(i) = E(i-1) + A[i]

Hence:

$$E(i) = max \{ E(i-1)+A[i], A[i] \}$$
$$E(1)=A[1]$$

Maximum Sub-array (MSA)

Example:

$E(i) = \max\{ E(i-1)+A[i],\ A[i] \}, \quad E(1)=A[1]$

A[1..n]	31	-41	59	26	-53	58	97	-93	-23	84

Maximum Sum for any sub Array ending at i^{th} location

E[1..n]	31	-10	59	85	32	90	187	94	71	155

Maximum so far

Vmax	31	31	59	85	85	90	187	187	187	187

Maximum Sub-array (MSA)

E(i) = max { E(i-1)+A[i], A[i] } , E(1)=A[1]

```
MSA(A[1..n])
E(1)=A[1], Vmax=A(1)
for i = 2 to n do
     E(i) = E(i-1)+A(i)
     if E(i) < A(i) E(i) = A(i)
     if E(i) > Vmax Vmax = E(i)
```

Time complexity: O(n)

Space Complexity: O(n) (for the array E)

O(1) if we remove the indices

Maximum Sub-array (MSA)

- What about the indices p, q of the optimal solution?
- Let P(i) be the start index of E(i)

```
MSA(A[1..n])
E(1)=A[1], Vmax=A(1), P(i)=1
For i = 2 to n do
     E(i) = E(i-1)+A(i), P(i)=P(i-1)
     if E(i) < A(i)
            E(i) = A(i)
            P(i) = i
     if E(i) > Vmax
             Vmax = E(i)
             p=P(i)
             q=i
```

Time complexity: O(n)

Space Complexity: O(n) (for arrays E, P), O(1) if we remove the indices