Advanced Programming Techniques

PART V

Problem solving

Beniamin Bogosel

Ecole Polytechnique
Department of Applied Mathematics

Generic approaches

- Brute force: simplest, direct method, starting from the definition, exhaustive research
- Divide and conquer: divide the problem into sub-problems, solve them and (eventually) fusion the solutions
- Dynamical programming: solve the current problem using smaller, possibly overlapping problems
- Greedy algorithms: construct the solution locally, by optimizing blindly a local criterion

Brute Force

- Build the most direct solution to the problem
- Examples:
  - Search an element in an array: linear loop
  - Compute a^n: multiply a with itself n times
  - Compute Fibonacci numbers: direct recursion (without thinking)
- Often, not efficient!
- Even if inefficient, use it to create and benchmark test cases on which you can test more refined algorithms

Examples

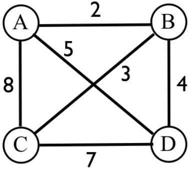
⋆ Searching: bubble sort: double loop, swap elements not respecting the order $O(n^2)$

⋆ Exhaustive search: generate all possible solutions until one verifies the desired properties

- Generate all permutations of an array and pick the sorted one...
- $O(n!)$

Traveling salesman

⋆ consider n cities and the distances between them
⋆ Find the shortest path going through all the cities exactly once before coming back to the original city.
⋆ Exhaustive search: $O(n!)$
⋆ polynomial algorithms are not known for this problem



A-B-C-D-A	17
A-B-D-C-A	21
A-C-B-D-A	20
A-C-D-B-A	21
A-D-B-C-A	20
A-D-C-B-A	17

Brute force/exhaustive search

Advantages:

- simple
- good starting point
- sometimes it's not worth going further

Inconvenients:

- It is rarely the best solution
- less elegant and creative than other techniques

In practice you can always start by giving the brute force solution before searching for something better.

Divide and Conquer

General principle:

- if the problem is trivial, solve it directly
- else:
  - divide the problem into smaller ones
  - solve the smaller problems (recursively)
  - fusion the solutions to subproblems to find a solution to the original problem

Examples already seen

- Merge Sort:
  1. Divide: split the array into two sub-arrays of equal size
  2. Conquer: sort recursively the two sub-arrays
  3. Fusion: fusion the sub-arrays

  Complexity $\Theta(n \log n)$
- Quick Sort:
  1. Divide: Partition the table according to the pivot
  2. Conquer: sort recursively the two sub-arrays
  3. Fusion: none

  Average Complexity $\Theta(n \log n)$
- Binary search (dichotomy):
  1. Divide: Control the central element of the array
  2. Conquer: Search recursively into the left/right sub-arrays
  3. Fusion: trivial

  Complexity $O(\log n)$ (brute force $O(n)$)

Example: search for spikes

- Consider a table A and assume $A[0] = A[A.length] + 1 = -\infty$
- Definition: $A[i]$ is a spike/peak if it is not smaller than its neighbors

$$A[i-1] \leq A[i] \geq A[i+1].$$

(local maximum)
- Objective: find a spike in an array
- A spike always exists (Exercise: prove it!)

Brute force approach

⋆ Test all possible positions sequentially:

```
PEAK1D(A)
for i = 1 to A.length
    if A[i − 1] ≤ A[i] ≥ A[i + 1]
        return i
```

⋆ Complexity: $\Theta(n)$ in worst case

⋆ Second variant: maximum element in the table is a peak. Search for a maximum: $\Theta(n)$

A more refined idea

Divide and conquer:

- Look at $A[i]$ and the neighbors $A[i-1], A[i+1]$
- If we have a peak, return i
- Otherwise:
  - the values must increase at least on one side

$$A[i-1] > A[i] \text{ or } A[i] < A[i+1].$$

  - if $A[i-1] > A[i]$ search for a peak in $A[1..i-1]$
  - if $A[i+1] > A[i]$ search for a peak in $A[i+1..A.length]$.
- At which position i should we look first?

Algorithm

```
PEAK1D(A, p, r)
  q = ⌊(p+r)/2⌋
  if A[q − 1] ≤ A[q] ≥ A[q + 1]
        return q
  elseif A[q − 1] > A[q]
        return PEAK1D(A, p, q − 1)
  elseif A[q] < A[q + 1]
        return PEAK1D(A, q + 1, r)
```

Initial call: Peak1D($A, 1, A.length$)

Analysis

- Is the algorithm correct? Yes
  - We need to prove this.
  - Assume $A[q+1] > A[q]$ and there's no peak in $A[q+1..r]$.
  - Then $A[q+1] < A[q+2]$ (otherwise $A[q+1]$ is a peak).
  - Repeat this until reaching the end of the array.
  - if $A[r-1] < A[r]$ (r is the endpoint) then by definition we have a peak!
- Complexity:
  - Worst case: $T(n) = T(n/2) + c_1$
  - $T(n) = O(\log n)$ (like the binary search)

Extending to a 2D array

- Consider a matrix $n \times n$ containing numbers
- Find an element which is largest than its neighbors
- Brute force $O(n^2)$, Search for a maximum $O(n^2)$

Divide and conquer

- Search for a maximum in the central column
- If it's a peak (in 2D) return it
- Otherwise, apply the function recursively to the left/right half of the matrix if the left/right neighbor is larger

Correct? Yes:

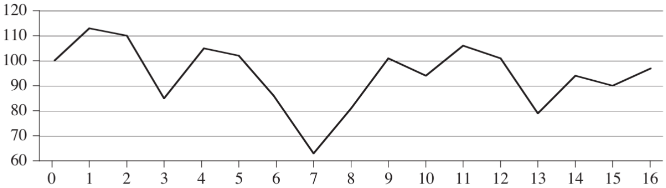
- A peak must exist on the half giving a larger value
- If not, then we can always find a neighbor with a larger value
- At some point we'll run out of points (finite number)

Complexity?

- $\Theta(n)$: finding the maximum on one column
- $O(\log n)$ iterations
- $O(n \log n)$ in total

Can we do better: yes, there exists a $O(n)$ algorithm.

Another example: Buy/Sell stocks



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- Consider the price of a stock on n consecutive days
- Determine retrospectively:
  - when should we have bought the stock
  - when should we have sold the stock

to maximize the profit

Strategies

First idea:

- Buy at minimum price, sell at maximum
- Not correct: the maximum is not necessarily after the minimum!

Second idea:

- Buy at minimum, sell at maximum price afterwards
- Sell at maximum, buy at minimum price before
- Not correct: if the max/min are at the beginning/end

Third idea:

- Test all pairs (brute force)
- Correct? Complexity?

Transform the problem

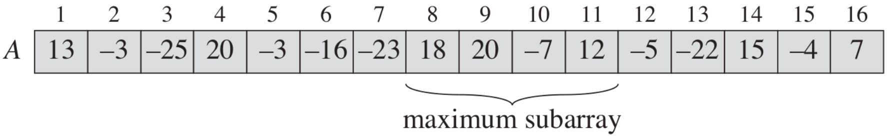
Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Assume the initial price table is labeled A
- Compute the difference table: $D[i] = A[i] - A[i-1]$
- Determine the non-void subsequence of maximal sum in D
- Let $D[i..j]$ be this sub-sequence: then it is optimal to buy on i-th day and sell on j-th day

⋆ In the example: buy on 8th day and sell 11th

⋆ If we can find the maximal sub-sequence in a table we have a solution for our problem

Brute force



- Generate all sub-arrays and compute all sums
- $O(n^2)$ sub-arrays and $O(n)$ for computing the sum: $O(n^3)$!

Divide and Conquer

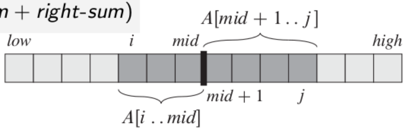
- Find maximum sub-array in $A[p..r]$
- Divide: split at midpoint $q = \lfloor (p+r)/2 \rfloor$
- Fusion?
  - Search for max sub-array crossing the midpoint!
  - Pick the best among the three options

New problem: maximum sub-array crossing the junction point!

- brute-force? $\Theta(n^2)$ ($n/2$ choices on the left, $n/2$ choices on the right)
- better solution: search independently the left/right parts $\Theta(n)$ for the two parts

Max Crossing Sub Array

```
MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```



```
MAX-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) = MAX-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) = MAX-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Analysis

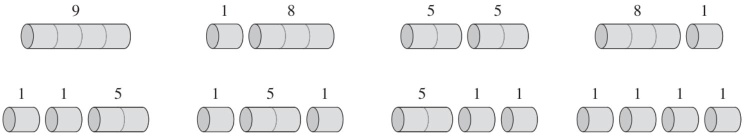
- The cost $T(n)$ verifies
$$T(n) = 2T(n/2) + cn, n \geq 2.$$
- Same complexity as merge sort: $\Theta(n \log n)$
- Can we do better? Yes.

Dynamical programming

- ⋆ use smaller subproblems to solve the current one!
- ⋆ Consider a steel rod to cut and sell piece by piece
- ⋆ the selling price depends non-linearly on the length
- ⋆ Find the maximum profit from selling a rod of n centimeters
  - Inputs: a price table: p_i, $i = 1, 2, ..., n$
  - Output: maximum revenue obtained from selling a rod of length n

Example:

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30



Ideas

Brute force approach

- enumerate all possible cuts, compute the revenue, select the maximum one
- Cost: exponentially in terms of n!
- Infeasible even for moderately sized n

Recursivity

- Re-formulate r_n recursively
- If n corresponds to a base case, return it
- Otherwise consider all possible sub-cuts using one admissible length.

Naive version

r_n is the maximum of

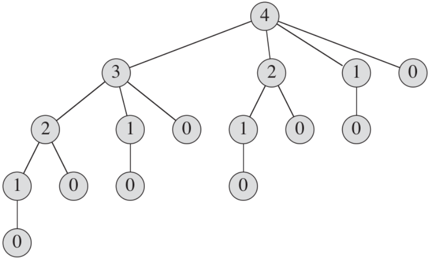
- p_n: the price without cutting (if admissible)
- $r_{n-1} + r_1$: rod of length 1 + rod of length $n-1$
- $r_{n-2} + r_2$:
- ...

Simplified:

- Consider the left-most cut part for an admissible cut size
- $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$.
- Only one sub-problem required

Direct implementation

- extremely inefficient due to redundant calls
- recursion tree for $n = 4$



- Number of nodes grows exponentially
- Store computed values in an array and re-use them when needed
- Two implementations possible:
  - top-down: memoization $\longrightarrow$ dictionaries or hash tables!
  - bottom-up

Top-down: memoization

```
MEMOIZED-CUT-ROD(p, n)
    Let r[0..n] be a new array
    for i = 1 to n
        r[i] = −∞
    return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
    if r[n] ≥ 0
        return r[n]
    if n == 0
        q = 0
    else q = −∞
        for i = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
    r[n] = q
    return q
```

⋆ if the current value is computed, use it

⋆ otherwise, use the recursive formula

Bottom-up

⋆ solve the subproblems which are smallest first and go upwards

```
BOTTOM-UP-CUT-ROD(p, n)
Let r[0 . . n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Analysis

- The ascending version is $\Theta(n^2)$
- The descending version is also $\Theta(n^2)$
- Reconstructing the solution??

Modified version

⋆ a new array s contains the left-most cut in the optimal solution for the size j

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
Let r[0..n] and s[1..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

⋆ To show the solution, use the values in s

i	0	1	2	3	4	5	6	7	8
$p[i]$	0	1	5	8	9	10	17	17	20
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

Generalities regarding dynamic programming

- optimization problems which can be decomposed into sub-problems of the same nature
  - optimal sub-structure: computing solution for problem of size n starting from solutions to subproblems
  - Subproblems may overlap
- Direct recursive implementation: exponential complexity
- saving previous results is helpful to decrease the cost

Fibonacci

- Direct recursion: exponential complexity
- Iterative version: linear complexity
- Matrix exponentiation: logarithmic complexity

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Exponentiation by squaring

- Divide and conquer:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{n } \textit{even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{n } \textit{odd} \end{cases}$$

Maximal subsequence

```
MAX-SUBARRAY-LINEAR(A)
Let m[1..n] be a new array
max-so-far = A[1]
m[1] = A[1]
for i = 2 to A.length
    if m[i − 1] > 0
        m[i] = m[i − 1] + A[i]
    else m[i] = A[i]
    if m[i] > max-so-far
        max-so-far = m[i]
return max-so-far
```

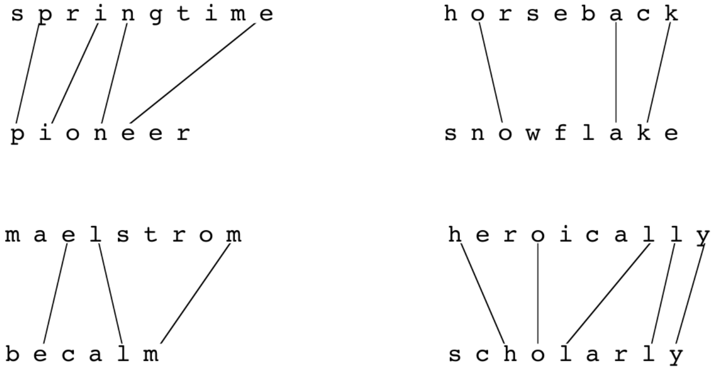
- Complexity: $\Theta(n)$ (vs $\Theta(n \log n)$ for divide and conquer)
- $m[i]$ is the maximal subsequence sum ending at i
- The algorithm computes $m[i]$ starting from $m[i-1]$.
- Ascending dynamical programming (very simple)
- Exercise: add variables to keep track of the sub-array bounds

Longest common sub-sequence

⋆ **Why?** Example: Compute diff between two text files to view modifications

Problem: Given two sequences $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ find the longest common sub-sequence

Examples:



Brute force

- Enumerate all subsequences of the shortest sequence
- For every one of them verify if it is a subsequence of the first one
- Complexity: $\Theta(n \cdot 2^m)$
  - 2^m possible subsequences
  - Testing if sub-sequence: $\Theta(n)$
- Exercise: implement this

Solve using dynamical programming

Sub-structure property:

- The longest common sub-sequence has prefixes which are longest common sub-sequences for some prefixes

Example: if $Z = z_1 ... z_k$ is the longest common sub-sequence (LCSS) then:

- z_1 is the LCSS for $x_1 ... x_{i_1}$; $y_1, ..., y_{j_1}$
- z_1, z_2 is the LCSS for $x_1 ... x_{i_2}$; $y_1, ..., y_{j_2}$
- etc...

Denote by $X_i = x_1 ... x_i$ a prefix for X and $Y_i = y_1 ... y_i$ a prefixe for Y for the index i.

Towards a solution

Let $c[i,j]$ the length of the LCSS in X_i and Y_j. Then

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & i,j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

⋆ **in other words**: if two elements are equal, the length of the common subsequence increases, else, it stays the same as the best previous case

Implementation

```
LCS-LENGTH(X, Y, m, n)
Let c[0..m, 0..n] be a new table
for i = 1 to m
        c[i,0]=0
for j = 0 to n
        c[0,j]=0
for i = 1 to m
        for j = 1 to n
                if x_i == y_j
                        c[i, j] = c[i − 1, j − 1] + 1
                elseif c[i − 1, j] ≥ c[i, j − 1]
                        c[i, j] = c[i − 1, j]
                else c[i, j] = c[i, j − 1]
return c
```

Complexity: $\Theta(m \cdot n)$

Example

		c	o	m	p	u	t	e	r	s	c	i	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
u	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
r	0	0	0	0	0	1	1	1	2	2	2	2	2	2	2	2
e	0	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3
l	0	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3
v	0	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3
l	0	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3
a	0	0	0	0	0	1	1	2	2	2	2	2	3	3	3	3
i	0	0	0	0	0	1	1	2	2	2	2	3	3	3	3	3
c	0	1	1	1	1	1	1	2	2	2	3	3	3	3	4	4
u	0	1	1	1	1	2	2	2	2	2	3	3	3	3	4	4

Recovering an example?

⋆ use the array $c[i,j]$
⋆ start from the bottom right which has value k
⋆ find the position (i,j) such that i and j are minimal and $c[i,j]=k$: this gives the k-th element of the common subsequence
⋆ Do the same for $k-1$, $k-2$, ... etc

Knapsack problem

Problem:

- A thief goes in a museum and wants to steal some objects which he can carry in his knapsack, of maximum weight W
- The museum has a list of n art objects each one having weight p_i and a value v_i
- Find the list of objects with total weight at most W and the maximum price!

⋆ Consider S a set of n objects having values v_i and weights p_i.

⋆ Find $x_1, ..., x_n \in \{0, 1\}$ such that

- $\sum_{i=1}^{n} x_i p_i \leq W$
- $\sum_{i=1}^{n} x_i v_i$ is maximal

Example

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Consider the knapsack capacity of $W = 11$
-
    - $\{5, 2, 1\}$ has weight 10 and value 35
    - $\{3, 4\}$ has weight 11 and value 40

Brute force approach

- Enumerate all subsets of S and compute their value: $O(n2^n)$
- Any amelioration that involves heuristics, but is still based on an enumeration of all possibilities will lead to a similar complexity

Dynamical programming

⋆ Define $M(k, w)$, $0 \leq k \leq n$ and $0 \leq w \leq W$ the maximum benefit that we can find using objects $1, 2, ..., k$ from S and a knapsack of maximum charge w. (assume all variables are integers)

⋆ We have two possibilities:

(a) The object k is not in the optimal choice: $M(k, w) = M(k-1, w)$

(b) The object k is among the optimal choice of objects: $M(k, w) = M(k-1, w-p_k) + v_k$

We obtain the recurrence:

$$M(k, w) = \begin{cases} 0 & \text{if } k = 0 \\ M(k-1, w) & \text{if } p_k > w \\ \max\{M(k-1, w), v_k + M(k-1, w-p_k)\} & \text{otherwise} \end{cases}$$

Code

```
KnapSack(p, v, n, W)
Let M[0..n, 0..W] be a new table
for w = 0 to W
        M[0,w]=0
for k = 1 to n
        M[k, 0] = 0
for k = 1 to n
        for w = 1 to W
                if p[k] > w
                        M[k, w] = M[k − 1, w]
                elseif M[k − 1, w] > v[k] + M[k − 1, w − p[k]]
                        M[k, w] = M[k − 1, w]
                else M[k, w] = v[k] + M[i − 1, w − p[k]]
return M[n, W]
```

Example

M	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1,2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1,2,3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1,2,3,4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1,2,3,4,5\}$	0	1	6	7	7	18	22	28	29	34	35	40

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Optimal solution: $\{3,4\}$
- Optimal value: $22+18=40$

Recover the solution

- ⋆ Go back through the table M starting from the down rigntmost position.
- ⋆ $k = n$
- ⋆ decrease k until the value of $M[k-1, w] < M[k, w]$
- ⋆ replace w by $w - p_k$ and repeat

Complexity of the solution

Time and space complexity: $\Theta(nW)$

- Filling the matrix: $\Theta(nW)$
- Searching the solution $\Theta(n)$

Dynamical programming: summary

- Define the value searched through a recurrence relation
- Compute the optimal solution (fill a table)
- Reconstruct the optimal solution

... vs divide and conquer

- For divide and conquer the size of the subproblems is significantly smaller $n \mapsto n/2$
- For dynamical programming: $n \to n-1$, in general
- For divide and conquer the subproblems are independent.
- Direct recursive implementations will not work well for dynamical programming!

- used for solving optimization problems (like dynamical programming)
- Main idea: if there is a local choice to be made, do it in the most greedy way possible!
  Example: for the knapsack problem always take the available object with the highest price.
- For such algorithms to work we need two properties:
  - Being able to get to an optimal solution via greedy choices
  - Optimal substructures: the solution to the problem can be found by solving similar sub problems
- Sometimes one can apply greedy algorithms even if they are not optimal.

Example 1: giving change

- Objective: having coins with values $1, 2, 5, 10, 20$, find a method for reimbursing x using the least number of coins.
- Example for $x = 34$:
  - $\{1, 1, 2, 5, 5, 20\}$: 6 coins
  - $\{2, 2, 10, 20\}$: 4 coins
- Simple greedy algorithm: at each step choose the coin with maximum value, smallest than the remaining sum
- Example: $x = 49$: $20, 20, 5, 2, 2$

Is the Greedy solution optimal?

Theorem. For $c = [20, 10, 5, 2, 1]$ the greedy algorithm is optimal.

By direct inspection one can prove the following:
(a) If x is the total sum, then the largest coin $c^* \leq x$ can be given.

- at most one coin equal to $1, 5, 10$, at most two coins with value 2!

(b) The solution for x is made of c^* and the solution for $x - c^*$!

For other coin values the greedy algorithm may not be optimal!
$C = [1, 10, 21, 34, 70, 100]$ and $x = 140$

- Greedy: $[100, 34, 1, 1, 1, 1, 1, 1]$
- Optimal: $[70, 70]$.

Example 2: Activity selection

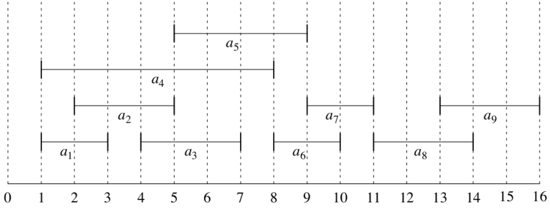
A room is used for different activities:

- $S = \{a_1, ..., a_n\}$ a set of n activities
- a_i starts at time s_i and ends at time f_i
- Two activities a_i, a_j are compatible if the intervals do not intersect: $f_i \leq s_j$ or $f_j \leq s_i$

Problem: Find the largest set of activities which are compatible.

⋆ Example:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Activity selection: greedy approach

- Define a natural order for the activities
- Select activities in this order

Example: Sort activities by starting time, final time, length $f_i - s_i$, etc.

Sorting: Final time

Assume activities are sorted with respect to the final time. If multiple activities start at the same time, the shortest comes first.

⋆ put the first activity in the list A.

⋆ iterate through the activities $k = 2, ..., n$:

- if a_k starts after the last activity in A finishes, put it in the list.

Complexity: $\Theta(n)$ $(+\Theta(n \log n)$ for sorting)

Proving that we have a correct approach

(a) Consider the activity a_x with the first end time f_x. Then there exists an optimal solution containing a_x.
Idea: replace the first activity in an optimal solution with a_x to obtain another solution.

(b) Optimal substructures: if a_x is the greedy choice and A^* is the optimal solution for the remaining activities then $\{a_x\} \cup A^*$ is a solution for the problem.

Other Greedy algorithms

- Dikkstra's algorithm: find path of minimal length between two points in a graph.
  Idea: at the current point, investigate all unvisited neighbors of the current point and compute its minimal distance to the source.

Algorithm conception – conclusion

- size of inputs/outputs
- complexity of brute force solution?
- can a simple rule lead to a solution (greedy: best local choice $\longrightarrow$ best global choice)
- can sorting help in some way?
- can I use smaller subproblems to solve bigger ones? (divide and conquer, dynamical programming)
- can a data structure help to find an efficient solution: tree, queue, file, heap, dictionary, hash table?
- relation to other algorithms
- find references online for optimized solution!