COT 6405
ANALYSIS OF ALGORITHMS

Greedy Algorithms

Computer & Electrical Engineering and Computer Science Department
Florida Atlantic University

Outline

- Greedy algorithms
- Problems solved using greedy
  - Scheduling all intervals (KT – chapter 4.1)
  - Scheduling to minimize lateness (KT – chapter 4.2)
  - Change making problem (CLRS-problem 16-1 page 446)
  - The knapsack problem (CLRS page 425-427)

KT book - *Algorithm Design* by J. Kleinberg and Eva Tardos
CLRS book - *Introduction to Algorithms*, 3rd edition, by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein

Greedy Algorithms

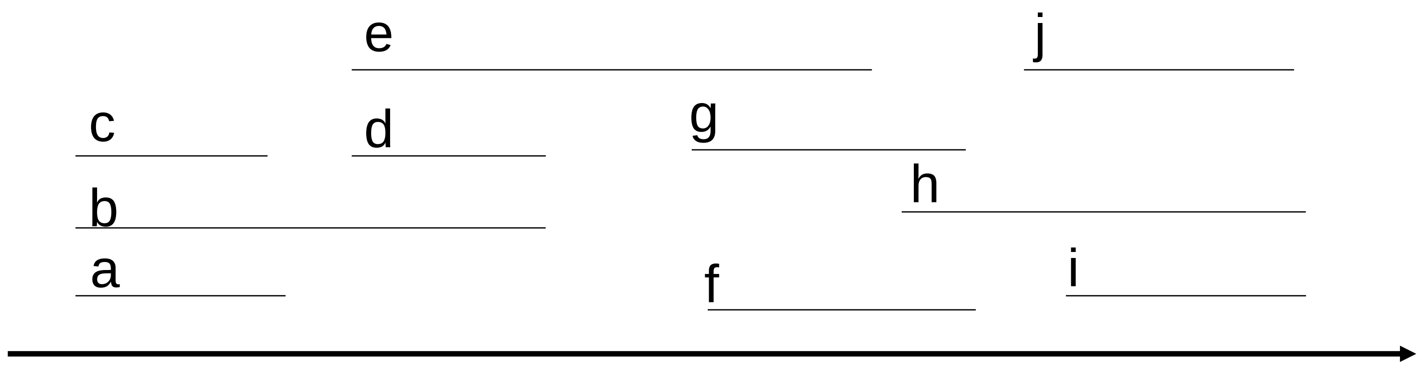
- Used for optimization problems
- When we have to make a choice, make the choice that looks best at the moment
- A greedy algorithm runs over a number of steps: at each step we make a greedy choice and we are left with one subproblem to solve
- A greedy algorithm does not necessarily produce an optimal solution. We need to prove it.

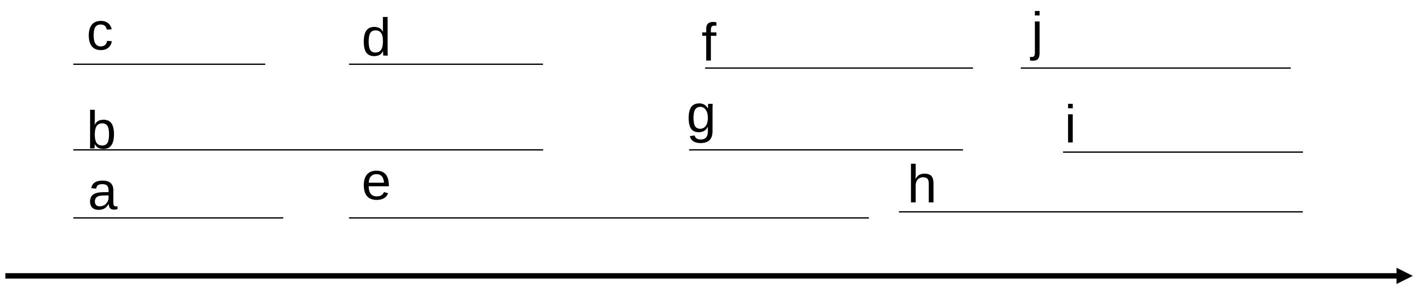
Problem: Scheduling All Intervals

- Interval Partitioning Problem: we have many identical resources available and we want to schedule all the requests using as few resources as possible

- Example:
  - Each request is a lecture to be scheduled in a classroom for a particular interval of time
  - Objective: satisfy all the requests using as few classrooms as possible
  - Constraints: any two classes that overlap in time must be scheduled in different classrooms

Example





- 10 intervals (*a* through *j*)
- all intervals can be scheduled using 3 resources: each row represents a set of intervals that can be scheduled on a single resource

Interval Partitioning Problem

- Define ***depth*** of a set of intervals as the maximum number of intervals that pass over a single point on the time-line
- Property: the number of resources needed is at least the depth of the set of intervals
- Design a greedy algorithm that schedules all intervals using a number of resources equal to the depth
  - Optimality of the algorithm results from the property

Greedy algorithm

- Let d – depth of the set of intervals

<u>Algorithm</u>

sort the intervals by their start times, breaking ties arbitrarily
let $I_1, I_2, \ldots, I_n$ denote the intervals in the sorted order
compute depth d
for j =1, 2, 3, …, n
    for each interval I_i that preceded I_j in the sorted order and overlaps it
        exclude the label of I_i from consideration for I_j
    if there is any label from {1, 2, …, d} that has not been excluded
        assign a nonexcluded label to I_j
    else
        leave I_j unlabeled

Analyzing the algorithm

- Property: using the greedy algorithm, every interval will be assigned a label, and no two overlapping intervals will receive the same label

  Proof:

  consider interval I_j, assume there are t intervals earlier in the sorted order that overlap it

  t+1 overlapping intervals $\Rightarrow$ $t+1 \leq d \Rightarrow t \leq d - 1 \Rightarrow$ there is at least one label available

  No two overlapping intervals receive the same label: the second interval in the list will be assigned a different label

- Property: the proposed greedy algorithm schedules each interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.

RT = $O(n^2)$

Outline

- Greedy algorithms
- Problems solved using greedy
    - Scheduling all intervals (KT – chapter 4.1)
    - **Scheduling to minimize lateness (KT – chapter 4.2)**
    - Change making problem (CLRS-problem 16-1 page 446)
    - The knapsack problem (CLRS page 425-427)

Problem: scheduling to minimize lateness

Problem description:

- a single resource
- a set of *n* requests, where each request *i* has a deadline d_i and requires a contiguous time interval of length t_i
- different requests must be assigned nonoverlapping intervals
- each request will be satisfied, request *i* will be scheduled [s(i),f(i)],

  where $f(i) = s(i) + t_i$
- a request *i* is *late* if it misses the deadline, $f(i) > d_i$
  - ***lateness*** defined as $\ell_i = f(i) - d_i$
  - if $\ell_i = 0$, then the request *i* is not late

Goal: schedule all requests (e.g. compute s(i), f(i) for all i=1..n), using nonoverlapping intervals, such that to minimize the maximum lateness,

$L = \max_i \ell_i$

Example

length = 1, deadline = 2

Job 1

length = 2, deadline = 4

Job 2

length = 3, deadline = 6

Job 3

- First job has length t_1 = 1 and deadline d_1 = 2
- Second job has length t_2 = 2 and deadline d_2 = 4
- Third job has length t_3 = 3 and deadline d_3 = 6

Possible schedules:

lateness = 0	lateness = 2	lateness = 2	
Job 3	Job 1	Job 2	Maximum lateness = 2

lateness = 0	lateness = 0	lateness = 0	
Job 1	Job 2	Job 3	Maximum lateness = 0

- Optimal solution: maximum lateness is 0

What greedy choice to choose?

Several greedy choices are possible for requests (t_i, d_i):

- Schedule jobs in the order of **increasing length t_i**
  - does not always lead to an optimal solution
  - example: two jobs J1(t_1=1,d_1=100) and J2(t_2=10,d_2=10)
    scheduling by increasing length: J1 + J2 $\Rightarrow$ L = 1
    optimal scheduling: J2 + J1 $\Rightarrow$ L =0
- Schedule jobs in the order of **increasing slack $d_i - t_i$**
  - does not always lead to an optimal solution
  - example: two jobs J1(t_1=1,d_1=2) and J2(t_2=10,d_2=10)
    scheduling by increasing slack: J2 + J1 $\Rightarrow$ L = 9
    optimal scheduling: J1 + J2 $\Rightarrow$ L = 1
- Schedule jobs in the order of **increasing deadline d_i**
  - Always yields an optimal solution!

Greedy choice: earliest deadline first

- Greedy choice that always produce an optimal solution:
  - sort the jobs in **increasing order of their deadlines d_i**
  - schedule jobs in this order
- Assume that jobs are labeled in the order of their deadlines (rename them if necessarily)

$$d_1 \leq d_2 \leq \ldots \leq d_n$$

- J1 starts at s and ends at $f(1) = s(1) + t_1$
- J2 starts at f(1) and ends at $f(2) = s(2) + t_2$

… so on

Greedy choice: earliest deadline first

Algorithm:

order the jobs in nondecreasing order of their deadlines

assume for simplicity of notation that $d_1 \le d_2 \le \ldots \le d_n$

initially f = s

for each job in the sorted order

assign job i to the interval s(i) = f and f(i) = f + t_i

f = f + t_i

return the set of scheduled intervals [s(i),f(i)] for i=1…n

RT = O(nlgn)

Correctness: using an exchange argument

- Observation:
  - our algorithm produces a schedule with no idle time
  - there is an optimal schedule with no idle time
- ***Exchange argument method***: start with an optimal solution and gradually modify it, preserving its optimality at each step, transforming it to the solution returned by the greedy algorithm
- Apply the “exchange argument” method to our problem:
  - let O be an optimal schedule
  - let A be the schedule returned by the greedy algorithm

Correctness

- a schedule A' has an inversion if for some $d_j < d_i$, the job i is scheduled before the job j
- the schedule A (greedy algorithm) has no inversion
- Property: ***all schedules with no inversions and no idle times have the same maximum lateness***
  - no inversions & no idle times $\Rightarrow$ schedules can differ in the order in which jobs with identical deadlines are scheduled
  - all these schedules have the same maximum lateness
- Property: ***there is an optimal schedule that has no inversions and no idle time***

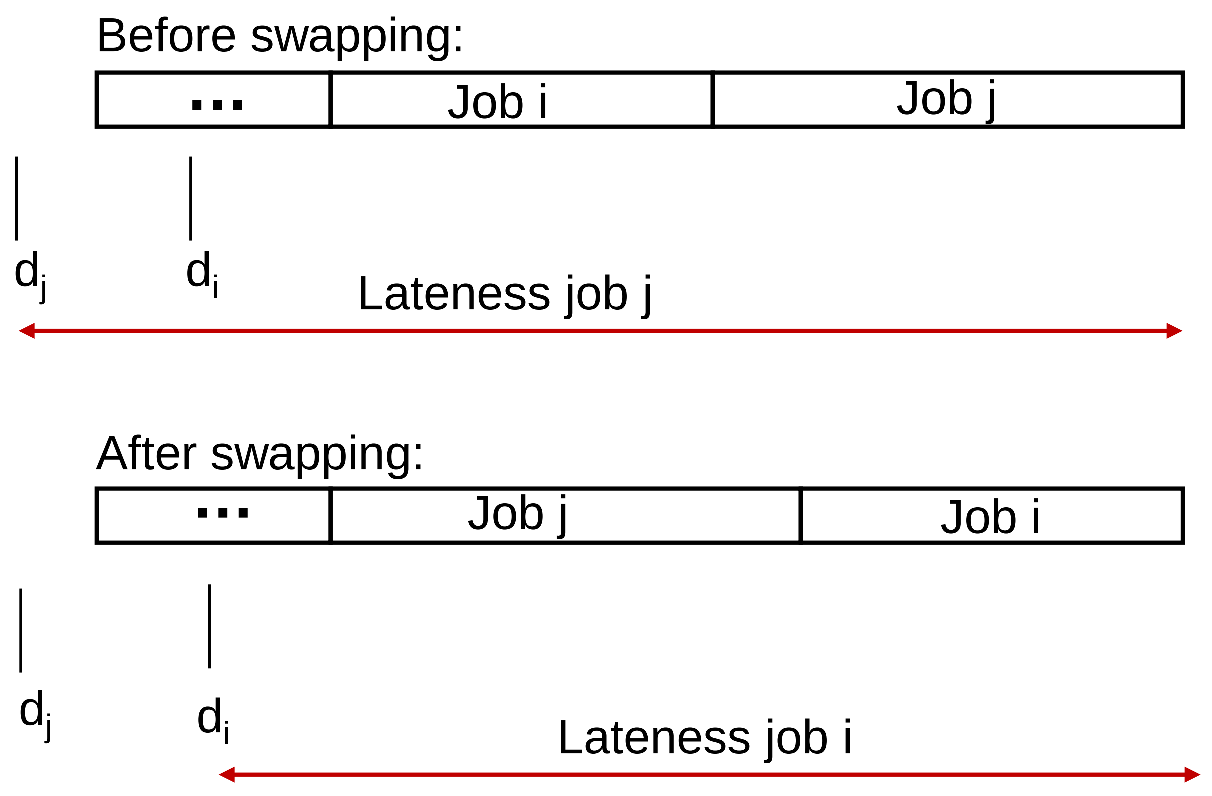
Correctness

Property: ***there is an optimal schedule that has no inversions and no idle time***

Proof:

- If O has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$
  - Examine the schedule starting from the beginning. At some point, the deadline decreases for the first time.
  - This pair of jobs J_i, J_j forms an inversion
- After swapping i and j we get a schedule with one less inversion
- The new swapped schedule has a maximum lateness no larger than that of O

Swapping two consecutive, inverted jobs



- All jobs other than i and j finish at the same time
- The swap does not increase the lateness of job j
- Lateness of job i is $\ell'_i = f'(i) - d_i = f(j) - d_i < f(j) - d_j = \ell_j$
- It follows that the swap does not increase the max lateness

Correctness

- The initial schedule O has at most $\binom{n}{2}$ inversions (all pairs inverted)
- After at most $\binom{n}{2}$ swaps we get an optimal schedule with no inversions

It follows that:

- *The schedule A produced by the greedy algorithm has optimum lateness L*

Outline

- Greedy algorithms
- Problems solved using greedy
  - Scheduling all intervals (KT – chapter 4.1)
  - Scheduling to minimize lateness (KT – chapter 4.2)
  - **Change making problem (CLRS-problem 16-1 page 446)**
  - The knapsack problem (CLRS page 425-427)

Change-making problem

Represent a given amount of money with the fewest number of coins, when the coins available are quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent).

- let q – number of quarters, d – number of dimes
  k – number of nickels, p – number of pennies

Change-making problem

Greedy algorithm:

<u>Make-change(n)</u>

```
S = ϕ
s = 0
while s ≠ n
    x is the largest coin such that s + x ≤ n
    if no such coin found
        return " no solution found"
    S = S ∪ {a coin of value x}
    s = s + x
return S
```

RT = O(n)

Change-making problem

<u>Algorithm 2 (n)</u>

$q = \lfloor n/25 \rfloor$ // number of quarters

$n_q = n \bmod 25$

$d = \lfloor n_q /10 \rfloor$ // number of dimes

$n_d = n_q \bmod 10$

$k = \lfloor n_d /5 \rfloor$ // number of nickels

$n_k = n_d \bmod 5$

$p = n_k$ // number of pennies

$RT = \Theta(1)$

Change-making problem

- example: 89 cents = 3Q + 1D + 4P
- the algorithm produces an optimal solution: there is an optimal solution that makes the greedy choice
- not all coin systems can be solved using the greedy algorithm
  - coin system: 25, 10, 6, 1
  - let n = 12
    - greedy: 10 + 1 + 1 $\rightarrow$ 3 coins
    - optimal: 6 + 6 $\rightarrow$ 2 coins

Change-making problem

Greedy algorithm:

- if $n = 0$, then the optimal solution has no coins
- if $n > 0$, take the largest coin with value $\leq n$

  Let c be this coin. Then use one coin c and recursively solve for $(n - c)$ cents

Greedy Choice Property

Greedy choice property: some optimal solution to the change-making problem for n cents includes a coin with value c, where c is the coin with the largest value $\leq$ n.

Proof:

let O be an optimal solution

- if O contains a coin c, then done
- if O does not contain a coin c:
  - if $1 \leq n < 5$, then $c = 1$; all solutions use only P
  - if $5 \leq n < 10$, then $c = 5$

  If O does not contain N, then it must use only P; replace 5P by 1N $\Rightarrow$ better solution

Greedy Choice Property

- if $10 \leq n < 25$, then $c = 10$

If O does not contain D, then it must use only N and P

Then by replacing a value of 10 (2N, 1N+5P, 10P) with 1D $\Rightarrow$ better solution (smaller number of coins)

- if $n \geq 25$, then $c = 25$

If O does not contain Q, then it must use only D, N, and P;

Then by replacing a value of 25 with 1Q $\Rightarrow$ better solution (smaller number of coins)

3D $\rightarrow$ 1Q + 1N

2D + 1N $\rightarrow$ 1Q

2D + 5P $\rightarrow$ 1Q

…. so on

Since there is always an optimal solution that contains the greedy choice $\Rightarrow$ greedy algorithm produces an optimal solution.

Change-making problem

Problem definition: Suppose that the available coins are in the denominations that are power of c, i.e. the denominations are $c^0, c^1, c^2, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution

Greedy algorithm:

- if $n = 0$, then the optimal solution has no coins
- if $n > 0$, take the largest coin with value $\leq n$

  Let c^j be this coin. Then use one coin c^j and recursively solve for $(n - c^j)$ cents

$\boxed{RT = O(k)}$

Greedy Choice Property

<u>Greedy choice property</u>: some optimal solution to the change-making problem for n cents includes a coin with value c^j, where c^j is the coin with the largest value $\leq$ n.

Proof:

Let O be an optimal solution

Let a_i be the number of coins of denomination c^i used by O

Note that $a_i < c$, otherwise replace c coins c^i with one coin c^{i+1} and improve the solution

- if O contains a coin c^j, then done
- if O does not contain a coin c^j:

  then O contains only coins $c^0, c^1, c^2, \ldots, c^{j-1}$

Greedy Choice Property

$c^j \leq n < c^{j+1}$

$$\sum_{i=0}^{j-1} a_i c^i = n \geq c^j$$

since O is optimal $\Rightarrow$ $a_i \leq c - 1$ for all i = 0,1,2 … j-1

$$\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c-1)c^i = (c-1)\sum_{i=0}^{j-1} c^i = (c-1)\frac{c^j - 1}{c-1} = c^j - 1$$

geometric series

contradiction $\Rightarrow$ greedy algorithm produces an optimal solution

Outline

- Greedy algorithms
- Problems solved using greedy
  - Scheduling all intervals (KT – chapter 4.1)
  - Scheduling to minimize lateness (KT – chapter 4.2)
  - Change making problem (CLRS-problem 16-1 page 446)
  - **The knapsack problem (CLRS page 425-427)**

The knapsack problem

Given:

- n objects and a knapsack
- i = 1,..,n object i has a positive weight w_i and a positive value v_i
- the knapsack can carry a weight $\leq W$

Objective: fill the knapsack such that to maximize the value of the included objects, while respecting the capacity constraints.

Two variations:

• **0-1 knapsack problem**: you can only take the whole object $\rightarrow$ solved optimally using dynamic programming

• **fractional knapsack problem**: you can take fractions of objects $\rightarrow$ solved optimally using greedy

Fractional knapsack problem

Greedy algorithm:

- **greedy choice**: choose the item with the largest v_i/w_i value
- this greedy choice produces an optimal solution

Greedy algorithm

Algorithm

sort objects in decreasing order of v_i/w_i

for $i = 1$ to n

    $x_i = 0$

$load = 0$

$value = 0$

$i = 1$

while $load < W$ and $i \le n$

    if $w_i \le W - load$

    then take whole object i, $x_i = 1$

    else take $x_i = (W\text{-}load)/w_i$ of item i

    $load = load + x_i\, w_i$

    $value = value + x_i\, v_i$

    $i = i + 1$

- RT = O(nlogn)
- example

The greedy algorithm does not work for the 0-1 knapsack problem

Example: