ALGORITHMS – COMTEK3,CCT3 & ESD3
Greedy Algorithms & Dynamic Programming

Ramoni Adeogun
Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

AALBORG UNIVERSITET

Today's journey

- Dynamic programming
- Greedy Algorithms
- Summary and Conclusion

Dynamic Programming

Initiating problem – finding factorial of a number

- Describe a recursive algorithm to find the factorial of n and draw the recursion tree for the factorial of 6.

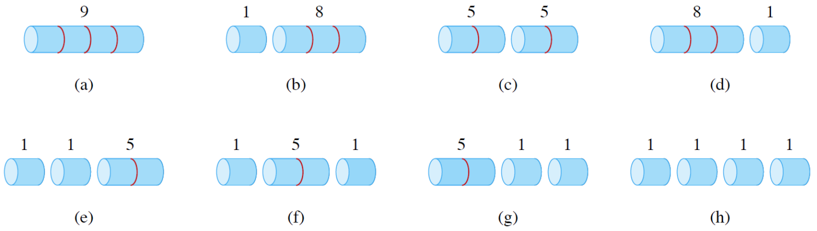
Dynamic Programming

- Not a specific algorithm, but **a technique** (like divide-and-conquer).
- Developed back in the day when “programming” meant “tabular method” (like linear programming).
- Doesn’t really refer to computer programming.
- Used for optimization problems:
  - Find a solution with the optimal value
    - Minimization or maximization

- **Four-step method**
  1. **Characterize** the structure of an optimal solution.
  2. **Recursively define** the value of an optimal solution.
  3. **Compute** the value of an optimal solution, typically in a **bottom-up** fashion.
  4. **Construct** an **optimal solution** from computed information

Example: Rod-Cutting Problem

- **The problem:** Given a rod of length n inches and a table of prices $p_i : i = 1, .., n$ determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
  - Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.
- Consider the case with n = 4 inches
  - $8(2^{n-1})$ cutting options
  - **Optimal cutting**
    - 2 pieces (5+5 = 10)



- **General problem:**

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

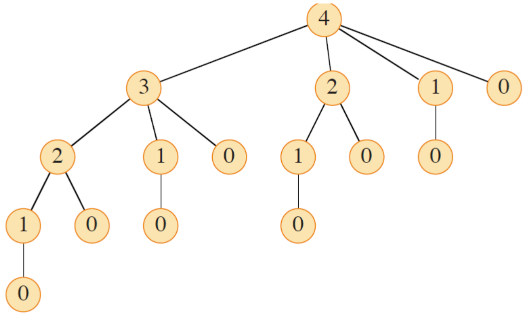
$$r_n = \max_{1 \le i \le n}(p_n, r_i + r_{i-1})$$

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

Recursive top-down implementation

Direct implementation of the formulation for r_n

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max {q, p[i] + CUT-ROD(p, n − i)}
return q
```



Running Time

- Recurrence: $$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$ $\Rightarrow$ $$T(n) = O(2^n)$$

- Exponential runtime in n => very in-efficient algorithm

Optimal Rod-Cutting Using Dynamic Programming

- Dynamic programming (DP) method:
  - Break a complex problem into a collection of subproblems
  - Store the solution to a solved subproblem in a memory-based data structure (array, map, table, etc)
  - Simple retrieval via table lookup
- Two approaches- same asymptotic running time
  - **Top-down** approach based on **memoization**
  - **Bottom-up** based on **tabulation**
  - How are these different?
- Running Time:
  - Double nested loops (recursive)

$$T(n) = O(2^n)$$

- From recursive to DP
  - Time complexity: $O(2^n) \rightarrow \Theta(n^2)$
  - Space complexity: $O(1) \rightarrow O(n)$

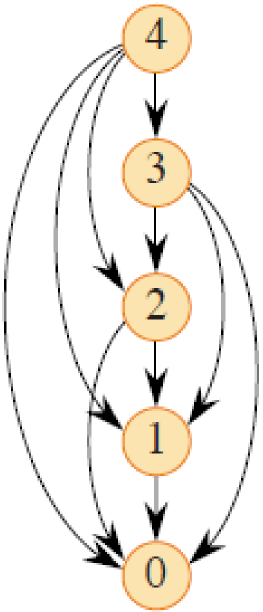
```
MEMOIZED-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                  // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n           // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q                     // remember the solution value for length n
return q

BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
r[0] = 0
for j = 1 to n                       // for increasing rod length j
    q = −∞
    for i = 1 to j                   // i is the position of the first cut
        q = max {q, p[i] + r[j − i]}
    r[j] = q                         // remember the solution value for length j
return r[n]
```

Subproblem Graphs

- Useful tool for dynamic programming
  - understand subproblems and dependencies between them
  - analyze the running time of DP algorithms
  - a reduced or collapsed version of recursion tree
- Definition:
  - a directed graph $G = (V, E)$
    - One vertex for each distinct subproblem.
    - Has a directed edge (x, y) if computing an optimal solution to subproblem x directly requires knowing an optimal solution to subproblem y.
  - edges are defined according to subproblem calls to solve a given subproblem
- Running time:
  - sum of time to solve all subproblems
    - the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph,
  - DP running time: $O(|V| + |E|)$



Subproblem graph for rod cutting problem with n=4

Reconstructing a solution

- DP solutions in our examples so far only returns the value of the optimal solution
  - but not the actual solution -> choices leading to the optimal solution
- Extend the solutions to also output the choices leading to the optimal value

```
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                      // for increasing rod length j
    q = −∞
    for i = 1 to j                  // i is the position of the first cut
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i                // best cut location so far for length j
    r[j] = q                        // remember the solution value for length j
return r and s

PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]            // cut location for length n
    n = n − s[n]          // length of the remainder of the rod
```

Applying Dynamic Programming

Approach:

1. Recursively define the problem, P.
2. Determine **a set S consisting of all subproblems** that must be solved during the computation of a solution to P.
3. Find an order $S_0, S_1, \ldots, S_k$ of the subproblems in S such that during the computation of a solution to S_i only subproblems $S_j (j < i)$ arise.
4. Solve $S_0, S_1, \ldots, S_k$ following this order and store the solution.

Elements of Dynamic Programming

What ingredients should an optimization problem have for it to be suitable for dynamic programming

1. **Optimal substructure**
    - A problem is said to have an optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems
    - Useful in determining the applicability of dynamic programming to the problem

2. **Overlapping subproblems**
    - a problem is said to have overlapping subproblems if the problem can be broken down into subproblems which are **reused several times** or
    - a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems

Greedy Algorithms

Introduction

Main Idea:

- When we have a **choice** to make, make the one **that looks best right now**. Make a locally optimal choice in the hope of getting a globally optimal solution.

- Greedy algorithms don't always yield an **optimal solution**.

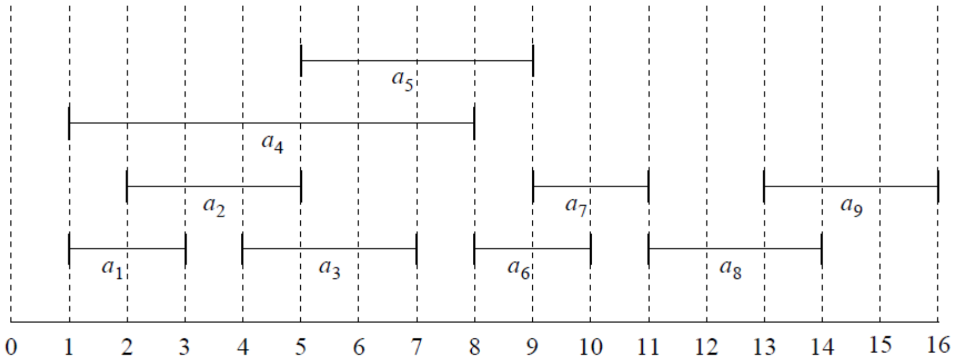
Activity Selection

- n activities require exclusive use of a common resource. For example, scheduling the use of a classroom.
- Set of activities $S = \{a_1, \dots, a_n\}$.
  - a_i needs resource during period $[s_i, f_i)$, which is a half-open interval.
- **Goal:**
  - Select the largest possible set of nonoverlapping (**mutually compatible**) activities.
- **Note:**
  - Could have many other objectives:
    - Schedule room for the longest time.
    - Maximize income rental fees.
    - Schedule tasks with varying processing times on a single machine.
- **Assumption**: activities are sorted by finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Activity Selection

Example: S sorted by finish time

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



Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$ or $\{a_2, a_5, a_7, a_9\}$.

Optimal substructure of activity selection

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$$
$$= \text{activities that start after } a_i \text{ finishes and finish before } a_j \text{ starts}$$

- Activities in S_{ij} are compatible with
  - all activities that finish by f_i , and
  - all activities that start no earlier than s_j

- Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij} .
- Let $a_k \in A_{ij}$ be some activity in A_{ij} . Then we have **two subproblems**:
  - Find mutually compatible activities in S_{ik} (activities that start after a_i finishes and that finish before a_k starts).
  - Find mutually compatible activities in S_{kj} (activities that start after a_k finishes and that finish before a_j starts).

Optimal substructure of activity selection

- Let:
  - $A_{ik} = A_{ij} \cap S_{ik} =$ activities in A_{ij} that finish before a_k starts;
  - $A_{kj} = A_{ij} \cap S_{kj} =$ activities in A_{ij} that start after a_k finishes.
  - Then: $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1.$

- **Claim**: Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj}.

- **Proof***: Suppose we could find a set A'_{kj} of mutually compatible activities in S_{kj}, where $|A'_{kj}| > |A_{kj}|$ and use A'_{kj} instead of A_{kj} when solving subproblem for S_{ij}. Size of the resulting set of mutually compatible activities would be $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A|$. This contradicts assumption that A_ij is optimal.

Recursive greedy algorithm

- Start and finish times are represented by arrays s and f, where f is assumed to be already sorted in monotonically increasing order.
- To start, add fictitious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.
- Procedure RECURSIVE-ACTIVITY-SELECTOR takes as parameters the arrays s and f, index k of the current subproblem, and number n of activities in the original problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

- Initial call: $\text{RECURSIVE} - \text{ACTIVITY} - \text{SELECTOR}(s, f, 0, n)$

Recursive greedy algorithm

Idea:

- The while loop checks $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_k (need $s_m \geq f_k$).
  - If the loop terminates because a_m is found $(m \leq n)$, then recursively solve S_m, and return this solution, along with a_m.
  - If the loop never finds a compatible am $(m > n)$, then just return empty set.

- **Running time:**
  - Each activity is examined exactly once, assuming that activities are already sorted by finish times

$$T(n) = \Theta(n)$$

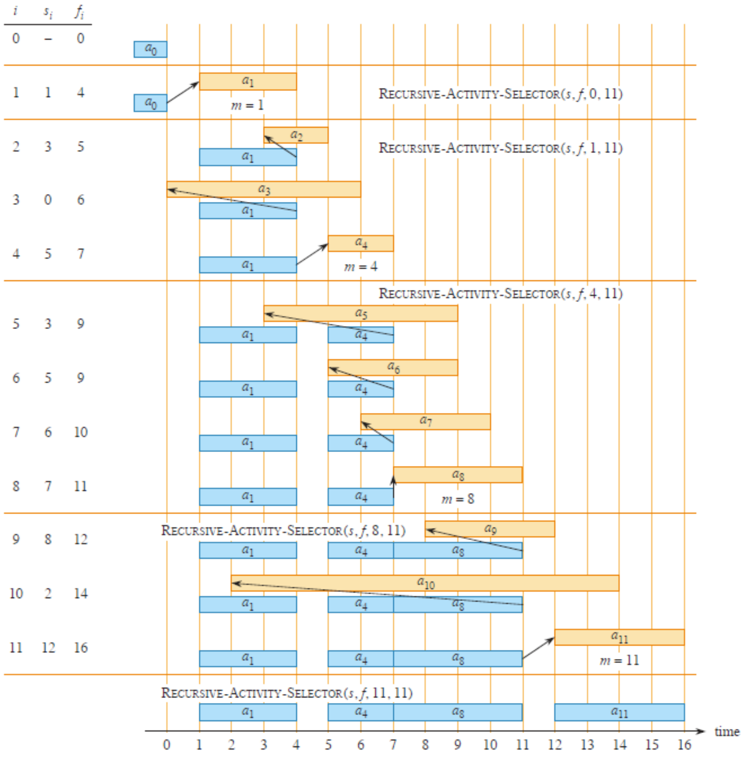
Recursive greedy algorithm - example

- Recursive greedy algorithm on:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

- Result: $\{a_1, a_4, a_8, a_{11}\}$.



AALBORG UNIVERSITET

Iterative greedy algorithm

- We can convert RECURSIVE-GREEDY-ALGORITHM into an iterative algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]          // is a_m in S_k?
        A = A ∪ {a_m}       // yes, so choose it
        k = m               // and continue from there
return A
```

- Exercise [5 mins]: Manually apply GREEDY-ACTIVITY-SELECTOR for the following activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- Running time: $T(n) = \Theta(n)$ if activities are pre-sorted by finish times.
    - What do we need to do if activities are not sorted?

Greedy Strategy

- Greedy approach: Select the choice that seems best now.
- What are the main steps?
  - Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  - Prove that there's always an optimal solution that makes the greedy choice so that it is always safe.
  - Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

- How do we tell if a greedy solution is optimal? Two main ingredients:
  1. **Greedy-choice** property
  2. **Optimal substructure**

Greedy choice property

A globally optimal solution can be assembled by making locally optimal (greedy) choices!

- Dynamic programming
  - Make a choice at each step.
  - Choice depends on *knowing* optimal solutions to subproblems.
    - Solve subproblems first.
  - Solve *bottom-up*.
- Greedy
  - Make a choice at each step.
  - Make the choice *before* solving the subproblems.
  - Solve *top-down*.
- How do we show the greedy-choice property?
  - Look for an optimal solution $\rightarrow \begin{cases} \textbf{Done}; & if\ it\ includes\ the\ greedy-choice \\ \textbf{Modify it to include the greedy choice}; & otherwise \end{cases}$

Greedy versus Dynamic Programming

- What's the difference? We answer this using the "knapsack problem".

- **0-1 knapsack problem:**
  - Given n items.
    - Item i is worth $\$\,i$ and weighs w_i pounds.
  - Find the most valuable subset of items with total weight W.
    - Have to either take an item or not take it—can't take part of it.

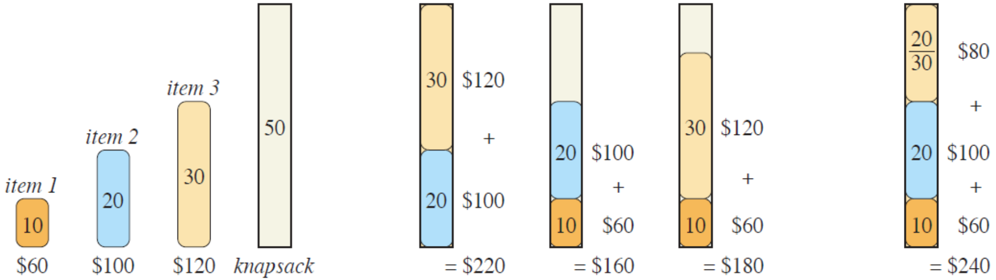
- **Fractional knapsack problem**:
  - Like the 0-1 knapsack problem but can take a **fraction** of an item.

Note:

- Both have **optimal substructure**.
- But the fractional knapsack problem has the **greedy-choice property**, and the $0-1$ knapsack problem does not.

Knapsack problem

- Solution to fractional knapsack problem
  - Rank items by value/weight ratio: $\frac{v_i}{w_i}$
  - Sort items such that $\frac{v_i}{w_i} \geq v_{i+1} / w_{i+1}$ for all i
  - Take items in decreasing order of value/weight => Take all of the items with the greatest value/weight ratio, and possibly a fraction of the next item.
- Example: 3 items and a knapsack that can hold 50 pounds.



Greedy strategy does not work for the 0-1 knapsack but works for fractional knapsack

Summary and Key Take-aways

- Two techniques for solving optimization problems:
  - Dynamic programming
  - Greedy algorithms

- **Dynamic programming:**
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information

- **Greedy algorithms**:
  - When we have a **choice** to make, make the one **that looks best right now**. Make a locally optimal choice in the hope of getting a globally optimal solution.