ETH zürich

Exercise Session 12 – DP, Greedy Algos

Data Structures and Algorithms

These slides are based on those of the lecture, but were adapted and extended by the teaching assistant Adel Gavranović

Today's Schedule

Intro
Follow-up
Learning Objectives
Example: Longest Common Subsequence
Example: Palindromes
Recap: Greedy Choice
Example: Activity Selection
Recursive Problem-Solving Strategies
Huffman Coding
In-Class-Exercise (practical)
Hints for current tasks
Outro

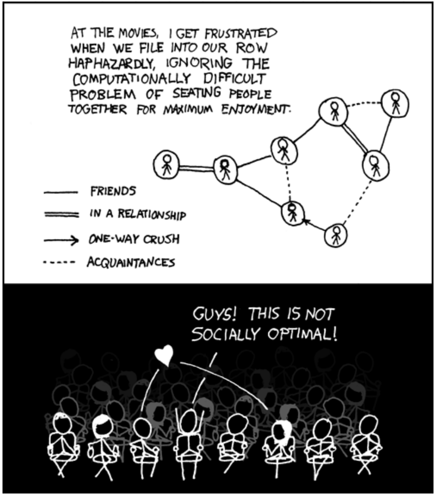
n.ethz.ch/~agavranovic

▸ Exercise Session Material

▸ Adel's Webpage

▸ Mail to Adel

Comic of the Week



▸ xkcd

1. Intro

Intro

- Lots to do; We’re mostly skipping the "Intro"

2. Follow-up

Follow-up from last exercise session

Old Max Flow Exam Question

- The Max Flow question from last time (that we skipped) was from the Exam[1] of 26.01.2018
- It's solvable via a bipartite matching approach

[1] ▸ Exam Solution

3. Learning Objectives

Learning Objectives

- ☐ Gather some intuition on how DP Algorithms look like and work
- ☐ Understand greedy approaches and when it’s reasonable to use
- ☐ Understand Huffman Coding and be able to perform it manually

4. Example: Longest Common Subsequence

DP Example: Longest Common Subsequence

Definition

A *subsequence* of a sequence is generated by removing some or none of the elements of the original sequence. For example, "AC" is a subsequence of "ABC".

Problem

Given two sequences X and Y, find the length of the longest common subsequence of X and Y.

Concrete Problem Instance

Example

X: PROGRAM

Y: ARMOR

X: PROGRAM

Y: ARMOR

Answer?

length 3: ROR

Subproblems?

String X of length m and string Y of length n:
Which subproblems are there?

- if last character matches: $+1$ and shorten both strings by one letter
- shorten X by one, leave Y the same
- shorten Y by one, leave X the same

Recursive Solution

```cpp
int lcs(const std::string& X, const std::string& Y, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }
    if (X[m - 1] == Y[n - 1]) {
        return 1 + lcs(X, Y, m - 1, n - 1);
    } else {
        return std::max(lcs(X, Y, m - 1, n),
                        lcs(X, Y, m, n - 1));
    }
}
```

Dynamic Programming

Instead, we can use dynamic programming to solve this problem by building a table to store the lengths of the longest common subsequences of the prefixes of X and Y:

- Update the table values from the top left to the bottom right.
- If the characters at the current position match, set the current cell value to the diagonal cell value incremented by one, or one if it doesn't exist.
- If they don't match, set the current cell value to the maximum of the left and top cell values, or zero if they don't exist.

DP Table

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R	0	1	1	1	1	1	1
M	0	1	1	1	1	1	2
O	0	1	2	2	2	2	2
R	0	1	2	2	3	3	3

Solution Reconstruction

find LCS (reconstruct solution)?

To find the LCS, trace backwards from the bottom right and mark the starting letter of each diagonal arrow.

X/Y	P	**R**	**O**	G	**R**	A	M
A	0 ↖	0	0	0	0	1	1
R	0	**1** ↑	1	1	1	1	1
M	0	1 ↖	1	1	1	1	2
O	0	1	**2** ←	2 ↖	2	2	2
R	0	1	2	2	**3** ←	3 ←	3

Time Complexity

Question

How does the time complexity of the DP algorithm compare to the naive recursive algorithm?

Naive (Recursive) Algorithm

The naive algorithm has an exponential time complexity of $\mathcal{O}(2^{n+m})$, where n and m are the lengths of the two sequences.

Dynamic Programming Algorithm

The dynamic programming algorithm has a polynomial time complexity of $\mathcal{O}(n \cdot m)$.

5. Example: Palindromes

DP Example: Palindromes

A *palindrome* is a word that reads the same way in either forward or reverse direction. Example: RACECAR.

Formally: $\langle a_1, \ldots, a_n \rangle$ is a palindrome $\iff$

- either $n = 1$, or
- $a_1 = a_n$ and $\langle a_2, \ldots, a_{n-1} \rangle$ is a palindrome [2]

We use an array $A[1..n]$ to store a string of length n. A subarray $A[i..j]$ is called *palindrome in* A if it is a palindrome. Examples:

- `[L, A, R, A]` contains palindromes A (2x), R, L and ARA
- `[A, N, N, A]` contains palindromes A (2x), N (2x), NN and $ANNA$

[2] for $n = 2$ we only require $a_1 = a_2$

DP Example: Palindromes

Task 1.1: Describe an efficient dynamic programming algorithm that finds all pairs (i, j) where $A[i] \dots A[j]$ is a palindrome.

Examples:

- `[L, A, R, A]` $\longrightarrow (1,1), (2,2), (3,3), (4,4), (2,4)$
- `[A, N, N, A]` $\longrightarrow (1,1), (2,2), (3,3), (4,4), (2,3), (1,4)$

Task 1.2: What is the running time of your solution?

- Try to find a DP algorithm!
- How does the table look like?
- How do we traverse the table?
- How do we compute an entry?

Palindromes Task 1.1: Solution

	R	A	C	E	C	A	R
R	1	0	0	0	0	0	1
A	-	1	0	0	0	1	0
C	-	-	1	0	1	0	0
E	-	-	-	1	0	0	0
C	-	-	-	-	1	0	0
E	-	-	-	-	-	1	0
R	-	-	-	-	-	-	1

Palindromes Task 1.1: Solution

Definition of the DP table: We use an $n \times n$ table T with entries that are 0 or 1. For $1 \le i \le j \le n$ let $T[i,j] = 1 \iff \langle A[i], \ldots, A[j] \rangle$ is a palindrome.

Computation of an entry: We distinguish three cases.

1. $1 \le i = j \le n$: $A[i]$ is a palindrome of length 1, thus we set
$$T[i,j] = T[i,i] = 1$$
2. $1 \le i \le n$, $j = i+1 \le n$: We consider palindromes of length 2, and set
$$T[i,i+1] = 1 \iff A[i] = A[i+1]$$
3. $1 \le i \le n$, $i+1 < j \le n$: Let $\langle A[i], \ldots, A[j] \rangle$ be the considered sequence. By definition it is a palindrome if $A[i] = A[j]$ and additionally, $\langle A[i+1], \ldots, A[j-1] \rangle$ is a palindrome. Thus we set
$$T[i,j] = 1 \iff A[i] = A[j] \text{ and } T[i+1, j-1] = 1$$

Palindromes Task 1.1: Solution

Example: $A =$ RACEC***E***R is not a palindrome, but contains non-trivial palindromes CEC and ECE.

	R	A	C	E	C	E	R
R	1	0	0	0	0	0	0
A	-	1	0	0	0	0	0
C	-	-	1	0	1	0	0
E	-	-	-	1	0	1	0
C	-	-	-	-	1	0	0
E	-	-	-	-	-	1	0
R	-	-	-	-	-	-	1

Palindromes: Solution

Task 1.2: What is the running time of the algorithm?

- The table has n^2 entries. We must effecively fill $\frac{n(n+1)}{2} \in \Theta(n^2)$ of these.
- Each table entry can be computed in time $\mathcal{O}(1)$.
- Hence, filling the table is done in $\mathcal{O}(n^2)$ steps.

Task 2.1: Describe how a longest palindrome in A can be extracted from the DP table constructed before.

Traverse table in opposite order of filling, starting from the entry $T[1, n]$. If $T[i, j] = 1$, then $A[i] \dots A[j]$ is a palindrome. The first such entry found is a longest palindrome.

Task 2.2: What is the running time of the reconstruction?

Same as before: $\mathcal{O}(n^2)$.

6. Recap: Greedy Choice

Recap: Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

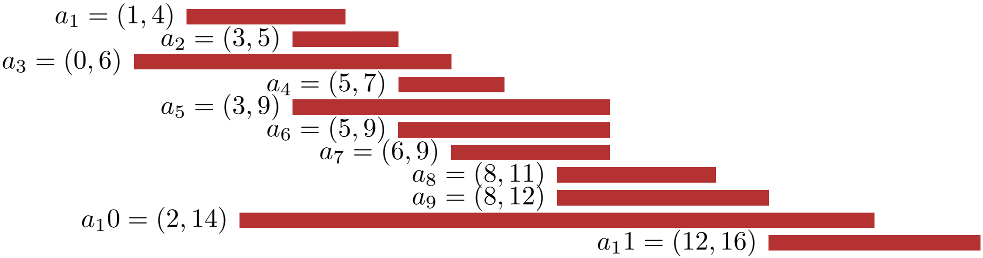
- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local property that does not depend on the solution of the sub-problems.

Examples: Fractional knapsack problem, Huffman coding
Counterexamples: Knapsack problem, optimal binary search tree.

7. Example: Activity Selection

Activity Selection

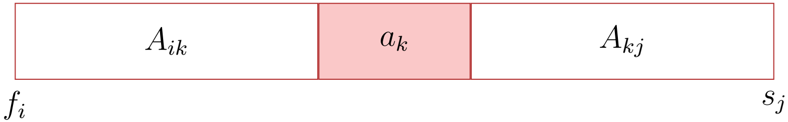
Coordination of activities that use a common resource exclusively. Activities $S = \{a_1, a_2, \ldots, a_n\}$ with start- and finishing times $0 \leq s_i \leq f_i < \infty$, sorted in ascending order by finishing times.



Activity Selection Problem: Find a maximal subset (maximum number of elements) of compatible (non-intersecting) activities.

Dynamic Programming Approach?

Let $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Let A_{ij} be a maximal subset of compatible activities from S_{ij}.
Let $a_k \in A_{ij}$ and $A_{ik} = S_{ik} \cap A_{ij}$, $A_{kj} = S_{kj} \cap A_{ij}$, thus $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



A_{ik} and A_{kj} must be maximal, otherwise $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$ would not be maximal – obviously?

Dynamic Programming Approach?

Why must A_{ik} and A_{kj} be maximal subsets of compatible activities for A_{ij} to be maximal as well?

The reason is that if either A_{ik} or A_{kj} were not maximal, there would exist additional compatible activities that could be added to these subsets.

Dynamic Programming Approach?

Let $c_{ij} = |A_{ij}|$.
Then the following recursion holds

$$c_{ij} = \begin{cases} 0 & \text{falls } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{falls } S_{ij} \neq \emptyset. \end{cases}$$

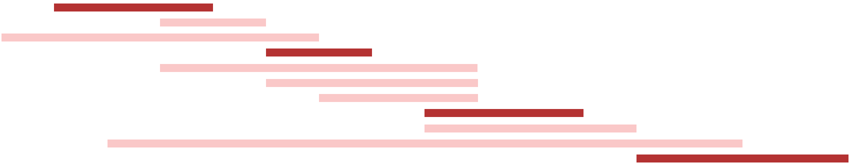
$\Rightarrow$ Dynamic programming.

But there is a simpler alternative.

Greedy

Intuition: Choose the activity that provides the earliest end time (a_1). That leaves maximal space for other activities.

Remaining problem: Activities that start after a_1 ends. (There are no activites that can end before a_1 starts.)



Greedy

Theorem 1

Given: The set of subproblem S_k, and an activity a_m from S_k with the earliest end time. Then a_m is contained in a maximal subset of compatible activities from S_k.

Let A_k be a maximal subset with compatible activities from S_k, and a_j be an activity from A_k with the earliest end time. If $a_j = a_m \Rightarrow$ done. If $a_j \neq a_m$, then consider $A'_k = A_k - \{a_j\} \cup \{a_m\}$. A'_k consists of compatible activities and is also maximal because $|A'_k| = |A_k|$.

■

Algorithm RecursiveActivitySelect(s, f, k, n)

Input: Sequence of start and end points (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ for all i. $1 \leq k \leq n$

Output: Set of all compatible activitivies.

$m \leftarrow k + 1$

while $m \leq n$ and $s_m \leq f_k$ **do**

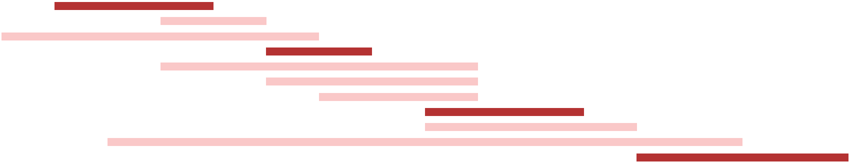
  $m \leftarrow m + 1$

if $m \leq n$ **then**

  return $\{a_m\} \cup$ RecursiveActivitySelect(s, f, m, n)

else

  return $\emptyset$



Algorithm IterativeActivitySelect(s, f, n)

Input: Sequence of start and end points (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ for all i.
Output: Maximal set of compatible activities.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ **to** n **do**
    if $s_m \geq f_k$ **then**
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

Runtime of both algorithms: $\Theta(n)$

Class Problem

Consider the following set of activities with their respective start and finish times:

Activity	Start Time	Finish Time
A	0	4
B	5	6
C	0	2
D	3	7
E	8	9
F	5	9

Exercise: Find the maximal set of compatible activities that can be scheduled using the greedy algorithm for activity selection.

Solution: Greedy Algorithm

1. Sort activities based on finish times:

$$C \to A \to B \to D \to E \to F$$

2. Initialize the list of selected activities:

$$\text{Selected} = \{C\}$$

3. Iterate through the remaining activities:

   - A is not compatible with C (skip A)
   - B is compatible with C $\implies$ Selected = {C, B}
   - $\dots$

4. The maximal set of compatible activities is:

$$\text{Selected} = \{C, B, E\}$$

8. Recursive Problem-Solving Strategies

Recursive Problem-Solving Strategies

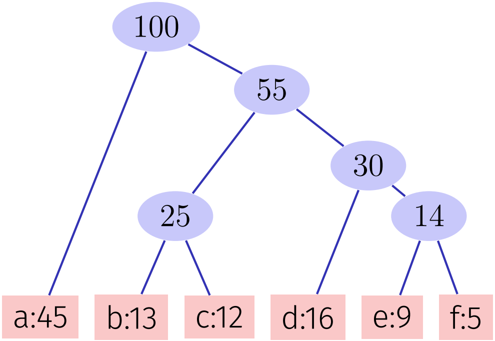
Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems	Optimal Substructure, Greedy Choice Property
DFS, BFS, all Permutations, Tree Traversal	n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP	Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT	Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

9. Huffman Coding

Huffman's Idea

Tree construction bottom up

- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Algorithm Huffman(C)

Input: code words $c \in C$
Output: Root of an optimal code tree

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
  allocate a new node z
  z.left $\leftarrow$ ExtractMin(Q) // extract word with minimal frequency.
  z.right $\leftarrow$ ExtractMin(Q)
  z.freq $\leftarrow$ z.left.freq $+$ z.right.freq
  Insert(Q, z)
return ExtractMin(Q)

10. In-Class-Exercise (practical)

Complement the DP implementation to compute an optimal search tree. $\longrightarrow$ CodeExpert

11. Hints for current tasks

Huffman Coding

Huffman: Frequencies

Use `std::unordered_map` **(**`#include <unordered_map>`**)**

```
std::unordered_map<char, int> frequencies;
// ...
++frequencies['a'];
++frequencies['x'];
++frequencies['a'];

// A map is a container of key-value pairs (std::pair).
// Output all entries:
for (auto x:observations){
  std::cout << "observations of " << x.first << ":" << x.second << '\n';
}
```

Huffman: Min Heap

Use `std::priority_queue` **(**`#include <queue>`**)**

```
struct MyClass {
  int x;
  MyClass(int X): x{X} {}
};

struct compare {
  bool operator() (const MyClass& a, const MyClass& b) const {
    return a.x < b.x;
  }
};

std::priority_queue<MyClass, std::vector<MyClass>, compare> q;
q.push(MyClass(10));
```

Huffman: Shared Pointers [optional]

Shared Pointers `std::shared_ptr` (`#include <memory>`)

```
struct SNode {
  int value;
  std::shared_ptr<SNode> left;
  std::shared_ptr<SNode> right;
  SNode(int v): value{v}, left{nullptr}, right{nullptr} {}
};

// A graph in which node 7 is shared:   //    0
SNode* root = new SNode(0);             //   / \
root->left = new SNode(1);              //  1   2
root->right = new SNode(2);             //     / \
root->right->left = new SNode(7);       //     \ /
root->right->right = root->right->left; //      7

root->left = nullptr; // Node 1 can and should be deallocated (deleted) now
root->right->left = nullptr; // Node 7 must not yet be deallocated
root->right->right = nullptr; // Node 7 can and should be deallocated now
```

Automated memory management, see Code Expert example

Huffman: Tree Nodes

```
using SharedNode = std::shared_ptr<Node>;

struct Node {
  char value;
  int frequency;
  SharedNode left;
  SharedNode right;

  // constructor for leafs
  Node(char v, int f):
    value{v}, frequency{f}, left{nullptr}, right{nullptr}
  {}

  // constructor for inner nodes
  Node(SharedNode l, SharedNode r):
    value{0}, frequency{l->frequency + r->frequency}, left{l}, right{r}
  {}
};
```

Huffman

Gegeben sind fünf Buchstaben mit relativer Häufigkeit (Anzahl Zugriffe) wie folgt. Erstellen Sie mit Hilfe des Huffman-Algorithmus einen optimalen Codierungsbaum. Tragen Sie den resultierenden Code in der Tabelle ein.

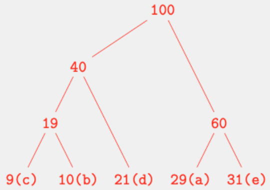
Five characters (keys) with relative frequency (number of accesses) are given as follows. Using the Huffman algorithm provide an optimal code tree. Enter the corresponding code into the table.

char	a	b	c	d	e
freq	29	10	9	21	31
Code					

Huffman – Solution

Gegeben sind fünf Buchstaben mit relativer Häufigkeit (Anzahl Zugriffe) wie folgt. Erstellen Sie mit Hilfe des Huffman-Algorithmus einen optimalen Codierungsbaum. Tragen Sie den resultierenden Code in der Tabelle ein.

Five characters (keys) with relative frequency (number of accesses) are given as follows. Using the Huffman algorithm provide an optimal code tree. Enter the corresponding code into the table.



b and c must have 3 bits and (only) share the first two bits,
a and e must have 2 bits and (only) share the first bit,
d must have two bits and (only) share the first bit with b and c.

char	a	b	c	d	e
freq	29	10	9	21	31
Code	10	001	000	01	11

12. Outro

General Questions?

See you next time

Have a nice week!