Algolab 2018 – Week 5

Overview

Today's tutorial: 'Advanced' Techniques

- Greedy Algorithms
  - Example 1: Knapsack variations
  - Proof Technique: Exchange Argument
  - Example 2: Interval Scheduling
  - Proof Technique: Staying Ahead
- Split & List

Greedy Algorithms

"Greed is good.
Greed is right.
Greed works."

— Wall Street 1987
by Gordon Gekko

- Sometimes **locally optimal** choices result in a **globally optimal** solution.
- This is when we can apply **greedy algorithms**.
- **However** often choices that seem best in a particular moment turn out not to be optimal in the long run (e.g. in chess, life, etc.).

A greedy approach typically has the following steps:

1. **Modelling**: realise that your task requires you to construct a set that is in some sense **globally optimal**.
2. **Greedy choice**: given already chosen elements $c_1, \ldots, c_{k-1}$, decide how to choose c_k, based on some **local optimality criterion**.
3. **Prove** that elements obtained in this way result in a **globally optimal** set.
4. **Implement** the greedy choice to be as efficient as possible.

Equal Weights Knapsack

Given a maximum weight W and a set of n items with **values** $v_1, \ldots, v_n$ and **weights** $w_1 = \ldots = w_n = 1$, find an **assignment** $x_1, \ldots, x_n$ that maximizes $\sum v_i x_i$ subject to $\sum w_i x_i \leq W$ and $x_i \in \{0, 1\}$.

Fractional Knapsack

Given a maximum weight W and a set of n items with **values** $v_1, \ldots, v_n$ and **weights** $w_1, \ldots, w_n$, find an **assignment** $x_1, \ldots, x_n$ that maximizes $\sum v_i x_i$ subject to $\sum w_i x_i \leq W$ and $0 \leq x_i \leq 1$.

Example: Two Knapsack Problems

Greedy choice

Idea:

- sort items **decreasingly** according to $\frac{v_i}{w_i}$ ratio
- choose items in that order until knapsack is **full**

Example: Two Knapsack Problems

Prove that this yields an optimal solution.

General method: Exchange Argument

- Let A be the choices made by the greedy algorithm.
- Let O be an optimal solution.
- **Goal:** Assuming A and O are 'not equal', modify O to create O' such that
  1. O' is at least as good as O, and
  2. O' is 'more like' A.

Tip: One good way to do the last bit is to assume O is an optimal solution which **'follows A the longest'**, that is has the **longest common prefix with A**.

Look at the first point at which O differs from A and exchange some (further) element to get O' which agrees with A at that point as well.

Example: Two Knapsack Problems

Prove that this yields an optimal solution.

Proof Sketch

- Let $A = \{x_1, \ldots, x_n\}$ be the choices made by the greedy algorithm.
- Let $O = \{x'_1, \ldots, x'_n\}$ be an optimal solution (which agrees with A the longest, i.e. shares the longest prefix).
- If $A = O$ we are done.
- Let $i \in [n-1]$ be the **smallest index** such that:

$$x_j = x'_j \qquad \text{for all } j < i \qquad \text{and} \qquad x_i \neq x'_i$$

- $A = \{x_1, \ldots, x_{i-1}, x_i, \ldots, x_n\}$
- $O = \{x_1, \ldots, x_{i-1}, x'_i, \ldots, x'_n\}$

Example: Two Knapsack Problems

Proof Sketch (cont.)

- Because of the greedy choice:
  - $x_i > x'_i$
  - $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ for all $j > i$
- How to make O 'closer' to A:
  - We want to **replace** x'_i by x_i in O
  - $(x_i - x'_i)w_i$ weight has to be available:
    - **Equal Weights**: All objects have the same weight
    - **Fractional**: Fractional assignments allow removing the exact weight
- O' has equal or better value
- O' is closer to A
- **Contradiction!**

Example: Two Knapsack Problems

Implement the algorithm efficiently.

1. **Sort** the items according to value to weight ratio.
2. Iterate over the items in this order.
3. While there is space, add the items to the knapsack.

This takes time $O(N \log N)$.

General Knapsack Problem

- Knapsack is NP-Complete, so greedy doesn't work! **Right?**
- If a greedy algorithm doesn't skip any item $\rightarrow$ Greedy is optimal!

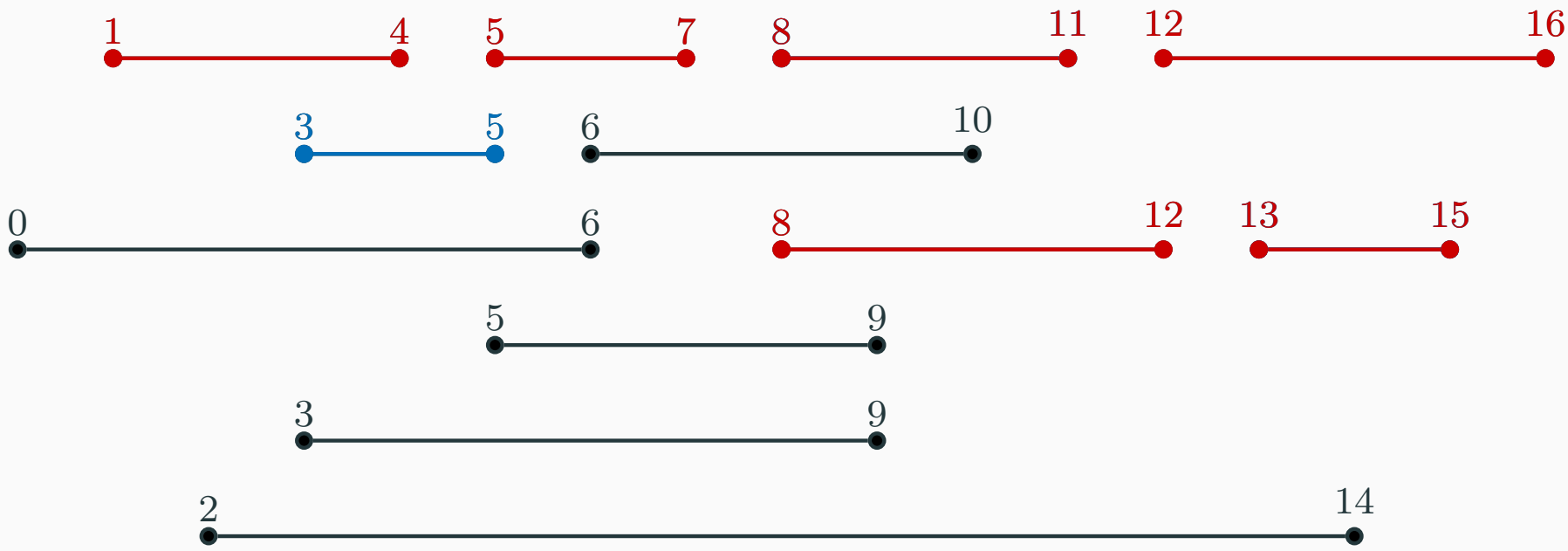
Warning

- Greedy **usually works** for many instances of most problems
- Easy to convince yourself greedy works
- Greedy **rarely** works

Example: Interval scheduling

- Your CPU needs to execute N jobs described by time intervals $[s_i, f_i]$.
- Job i starts at time s_i and ends at time f_i.
- Two jobs are **compatible** if their intervals are disjoint.
- **Goal:** find the maximum number of mutually compatible jobs.

Example: Interval Scheduling



$$A = \{[3, 5], [8, 11], [13, 15]\}$$

Optimal:

$B = \{[1, 4], [5, 7], [8, 11], [12, 16]\}$ also $C = \{[1, 4], [5, 7], [8, 12], [13, 15]\}$

Example: Interval scheduling

Modelling done for us in the problem description—find the maximum set of compatible jobs.

Example: Interval scheduling

Greedy choice: decide how to choose the job i_k given already chosen jobs $i_1, \ldots, i_{k-1}$.

Natural candidates:

- **Earliest start time** – among compatible jobs, take the one with smallest s_k.
- **Earliest finish time** – among compatible jobs, take the one with smallest f_k.
- **Shortest length** – among compatible jobs, take the one with smallest $f_k - s_k$.
- **Fewest conflicts** – among compatible jobs, take the one which conflicts with the least amount of other compatible jobs.

Example: Interval scheduling

Earliest start time

Earliest finish time

Shortest length

Fewest conflicts

Which one do you think will work?

Example: Interval scheduling

Earliest start time

WRONG

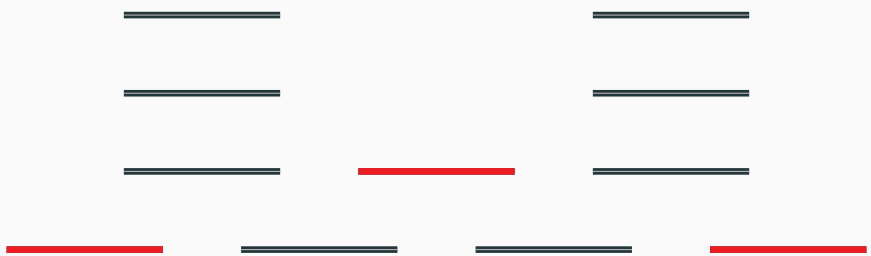
Example: Interval scheduling

Shortest length

WRONG

Example: Interval scheduling

Fewest conflicts



WRONG

Example: Interval scheduling

Earliest finish time

Maybe???

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Staying Ahead

- Let $A = \{i_1, \ldots, i_n\}$ be the jobs chosen according to earliest finish time.
- Let $O = \{j_1, \ldots, j_m\}$ be an optimal solution (sorted by finish time).
- If $|A| = |O|$ we are done.
- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').

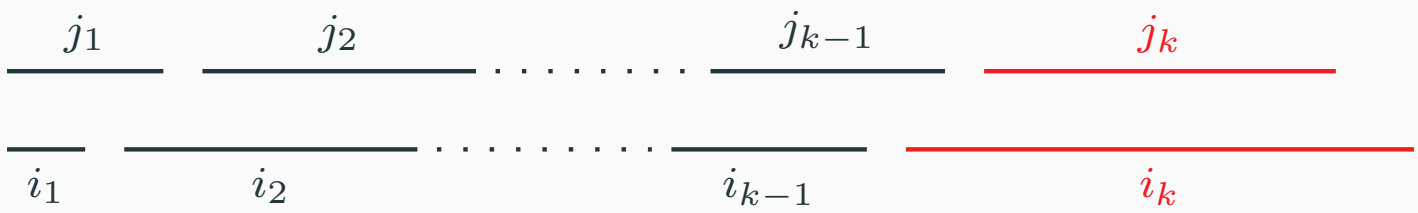
Example: Interval scheduling

Prove that earliest finish time is correct.

Proof Sketch

- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').
- Proof by induction on k.
- **Base case, $k = 1$: Clearly holds!**
- Let $k > 1$ and assume it holds for $k - 1$ (i.e. $f_{i_{k-1}} \leq f_{j_{k-1}}$).
- Could it happen that $f_{i_k} > f_{j_k}$? **NO!**

  WHY?! $f_{i_{k-1}} \leq f_{j_{k-1}}$ and j_k is **compatible** with j_{k-1}, thus with i_{k-1} as well. The greedy algorithm would select j_k instead of i_k.



Example: Interval scheduling

Prove that earliest finish time is correct.

Proof Sketch (cont.)

- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').
- For all $k \leq n$, we have $f_{i_k} \leq f_{j_k}$.
- Since $m > n$, there is j_{n+1} in O with:

$$s_{j_{n+1}} > f_{j_n} \qquad \text{and thus} \qquad s_{j_{n+1}} > f_{i_n}.$$

- Therefore, j_{n+1} is **compatible** with $i_1, \ldots, i_n$, but **does not** belong to A.

Contradiction! □

Example: Interval scheduling

Implement the algorithm efficiently.

1. **Sort** the jobs according to increasing finish time.
2. Iterate over the jobs in this order.
3. For each job with interval $[s_i, f_i]$, add the job if s_i is greater than the finish time of the last job that was added.

This takes time $O(N \log N)$.

Example: Checking Change

ATM has bills with values $1, 10$, and 25 and is supposed to give you 42. What is the minimum number of bills used?

Greedy choice

$$1 \times 25 + 1 \times 10 + 7 \times 1 = 42$$

Bills used: **9**.

Optimal

$$4 \times 10 + 2 \times 1 = 42$$

Bills used: **6**.

Conclusion:

- Some (but not all!) problems can be solved with a greedy approach.
- Deciding how to make the greedy choice can be non-obvious.
- We can check whether the greedy solution works using an **exchange argument** or a **staying ahead** argument.
- Disproving greedy solutions tends to be easy
- Convincing yourself greedy works is easy (**even when it doesn't!**)

Split & List

Brute Force

Brute force: some problems are **hard** and we only know how to solve them by **trying everything**.

However, one can often do it a **little bit smarter**:

1. Heuristics (important in practice, not in AlgoLab)
2. **Improve worst case complexity :)**

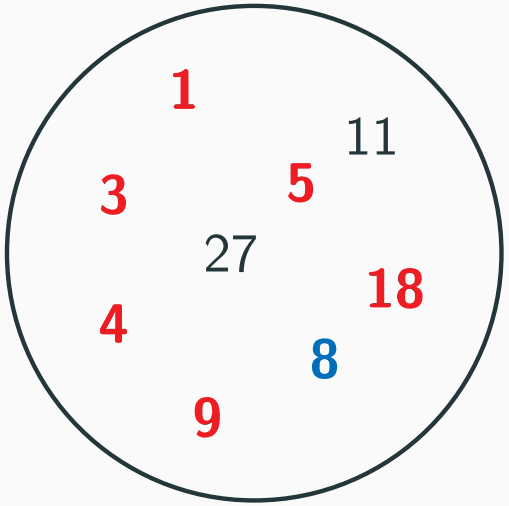
We will see a technique called **Split & List**.

This technique is why there is 'DES' and 'triple-DES' but no 'double-DES'...

Example: Subset Sum

Given a set $S \subseteq \mathbb{N}$, is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

- $S = \{1, 3, 4, 5, 8, 9, 11, 18, 27\}$
- $k = 8$? **YES!** $S' = \{1, 3, 4\}$ or $S' = \{8\}$
- $k = 1000$? **NO!**
- $k = 37$? **YES!** $S' = \{1, 4, 5, 9, 18\}$



NP-Complete :(

n is small: brute force **n is small: brute force**

Check **all subsets**!

Recursive/Iterative algorithm

k is small: DP **k is small: DP**

EXERCISE!

Subset Sum — Recursive

Example: Subset Sum

Given a set $S = \{s_1, \ldots, s_n\} \subseteq \mathbb{N}$, is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

We want a recursive definition of $f(i, j) :=$ 'is there $S' \subseteq \{s_1, \ldots, s_i\}$ s.t. $\sum_{s \in S'} s = j$'.

- Base cases:
  $f(i, 0) =$ `true`, for all i, and
  $f(0, j) =$ `false`, for all $j > 0$.
- $f(i, j) = f(i - 1, j - s_i) \vee f(i - 1, j)$

Recursive algorithm:

```
bool f(int i, int j) {
    if (j == 0) return true;
    if ((i == 0 && j > 0) || j < 0) return false;
    return f(i - 1, j - elements[i]) || f(i - 1, j);
}
```

Time complexity: $O(2^n)$, ok for $n \approx 25$.

Subset Sum — Iterative

How can we iterate over all subsets of an n element set?

Trick: encode the set in an integer.

```
bool subsetsum(int k) {
  for (int s = 0; s < 1<<n; ++s) { // Iterate through all subsets
      int sum = 0;
      for (int i = 0; i < n; ++i) {
          if (s & 1<<i) sum += elements[i]; // If i-th element in subset
      }
      if (sum == k) return true;
  }
  return false;
}
```

Time complexity: $O(n \cdot 2^n)$, ok for $n \approx 25$.

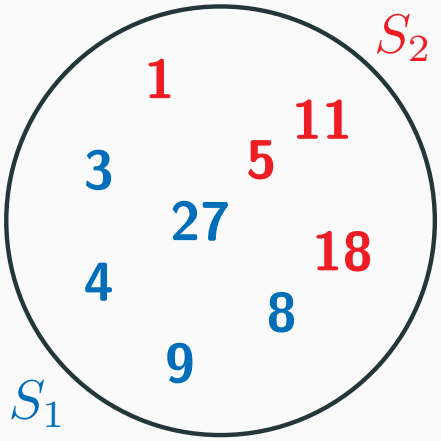
Subset Sum — Faster? Split & List

Split S into $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \varnothing$ of size $\approx \frac{n}{2}$.

List all subset sums of S_1 and S_2 into L_1 and L_2

Lemma: The following statements are equivalent:

- There is a $S' \subseteq S$ with $\sum_{s \in S'} s = k$
- There are $S_1' \subseteq S_1$ and $S_2' \subseteq S_2$ such that $\sum_{s \in S_1'} s + \sum_{s \in S_2'} s = k$



Idea: use second statement to check the first.

Algorithm sketch:

- Sort L_2
- For each k_1 in L_1 check if there is k_2 in L_2 (**binary search**!) such that $k_1 + k_2 = k$.

Time complexity: $O(n \cdot 2^{n/2})$, ok for $n \approx 50$. :)