CS 341: Algorithms

Lecture 5: Greedy algorithms

Slides due to Éric Schost and based on lecture notes by many other CS341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2026

Master theorem – correction

Suppose that $\boldsymbol{a \geq 1}$ and $\boldsymbol{b > 1}$. Consider the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + \boldsymbol{f(n)}, \qquad T(n) = d \quad (n \leq 1), \qquad \inf_n f(n) > 0$$

Let $\boldsymbol{x = \log_b a}$ (so $a = b^x$).

$$\text{Then } T(n) \in \begin{cases} \Theta(\boldsymbol{f(n)}) & \text{if } f(n) \in \Omega(n^{x+\varepsilon}), \text{ for some } \varepsilon > 0 \\ & \text{and } \textbf{regularity condition} \\ \Theta(\boldsymbol{n^x \log n}) & \text{if } f(n) \in \Theta(n^x) \\ \Theta(\boldsymbol{n^x}) & \text{if } f(n) \in O(n^{x-\varepsilon}), \text{ for some } \varepsilon > 0 \end{cases}$$

Regularity: $af(n/b) \leq cf(n)$ for all sufficiently large n and some $c < 1$.

Regularity

Theorem

If $f(n)/n^{x+\varepsilon}$ is non-decreasing for some $\varepsilon > 0$ then the regularity condition holds.

Proof

$$\frac{f(n/b)}{(n/b)^{x+\varepsilon}} \leq \frac{f(n)}{n^{x+\varepsilon}}$$

$$b^x f(n/b) \leq b^{-\varepsilon} f(n)$$

$$a f(n/b) \leq c f(n)$$

where we recall $a = b^x$ and let $c = b^{-\varepsilon} < 1$.

Closest Pair

ClosestPair(A)
A: array of points of size n sorted by y
$x_{median} \leftarrow$ **Median**(A)
$L, R \leftarrow$ **Partition**(A, x_{median})
$\delta_L \leftarrow$ **ClosestPair**(L)
$\delta_R \leftarrow$ **ClosestPair**(R)
$\delta \leftarrow \min(\delta_L, \delta_R)$
return **TransversePairs**(A, x_{median}, δ)

TransversePairs(A, x_{median}, δ)
A: array of points of size n sorted by y
opt $\leftarrow \delta$
$A \leftarrow [(x, y) \in A \mid -\delta < x - x_{median} < \delta]$
for $P \in A$:
    for $Q \in A$ such that $y_P \leq y_Q < y_P + \delta$
        compute $d(P, Q)$, replace opt if better
return opt

Greedy Algorithms

Goals

This chapter: the greedy paradigm through examples

- job scheduling
- interval scheduling
- more scheduling
- fractional knapsack
- and so on

Computational model:

- all input quantities we work with (weights, capacities, deadlines, ...) fit in a word
- unit cost

Greedy algorithms

Context: we are trying to solve a **combinatorial optimization** problem:

- have a **large, but finite,** set $\mathcal{S}$ (orderings of tasks, sets of possible tasks, trees, ...)
- want to find an element E in $\mathcal{S}$ that **minimizes / maximizes** a cost function

Greedy strategy:

- build E step-by-step
- don't think ahead, just try to improve as much as you can at every step
- simple algorithms, but it is often **hard** to prove correctness

A recurrent proof pattern

- let $\boldsymbol{E}_{\mathbf{greedy}}$ be the greedy solution
- let $\boldsymbol{E}$ be any other solution
- transform $\boldsymbol{E}$ into $\boldsymbol{E}_{\mathbf{greedy}}$ progressively, making sure the cost never increases

Example: Huffman

Review from CS240: the **Huffman tree**

- we are given **"frequencies"** $f_1, \dots, f_n$ **for characters** $c_1, \dots, c_n$
- want a code (character $c_i \mapsto$ word w_i in $\{0, 1\}$)
- want prefix-free: build a **binary tree**
- minimize expected codeword length, $\mathbb{E}[\mathrm{length}(w_i)] = \sum_i f_i \mathrm{length}(w_i)$

Greedy strategy: we build the tree **bottom up**.

- create n single-letter trees
- define the **frequency** of a tree as the sum of the frequencies of the letters in it
- build the final tree by joining smaller trees
- greedy choice: **join the two trees with the least frequencies**

Claim

this minimizes $\sum_i f_i \times \{\text{length of } w_i\}$

Proof: takes some work. Progressively transform any other tree into the greedy one.

Minimizing completion time

The problem

Input:

- n jobs, with processing times $[t(1), \ldots, t(n)]$

Output:

- an ordering of the jobs that minimizes the **sum T of the completion times**
- **completion time:** how long it took **(since the beginning)** to complete a job

Example:

- $n = 5$, processing times $[2, 8, 1, 10, 5]$
- in this order,
  $T = 2 \ + \ (8+2) \ + \ (1+8+2) \ + \ (10+1+8+2) \ + \ (5+10+1+8+2) = 70$
- in the order $[1, 2, 8, 5, 10]$,
  $T = 1 \ + \ (2+1) \ + \ (8+2+1) \ + \ (5+8+2+1) \ + \ (10+5+8+2+1) = 57$
- in the order $[1, 2, 5, 8, 10]$,
  $T = 1 \ + \ (2+1) \ + \ (5+2+1) \ + \ (8+5+2+1) \ + \ (10+8+5+2+1) = 54$

Greedy algorithm

Lemma

Let $L = [e_1, \ldots, e_i, e_{i+1}, \ldots, e_n]$ and $L' = [e_1, \ldots, e_{i+1}, e_i, \ldots, e_n]$ be permutations of $[1, \ldots, n]$ that differ by a swap of e_i and e_{i+1}.

The cost difference is $\textbf{cost}(L') - \textbf{cost}(L) = t(e_{i+1}) - t(e_i)$.

Proof

$$\begin{aligned} \textbf{cost}(L') - \textbf{cost}(L) = \quad & (n-i+1)t(e_{i+1}) + (n-i)t(e_i) \\ & - (n-i-1)t(e_i) - (n-i)t(e_{i+1}) \\ = \; & t(e_{i+1}) - t(e_i). \end{aligned}$$

Greedy algorithm

Algorithm: order the jobs in **non-decreasing** processing times

To prove correctness

- let $L = [e_1, \ldots, e_n]$ be a permutation of $[1, \ldots, n]$
- suppose that **we don't** have $t(e_1) \leq t(e_2) \cdots \leq t(e_n)$
- then we can find a better permutation L' by removing an inversion

Greedy algorithm

Algorithm: order the jobs in **non-decreasing** processing times

To prove correctness

- let $L = [e_1, \ldots, e_n]$ be a permutation of $[1, \ldots, n]$
- suppose that **we don't** have $t(e_1) \leq t(e_2) \cdots \leq t(e_n)$
- then we can find a better permutation L' by removing an inversion

1. by assumption there exists i such that $\boldsymbol{t(e_i) > t(e_{i+1})}$
2. use the lemma: $\textbf{cost}(L') - \textbf{cost}(L) = t(e_{i+1}) - t(e_i) < 0$
3. cost is improved

Greedy algorithm

Algorithm: order the jobs in **non-decreasing** processing times

Review from CS240

- optimal static order for linked list implementation of dictionaries
- same result (up to reverse), same proof

$$\textbf{cost}(L) = \sum_{i=1}^{n} i f(e_i)$$

Interval scheduling

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

Output:

- a choice T of intervals that **do not overlap** and that has **maximal cardinality**
- $\textbf{finish}(I_j) = \textbf{start}(I_k)$ not an overlap

Example: A car rental company has the following requests for a given day:

I_1: 2pm to 8pm

I_2: 3pm to 4pm

I_3: 5pm to 6pm

Optimum is $T = [I_2, I_3]$.

A few attempts

Attempt 1:

- pick the interval with the **earliest starting time** that creates no conflict

A few attempts

Attempt 1:

- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

A few attempts

Attempt 1:

- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

Attempt 2:

- pick the **shortest interval** that creates no conflict

A few attempts

Attempt 1:

- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

Attempt 2:

- pick the **shortest interval** that creates no conflict
- **no**, for example

A few attempts

Attempt 1:

- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

Attempt 2:

- pick the **shortest interval** that creates no conflict
- **no**, for example 

Attempt 3:

- pick the interval with the **fewest overlaps** that creates no conflict

A few attempts

Attempt 1:

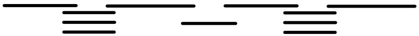
- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

Attempt 2:

- pick the **shortest interval** that creates no conflict
- **no**, for example

Attempt 3:

- pick the interval with the **fewest overlaps** that creates no conflict
- **no**, for example



A few attempts

Attempt 1:

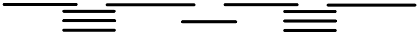
- pick the interval with the **earliest starting time** that creates no conflict
- **no**, previous example

Attempt 2:

- pick the **shortest interval** that creates no conflict
- **no**, for example

Attempt 3:

- pick the interval with the **fewest overlaps** that creates no conflict
- **no**, for example



Attempt 4:

- pick the interval with the **earliest finish time**, that creates no conflict

A $\Theta(n \log(n))$ implementation

Greedy($\boldsymbol{I} = [I_1, \ldots, I_n]$)

1. $T \leftarrow [\,]$
2. sort $\boldsymbol{I}$ by non-decreasing finish time
3. **for** $k = 1, \ldots, n$ **do**
4.   if I_k does not overlap the last entry in T
5.   append I_k to T

Correctness: greedy is optimal

To prove correctness

- $T = [\boldsymbol{x_1}, \ldots, \boldsymbol{x_p}]$ be the intervals chosen by algorithm
- $S = [\boldsymbol{y_1}, \ldots, \boldsymbol{y_q}]$ be any feasible choice (sorted by increasing finish time)
- want to prove $\boldsymbol{p} \geq \boldsymbol{q}$

Correctness: greedy is optimal

To prove correctness

- $T = [\boldsymbol{x_1}, \ldots, \boldsymbol{x_p}]$ be the intervals chosen by algorithm
- $S = [\boldsymbol{y_1}, \ldots, \boldsymbol{y_q}]$ be any feasible choice (sorted by increasing finish time)
- want to prove $\boldsymbol{p} \geq \boldsymbol{q}$

Proof by induction: **for** $\boldsymbol{k = 0, \ldots, q}$, $\boldsymbol{p \geq k}$, $[\boldsymbol{x_1, \ldots, x_k, y_{k+1}, \ldots, y_q}]$ **feasible and sorted by increasing finish time**

Correctness: greedy is optimal

To prove correctness

- $T = [\boldsymbol{x_1, \ldots, x_p}]$ be the intervals chosen by algorithm
- $S = [\boldsymbol{y_1, \ldots, y_q}]$ be any feasible choice (sorted by increasing finish time)
- want to prove $\boldsymbol{p \geq q}$

Proof by induction: for $\boldsymbol{k = 0, \ldots, q}$, $\boldsymbol{p \geq k}$, $[\boldsymbol{x_1, \ldots, x_k, y_{k+1}, \ldots, y_q}]$ feasible and sorted by increasing finish time

- OK for $k = 0$, so we suppose true for some $k < q$, and prove for $k + 1$
- **finish$(\boldsymbol{x_k}) \leq$ finish$(\boldsymbol{y_{k+1}})$** and $[\boldsymbol{x_1, \ldots, x_k, y_{k+1}}]$ **feasible**, so the algorithm didn't stop at x_k. So $\boldsymbol{p \geq k + 1}$.
- by definition, **finish$(\boldsymbol{x_{k+1}}) \leq$ finish$(\boldsymbol{y_{k+1}})$**. So we can replace y_{k+1} by x_{k+1} and we get $[\boldsymbol{x_1, \ldots, x_{k+1}, y_{k+2}, \ldots, y_q}]$, which is still feasible and sorted by increasing finish time

Interval coloring

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textsf{start}(I_j)$, $f_j = \textsf{finish}(I_j)$

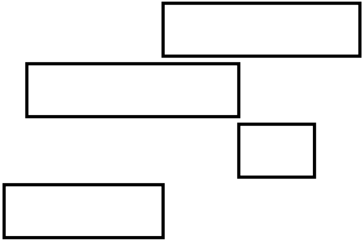
Output:

- assignment of **colors** to each interval
- overlapping intervals get **different colors**
- **minimize** the number of colors used overall

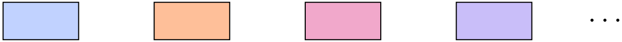
Remarks:

- another version: finding classrooms for lectures
- colors $\leftrightarrow$ indices $1, 2, 3, \ldots$
- $\textsf{finish}(I_j) = \textsf{start}(I_k)$ not an overlap

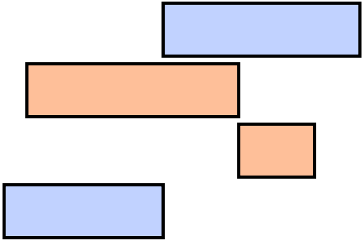
An example



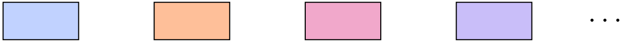
Available colors:



An example

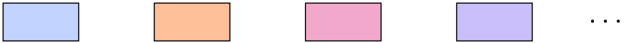


Available colors:



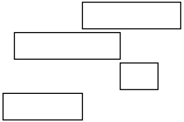
A few attempts

Available colors:



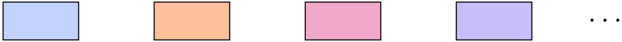
Attempt 1:

- sort intervals by **non-decreasing finish times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



A few attempts

Available colors:

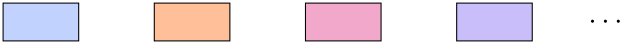


Attempt 1:

- sort intervals by **non-decreasing finish times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed

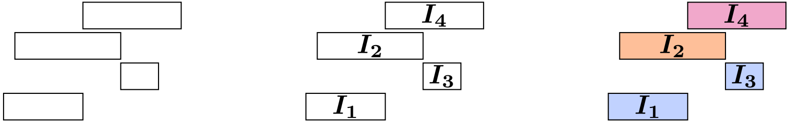
A few attempts

Available colors:



Attempt 1:

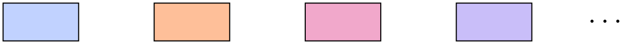
- sort intervals by **non-decreasing finish times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



- does not give the optimal

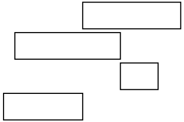
A few attempts

Available colors:



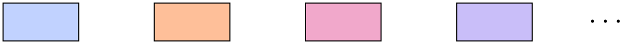
Attempt 2:

- sort intervals **from shortest to longest**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



A few attempts

Available colors:

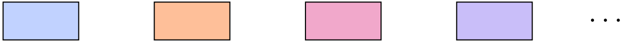


Attempt 2:

- sort intervals **from shortest to longest**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed

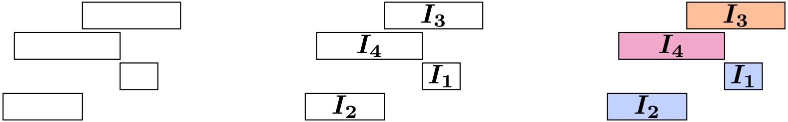
A few attempts

Available colors:



Attempt 2:

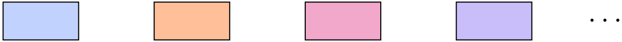
- sort intervals **from shortest to longest**
- for $j = 1, \dots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



- does not give the optimal

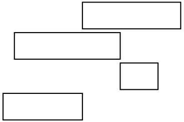
A few attempts

Available colors:



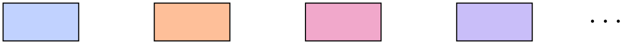
Attempt 3:

- sort intervals **by non-decreasing start times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



A few attempts

Available colors:

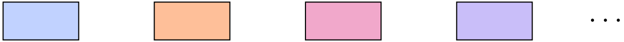


Attempt 3:

- sort intervals **by non-decreasing start times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed

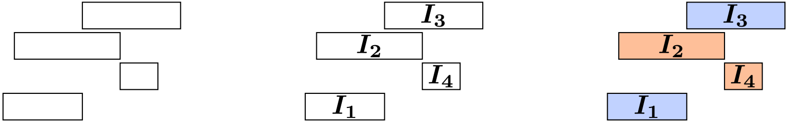
A few attempts

Available colors:



Attempt 3:

- sort intervals **by non-decreasing start times**
- for $j = 1, \ldots, n$, use for I_j **the smallest existing color** (smallest index) that creates no conflict, or **a new color** if needed



- maybe, needs proof

Correctness of the third attempt

Claim

Suppose the output uses $\boldsymbol{k}$ colors. Then, **we cannot use fewer than $\boldsymbol{k}$**.

Proof

- suppose we color I_t with color k
- so I_t overlaps with $k-1$ intervals, say $I_{\alpha_1}, \ldots, I_{\alpha_{k-1}}$ seen previously
- **because we sorted by start time**, for all $j = 1, \ldots, k-1$, $\boldsymbol{s_{\alpha_j} \leq s_t < f_{\alpha_j}}$
- so at time s_t, we can't do with less than k colors

Correctness of the third attempt

Claim

Suppose the output uses $\boldsymbol{k}$ colors. Then, **we cannot use fewer than $\boldsymbol{k}$**.

Proof

- suppose we color I_t with color k
- so I_t overlaps with $k-1$ intervals, say $I_{\alpha_1}, \ldots, I_{\alpha_{k-1}}$ seen previously
- **because we sorted by start time**, for all $j = 1, \ldots, k-1$, $\boldsymbol{s_{\alpha_j} \leq s_t < f_{\alpha_j}}$
- so at time s_t, we can't do with less than k colors

Remark: could also write a proof closer in spirit to the previous ones:

- $T = [\boldsymbol{c_1, \ldots, c_n}]$ are the colors chosen by algorithm
- $S = [\boldsymbol{d_1, \ldots, d_n}]$ is any other feasible choice
- prove that S uses more (or same) number of colors by transforming it into T progressively

A $\Theta(n \log n)$ implementation

Color

1. sort the array $A = [[s_i, \text{“start”}, i]]_{1..n}$ **cat** $[[f_i, \text{“finish”}, i]]_{1..n}$ by time
   (to break ties: **finish** comes before **start**)
2. $C[1..n] \leftarrow$ array (of color indices)
3. $H \leftarrow$ **min-heap** of available color indices, initially empty
4. $k \leftarrow 0$
5. **for all entries of** A (in increasing order)
   if interval i starts, set $C[i]$ = min element in H (if not empty) or $++k$ (if empty)
   if interval i ends, insert $C[i]$ in H
6. **return** C

Remark: picking **any** available color would work too.