Plan for Today

1. Recap
2. Interval Coloring
3. Set Cover
4. Knapsack
Plan

1. Recap
2. Interval Coloring
3. Set Cover
4. Knapsack
The greedy method

*Building a feasible solution incrementally, and “hope” it is optimal.*

Suppose any feasible solution is an $n$ tuple $[x_1, \ldots, x_n]$; the greedy method usually proceeds as follows:

- Select a suitable ordering/permutation $\pi$ of indexes
- Start with $X = []$, and follow the selected ordering to “grow” it
- Given $x_{\pi(1)}, \ldots, x_{\pi(i)}$, select $x_{\pi(i+1)}$ such that
  - no constraint is violated or cannot be repaired in later stages
  - $f$ is increased most among all possible choices (greedy)
Minimizing sum of completion times

**Minimum total completion time**

**Input:** A set of $n$ jobs, taking $[\ell_1, \ldots, \ell_n]$ time to complete.

**Output:** A sequential schedule with minimum sum of completion times.

Formulate mathematically:

$$\min_{\pi \in P_n} \sum_{i=1}^{n} \sum_{j=1}^{i} \ell_{\pi}(j) = \min_{\pi \in P_n} \sum_{j=1}^{n} (n-j+1) \ell_{\pi}(j)$$

**Claim:** $\ell_{\pi}(1) \leq \ell_{\pi}(2) \leq \cdots \leq \ell_{\pi}(n)$.

So, schedule the jobs according to their lengths (the shorter the sooner).
Minimizing weighted sum of completion times

Minimum total weighted completion time

Input: $n$ jobs, job $i$ takes $\ell_i$ time to complete and has weight $w_i$.
Output: A schedule with minimum weighted sum of completion times.

Formulate mathematically:

$$\min_{\pi \in P_n} \sum_{i=1}^{n} w_{\pi(i)} \sum_{j=1}^{i} \ell_{\pi(j)} = \min_{\pi \in P_n} \sum_{j=1}^{n} \ell_{\pi(j)} \sum_{i=j}^{n} w_{\pi(i)}$$

- take an arbitrary pair $i$ and $j$
- swap them
- figure out the change of the cost
Change of cost

\[\ell_{\pi(1)} \sum_{k=1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(i-1)} \sum_{k=i-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i)} \sum_{k=i}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i+1)} \sum_{k=i+1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(j-1)} \sum_{k=j-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j)} \sum_{k=j}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j+1)} \sum_{k=j+1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(n)} \sum_{k=n}^{n} W_{\pi(k)} \]

\[\ell_{\pi(1)} \sum_{k=1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(i-1)} \sum_{k=i-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i)} \sum_{k=i}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i+1)} \sum_{k=i+1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j-1)} \sum_{k=j-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j)} \sum_{k=j}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j+1)} \sum_{k=j+1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(n)} \sum_{k=n}^{n} W_{\pi(k)} \]

\[\ell_{\pi(1)} \sum_{k=1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(i-1)} \sum_{k=i-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i)} \sum_{k=i}^{n} W_{\pi(k)} \]

\[\ell_{\pi(i+1)} \sum_{k=i+1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j-1)} \sum_{k=j-1}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j)} \sum_{k=j}^{n} W_{\pi(k)} \]

\[\ell_{\pi(j+1)} \sum_{k=j+1}^{n} W_{\pi(k)} \]

\[\vdots\]

\[\ell_{\pi(n)} \sum_{k=n}^{n} W_{\pi(k)} \]
An important trick

Suppose there is an “optimal” permutation $\sigma$.

How can we get to $\sigma$ from an arbitrary permutation $\pi$?
An important trick

Suppose there is an “optimal” permutation $\sigma$.

How can we get to $\sigma$ from an arbitrary permutation $\pi$?

How did bubble sort work?
An important trick

Suppose there is an “optimal” permutation $\sigma$.

How can we get to $\sigma$ from an arbitrary permutation $\pi$?

How did bubble sort work?

By swapping neighboring positions!

Swap the pair $i$ and $i + 1$, how would the cost change?

$$\ell_{\pi(i)} w_{\pi(i+1)} \text{ vs. } \ell_{\pi(i+1)} w_{\pi(i)}$$

i.e. $$\frac{\ell_{\pi(i)}}{w_{\pi(i)}} \text{ vs. } \frac{\ell_{\pi(i+1)}}{w_{\pi(i+1)}}$$

So, sort the jobs by the ratio $$\frac{\ell_{\pi(i)}}{w_{\pi(i)}}$$
Interval Selection

Input: A set of $n$ intervals $\{[s_i, f_i]\}$ with start time $s_i$ and finish time $f_i$
Output: A maximum number of pairwise disjoint intervals

Suppose $[s_{i_1}, f_{i_1}), \ldots, [s_{i_k}, f_{i_k})$ is an optimal solution, arranged increasingly.

If there exists a job $[s_{i_0}, f_{i_0})$ with a smaller finishing time $f_{i_0}$ than $f_{i_1}$
Then claim that $[s_{i_0}, f_{i_0}), [s_{i_2}, f_{i_2}), \ldots, [s_{i_k}, f_{i_k})$ is also an optimal solution.

In particular, can choose $f_{i_0}$ to be the smallest finishing time.

Iterate the argument for $[s_{i_2}, f_{i_2}), \ldots, [s_{i_k}, f_{i_k})$. 
Plan

1. Recap
2. Interval Coloring
3. Set Cover
4. Knapsack
Interval Coloring

Input: A set of \( n \) intervals \([s_i, f_i)\) with start time \( s_i \) and finish time \( f_i \)
Output: Least number of colors such that overlapping intervals are colored differently

Possible greedy algorithm:
Interval Coloring

Input: A set of $n$ intervals $\{[s_i, f_i]\}$ with start time $s_i$ and finish time $f_i$
Output: Least number of colors such that overlapping intervals are colored differently

Possible greedy algorithm:
- select a maximum number of pairwise disjoint intervals
- color them with a new color
- delete them
- repeat
A lower bound

**Definition (Depth).** Maximum number of intervals that pass same point.

**Theorem.** We need at least as many colors as the depth.

**Corollary.** If an algorithm uses $d$ colors, then it is optimal.
Algorithm 1: \texttt{greedyIntervalColor}(A)

1. \( A \leftarrow \text{sort}(A) \) // by starting time \( s_i \)
2. \( d \leftarrow 1 \)
3. \( \text{color}[1] \leftarrow 1 \)
4. \( \text{finish}[1] \leftarrow f_1 \)
5. \( \textbf{for} \ i = 2, \ldots, n \ \textbf{do} \)
6. \( \quad \text{neednewcolor} \leftarrow \text{true} \)
7. \( \quad c \leftarrow 1 \)
8. \( \quad \textbf{while} \ c \leq d \ \textbf{do} \)
9. \( \quad \quad \textbf{if} \ \text{finish}[c] \leq s_i \ \textbf{then} \)
10. \( \quad \quad \quad \text{color}[i] \leftarrow c \)
11. \( \quad \quad \quad \text{finish}[c] \leftarrow f_i \)
12. \( \quad \quad \quad \text{neednewcolor} \leftarrow \text{false} \)
13. \( \quad \quad \textbf{break} \)
14. \( \quad \textbf{else} \)
15. \( \quad \quad c++ \)
16. \( \quad \textbf{if} \ \text{neednewcolor} \ \textbf{then} \)
17. \( \quad \quad d++ \)
18. \( \quad \quad \text{color}[i] \leftarrow d \)
19. \( \quad \quad \text{finish}[d] \leftarrow f_i \)
20. \( \textbf{return} \ (d, \text{color}) \)
**Analysis**

- \( \textit{finish}[c] \) denotes the latest finishing time under color \( c \)
- new interval \( A \) can be assigned color \( c \) if \( s_i \geq \textit{finish}[c] \)
- otherwise start a new color
- figuring out the depth \( d \) on the fly
- time complexity \( O(nd) \)
- in the worst case \( d = \Theta(n) \), hence time complexity \( O(n^2) \)
- any idea on improvement?
- construct a counterexample for our initial try?
Plan

1. Recap
2. Interval Coloring
3. Set Cover
4. Knapsack
Set cover

Set Cover

Input: A set $U = \{1, \ldots, n\}$ and a collection of subsets
$\mathcal{F} = \{A_i \subseteq U : i = 1, \ldots, m\}$
Output: “Smallest” $C \subseteq \mathcal{F}$ such that $\bigcup_{C \in C} C = U$
A greedy approximation algorithm

Algorithm 2: greedySetCover($U, \mathcal{F}$)

1. $C \leftarrow \emptyset$
2. while $U \neq \emptyset$ do
3.     choose $A \in \mathcal{F}$ that maximizes $|A \cap U|$.
4.     $U \leftarrow U \setminus A$
5.     $C \leftarrow C \cup \{A\}$
6. return $(C)$
Analysis

At the $i$-th iteration, let $U_i$ be the uncovered and let greedy selects $A_i$. $U_i$ has a cover of size at most $\text{opt}$, which does not include $A_1, \ldots, A_{i-1}$. At least one set that covers at least $|U_i|/\text{opt}$ elements in $U_i$.

Thus, $|U_{i+1}| = |U_i| - |U_i \cap A_i| \leq \left(1 - \frac{1}{\text{opt}}\right)|U_i|$

Greedy will run at most $O(\text{opt} \log n)$ iterations.
Tightness

- choose \( n = 2^{k+1} - 2 \)
- cannot be improved unless \( P = NP \) (Dinur & Steurer, 2014)
- weighted version, follow the textbook
- linear-time implementation (Bar-Yehuda & Even, 1981)
Plan

1. Recap
2. Interval Coloring
3. Set Cover
4. Knapsack
Knapsack

Input: Profits $P = [p_1, \ldots, p_n]$; weights $W = [w_1, \ldots, w_n]$; capacity $C$.

Feasible solution: $X = [x_1, \ldots, x_n]$ such that $\sum_i w_i x_i \leq C$, for all $i$, $x_i \in \{0, 1\}$ or $x_i \in \mathbb{N}$ or more relaxedly $x_i \in [0, 1]$

Output: A feasible solution $X$ that maximizes $\sum_i p_i x_i$.

Possible greedy choices:

- choose the items in decreasing order of profit
- choose the items in increasing order of weight
- choose the items in decreasing order of profit per weight
Justification for relaxed Knapsack

Mathematically:

\[
\max \sum p_i x_i \quad 0 \leq x_i \leq 1, \sum_i x_i w_i \leq C
\]

Do a change-of-variable:

\[
\max \sum y_i p_i / w_i \quad 0 \leq y_i \leq w_i, \sum_i y_i \leq C
\]

Pick a pair \(i\) and \(j\) in any feasible solution \(Y\).

Swap \(i\) with \(j\); how would the cost change?
Justification for relaxed Knapsack

Mathematically:

$$\max \sum_i x_i \text{ s.t. } 0 \leq x_i \leq 1, \sum_i x_i w_i \leq C$$

Do a change-of-variable:

$$\max \sum_i y_i \text{ s.t. } 0 \leq y_i \leq w_i, \sum_i y_i \leq C$$

Pick a pair $i$ and $j$ in any feasible solution $Y$.

Swap $i$ with $j$; how would the cost change?

Ooops, may violate the constraint $y_i \leq w_i$. 
Algorithm 3: greedyRelaxedKnapsack\((P, W, C)\)

1. \((P, W) \leftarrow \text{sort}(P, W)\) // sort by \(p_i/w_i\)
2. \(i \leftarrow 1\)
3. \(curW \leftarrow 0\) // current total weight
4. \textbf{while} \(curW < C\) and \(i \leq n\) \textbf{do}
5. \(x_i \leftarrow \min\{1, (C - curW)/w_i\}\)
6. \(curW + = x_i w_i\)
7. \(i + +\)
8. \textbf{return} \(X\)

- Clearly, time complexity \(O(n \log n)\)
What if $x_i \in \{0, 1\}$

Would straightforward modification of our greedy algorithm work?

Any ideas on other possible greedy algorithms?

Any ideas on any polytime algorithm?

In fact, this is an NP-hard problem.
Pseudo-polynomial time algorithm using dynamic programming.
What if $x_i \in \{0, 1\}$

Would straightforward modification of our greedy algorithm work?

Any ideas on other possible greedy algorithms?
What if $x_i \in \{0, 1\}$

Would straightforward modification of our greedy algorithm work?

Any ideas on other possible greedy algorithms?

Any ideas on any polytime algorithm?
What if $x_i \in \{0, 1\}$

Would straightforward modification of our greedy algorithm work?

Any ideas on other possible greedy algorithms?

Any ideas on any polytime algorithm?

In fact, this is an NP-hard problem.
What if $x_i \in \{0, 1\}$

Would straightforward modification of our greedy algorithm work?

Any ideas on other possible greedy algorithms?

Any ideas on any polytime algorithm?

In fact, this is an **NP-hard** problem.

Pseudo-polynomial time algorithm using dynamic programming.
Greedy is not too bad

W.l.o.g., assume \( w_i \leq C \) for all \( i \).

- sort by \( p_i/w_i \) decreasingly
- find \( k \) such that \( \sum_{i=1}^{k} w_i \leq C < \sum_{i=1}^{k+1} w_i \)
- return \( \max\{\sum_{i=1}^{k} p_i, p_{k+1}\} \)

Claim: \( \sum_{i=1}^{k} p_i \leq \text{opt} \leq \text{opt}_{\text{relaxed}} < \sum_{i=1}^{k+1} p_i \)

Hence greedy \( \geq \text{opt}/2 \).

Example: \( P = [2, C], W = [1, C] \)