Algorithms: COMP3121/3821/9101/9801

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

LECTURE 5: THE GREEDY METHOD

Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Activity selection problem

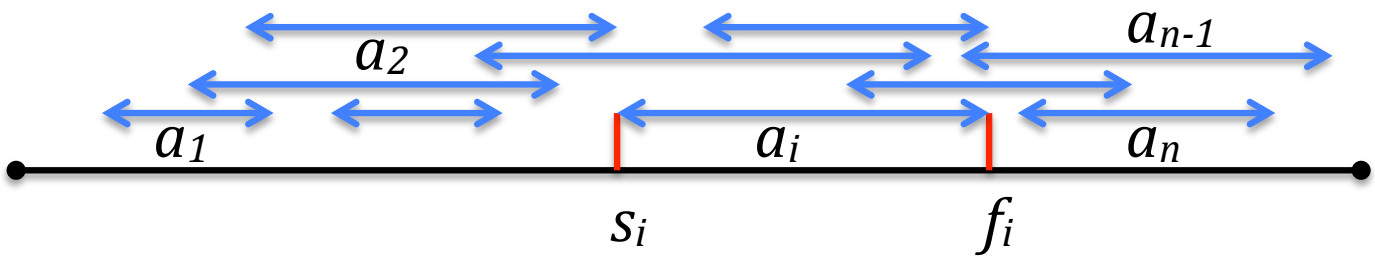
- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a *maximum size* subset of compatible activities.

Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

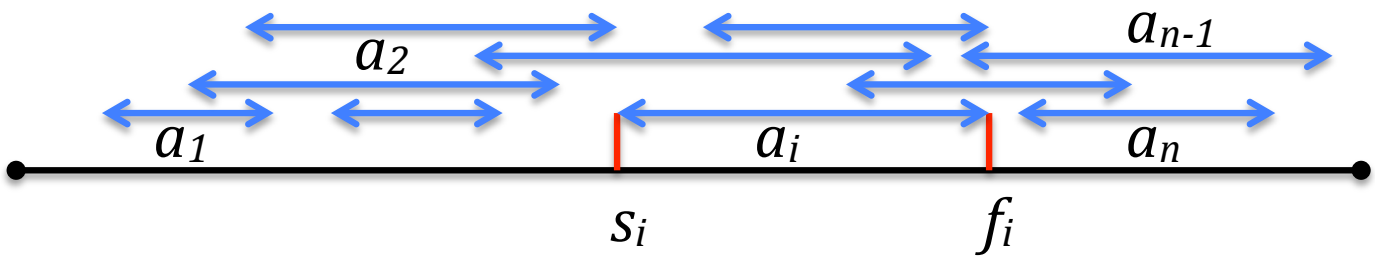
- **Task:** Find a *maximum size* subset of compatible activities.



Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a *maximum size* subset of compatible activities.

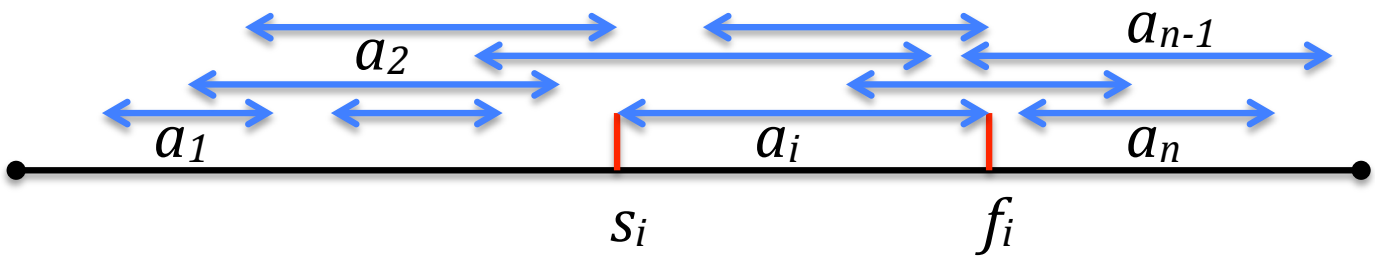


Attempt 1: always choose the shortest activity which does not conflict with the previously chosen activities, remove the conflicting activities and repeat?

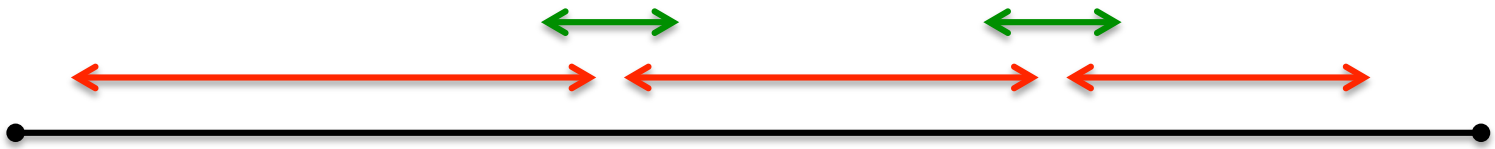
Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a *maximum size* subset of compatible activities.



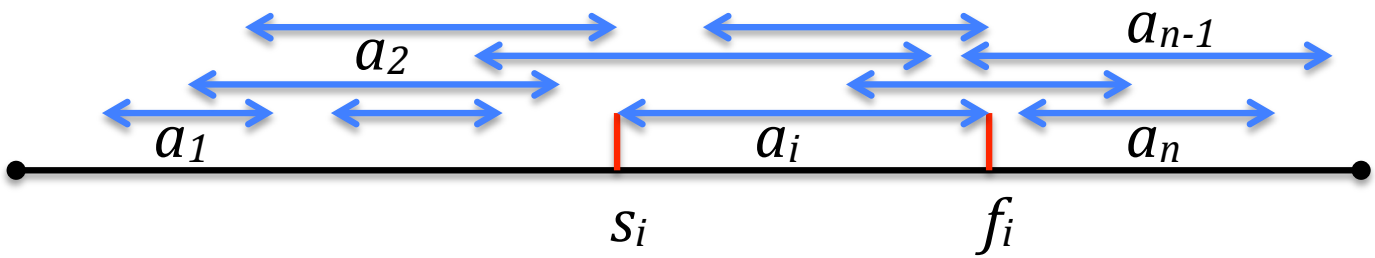
Attempt 1: always choose the shortest activity which does not conflict with the previously chosen activities, remove the conflicting activities and repeat?



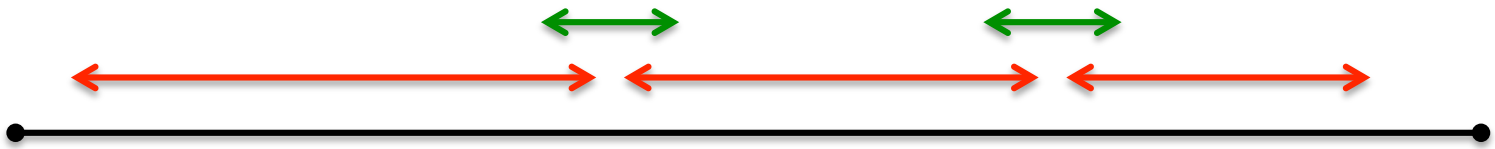
Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

- **Task:** Find a *maximum size* subset of compatible activities.



Attempt 1: always choose the shortest activity which does not conflict with the previously chosen activities, remove the conflicting activities and repeat?



- The above figure shows this does not work...
(chosen activities in green, conflicting in red)

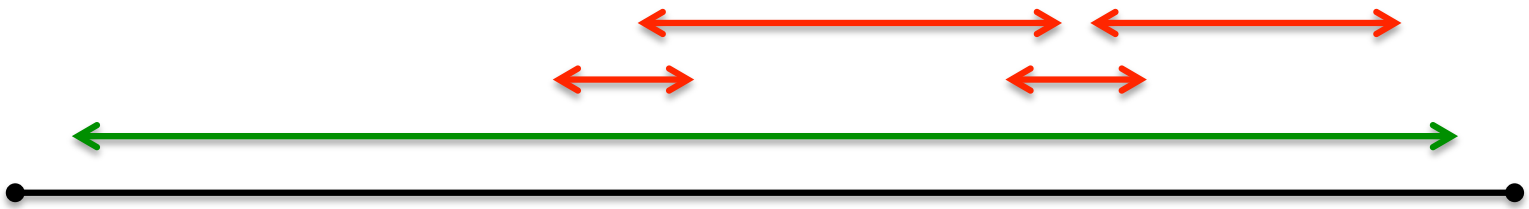
Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

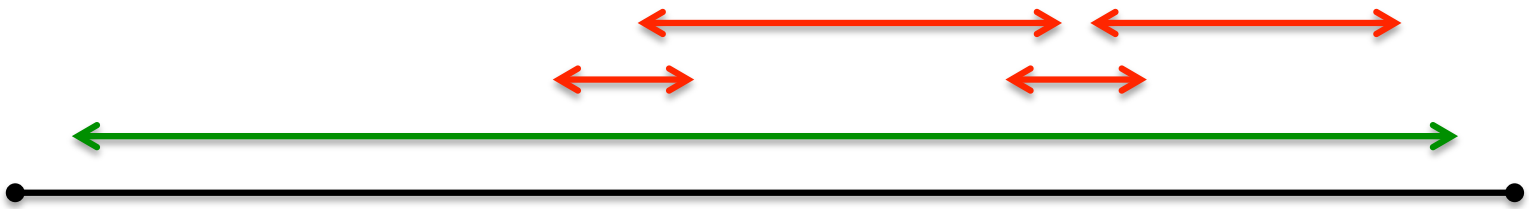
- **Attempt 2:** Maybe among the activities which do not conflict with the previously chosen activities we should always choose the one with the earliest possible start?



Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

- **Attempt 2:** Maybe among the activities which do not conflict with the previously chosen activities we should always choose the one with the earliest possible start?



- The above figure shows this does not work either...

Activity selection problem.

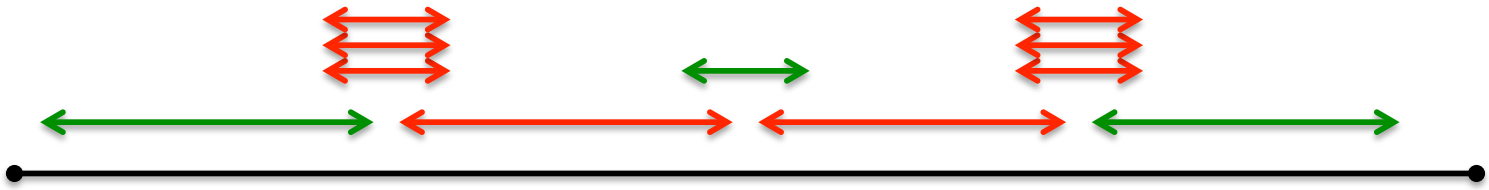
- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

- **Attempt 3:** Maybe we should always choose an activity which conflicts with the fewest possible number of the remaining activities? It may appear that in this way we minimally restrict our next choice....

Activity selection problem.

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

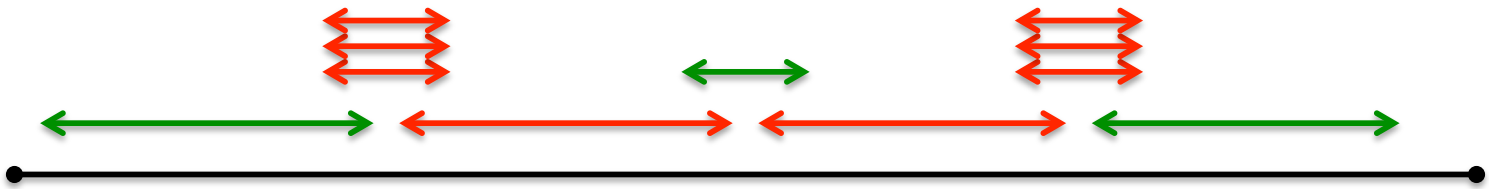
- **Attempt 3:** Maybe we should always choose an activity which conflicts with the fewest possible number of the remaining activities? It may appear that in this way we minimally restrict our next choice....



Activity selection problem.

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

- **Attempt 3:** Maybe we should always choose an activity which conflicts with the fewest possible number of the remaining activities? It may appear that in this way we minimally restrict our next choice....



- As appealing the idea is, the above figure shows this again does not work ...

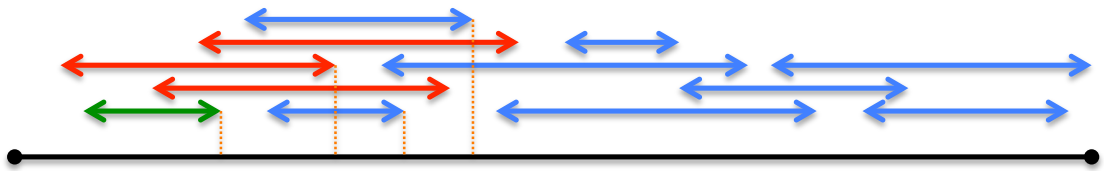
Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

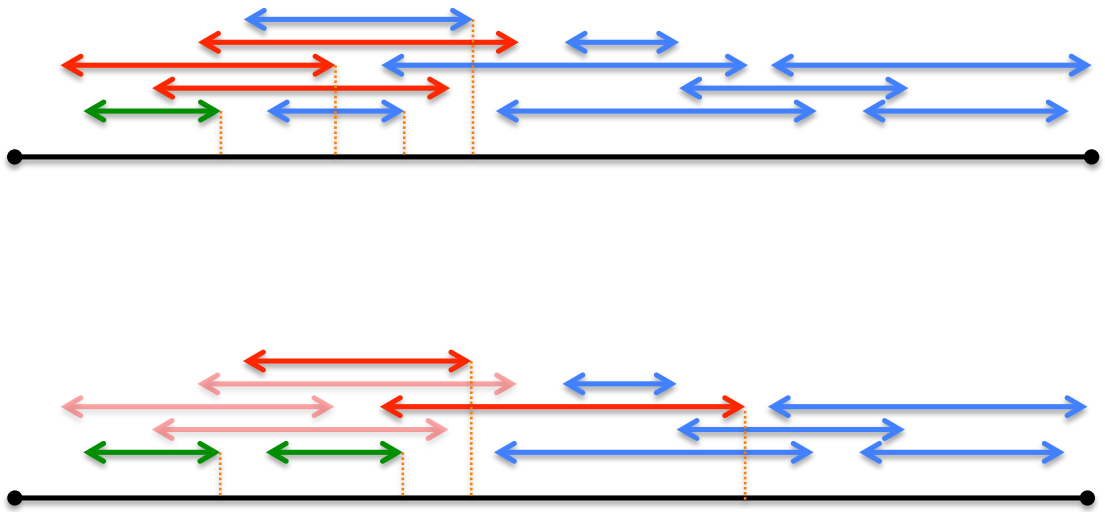
- **The correct solution:** Among the activities which do not conflict with the previously chosen activities always chose the one with the earliest end time.



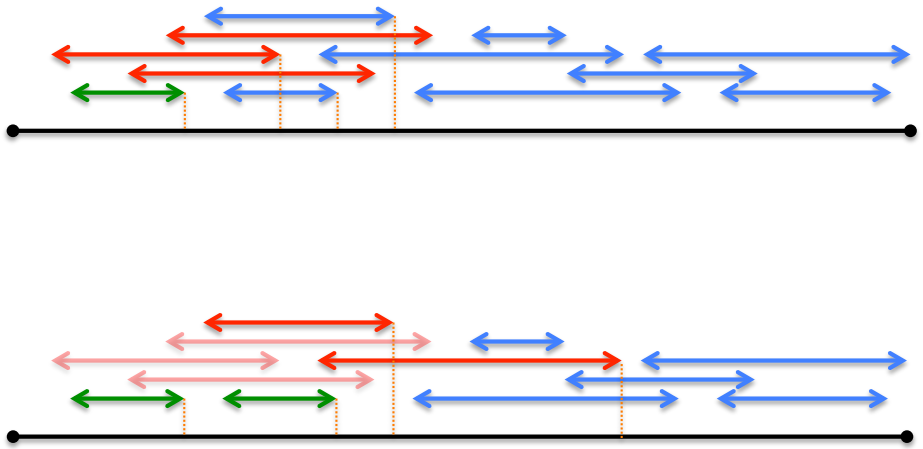
Activity selection problem

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.
- **Task:** Find a *maximum size* subset of compatible activities.

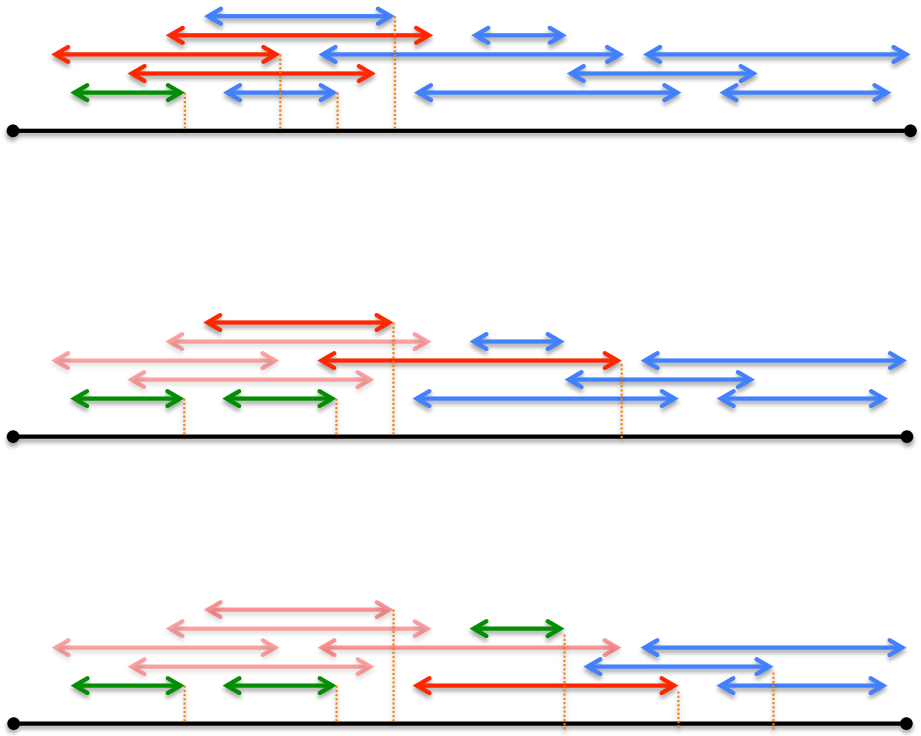
- **The correct solution:** Among the activities which do not conflict with the previously chosen activities always chose the one with the earliest end time.



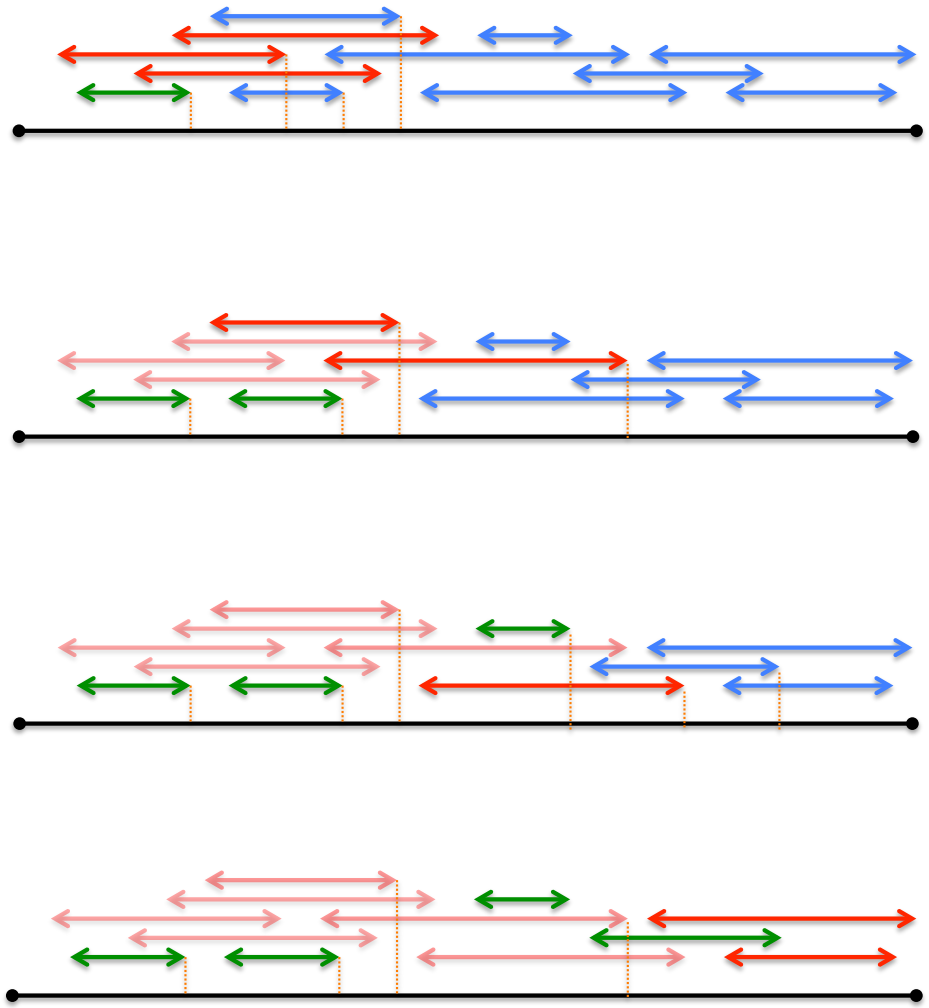
Activity selection problem



Activity selection problem

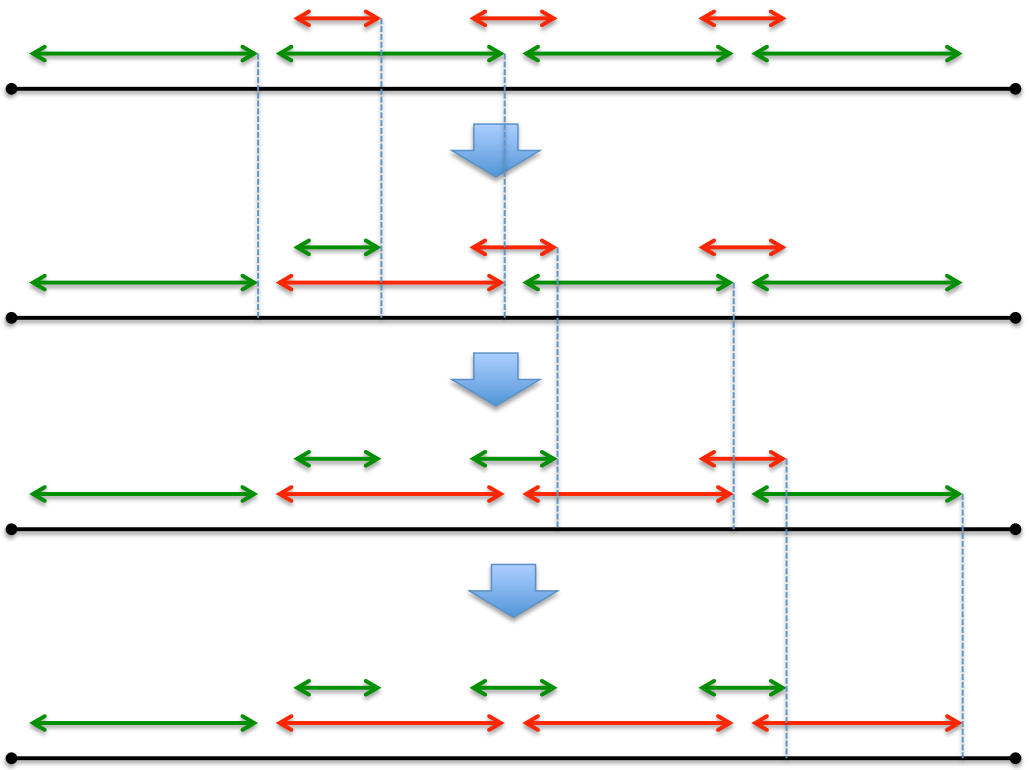


Activity selection problem



The Greedy Method

- Transforming any optimal solution to the greedy solution with equal number of activities: find the first place where the chosen activity violates the greedy choice and show that replacing that activity with the greedy choice produces a non conflicting selection with the same number of activities. Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of *maximal total duration*.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of *maximal total duration*.
- **Solution:** Since all activities are of the same duration, this is equivalent to finding a selection with a largest number of non conflicting activities, i.e., the previous problem.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of *maximal total duration*.
- **Solution:** Since all activities are of the same duration, this is equivalent to finding a selection with a largest number of non conflicting activities, i.e., the previous problem.
- **Question:** What happens if the activities are not all of the same duration?

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.
- **Task:** Find a subset of compatible activities of *maximal total duration*.
- **Solution:** Since all activities are of the same duration, this is equivalent to finding a selection with a largest number of non conflicting activities, i.e., the previous problem.
- **Question:** What happens if the activities are not all of the same duration?
- Greedy strategy no longer works - we will need a more sophisticated technique.

Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time; all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.

Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time; all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.

- **Task:** Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time; all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.

- **Task:** Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

- **Solution:** Ignore job durations and schedule jobs in the increasing order of deadlines.

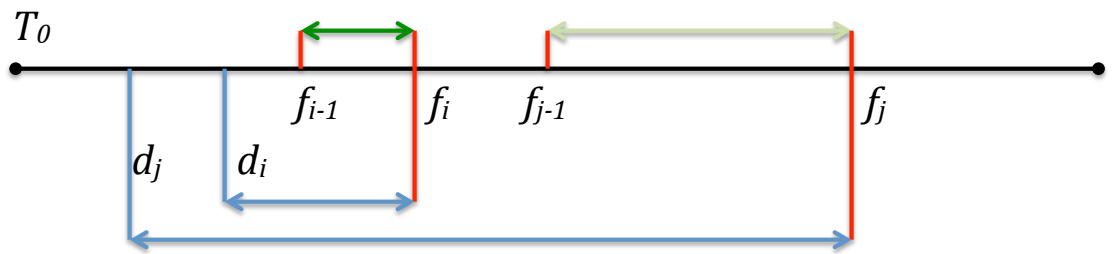
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time; all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.

- **Task:** Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

- **Solution:** Ignore job durations and schedule jobs in the increasing order of deadlines.

- **Optimality:** Consider any optimal solution. We say that jobs a_i and jobs a_j form an inversion if job a_i is scheduled before job a_j but $d_j < d_i$.



Minimising job lateness

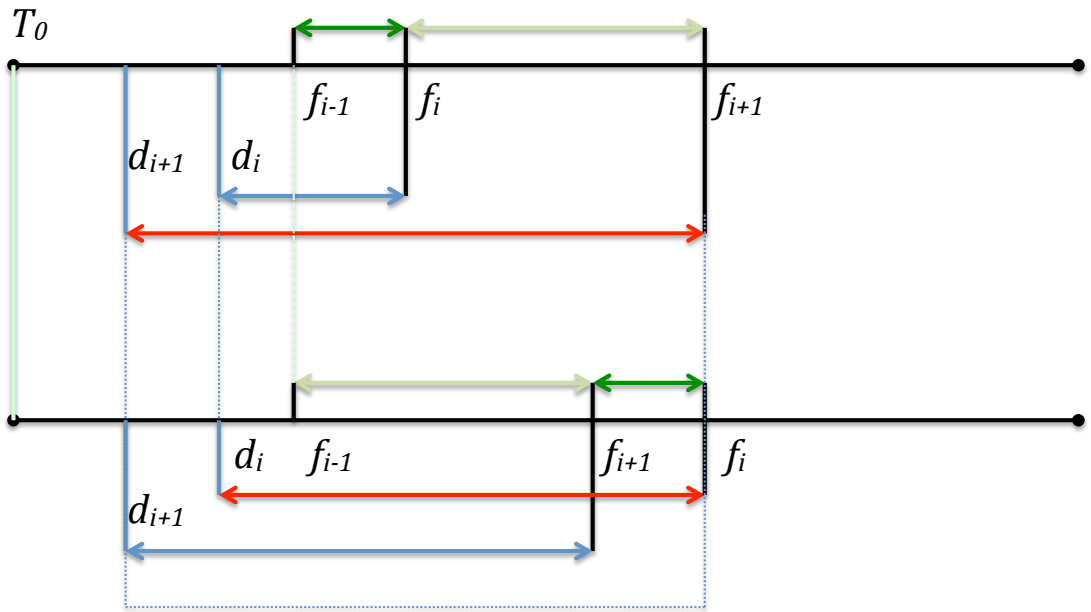
- We will show that there exists a scheduling without inversions which is also optimal.

Minimising job lateness

- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the Bubble Sort: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.

Minimising job lateness

- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the Bubble Sort: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.



- Note that swapping adjacent inverted jobs reduces the larger lateness!

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.

- **Task:** Partition the vertices of G into k disjoint subset so that the minimal distance between two points belonging to different sets of the partition is as large as possible. Thus, we want a partition into k disjoint sets which are as far apart as possible.

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subset so that the minimal distance between two points belonging to different sets of the partition is as large as possible. Thus, we want a partition into k disjoint sets which are as far apart as possible.
- **Solution:** Sort the edges in an increasing order and perform the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k trees, rather than a single spanning tree.

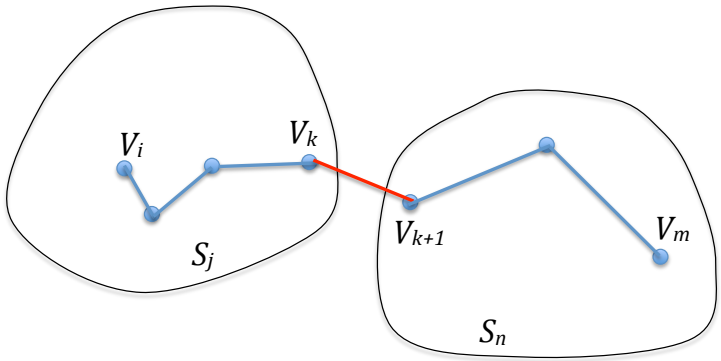
k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subset so that the minimal distance between two points belonging to different sets of the partition is as large as possible. Thus, we want a partition into k disjoint sets which are as far apart as possible.
- **Solution:** Sort the edges in an increasing order and perform the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k trees, rather than a single spanning tree.
- **Proof of optimality:** Let d be the distance associated with the first edge of the minimal spanning tree which was not added to our k trees; it is clearly the minimal distance between two vertices belonging to two of our k trees. Clearly, all the edges included in k many trees produced by our algorithm are of length smaller or equal to d.

k-clustering of maximum spacing

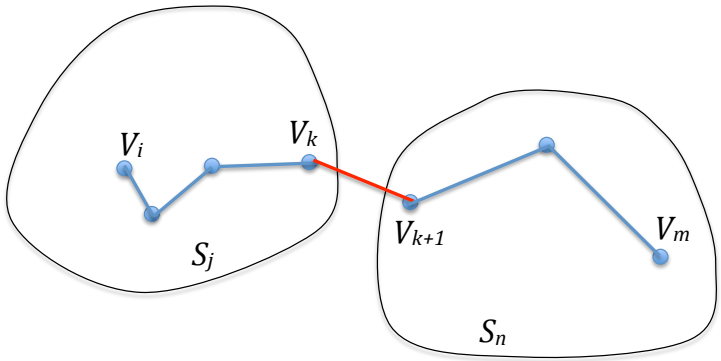
k-clustering of maximum spacing

- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm. This means that there is a tree produced by our algorithm which contains vertices V_i and V_m from two different subsets from $\mathcal{S}$, say S_j and S_n.



k-clustering of maximum spacing

- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm. This means that there is a tree produced by our algorithm which contains vertices V_i and V_m from two different subsets from $\mathcal{S}$, say S_j and S_n.



- Since V_i and V_m belong to the same tree, there is a path along the edges of that tree connecting V_i and V_m. Let V_k and V_{k+1} be two consecutive vertices on that path such that V_k belongs to S_j and V_{k+1} belongs to S_n. Note that $d(V_k, V_{k+1}) \leq d$ which implies that the minimal distance between clusters of $\mathcal{S}$ is smaller or equal to the minimal distance d between the k trees produced by our algorithm. Thus, such a partition cannot be a more optimal clustering than the one produced by our algorithm

Tape storrage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is found and read.

Tape storrage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is found and read.

- **Task:** Order the files on the tape so that the average (expected) retrieval time is minimised.

Tape storrage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is found and read.
- **Task:** Order the files on the tape so that the average (expected) retrieval time is minimised.
- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n) = \\ nl_1 + (n-1)l_2 + (n-2)l_3 + \ldots + 2l_{n-1} + l_n$$

Tape storrage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is found and read.
- **Task:** Order the files on the tape so that the average (expected) retrieval time is minimised.
- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n) = \\ nl_1 + (n-1)l_2 + (n-2)l_3 + \ldots + 2l_{n-1} + l_n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

Tape storrage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is fund and read.

Tape storrage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is fund and read.

- **Task:** Order the files on the tape so that the expected retrieval time is minimised.

Tape storrage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is fund and read.

- **Task:** Order the files on the tape so that the expected retrieval time is minimised.

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

Tape storrage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the tape is fund and read.

- **Task:** Order the files on the tape so that the expected retrieval time is minimised.

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i/l_i.

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$\begin{aligned} E =& p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + \ldots \\ &+ (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n \end{aligned}$$

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive just in case $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k / l_k < p_{k+1} / l_{k+1}$.

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive just in case $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if $p_k/l_k < p_{k+1}/l_{k+1}$, which means that the swap decreases the expected time just in case $p_k/l_k < p_{k+1}/l_{k+1}$, i.e., if there is an inversion: a file f_{i+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_i with a smaller ratio p_k/l_k.

The Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive just in case $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if $p_k/l_k < p_{k+1}/l_{k+1}$, which means that the swap decreases the expected time just in case $p_k/l_k < p_{k+1}/l_{k+1}$, i.e., if there is an inversion: a file f_{i+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_i with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.
- Can we always choose the item with the highest value per unit weight?

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.
- Can we always choose the item with the highest value per unit weight?
- Assume there are just three items with weights and values: (10kg, \$60), (20kg, \$100), (30kg, \$120) and a knapsack of capacity W = 50kg.

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.
- Can we always choose the item with the highest value per unit weight?
- Assume there are just three items with weights and values: (10kg, \$60), (20kg, \$100), (30kg, \$120) and a knapsack of capacity W = 50kg.
- Greedy would choose (10kg, \$60) and (20kg, \$100), while the optimal solution is to take (20kg, \$100) and (30kg, \$120)!

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.
- Can we always choose the item with the highest value per unit weight?
- Assume there are just three items with weights and values: (10kg, \$60), (20kg, \$100), (30kg, \$120) and a knapsack of capacity W = 50kg.
- Greedy would choose (10kg, \$60) and (20kg, \$100), while the optimal solution is to take (20kg, \$100) and (30kg, \$120)!
- So when does the Greedy Strategy work??

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.
- Can we always choose the item with the highest value per unit weight?
- Assume there are just three items with weights and values: (10kg, \$60), (20kg, \$100), (30kg, \$120) and a knapsack of capacity W = 50kg.
- Greedy would choose (10kg, \$60) and (20kg, \$100), while the optimal solution is to take (20kg, \$100) and (30kg, \$120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...