Greedy Algorithms

Greedy Algorithms

- Not an algorithm, But a technique.
- As the name suggests, always makes the choice that seems to be the best at that moment.
- It makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Pros and Cons

- It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques.
- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct.

Container Loading

Container Loading Problem

- A large ship is to be loaded with cargo.
- The cargo is containerized, and all containers are the same size.
- Different containers may have different weights.
- Let w_i be the weight of the i^{th} container, $1 <= i <= n$
- The cargo capacity of the ship is c.
- We wish to load the ship with the maximum number of containers.

Container Loading Problem

- Formulation of the problem :
  - Variables : x_i $(1 \leq i \leq n)$ is set to 0 if the container i is not to be loaded and 1 in the other case.
  - Constraints : $\sum_{i=1}^{n} w_i x_i \leq c$.
  - Optimization function : $\sum_{i=1}^{n} x_i$
- Every set of x_is that satisfies the constraints is a feasible solution.
- Every feasible solution that maximizes $\sum_{i=1}^{n} x_i$ is an optimal solution.

Approach

- Using the greedy algorithm the ship may be loaded in stages;
  - one container per stage.
- At each stage, the greedy criterion used to decide which container to load is the following :
  - From the remaining containers, select the one with least weight.
- This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers.

Example

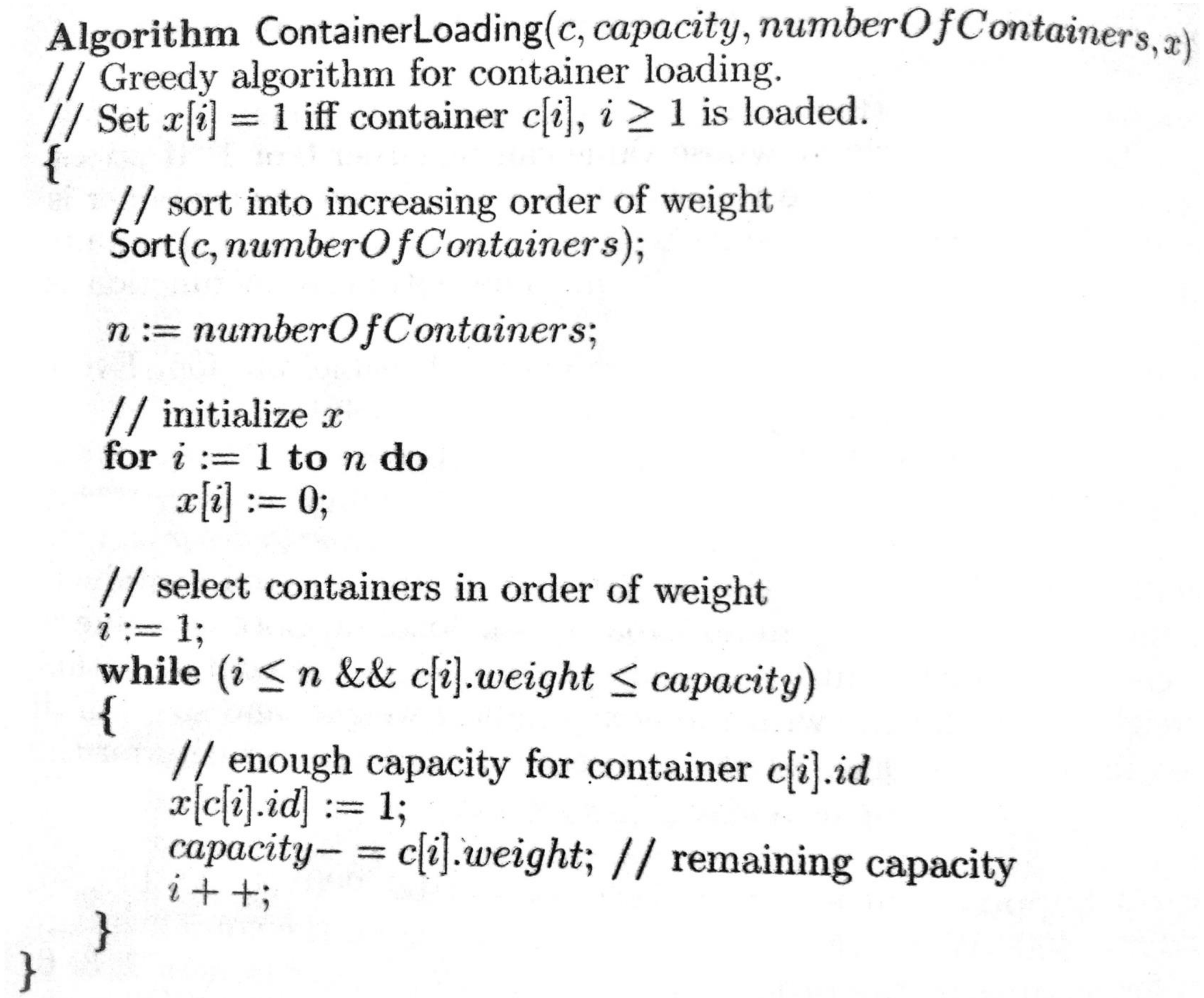
- Suppose that :
  - $n = 8$,
  - $[w_1, \ldots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$,
  - and $c = 400$.

Solution

- When the greedy algorithm is used, the containers are considered for loading in the order 7,3,6,8,4,1,5,2.
- Containers 7,3,6,8,4 and 1 together weight 390 units and are loaded.
- The available capacity is now 10 units, which is inadequate for any of the remaining containers.
- In the greedy solution we have :

$$[x_1, \ldots, x_8] = [1, 0, 1, 1, 0, 1, 1, 1] \text{ and } \sum x_i = 6.$$

Algorithm



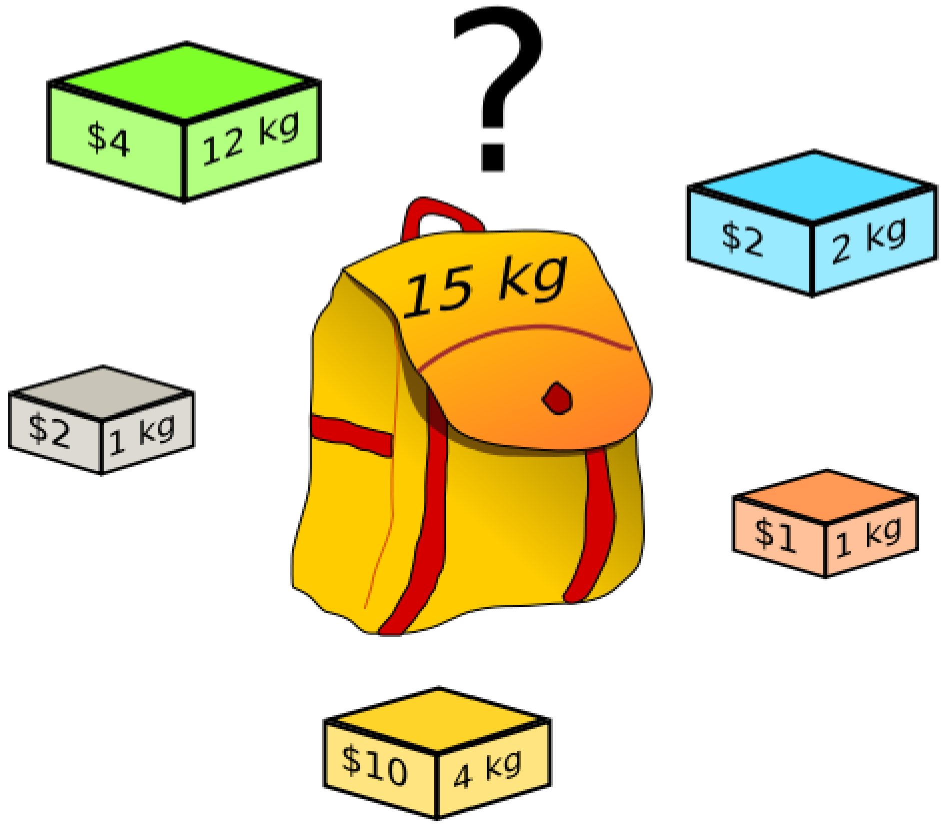
```
Algorithm ContainerLoading($c, capacity, numberOfContainers, x$)
// Greedy algorithm for container loading.
// Set $x[i] = 1$ iff container $c[i]$, $i \geq 1$ is loaded.
{
    // sort into increasing order of weight
    Sort($c, numberOfContainers$);

    $n := numberOfContainers$;

    // initialize $x$
    for $i := 1$ to $n$ do
        $x[i] := 0$;

    // select containers in order of weight
    $i := 1$;
    while ($i \leq n$ && $c[i].weight \leq capacity$)
    {
        // enough capacity for container $c[i].id$
        $x[c[i].id] := 1$;
        $capacity -= c[i].weight$; // remaining capacity
        $i++$;
    }
}
```

Knapsack Problem



0/1 Knapsack Problem

- The 0/1 knapsack problem is a generalization of the container loading problem to the case where the profit earned from each container is different.
- In the 0/1 knapsack problem, we wish to pack a knapsack (bag or sack) with a capacity of C.
- From a list of n items, we must select the items that are to be packed into the knapsack.
- Each object i has a weight w_i and a profit p_i

0/1 Knapsack Problem

- In a **feasible** knapsack packing, the sum of the weights of the packed objects does not exceed the knapsack capacity :

$$\sum_{i=1}^{n} w_i x_i \leq c \text{ and } x_i \in [0,1], 1 \leq i \leq n$$

- An **optimal** packing is a feasible one with maximum profit :

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

Approach 1 : Greedy on profit

- Several greedy strategies for the 0/1 knapsack problem are possible.
- In each of these strategies, the knapsack is packed in several stages. In each stage one object is selected for inclusion into the knapsack using a greedy criterion.
- First possible criterion : from the remaining objects, select the object with the maximum profit that fits into the knapsack.
- This strategy does not guarantee an optimal solution.

Example

- n = 3, w = [100,10,10], p = [20,15,15], c = 105
- Solution using the above criterion : x = [1,0,0], profit = 20
- Optimal solution : x = [0,1,1], profit = 30

Approach 2 : Greedy on weight

- *From the remaining objects, select the one that has minimum weight and also fits into the knapsack.*
- This approach does not yield in general to an optimal solution.

Example : $n = 2, w = [10, 20], p = [5, 100], c = 25.$

Solution : $x = [1, 0]$ inferior to the solution : $x = [0, 1].$

Approach 3 : Greedy on the profit density p_i/w_i

- From the remaining objects, select the one with maximum p_i/w_i that fits into the knapsack.
- This strategy does not guarantee optimal solutions either.
- Example : n = 3, w = [20,15,15], p = [40,25,25] and c = 30

Point to Ponder

The 0/1 knapsack problem is an NP-hard problem. This is the reason why we cannot find a polynomial-time algorithm to solve it.

Fractional Knapsack

- We are given n objects and a knapsack.
- Object *i* has a weight w_i and the knapsack has a capacity *M*. If a fraction x_i, $0 <= x <= 1$, of object *i* is placed into the knapsack then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is M, we require the total weight of all chosen objects to be at most M.

Fractional Knapsack

- Formal Definition

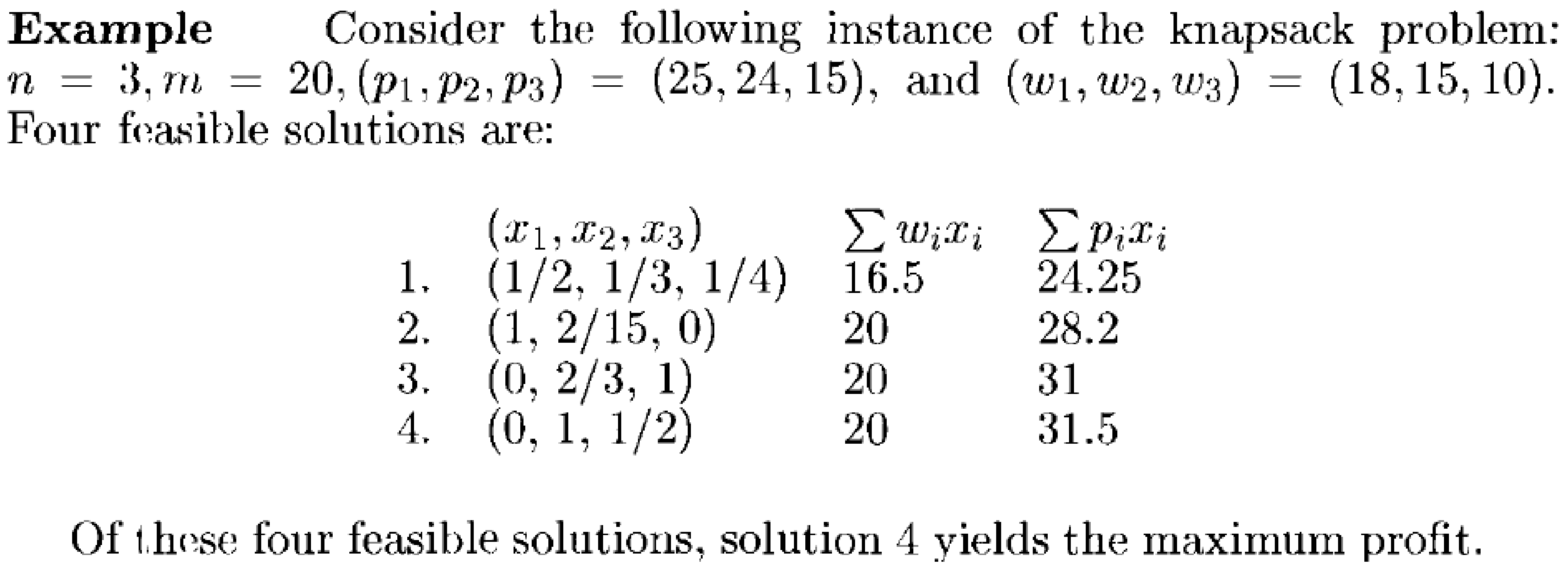
$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le M$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The profits and weights are positive numbers.

Fractional Knapsack : Example

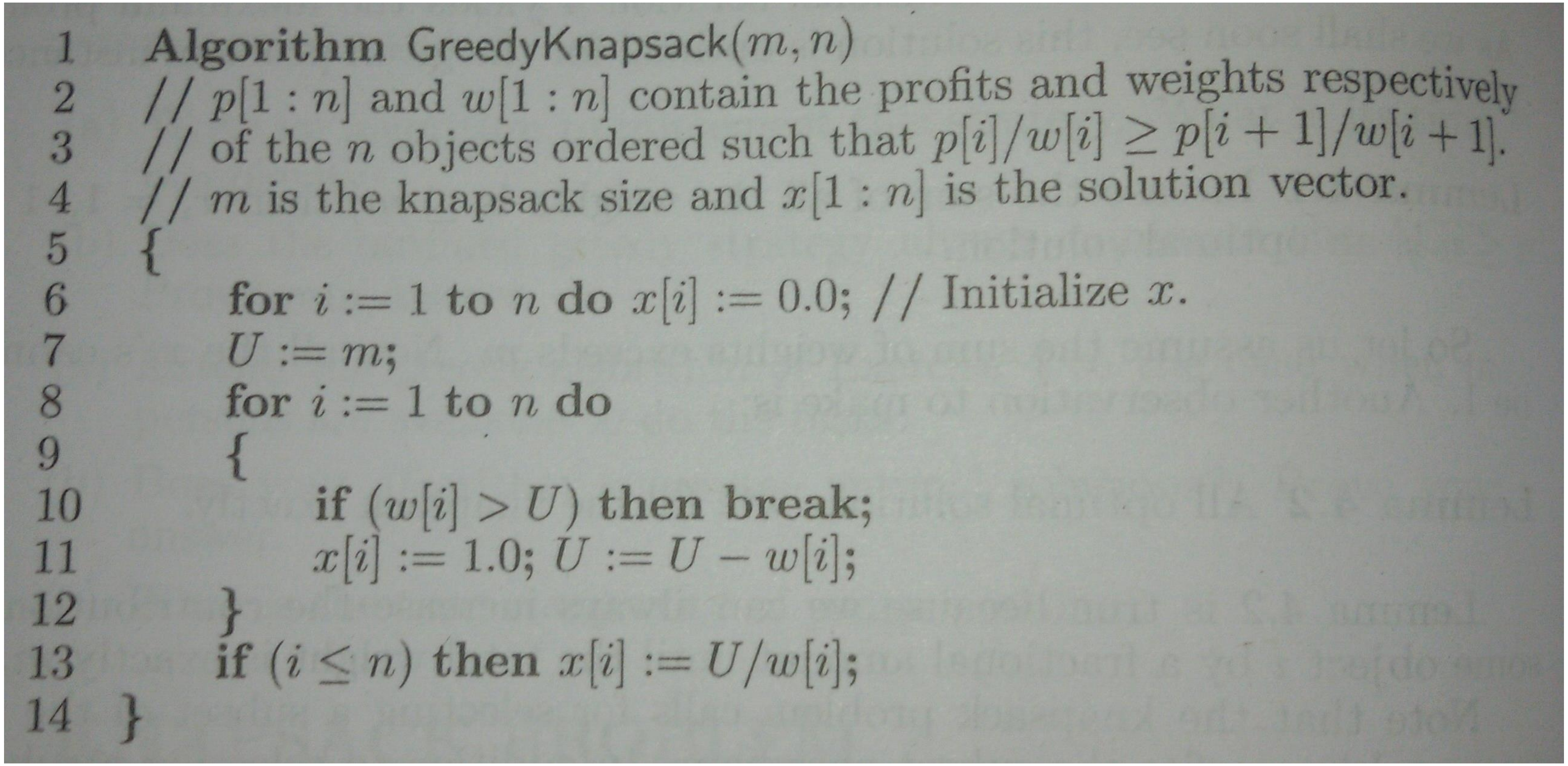


Example Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit.

Fractional Knapsack : Algorithm



```
Algorithm GreedyKnapsack($m, n$)
// $p[1:n]$ and $w[1:n]$ contain the profits and weights respectively
// of the $n$ objects ordered such that $p[i]/w[i] \geq p[i+1]/w[i+1]$.
// $m$ is the knapsack size and $x[1:n]$ is the solution vector.
{
    for $i := 1$ to $n$ do $x[i] := 0.0$; // Initialize $x$.
    $U := m$;
    for $i := 1$ to $n$ do
    {
        if ($w[i] > U$) then break;
        $x[i] := 1.0$; $U := U - w[i]$;
    }
    if ($i \leq n$) then $x[i] := U/w[i]$;
}
```

Job Sequencing with Deadlines

Job Sequencing with Deadlines : Problem

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value.

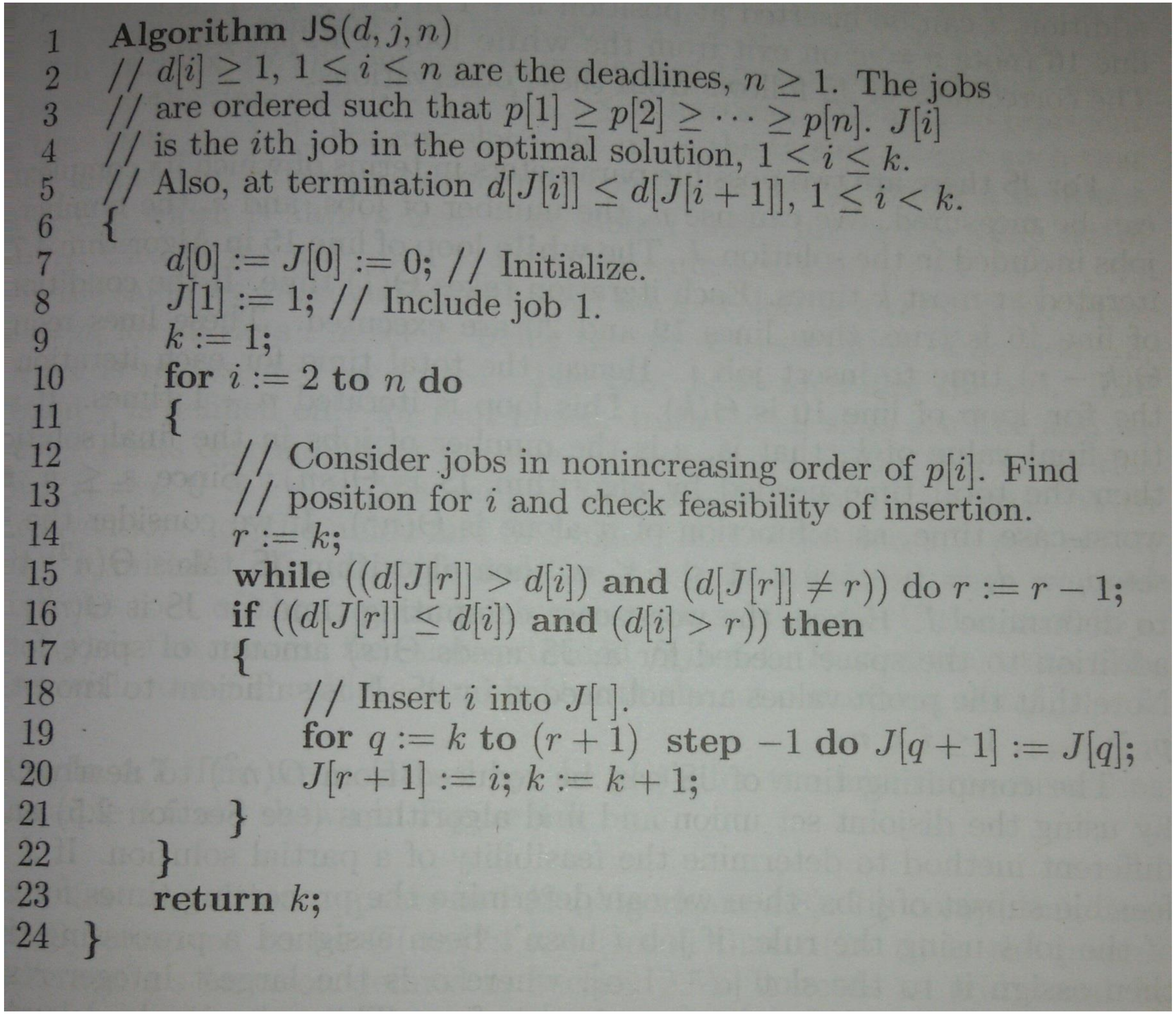
Example

Let $n = 4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	**processing sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Example

- n = 5
- p = [20,15,10,5,1]
- d = [2,2,1,3,3]

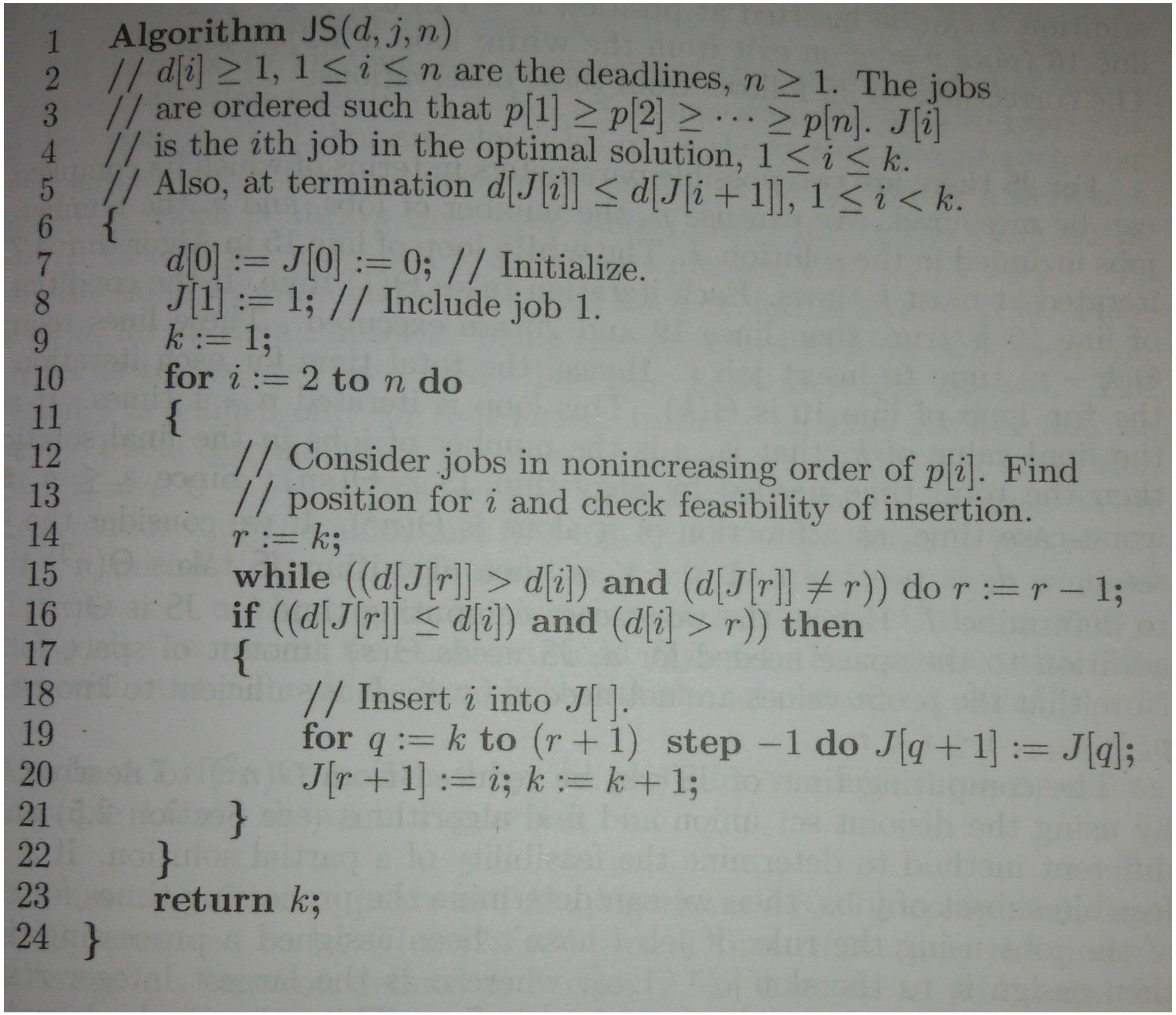


```
Algorithm JS($d, j, n$)
// $d[i] \geq 1$, $1 \leq i \leq n$ are the deadlines, $n \geq 1$. The jobs
// are ordered such that $p[1] \geq p[2] \geq \cdots \geq p[n]$. $J[i]$
// is the $i$th job in the optimal solution, $1 \leq i \leq k$.
// Also, at termination $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$.
{
    $d[0] := J[0] := 0$; // Initialize.
    $J[1] := 1$; // Include job 1.
    $k := 1$;
    for $i := 2$ to $n$ do
    {
        // Consider jobs in nonincreasing order of $p[i]$. Find
        // position for $i$ and check feasibility of insertion.
        $r := k$;
        while $((d[J[r]] > d[i])$ and $(d[J[r]] \neq r))$ do $r := r - 1$;
        if $((d[J[r]] \leq d[i])$ and $(d[i] > r))$ then
        {
            // Insert $i$ into $J[\,]$.
            for $q := k$ to $(r + 1)$ step $-1$ do $J[q + 1] := J[q]$;
            $J[r + 1] := i$; $k := k + 1$;
        }
    }
    return $k$;
}
```

Explanation of Algorithm

- YouTube Reference Link
- https://youtu.be/Z67k4SWcY2c

Algorithm



```
Algorithm JS($d, j, n$)
// $d[i] \geq 1$, $1 \leq i \leq n$ are the deadlines, $n \geq 1$. The jobs
// are ordered such that $p[1] \geq p[2] \geq \cdots \geq p[n]$. $J[i]$
// is the $i$th job in the optimal solution, $1 \leq i \leq k$.
// Also, at termination $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$.
{
    $d[0] := J[0] := 0$; // Initialize.
    $J[1] := 1$; // Include job 1.
    $k := 1$;
    for $i := 2$ to $n$ do
    {
        // Consider jobs in nonincreasing order of $p[i]$. Find
        // position for $i$ and check feasibility of insertion.
        $r := k$;
        while $((d[J[r]] > d[i])$ and $(d[J[r]] \neq r))$ do $r := r - 1$;
        if $((d[J[r]] \leq d[i])$ and $(d[i] > r))$ then
        {
            // Insert $i$ into $J[\,]$.
            for $q := k$ to $(r+1)$ step $-1$ do $J[q+1] := J[q]$;
            $J[r+1] := i$; $k := k + 1$;
        }
    }
    return $k$;
}
```

Minimum Cost Spanning Trees

Prim’s and Kruskal’s Algorithm

(Refer another ppt)

Prim’s Algorithm

```
Algorithm Prim($E, cost, n, t$)
// $E$ is the set of edges in $G$. $cost[1:n, 1:n]$ is the cost
// adjacency matrix of an $n$ vertex graph such that $cost[i,j]$ is
// either a positive real number or $\infty$ if no edge $(i,j)$ exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array $t[1:n-1, 1:2]$. $(t[i,1], t[i,2])$ is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let $(k,l)$ be an edge of minimum cost in $E$;
    $mincost := cost[k,l]$;
    $t[1,1] := k$; $t[1,2] := l$;
    for $i := 1$ to $n$ do // Initialize near.
        if $(cost[i,l] < cost[i,k])$ then $near[i] := l$;
        else $near[i] := k$;
```

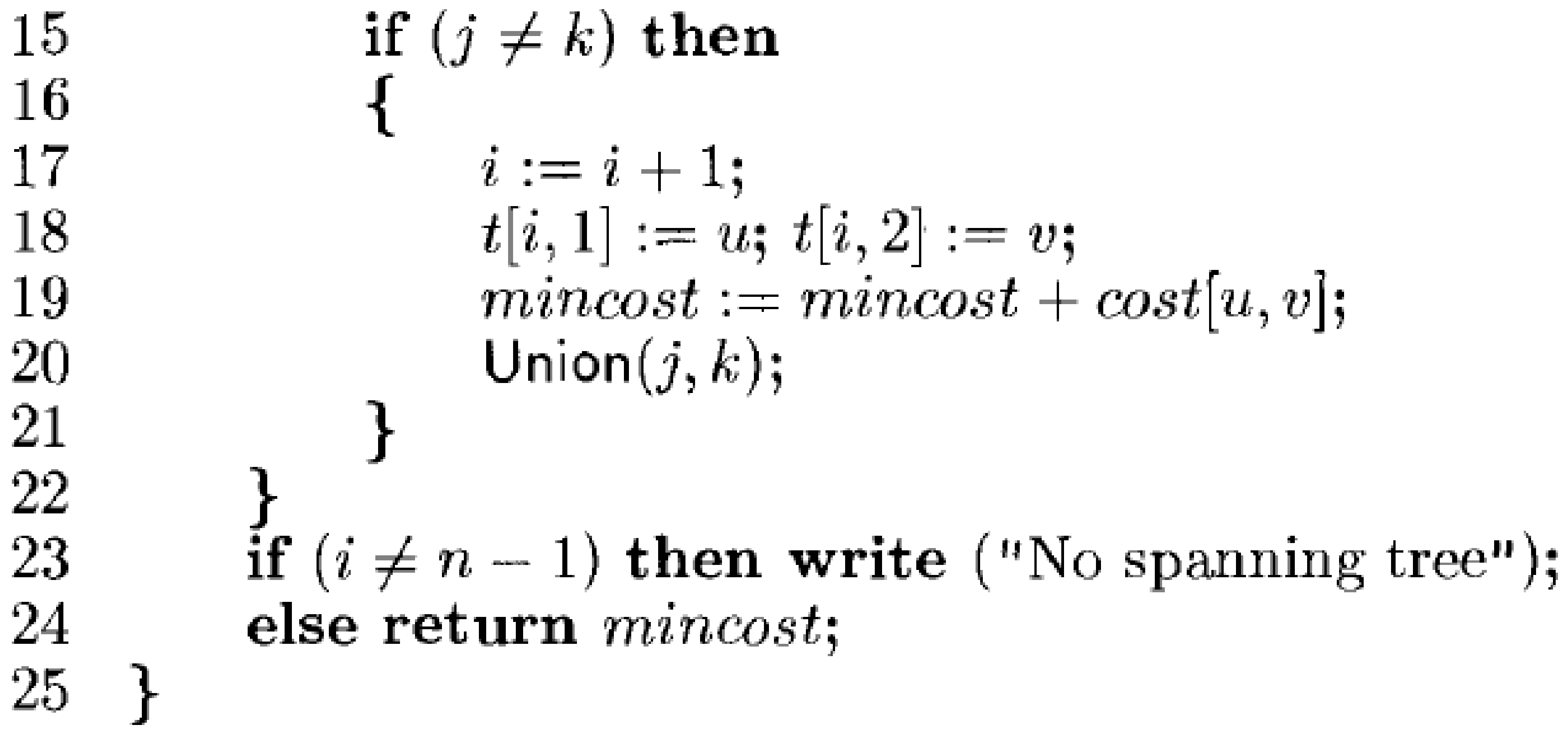
Prim’s Algorithm

```
        $near[k] := near[l] := 0;$
        **for** $i := 2$ **to** $n - 1$ **do**
        { // Find $n - 2$ additional edges for $t$.
            Let $j$ be an index such that $near[j] \neq 0$ and
            $cost[j, near[j]]$ is minimum;
            $t[i, 1] := j;\ t[i, 2] := near[j];$
            $mincost := mincost + cost[j, near[j]];$
            $near[j] := 0;$
            **for** $k := 1$ **to** $n$ **do** // Update $near[\ ]$.
                **if** $((near[k] \neq 0)$ **and** $(cost[k, near[k]] > cost[k, j]))$
                    **then** $near[k] := j;$
        }
        **return** $mincost;$
}
```

Kruskal’s Algorithm

```
Algorithm Kruskal($E, cost, n, t$)
// $E$ is the set of edges in $G$. $G$ has $n$ vertices. $cost[u, v]$ is the
// cost of edge $(u, v)$. $t$ is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for $i := 1$ to $n$ do $parent[i] := -1$;
    // Each vertex is in a different set.
    $i := 0$; $mincost := 0.0$;
    while (($i < n - 1$) and (heap not empty)) do
    {
        Delete a minimum cost edge $(u, v)$ from the heap
        and reheapify using Adjust;
        $j :=$ Find($u$); $k :=$ Find($v$);
```

Kruskal’s Algorithm



```
            if ($j \neq k$) then
            {
                $i := i + 1$;
                $t[i, 1] := u$; $t[i, 2] := v$;
                $mincost := mincost + cost[u, v]$;
                Union($j, k$);
            }
        }
        if ($i \neq n - 1$) then write ("No spanning tree");
        else return $mincost$;
}
```

End.