Algorithmic approaches: scheduling case study[1]

A Greedy Algorithm for Scheduling Jobs with Deadlines and Profits

The setting is that we have n jobs, each of which takes unit time, and a processor on which we would like to schedule them in as profitable a manner as possible. Each job has a profit associated with it, as well as a deadline; if the job is not scheduled by the deadline, then we don't get the profit.[2] Because each job takes the same amount of time, we will think of a Schedule S as consisting of a sequence of job "slots" $1, 2, 3, \ldots$ where $S(t)$ is the job scheduled in slot t.
(If one wishes, one can think of a job scheduled in slot t as beginning at time $t-1$ and ending at time t, but this is not really necessary.)

More formally, the input is a sequence $(d_1, g_1), (d_2, g_2), \cdots, (d_n, g_n)$ where g_i is a nonnegative real number representing the profit obtainable from job i, and $d_i \in \mathbb{N}$ is the deadline for job i; it doesn't hurt to assume that $1 \leq d_i \leq n$. (The reason why we can assume that every deadline is less than or equal to n is because even if some deadlines were bigger, every feasible schedule could be "contracted" so that no job was placed in a slot bigger than n.)

Definition 1 *A* schedule S *is an array:* $S(1), S(2), ..., S(n)$ *where* $S(t) \in \{0, 1, 2, \cdots n\}$ *for each* $t \in \{1, 2, \cdots, n\}$.

The intuition is that $S(t)$ is the job scheduled by S in slot t; if $S(t) = 0$, this means that no job is scheduled in slot t.

Definition 2 S *is* feasible *if*
(a) If $S(t) = i > 0$, *then* $t \leq d_i$. *(Every scheduled job meets its deadline) (b) If* $t_1 \neq t_2$ *and* $S(t_1) \neq 0$, *then* $S(t_1) \neq S(t_2)$. *(Each job is scheduled at most once.)*

We define the *profit* of a feasible schedule S by
$P(S) = g_{S(1)} + g_{S(2)} + ... + g_{S(n)}$, where $g_0 = 0$ by definition.

Goal: Find a feasible schedule S whose profit $P(S)$ is as large as possible; we call such a schedule *optimal*.

We shall consider the following greedy algorithm. This algorithm begins by sorting the jobs in order of decreasing (actually nonincreasing) profits. Then, starting with the empty schedule, it considers the jobs one at a time; if a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

[1] Based, for the most parts, on University of Toronto CSC 364 notes, original lectures by Stephen Cook

[2] A simplified setting is one in which all jobs have the same profit and we are trying to maximize the number of jobs in the schedule. In that case, our algorithm will consider jobs in order of increasing deadlines.

Greedy:
Sort the jobs so that: $g_1 \geq g_2 \geq \ldots \geq g_n$
for $t : 1..n$
    $S(t) \leftarrow 0$ {Initialize array $S(1), S(2), ..., S(n)$}
end for
for $i : 1..n$
    Schedule job i in the latest possible free slot meeting its deadline;
    if there is no such slot, do not schedule i.
end for

Example. Input of **Greedy**:

Job i:	1	2	3	4	Comments
Deadline d_i:	3	2	3	1	(when job must finish by)
Profit g_i:	9	7	7	2	(already sorted in order of profits)

Initialize $S(t)$:

t	1	2	3	4
$S(t)$	0	0	0	0

Apply **Greedy**: Job 1 is the most profitable, and we consider it first. After 4 iterations:

t	1	2	3	4
$S(t)$	3	2	1	0

Job 3 is scheduled in slot 1 because its deadline $t = 3$, as well as slot $t = 2$, has already been filled.

$P(S) = g_3 + g_2 + g_1 = 7 + 7 + 9 = 23$.

Theorem 1 *The schedule output by the greedy algorithm is optimal, that is, it is feasible and the profit is as large as possible among all feasible solutions.*

We will prove this using our standard method for proving correctness of greedy algorithms.
We say feasible schedule S' *extends* feasible schedule S iff for all t $(1 \leq t \leq n)$,
if $S(t) \neq 0$ then $S'(t) = S(t)$.

Definition 3 *A feasible schedule is* promising after stage i *if it can be extended to an optimal feasible schedule by adding only jobs from* $\{i + 1, \cdots, n\}$.

Lemma 1 *For $0 \leq i \leq n$, let S_i be the value of S after i stages of the greedy algorithm, that is, after examining jobs $1, \cdots, i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq n$:*

$P(i)$: S_i is promising after stage i.

This Lemma implies that the result of **Greedy** is optimal. This is because $P(n)$ tells us that the result of **Greedy** can be extended to an optimal schedule using only jobs from $\emptyset$. Therefore the result of **Greedy** must be an optimal schedule.

Proof of Lemma: To see that $P(0)$ holds, consider any optimal schedule S_{opt}. Clearly S_{opt} extends the empty schedule, using only jobs from $\{1, \cdots, n\}$.

So let $0 \leq i < n$ and assume $P(i)$. We want to show $P(i+1)$. By assumption, S_i can be extended to some optimal schedule S_{opt} using only jobs from $\{i+1, \cdots, n\}$. **Case 1:** Job $i+1$ cannot be scheduled, so $S_{i+1} = S_i$.
Since S_{opt} extends S_i, we know that S_{opt} does not schedule job $i+1$. So S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

Case 2: Job $i+1$ is scheduled by the algorithm, say at time t_0 (so $S_{i+1}(t_0) = i+1$ and t_0 is the latest free slot in S_i that is $\leq d_{i+1}$).

Subcase 2A: Job $i+1$ occurs in S_{opt} at some time t_1 (where t_1 may or may not be equal to t_0).

Then $t_1 \leq t_0$ (because S_{opt} extends S_i and t_0 is as large as possible) and $S_{opt}(t_1) = i+1 = S_{i+1}(t_0)$.

If $t_0 = t_1$ we are finished with this case, since then S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. Otherwise, we have $t_1 < t_0$. Say that $S_{opt}(t_0) = j \neq i+1$. Form S'_{opt} by interchanging the values in slots t_1 and t_0 in S_{opt}. Thus $S'_{opt}(t_1) = S_{opt}(t_0) = j$ and $S'_{opt}(t_0) = S_{opt}(t_1) = i+1$. The new schedule S'_{opt} is feasible (since if $j \neq 0$, we have moved job j to an earlier slot), and S'_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. We also have $P(S_{opt}) = P(S'_{opt})$, and therefore S'_{opt} is also optimal.

Subcase 2B: Job $i+1$ does not occur in S_{opt}.

Define a new schedule S'_{opt} to be the same as S_{opt} except for time t_0, where we define $S'_{opt}(t_0) = i+1$. Then S'_{opt} is feasible and extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

To finish the proof for this case, we must show that S'_{opt} is optimal. If $S_{opt}(t_0) = 0$, then we have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} \geq P(S_{opt})$. Since S_{opt} is optimal, we must have $P(S'_{opt}) = P(S_{opt})$ and S'_{opt} is optimal. So say that $S_{opt}(t_0) = j$, $j > 0$, $j \neq i+1$. Recall that S_{opt} extends S_i using only jobs from $\{i+1, \cdots, n\}$. So $j > i+1$, so $g_j \leq g_{i+1}$. We have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} - g_j \geq P(S_{opt})$. As above, this implies tha t S'_{opt} is optimal. $\square$

We still have to discuss the running time of the algorithm. The initial sorting can be done in time $O(n \log n)$, and the first loop takes time $O(n)$. It is not hard to implement each body of the second loop in time $O(n)$, so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to $O(n \log n)$, but in any case it is a polynomial-time algorithm.

Greedy Summary: In general, greedy algorithms are very fast. Unfortunately, for some kinds of problems they only do not always yield an optimal solution (such as for Simple Knapsack). However for other problems (such as the scheduling problem above, and finding a minimum cost spanning tree) they *always* find an optimal solution. For these problems, greedy algorithms are great.

Scheduling Jobs With Deadlines, Profits, and Durations

In the notes on Greedy Algorithms, we saw an efficient greedy algorithm for the problem of scheduling unit-length jobs which have deadlines and profits. We will now consider a generalization of this problem, where instead of being unit-length, each job now has a *duration* (or processing time).

More specifically, the input will consist of information about n jobs, where for job i we have a nonnegative real valued profit $g_i \in \mathbb{R}^{\geq 0}$, a deadline $d_i \in \mathbb{N}$, and a duration $t_i \in \mathbb{N}$. It is convenient to think of a schedule as being a sequence $C = C(1), C(2), \cdots, C(n)$; if $C(i) = -1$, this means that job i is not scheduled, otherwise $C(i) \in \mathbb{N}$ is the time at which job i is scheduled to begin.

We say that schedule C is *feasible* if the following two properties hold.

(a) Each job finishes by its deadline. That is, for every i, if $C(i) \geq 0$, then $C(i) + t_i \leq d_i$.

(b) No two jobs overlap in the schedule. That is, If $C(i) \geq 0$ and $C(j) \geq 0$ and $i \neq j$, then either $C(i) + t_i \leq C(j)$ or $C(j) + t_j \leq C(i)$. (Note that we permit one job to finish at exactly the same time as another begins.)

We define the profit of a feasible schedule C by $P(C) = \sum_{C(i) \geq 0} g_i$.

We now define the problem of Job Scheduling with Deadlines, Profits and Durations:

Input A list of jobs (d_1, t_1, g_1),..., (d_n, t_n, g_n)

Output A feasible schedule $C = C(1), ..., C(n)$ such that the profit $P(C)$ is the maximum possible among all feasible schedules.

Before beginning the main part of our dynamic programming algorithm, we will sort the jobs according to deadline, so that $d_1 \leq d_2 \leq \cdots \leq d_n = d$, where d is the largest deadline. Looking ahead to how our dynamic programming algorithm will work, it turns out that it is important that we prove the following lemma.

Lemma 2 *Let C be a feasible schedule such that at least one job is scheduled; let $i > 0$ be the largest job number that is scheduled in C. Say that every job that is scheduled in C finishes by time t. Then there is feasible schedule C' that schedules exactly the same jobs as C, and such that $C'(i) = \min\{t, d_i\} - t_i$, and such that all other jobs scheduled by C' end at or before time $\min\{t, d_i\} - t_i$.*

Proof: This proof uses the fact the the jobs are sorted according to deadline. The details are left as an exercise.

We now perform the four steps of a dynamic programming algorithm.

Step 1: Describe an array of values we want to compute.
Define the array $A(i, t)$ for $0 \leq i \leq n$, $0 \leq t \leq d$ by

$A(i, t) = \max\{P(C) | C$ is a feasible schedule in which only jobs from $\{1, \cdots, i\}$ are scheduled, and all scheduled jobs finish by time t $\}$.

Note that the value of the profit of the optimal schedule that we are ultimately interested in, is exactly $A(n, d)$.

Step 2: Give a recurrence.
This recurrence will allow us to compute the values of A one row at a time, where by the ith row of A we mean $A(i, 0), \cdots, A(i, d)$.

- $A(0, t) = 0$ for all t, $0 \leq t \leq d$.
- Let $1 \leq i \leq n$, $0 \leq t \leq d$. Define $t' = \min\{t, d_i\} - t_i$. Clearly t' is the latest possible time that we can schedule job i, so that it ends both by its deadline and by time t. Then we have:

If $t' < 0$, then $A(i, t) = A(i - 1, t)$.
If $t' \geq 0$, then $A(i, t) = \max\{A(i - 1, t), g_i + A(i - 1, t')\}$.

We now must explain (or prove) why this recurrence is true. Clearly $A(0, t) = 0$.

To see why the second part of the recurrence is true, first consider the case where $t' < 0$. Then we cannot (feasibly) schedule job i so as to end by ti me t, so clearly $A(i, t) = A(i-1, t)$. Now assume that $t' \geq 0$. We have a choic e of whether or not to schedule job i. If we don't schedule job i, then the best profit we can get (from scheduling some jobs from $\{1, \cdots i\}$ so that all end by time t) is $A(i - 1, t)$. If we do schedule job i, then the previous lemma tells us that we can assume job i is scheduled at time t' and all the other scheduled jobs end by time t', and so the best profit we can get is $g_i + A(i - 1, t')$.

Although the above argument is pretty convincing, sometimes we want to give a more rigorous proof of our recurrence, or at least the most difficult part of the recurrence.

THIS ARGUMENT CAN BE OMITTED ON THE FIRST READING.
Say that $1 \leq i \leq n$, $0 \leq t \leq d$, and $t' = \min\{t, d_i\} - t_i \geq 0$; we want to prove that $A(i,t) = \max\{A(i-1,t), g_i + A(i-1,t')\}$. To prove such an equality, it is usually convenient to prove two *inequalities*:
$A(i,t) \geq \max\{A(i-1,t), g_i + A(i-1,t')\}$ and
$A(i,t) \leq \max\{A(i-1,t), g_i + A(i-1,t')\}$.

- To prove $A(i,t) \geq \max\{A(i-1,t), g_i + A(i-1,t')\}$, we prove that *both* $A(i,t) \geq A(i-1,t)$ and $A(i,t) \geq g_i + A(i-1,t')$ hold.
  - To show that $A(i,t) \geq A(i-1,t)$, consider a feasible schedule C of profit $A(i-1,t)$ that schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t. Clearly C schedules only jobs from $\{1, \cdots, i\}$ so that all jobs finish by time t, so C's profit is less than or equal to the best such profit, so $A(i-1,t) \leq A(i,t)$.
  - To show that $A(i,t) \geq g_i + A(i-1,t')$, Consider a feasible schedule C of profit $A(i-1,t')$ that schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t'. Let C' be the feasible schedule that extends C by scheduling job i beginning at time t'. Then C' is a feasible schedule that schedules only jobs from $\{1, \cdots, i\}$, all ending by time t, and $P(C') = g_i + A(i-1,t')$. C' has a profit that is less than or equal to the best such profit, so $g_i + A(i-1,t) \leq A(i,t)$.
- To prove that $A(i,t) \leq \max\{A(i-1,t), g_i + A(i-1,t')\}$, we prove that *either* $A(i,t) \leq A(i-1,t)$ or $A(i,t) \leq g_i + A(i-1,t')$ holds. Let C be a feasible schedule that only schedules jobs from $\{1, \cdots, i\}$ such that all jobs end by time t, and such that $P(C) = A(i,t)$. We consider two cases:
  - **Case 1:** C does not schedule job i. Then C schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t, so C's profit is less than or equal to the best such profit, so $A(i,t) \leq A(i-1,t)$.
  - **Case 2:** C schedules job i. Then by the previous lemma, there is a feasible schedule C' such that C' schedules the same jobs as C, C' schedules job i beginning at time t', and C' schedules all its other jobs so that they end by time t'; we have $P(C') = P(C) = A(i,t)$. Let C'' be the feasible schedule that is the same as C', except that C'' doesn't schedule job i; so $P(C'') = A(i,t) - g_i$. Since C'' is a feasible schedule such that only jobs from $\{1, \cdots i-1\}$ are scheduled and such that all jobs end by time t', its profit is less than or equal to the best such profit, so $A(i,t) - g_i \leq A(i-1,t')$, so $A(i,t) \leq g_i + A(i-1,t')$.

Step 3: Give a high-level program.
We now give a high-level program that computes values into an array B, so that we will have $B[i,t] = A(i,t)$. (The reason we call our array B instead of A, is to make it convenient to prove that the values computed into B actually are the values of the array A defined above. This proof is usually a simple induction proof that uses the above recurrence, and so usually this proof is omitted.)

```
for every t ∈ {0, ··· , d}
    B[0, t] ← 0
end for
for i : 1..n
    for every t ∈ {0, ··· , d}
        t' ← min{t, d_i} − t_i
        if t' < 0 then
            B[i, t] ← B[i − 1, t]
        else
            B[i, t] ← max{B[i − 1, t], g_i + B[i − 1, t']}
        end if
    end for
end for
```

Step 4: Compute an optimal solution.
Let us assume we have correctly computed the values of the array A into the array B. It is now convenient to define a "helping" procedure PrintOpt(i,t) that will call itself recursively. Whenever we use a helping procedure, it is important that we specify the appropriate precondition/postcondition for it. In this case, we have:
Precondition: i and t are integers, $0 \leq i \leq n$ and $0 \leq t \leq d$.
Postcondition: A schedule is printed out that is an optimal way of scheduling only jobs from $\{1, \cdots, i\}$ so that all jobs end by time t.

We can now print out an optimal schedule by calling
PrintOpt(n,d)
Note that we have written a recursive program since a simple iterative version would print out the schedule in reverse order. It is easy to prove that PrintOpt(i,t) works, by induction on i. The full program (assuming we have already computed the correct values into B) is as follows:

```
procedure PrintOpt(i, t)
    if i = 0 then return end if
    if B[i, t] = B[i − 1, t] then
        PrintOpt(i − 1, t)
    else
```

```
        t' ← min{t, d_i} − t_i
        PrintOpt(i − 1, t')
        put "Schedule job", i, "at time", t'
    end if
end PrintOpt

PrintOpt(n, d)
```

Analysis of the Running Time
The initial sorting can be done in time $O(n \log n)$. The program in Step 3 clearly takes time $O(nd)$. Therefore we can compute the entire array A in total time $O(nd + n \log n)$. When d is large, this expression is dominated by the term nd. It would be nice if we could state a running time of simply $O(nd)$. Here is one way to do this. When $d \leq n$, instead of using an $n \log n$ sorting algorithm, we can so something faster by noting that we are sorting n numbers from the range 0 to n; this can easily be done (using only $O(n)$ extra storage) in time $O(n)$. Therefore, we can compute the entire array A within total time $O(nd)$. Step 4 runs in time $O(n)$. So our total time to compute an optimal schedule is in $O(nd)$. Keep in mind that we are assuming that each arithmetic operation can be done in constant time.

Should this be considered a polynomial-time algorithm? If we are guaranteed that on all inputs d will be less than, say, n^2, then the algorithm can be considered a polynomial-time algorithm.

More generally, however, the best way to address this question is to view the input as a sequence of bits rather than integers or real numbers. In this model, let us assume that all numbers are integers represented in binary notation. So if the $3n$ integers are represented with about k bits each, then the actual bit-size of the input is about nk bits. It is not hard to see that each arithmetic operation can be done in time polynomial in the bit-size of the input, but how many operations will the algorithm perform? Since d is a k bit number, d can be as large as 2^k. Since 2^k is not polynomial in nk, the algorithm is *not* a polynomial-time algorithm in this setting.

Now consider a slightly different setting. As before, the profits are expressed as binary integers. The durations and deadlines, however, are expressed in *unary* notation. This means that the integer m is expressed as a string of m ones. Hence, d is now less than the bit-size of the input, and so the algorithm is polynomial-time in this setting.

Actually, in order to decide if this algorithm should be used in a specific application, all you really have to know is that it performs about nd arithmetic operations. You can then compare it with other algorithms you know. In this case, perhaps the only other algorithm you know is the "brute force" algorithm that tries all possible subsets of the jobs, seeing which ones can be (feasibly) scheduled. Using the previous lemma, we can test if a set of jobs can be feasibly scheduled in time $O(n)$, so the brute-force algorithm can be implemented to run in time about $n2^n$. Therefore, if d is much less than 2^n then the dynamic programming

algorithm is better; if d is much bigger than 2^n then the brute-force algorithm is better; if d is comparable with 2^n, then probably one has to do some program testing to see which algorithm is better for a particular application.

The (General) Knapsack Problem

First, recall the Simple Knapsack Problem from the notes on Greedy algorithms. We are given a sequence of nonnegative integer weights $w_1, \cdots, w_n$ and a nonnegative integer capacity C, and we wish to find a subset of the weights that adds up to as large a number as possible without exceeding C. In effect, in the Simple Knapsack Problem we treat the weight of each item as its profit. In the (general) Knapsack Problem, we have a separate (nonnegative) profit for each job; we wish to find the most profitable knapsack possible among those whose weight does not exceed C. More formally, we define the problem as follows.

Let $w_1, \cdots, w_n \in \mathbb{N}$ be weights, let $g_1, \cdots, g_n \in \mathbb{R}^{\geq 0}$ be profits, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \cdots, n\}$ let $K(S) = \sum_{i \in S} w_i$ and let $P(S) = \sum_{i \in S} g_i$. (Note that $K(\emptyset) = P(\emptyset) = 0$.) We call $S \subseteq \{1, \cdots, n\}$ *feasible* if $K(S) \leq C$.
The goal is to find a feasible S so that $P(S)$ is as large as possible.

We can view a Simple Knapsack Problem as a special case of a General Knapsack Problem, where for every i, $g_i = w_i$.

Furthermore, we can view a General Knapsack Problem as a special case of a Scheduling With Deadlines, Profits and Durations Problem, where all the deadlines are the same. To see this, say that we are given a General Knapsack Problem with capacity C and with n items, where the ith item has weight w_i and profit g_i. We then create a scheduling problem with n jobs, where the ith job has duration w_i, profit g_i, and deadline C. A solution to this scheduling problem yields a solution to the knapsack problem, and so we can solve the General Knapsack Problem with a dynamic programming algorithm that runs in time $O(nC)$.

This is an example of a central concept in algorithms and complexity theory: a reduction of one problem to the other. We have just shown that if we would have a fast algorithm for the Scheduling with Deadlines, Profits and Durations, then we could solve the Knapsack problem efficiently by "disguising" it as scheduling and running scheduling algorithm. It is a big open question whether there exists such an efficient algorithm for scheduling, though: although most people believe that such an algorithm does not exist, nobody has been able to prove it (here, "efficient" means running in time "polynomial in the length of the input"). This problem is called "P vs. NP" problem, and is stated, loosely, as follows: "is it true that there are problems that have easily verifiable solutions, for which solutions cannot be found more efficiently than doing a (form of) brute-force search?" Clay Mathematical Institute lists this problem as one of the 7 main problems in mathematics for the new millenium, and offers 1 million dollars for resolving it.

Scheduling and knapsack are both problems with "easily verifiable solutions" in a sense that if somebody presents us a schedule or a content of a knapsack we can check easily if the schedule contains conflicts / items fit in a knapsack; we can even check whether the profit is greater than a certain given value. This is the property that the brute-force search algorithm, as well as the backtracking algorithm that we will see in the next section, employ: note that we do not say that it is easy to check that a certain solution is in fact the best one by only looking at that solution! In complexity theory, the class of problems with easily verifiable solutions is (modulo technical details[3]) called "NP", and (a specially phrased version of) the Scheduling problem is "NP-complete" meaning that any other problem in NP such as Knapsack, can be "easily disguised" as a Scheduling problem.

1 Backtracking

If the largest deadline d has a value much larger than n (e.g., 2^n or larger), then the dynamic programming algorithm is not polynomial-time. Brute-force search (check all subsets of jobs) takes time $n2^n$. A version of the brute-force search that performs better in practice is a backtracking algorithm: iteratively pick every job from 1 to n and recurse both for the case when it can be scheduled and when it cannot. The advantage of this approach is that it allows us to stop the moment we discovered a conflict in a subset of jobs, without considering all extensions of that subset. The main part of the backtracking algorithm is its recursive method.

RecBtSched$(S, (d_i, t_i, g_i) \dots (d_n, t_n, g_n))$
$S_0 \leftarrow S; S_1 \leftarrow S \cup \{i\}$
$p_0 \leftarrow RecBtSched(S_0, (d_{i+1}, t_{i+1}, g_{i+1}) \dots (d_n, t_n, g_n))$
if (d_i, t_i, g_i) conflicts with S **then**
  $p_1 \leftarrow -1$
  else $p_1 \leftarrow RecBtSched(S_1, (d_{i+1}, t_{i+1}, g_{i+1}) \dots (d_n, t_n, g_n))$
end if
if $p_0 \geq p_1$ **then**
  $S \leftarrow S_0$
  return p_0
else
  $S \leftarrow S_1$
  return p_1
end if

[3]The major technical detail here is that NP is a class of "decision" problems for which the algorithm returns an answer "yes/no" as opposed to a value or a schedule: "is there a schedule with profit $> B$?" for a B given as part of the input.

Here, we skipped the check that a set of jobs S can be scheduled feasibly: this can be done in linear time (assuming jobs are sorted by the deadlines) using the lemma about scheduling jobs at the latest possible moment.

Now, call RecBtSched() as follows:

Algorithm BacktrackSchedule($(d_1, t_1, g_1) \dots (d_n, t_n, g_n)$)
$S \leftarrow \emptyset$
RecBtSched($S, (d_1, t_1, g_1) \dots (d_n, t_n, g_n)$)
return S

The idea of this algorithm is that it "backtracks" when it sees that a certain choice cannot be made. Suppose, for example, that there are n jobs, and job 3 conflicts with job 1, but none of the others conflict. The backtracking algorithm will first consider schedule with job 1 and without (both are possible), then on the next level of the recursion will try to add job 2 to the schedule (having 4 possibilities now). But on the level 3 of the recursion, it will rule out schedules starting with $\{1, 2, 3\}$ and $\{1, 3\}$, limiting the number of recursion calls to 6 rather than 8. Therefore, these $(8 - 6) * 2^{n-1}$ possible sets which contain both 1 and 3 will never be considered.

In the worst case, this algorithm has the same time complexity as brute-force search (think of n jobs for which any $n - 1$ jobs form a schedule, but all n cannot, e.g., as in the knapsack case). However it performs noticeably better in practice.