An algorithm is *greedy* if it computes its solution piece by piece: at each step, it adds one more piece to its current solution, stopping and returning the solution once it is complete. To prove that such an algorithm is correct, we will always prove that *the algorithm never makes a mistake.* That is, it never adds a piece that makes it impossible to reach a correct solution. In other words, we always prove an invariant of the following form:

greedy invariant: *The current partial solution can be extended (somehow) to a full, correct solution.*

To design the algorithm, the challenge is to make sure each greedy step maintains the invariant. This approach appears different than the one in the text (CLRS), based the so-called *greedy-choice and optimal-substructure* properties. But the two approaches are equivalent — if one works, so does the other. (More on this later.) To illustrate the approach, we design an algorithm for a problem called ACTIVITY SELECTION. We follow the four steps for algorithm design:

1. Define the problem.
2. Define the algorithm.
3. Bound the worst-case running time.
4. Prove that the algorithm is correct.

Except (since this the first half of the course) we don't consider Step 3 (running time).

ACTIVITY SELECTION ....

input: A collection of real intervals $I = ([s_1, f_1], [s_2, f_2], , [s_n, f_n])$, with $s_j \le f_j$ for all j. We use J_j to denote $[s_j, t_j]$, and call J_j a *job*. We think of $[s_j, f_j]$ as the time interval the job requires.

output: A maximum-size pairwise-disjoint subset S of I. (*Pairwise disjoint* means that every two intervals in S are disjoint. *Disjoint* means their intersection is empty. We want to do as many jobs as we can, but for any two jobs whose intervals overlap, we can't do both.)

(For intuition, draw some intervals and find a maximum pairwise-disjoint subset of them.)

We use the following standard terms. Given an instance I, define a set S to be a *feasible solution (for I)* if it meets the constraint of being a pairwise-disjoint subset of I. A feasible solution is a correct solution if it has maximum size. Let $\mathsf{opt}(I)$ be the maximum size of any feasible solution:

$$\mathsf{opt}(I) \;=\; \max\{|S| : S \subseteq I \textit{ and } I \textit{ is pairwise disjoint}\} \;=\; \max\{|S| : S \textit{ is feasible for } I\}.$$

Given any feasible solution S, say that S is *optimal (for I)* if $|S| = \mathsf{opt}(I)$. So, given I, the problem is to find a feasible solution of maximum size, that is, an optimal solution.

A greedy algorithm for this problem should maintain a subset $X \subseteq I$ of the given jobs. The subset X should initially be empty. The algorithm should add activities one by one, maintaining the invariant, until no more activities can be added to X. It should then return the final set X. That is, the algorithm should have the following form:

input: ACTIVITY SELECTION instance $I = \big\{J_j = [s_j, f_j] : i \in \{1, 2, \ldots, n\}\big\}$
output: solution X
1. $X \leftarrow \emptyset$ *(X is the partial solution so far.)*
2. while some job in I can be added to X:
3.1. Somehow choose some job J_j in I. *(But how?)*
3.2. $X \leftarrow X \cup \{J_j\}$. *(Add the next piece to the partial solution.)*
4. return X

The algorithm should maintain the generic greedy invariant, which in this case is

greedy invariant: *The current set X is a subset of some optimal solution.*

To design the algorithm, we need to figure out how to choose the job J_j in each iteration to maintain this invariant. To get started, consider just the *first step*. Given I, how can we find one job that *has to* be in some optimal solution for I? We'll see later that once we answer this question, our answer will determine the entire algorithm.

We need a rule for choosing the first job J_j. Here are some candidates to explore:

1. Choose one with earliest start time (minimizing s_j).
2. Choose a smallest job (minimizing $f_j - s_j$).
3. Choose one with the fewest *conflicts* (i.e., other jobs that overlap).
4. Choose one with earliest finish time (minimizing f_j).

To build intuition, for each of the first three rules, find a counter-example, an instance for which the rule chooses a job that is not in any optimal solution. For example, for the third rule, consider

$$I = \{[0, 2], [3, 6], [7, 10], [11, 13], [1, 4], [1, 4], [1, 4], [5, 8], [9, 12], [9, 12], [9, 12]\}.$$

The rule chooses $[5, 8]$ first, but the only optimal solution is $S = \{[0, 2], [3, 6], [7, 10], [11, 13]\}$.

But the earliest-finishing rule works. We give both short- and long-form proofs for intuition.

From this point on, we assume for ease of notation that the jobs in I are ordered by finish time. That is, $f_1 \le f_2 \le \cdots \le f_n$. So job J_1 is an earliest-ending job.

Since any feasible solution $S \subseteq J$ is pairwise disjoint, we'll order the jobs in S by increasing finish time, and write the ordered sequence as $S = (S_1, S_2, \ldots, S_m)$. Since S is pairwise disjoint, each job ends before the next one starts, and the jobs are also ordered by increasing start time.

Lemma 1. *For any non-empty instance I, job J_1 is in some optimal solution.*

Short-form proof of Lemma 1. Fix any optimal solution $S = (S_1, S_2, \ldots, S_m)$. In the case that $J_1 \in S$, we are done. In the remaining case, consider the modified job sequence $S' = (J_1, S_2, S_3, \ldots, S_m)$ obtained by replacing the first job S_1 in S by J_1. Since J_1 ends as early as S_1, and S_1 ends before all other jobs in S, job J_1 also ends before all other jobs in S. So S' is a feasible solution. And S' has the same size as S, so S' is also an optimal solution, but one that contains J_1. □

Long-form proof of Lemma 1.

1. Consider an arbitrary non-empty instance I, and any job $J_1 \in I$ with earliest finish time.
2. Let $S = (S_1, S_2, \ldots, S_K)$ be the jobs in an optimal solution for I, ordered as described above.

3.1. <u>Case 1.</u> Consider first the "easy" case, when $J_1 \in S$.
3.2. In this case J_1 is in some optimal solution (namely, S).

4.1. <u>Case 2.</u> In the remaining case, J_1 is not in S.
4.2. Let $S' = (J_1, S_2, S_3, \ldots, S_m)$ be obtained from S by replacing S_1 by J_1.
4.3. Job S_1 ends before all other jobs in S start, and J_1 ends no later than S_1.
4.4. So J_1 ends before all other jobs in S' start,
4.5. So S' is also a feasible solution. And S' has the same size as S, so S' is optimal.
4.6. We conclude that Case 2, J_1 is also in some optimal solution (S').

5. By Blocks 3 and 4, J_1 is in some optimal solution. □

Extending from the first step. In its first iteration, the algorithm can commit to choosing J_1. By Lemma 1, that choice is guaranteed to maintain the invariant for the first iteration. That is, we know there is an optimal solution of the form $\{J_1\} \cup R$, for some R. It's enough to find an optimal solution of this form. So the remaining problem is to compute an R such that $\{J_1\} \cup R$ is optimal.

Define $D = \{J_j \in I : s_j > f_1\}$ to contain the jobs in I that are disjoint from J_1. For $\{J_1\} \cup R$ to be feasible, each job in the desired set R must be disjoint from J_1. So R has to be a subset of D, and of course R has to be pairwise disjoint. Subject to these two constraints, R should be as large as possible. In other words, R should be a maximum-size pairwise-disjoint subset of D. That is, R should be an optimal solution to the ACTIVITY SELECTION instance specified by D.

The next lemma captures this intuition. Recall that I is any non-empty instance, J_1 is an earliest-finishing job in I, and $D = \{J_j \in I : s_j > f_1\}$ contains all jobs that are disjoint from J_1.

Lemma 2 (optimal substructure)**.** *Let R be any optimal solution to* ACTIVITY SELECTION *instance D (containing the jobs that are disjoint from J_1). Then $\{J_1\} \cup R$ is an optimal solution to I.*

Proof (long form).

1. Suppose for contradiction that $\{J_1\} \cup R$ is not optimal for I.
2. Since $R \subseteq D$, each job in R is pairwise disjoint from J_1, so $\{J_1\} \cup R$ is feasible for I.
3. Since $\{J_1\} \cup R$ is not optimal for I, it must be that there exists a larger feasible solution for I.
4. Let S be a larger optimal solution for I, with J_1 in S. (S exists by Lemma 1.)
5. Let $S = (J_1, S'_1, S'_2, \ldots, S'_m)$ be the sequence of jobs in S, ordered by end time.
6. Let $S' = (S'_1, S'_2, \ldots, S'_m)$ be obtained from S by deleting D.
7. Because S is pairwise disjoint, each S'_i is disjoint from J_1, so each S'_i is in D.
8. So S' is a feasible solution to instance D.
9. But $S = \{J_1\} \cup S'$ is larger than $\{J_1\} \cup R$ (Step 4), so S' is larger than R.
10. This contradicts the optimality of R for D. □

By the way, the converse of the lemma also holds. That is, if $\{J_1\} \cup R$ is any optimal solution for I (and $J_1 \notin R$) then R is an optimal solution for D. The underlying fact is that *the feasible solutions for I that contain J_1 correspond one-to-one with the feasible solutions R for D, via the bijection $\{J_1\} \cup R \leftrightarrow R$.* We don't need this fact for our proofs here, but it's the "right way" to understand what's going on, and proving it is a good exercise.

Lemma 2 gives the following recursive algorithm. Recall that jobs in I are ordered by finish time.

```
rselect(I):                       — recursive ACTIVITY SELECTION algorithm —
1. if I = ∅: return ∅.                      (note: ∅ denotes the empty set)
2. Let J_1 be an earliest-finishing job in I.
3. Let D contain the jobs in I that are disjoint from J_1 (appropriately ordered and reindexed).
4. Recursively compute R = rselect(D), then return {J_1} ∪ R.
```

The correctness of this algorithm follows from Lemma 2 by a simple induction.

Theorem 1. *The recursive algorithm is correct.*

Proof (long form).

1. Say that the algorithm (rselect) is *correct for I* if rselect(I) returns an optimal solution for I.
2. We prove that the algorithm is correct for all instances I. The proof is by induction on $|I|$.
3. For the base case, $I = \emptyset$, the only feasible solution is $\emptyset$, so the algorithm is correct for I.
4.1. Consider any non-empty instance I.
4.2.1. Assume that the algorithm is correct for all instances smaller than I.
4.2.2. Consider the execution of the algorithm on I.
4.2.3. Let J_1, D, and $R =$ rselect(D) be as computed by Lines 2 and 3 of the algorithm.
4.2.4. By induction, R is optimal for the smaller instance D.
4.2.5. So, by Lemma 2, the solution $\{J_1\} \cup R$ returned by rselect(I) is optimal for I.
4.3. By Block 4.2, if rselect is correct for all smaller instances, then rselect is correct for I.
5. By Block 4, for any I, if rselect is correct for all smaller instances, then rselect is correct for I.
6. By Step 3, rselect is correct for the smallest instance $I = \emptyset$.
7. Inductively, rselect is correct for all instances. □

The proofs above follow the so-called *greedy-choice / optimal-substructure* approach in the text (CLRS). That approach is elegant, but isn't as general as using the greedy invariant. To see how that works, let's do another proof, this time using the invariant. Unwinding the tail-recursion in the recursive algorithm gives the following equivalent iterative algorithm:

```
iselect(I = (J_1, J_2, ..., J_n)):          — iterative ACTIVITY SELECTION algorithm —
1. X ← ()                  (X is the partial solution so far, initially the empty sequence)
2. for t = 1, 2, ..., n:
3.1. If J_t is disjoint from each job in X, then append J_t to X.   (Add next piece to partial solution.)
4. return X
```

To prove that this iterative algorithm is correct, we will show that it maintains the greedy invariant: *The current set X is a subset of some optimal solution.* The invariant is initially true (when $S = \emptyset$). By Lemma 1, the first step maintains the invariant. To show that every step maintains the invariant, we generalize Lemma 1 as follows:

Lemma 3 (generalizes Lemma 1)**.** *The iterative algorithm maintains the following invariant: the partial solution X is a prefix of some optimal solution.*

Proof (long form).

1. Consider the execution of the algorithm on any instance $I = (J_1, J_2, \ldots, J_n)$.
2. The invariant holds at the start of the first iteration, when $X = ()$.
3.1. Consider any iteration t such that the invariant holds at the start of the iteration.
3.2. Let $X = (X_1, \ldots, X_k)$ be the jobs in X at the start of the iteration.
3.3.1. <u>Case 1.</u> Consider the case that iteration t doesn't change X.
3.3.2. By inspection of the invariant, the iteration preserves the invariant.
3.4.1. <u>Case 2.</u> Otherwise iteration t changes X, to, say, $X' = (X_1, \ldots, X_k, J_t)$.
3.4.2. Let $S = (X_1, \ldots, X_k, J_s, Z_1, \ldots, Z_\ell)$ be an optimal solution that X is a prefix of (per invariant).
3.4.3. Let $S' = (X_1, \ldots, X_k, J_t, Z_1, , \ldots, Z_\ell)$ be obtained by replacing J_s in S' by J_t.
3.4.4. Then X' is a prefix of S'. To show the invariant is maintained, we show S' is optimal for I.
3.4.5. Since S is pairwise disjoint, the job $J_s \in S$ is disjoint from each $X_i \in S$.

3.4.6. So job J_s was not considered in any previous iteration (as it would've been added to X then).
3.4.7. So $s \geq t$, and job J_t ends no later than J_s ends. (Using here the ordering of the jobs.)
3.4.8. And J_s ends before any subsequent job $Z_j \in S'$ starts, so J_t must also.
3.4.9. And Alg. Line 3.1 ensures that J_t is disjoint from each preceding job $X_i \in S'$.
3.4.10. So S' is also pairwise disjoint. It has the same size as S, so is also optimal.
3.4.11. So the invariant is maintained.

3.5. By Blocks 3.3 and 3.4, the invariant holds after the iteration.
4. By Block 3, each iteration that starts with the invariant true ends with it true.
5. And the invariant holds initially. So it holds throughout. □

Correctness follows easily from the invariant.

Theorem 2. *The iterative algorithm is correct.*

Proof (long form).

1. Consider the execution of the algorithm on any instance I.
2. Consider the state after the final iteration n. Let X be the set at that time.
3. By Lemma 3, the invariant holds then, so $X \subseteq S$ for some optimal solution S.
4. But S can't contain any job J_t that isn't in X (otherwise job J_t, being in S, is disjoint from each job in X, and would have been added to X in iteration t).
5. So $X = S$, and the set X returned by the algorithm is an optimal solution for I. □

Other resources on the greedy Activity Selection algorithm.

- text: CLRS Section 16.1
- text: KT Section 4.1 (Kleinberg and Tardos)
- lecture slides for KT 4.1:
  `https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf`
- online text: Edmonds 7.2:
  `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/07-greedy.pdf`