1 A Scheduling Problem

- You manage a ginormous space telescope.
- Lots of astronomers want to use it to make observations.
- Each astronomer’s project $p_i$ requires use of the telescope starting at a fixed time $s_i$ (when their grant starts) and running for $\ell_i$ days.
- Only one project can use the telescope at a time.
- Your goal: justify your outrageous budget to NASA by scheduling as many projects as possible!
- More formally: given a set $P$ of projects $p_i$, each occupying half-open interval $[s_i, s_i + \ell_i)$...
- Choose a subset $\Pi \subseteq P$ of projects for which
  - No two projects’ intervals overlap (“conflict”);
  - The number of projects in $\Pi$ is maximized.
- (this is one of many variants of scheduling or activity selection problem)

1.1 Examples - learning from failure

OK – I’m open to suggestions. How should we solve this problem?

- **Suggestion 1**: repeatedly pick shortest non-conflicting, unscheduled project (i.e. that does not conflict with any scheduled project).
- Does this strategy always yield an optimal solution? Prove or disprove.
• **Counterexample:**

• **Suggestion 2:** repeatedly pick non-conflicting project with earliest starting time.

• Does this always yield an optimal solution? Prove or disprove.

• **Counterexample:**

• **Suggestion 3:** first, label each project with number of other projects with which it conflicts. Then, repeatedly pick nonconflicting project with fewest total conflicts.

• Does this always yield an optimal solution? Prove or disprove.

• **Counterexample:**

Aaaaargh! We need a *principle* to stop the endless flailing!

### 1.2 An approach that works

What structure do all above solutions have in common?

• Repeatedly pick an element until no more feasible choices remain.
Among all feasible choices, we always pick the one that minimizes or maximizes some property (project length, start time, # conflicts).

Such algorithms are called greedy.

As we've seen, greedy algorithms are frequently not optimal.

Ah, but maybe we have been using the wrong property!

Let’s take another wild guess...

- For each project $p_i$, define its finishing time $f_i$ to be $s_i + \ell_i$.
- Repeatedly pick non-conflicting, unscheduled project with earliest finishing time.
- Here’s a reasonably efficient implementation of this strategy in pseudocode.

\[
\text{Schedule}(P) \\
\text{sort } P \text{ in increasing order } \{p_1 \ldots p_n\} \text{ of finishing time } f_i \\
\Pi \leftarrow \{p_1\} \\
j \leftarrow 1 \\
\text{for } i = 2 \ldots n \{ \\
\quad \text{if } s_i \geq f_j \{ \\
\quad \quad \Pi \leftarrow \Pi \cup \{p_i\} \\
\quad \quad j \leftarrow i \} \} \\
\text{return } \Pi
\]

- For unrestricted times, sorting is $O(n \log n)$.
- Selection procedure is $O(1)$ for each $i$, so $O(n)$ overall.
- Hence, total complexity of Schedule is $O(n \log n)$.
- But does it work????

1.3 Proving correctness

Why should this greedy algorithm work when all others failed? Three key observations do it for us:

1. **Greedy Choice**: For every problem instance $P$, there exists an optimal solution that includes first element $\hat{p}$ picked by greedy algo.
2. **Inductive Structure**: After making greedy first choice \( \hat{p} \) for problem instance \( P \), we are left with smaller subproblem \( P' \), such that, if \( \Pi' \) is a feasible solution to \( P' \), then \( \Pi' \cup \{ \hat{p} \} \) is a feasible solution to \( P \).

Equivalently, we say that subproblem \( P' \) has *no external constraints* restricting its feasible solutions.

3. **Optimal Substructure**: If \( P' \) is subproblem left from \( P \) after greedy choice \( \{ \hat{p} \} \), and \( \Pi' \) is an optimal solution to \( P' \), then \( \Pi' \cup \{ \hat{p} \} \) is an optimal solution to \( P \).

Let’s prove these properties for SCHEDULE’s greedy choice.

- **Greedy Choice**: Let \( P \) be instance of scheduling problem, and let \( \hat{p} \in P \) be first project picked by SCHEDULE. Then there exists an optimal solution to \( P \) that contains \( \hat{p} \).

- **Pf**: we use an *exchange argument*.

- Let \( \Pi^* \) be *any optimal solution* to \( P \).

- If \( \hat{p} \in \Pi^* \), we are done.

- Otherwise, let \( \Pi' \) be solution obtained by removing earliest project \( p \in \Pi^* \) and adding \( \hat{p} \).

- By construction, \( \hat{p} \) ends no later than \( p \), so if \( p \) does not conflict with any later project of \( \Pi^* \), neither does \( \hat{p} \). Hence, \( \Pi' \) is feasible.

- Moreover, \( |\Pi'| = |\Pi^*| \), so \( \Pi^* \) is optimal. QED

One down, two to go.

- **Inductive Structure**: After making greedy first choice \( \hat{p} \) for problem instance \( P \), we are left with smaller subproblem \( P' \), with no external constraints.

- After we select the first project \( \hat{p} \), what is remaining subproblem?
• It’s not just $P - \{\hat{p}\}$!

• Having selected $\hat{p}$, we cannot pick any other project that conflicts with it.

• Put another way, choosing $\hat{p}$ imposes an external constraint on subproblem $P - \{\hat{p}\}$, because not every feasible solution to the subproblem can be combined with the greedy choice.

• Simple fix: define the subproblem to be

$$P' = P - \{\hat{p}\} - \{\text{projects that conflict with } \hat{p}\}.$$ 

Now any feasible solution to $P'$ can be combined with $\hat{p}$, and so there is no external constraint on $P'$.

Two down, one to go.

• **Optimal Substructure**: If $\Pi'$ is an optimal solution to subproblem $P'$, then $\Pi' \cup \{\hat{p}\}$ is an optimal solution to $P$.

• **Pf**: let $\Pi'$ be as given.

• Then $\Pi = \Pi' \cup \{\hat{p}\}$ is a feasible solution to $P$, with size $|\Pi| = |\Pi'| + 1$.

• Now suppose $\Pi$ were not optimal.

• Let $\Pi^*$ be an optimal solution containing $\hat{p}$. (Such a solution must exist by the Greedy Choice Property.)

• Then $\Pi^* - \{\hat{p}\}$ is a feasible schedule for $P'$ with

$$|\Pi^* - \{\hat{p}\}| > |\Pi - \{\hat{p}\}| = |\Pi'|,$$

contradicting optimality of $\Pi'$.

• Conclude that $\Pi$ must be optimal. QED

OK, that was fun. But why do these three facts constitute a proof that SCHEDULE always obtains an optimal solution?

• **Claim**: SCHEDULE’s solution is optimal for every problem instance $P$.

• **Pf**: by induction on size of problem $P$. 
• **Bas:** if $P$ has size 1, greedy solution is trivially as good as optimal (it picks the one element).

• **Ind:** suppose SCHEDULE’s solution is optimal for problem instances of size $< k$.

• Consider an instance $P$ of size $k$.

• Let $P'$ be subproblem obtained from $P$ after making first greedy choice, and let $\hat{p}$ be the greedy choice. Observe that $|P'| < |P|$.

• By IH, SCHEDULE optimally solves $P'$. Let $\Pi'$ be the solution it produces.

• Inductive structure property guarantees that $\Pi' \cup \{\hat{p}\}$ is a feasible solution.

• Moreover, optimal substructure property guarantees that $\Pi' \cup \{\hat{p}\}$ is an optimal solution for $P$.

• Hence, SCHEDULE optimally solves $P$ of size $k$. QED

**Key Observation:** the inductive proof uses the two structural properties as subroutines. The optimal substructure property in turn uses the greedy choice property in its proof. This form of argument is a “design pattern” for proving correctness of a greedy algorithm. It also serves as a guide to algorithm design: pick your greedy choice to satisfy G.C.P. while leaving behind a subproblem with optimal substructure!

## 2 Knapsack Problem

A classic problem for which one might want to apply a greedy algo is knapsack.

• Given: a knapsack of capacity $M$, and $n$ items.

• Item $i$ has weight $w_i > 0$, value $v_i > 0$.

• **Problem:** choose contents for the knapsack so that the total weight is at most $M$ and total value is maximized.

• (To make this interesting, we assume that $\sum_i w_i > M$, so we cannot choose everything.)

• Many versions of this problem exist, but let’s look at two.
2.1 Fractional Knapsack problem

First variant: fractional knapsack

- Can take any real-valued amount up to \( w_i \) of item \( i \).
- Examples: gold dust, gasoline, cocaine . . .
- Could also model return on activities, e.g. time spent coding vs time spent writing grants vs time spent reading papers.
- Suggestions? (Wait)
- Intuition: to maximize value, we want to take items with greatest "value density."
- Define \( d_i = \frac{v_i}{w_i} \).
- Density measures "bang for buck" from taking a fixed amount of a given item.

OK, so let’s design a (greedy) algorithm.

- Sort items in decreasing order of value density.
- Initial weight of knapsack is 0 (empty).
- For each item \( i \) in this order, add item to knapsack until it is used up or until total weight reaches \( M \).
- Cost is trivially \( O(n \log n) \). Is it correct?
- Let’s formulate and prove our key properties!

2.2 Proof that fractional Knapsack is optimal

- Greedy Choice: Consider a knapsack instance \( P \), and let item 1 be item of highest value density. Then there exists an optimal solution to \( P \) that uses as much of item 1 as possible (that is, \( \min(w_1, M) \)).

- Pf: suppose we have a solution \( \Pi \) that uses weight \( w < \min(w_1, M) \) of item 1. Let \( w' = \min(w_1, M) - w \).

- \( \Pi \) must contain at least weight \( w' \) of some other item(s), since it never pays to leave the knapsack partly empty.
• Construct $\Pi^*$ from $\Pi$ by removing $w'$ worth of other items and replacing with $w'$ worth of item 1.

• Because item 1 has max value density, $\Pi^*$ has total value at least as big as $\Pi$. QED

One down, two to go.

• **Inductive Structure**: after making greedy first choice for $P$, we are left with a smaller problem instance $P'$.

• **Pf**: We are left with a knapsack of capacity $M' < M$ (possibly 0) and a collection of remaining items to fill it. Any feasible knapsack for $P'$ may be combined with the remaining weight $M - M'$ of item 1 to form a feasible knapsack for $P$.

Two down, one to go.

• **Optimal Substructure**: optimal solution to $P'$ yields optimal solution to $P$.

• **Pf**: suppose we find optimal solution $\Pi'$ for $P'$ and combine with greedy choice to get solution $\Pi$ to $P$. Let $\hat{v} \leq v_1$ be value associated with initial greedy choice.

• Then $\text{value}(\Pi) = \text{value}(\Pi') + \hat{v}$.

• If $\Pi$ is not optimal, let $\Pi^*$ be an optimal solution that also makes greedy choice, i.e. uses as much of item 1 as possible. Remainder of knapsack after this item is removed has value greater than $\text{value}(\Pi')$, which is impossible. QED

• **Note**: that last bit of argument gets repetitive to write. It is enough for your homework to recognize that

\[ \text{value}(\Pi) = \text{value}(\Pi') + \hat{v}, \]

that is, that the value of the full solution is the value of the subproblem’s solution plus that of the greedy choice.
2.3 And now, a classic failure

Lest you get all excited and think that greed always works... 

- **0-1 knapsack** problem does not permit items to be subdivided.
- **Example:** gold bar, painting, iPod
- Each item still has weight $w_i$ and value $v_i$.
- Goal is to maximize value of knapsack without going over weight $M$.

Fractional algo makes no sense in this context. What to do?

- Well, we can still assign value density $d_i$ to each item.
- Intuitively, items of high value density are more attractive (diamond vs an equal-sized chunk of coal).
- **Suggestion:** sort items by decreasing value density as before, then choose items of highest density until next item would exceed total weight of $M$.
- Does this work? (wait)
- Counterexample with 3 items and $M = 5$:

- values = 20, 30, 40; weights = 2, 3, 3.5
- densities are 10, 10, 11.4
- Greedy algo picks item of weight 3.5 first, then stops with value 40.
- Optimal solution would take other two items for total value 50.

What broke?

- There is no optimal solution that contains the greedy choice!
- Hence, greedy choice property fails for this problem.
- (In fact, it is NP-hard, but we don’t know that yet!)
3 Huffman Coding

3.1 Set up for Huffman coding

We’re now going to look at a very important application of greedy algorithm design. In fact, you probably use it every day at least a few times without knowing it.

- **Setting**: data compression (ZIP, gzip, etc)
- Suppose we have a text consisting of a collection of letters.
- We want to encode this text efficiently on a computer.
- If there are at most $2^k$ distinct letters, each one can be represented by a $k$-bit code. For $n$-letter text, total size in bits is $kn$.
- However, what if letter frequencies are very unequal?
- **Example**: in *Moby Dick*, there are 117194 occurrences of ‘e’, but only 640 occurrences of ‘z’. Other letters are in-between.
- **Idea**: encode common letters with fewer bits!
- **Example** on text of 100k letters from 6-letter alphabet:
  - Smallest fixed-length encoding uses 3 bits per letter, hence 300,000 total bits.
  - Specified encoding uses only 224,000 bits.

Wait, are variable-length encodings possible?
- **Stupid example**: consider the following encoding of a 3-letter alphabet:
  - \( a \to 0 \)
  - \( b \to 1 \)
  - \( c \to 01 \)

- Hence, the message “ababc” would be encoded as “010101”.

- Anyone see a problem here?

- “ccc” is also encoded as “010101.” If you receive these bits, which message was sent?

- This is the *ambiguity problem*.

- In contrast, first code is *unambiguous* – no encoded message can be decoded as two different strings.

- *sufficient condition*: no code word is a prefix of any other code word. (Pf left as exercise)

- Such codes are called *prefix-free*, or simply *prefix codes*.

We will look at *Huffman coding*, a technique for optimal prefix code generation.

- **Problem**: given a sequence \( T \) built from letters \( X = \{x_1 \ldots x_n\} \), such that \( x_i \) occurs \( f_i \) times in \( T \).

- Produce an encoding function \( C(x) \) that maps characters to bit strings, s.t.
  1. \( C(X) \) is a prefix code.
  2. Total number of bits used to represent \( T \) is minimized. That is, minimize \( B(T) = \sum_i f_i \cdot |C(x_i)| \).

### 3.2 Prefix codes as trees

Now, Huffman coding algorithm and proof. We first need to come up with a framework for designing prefix codes.

- **Idea**: represent a code as a binary tree.
Each edge of tree is labeled with a bit (0 or 1).

- Left edges get 0’s, right edges get 1’s.

- Each letter $x_i$ labels one leaf $\ell_i$ of tree.

- Codeword corresponding to $x_i$ is given by the bitstring labeling path from root down to $\ell_i$.

- Example:

A few important observations...

- **Fact**: no two leaves get the same codeword (they have different paths from root).

- **Fact**: because letters appear only at leaves, code corresponding to tree is a prefix code.

- (Otherwise, some codeword would end at an internal node.)

- **Fact**: in tree for an optimal code (min total # of bits), every internal node has two children.

- **Pf**: Let $R$ be tree corresponding to a code, and suppose some int node $v \in R$ has one child $w$.

- Consider revised tree $R'$ that deletes edge $(v, w)$ and hangs subtree rooted at $w$ off of $v$.

- Codewords for all letters below $w$ in $R$ are one bit shorter in $R'$, and they do not collide with any other codewords from $R$. Hence, $R$ does not yield optimal code.
One more important definition.

- Let depth of leaf $\ell_i$, denoted $d(\ell_i)$, be length of codeword labeling path from root to $\ell_i$ in $R$.
- So, how many bits are used to represent text $T$ with tree $R$?
- All copies of letter $x_i$ together use $f_i \cdot d(\ell_i)$ bits.
- Hence,
  \[ B(T) = \sum_i f_i \cdot d(\ell_i). \]

### 3.3 Finding an optimal tree

We’ve reduced the problem to searching the space of all binary trees with $n$ leaves labeled with letters $x_i$, with 2 children per internal node. Goal is to find one labeled tree that minimizes $B(T)$.

- Intuitively, we want a tree that puts rare letters at high depth and common letters at low depth.
- **Idea**: build tree from bottom up. We will stick together subtrees until we have one full tree.
- Let $L = \{\ell_1, \ldots, \ell_n\}$ be set of leaves for all chars. Let $f_i$ be frequency of letter $x_i$ corresponding to leaf $\ell_i$.
- Find the two leaves $\ell_a$ and $\ell_b$ in $L$ with two lowest frequencies $f_a$ and $f_b$.
- Link these leaves into a single subtree $R_{ab}$, and create a new “megaleaf” $\ell_{ab}$ with frequency $f_a + f_b$.

- Recursively solve problem on reduced input
  \[ L \cup \{\ell_{ab}\} - \{\ell_a, \ell_b\}. \]
- Stop when $L$ contains one megaleaf for whole tree.
• Finally, expand each megaleaf recursively from root to get final tree.

Note that this is a greedy algorithm: repeatedly join two least frequent leaves into one, until only one leaf remains.

3.4 Proof of optimality

We take the usual route to an optimality proof for greedy algorithms.

• Greedy Choice: Let $L$ be an instance of the encoding problem, and let $\ell_a, \ell_b \in S$ be two leaves chosen for linking by greedy algorithm. Then there exists an optimal tree for $L$ containing $R_{ab}$.

• Pf: Let $R$ be any optimal tree for $L$. If $R_{ab}$ is subtree of $R$, we are happy!

• Otherwise, let $\ell_x$ and $\ell_y$ be a pair of leaves in $R$ w/common parent, such that $\delta = d(\ell_x) = d(\ell_y)$ is maximal.

• Assume that neither $x$ nor $y$ is one of $a, b$. (If it is, simplified version of following argument still works.)

• Modify $R$ to obtain a new tree $R^*$ by exchanging the positions of $\ell_a$ with $\ell_x$ and $\ell_b$ with $\ell_y$. Now $R^*$ contains $R_{ab}$.

• Let $B_R(T)$ be number of bits used for text $T$ by $R$’s encoding.

• For new tree $R^*$, number of bits is given by

$$B_{R^*}(T) = B_R(T) - (f_x + f_y)\delta - f_a d(\ell_a) - f_b d(\ell_b) + (f_a + f_b)\delta + f_x d(\ell_a) + f_y d(\ell_b)$$

$$= B_R(T) + (f_a - f_x)(\delta - d(\ell_a)) + (f_b - f_y)(\delta - d(\ell_b)).$$
Note that, by greedy choices of \( a \) and \( b \), \((f_a - f_x) \leq 0 \) and \((f_b - f_y) \leq 0\).

Moreover, by choice of \( x \) and \( y \), \((\delta - d(\ell_a)) \geq 0 \) and \((\delta - d(\ell_b)) \geq 0\).

Conclude that \( B_{R'}(T) \leq B_R(T) \), and so \( R^* \) is optimal too. QED

One down, two to go.

**Inductive Structure**: first step of greedy algorithm leaves us with smaller instance \( L' \) of same problem.

**Pf**: let \( L' \) be set of leaves after first linking step. \( L' \) contains a smaller set of leaves with associated frequencies. Any strategy for joining the leaves of \( L' \) into a tree is compatible with our greedy choice, since we can replace the megaleaf \( R_{ab} \) in the final tree by the subtree with leaves \( \ell_a \) and \( \ell_b \).

Two down, one to go.

**Optimal Substructure**: let \( R' \) be an optimal tree constructed from leaves of \( L' \) after linking \( \ell_a \) and \( \ell_b \). Then \( R \), the tree obtained by replacing \( \ell_{ab} \) in \( R' \) with subtree \( R_{ab} \), is optimal.

**Pf**: Let \( d'(\ell_{ab}) \) be depth of megaleaf \( \ell_{ab} \) in \( R' \). Then in \( R \), we have

\[
d(\ell_a) = d(\ell_b) = d'(\ell_{ab}) + 1.
\]

Conclude that \( B_R(T) \) is given by

\[
B_R(T) = B_{R'}(T) - (f_a + f_b)d'(\ell_{ab}) + f_a d(\ell_a) + f_b d(\ell_b)
= B_{R'}(T) - (f_a + f_b)d'(\ell_{ab}) + (f_a + f_b)(d'(\ell_{ab}) + 1)
= B_{R'}(T) + f_a + f_b.
\]

Does usual contradiction argument work? Yes!

(Exercise: work through it.) QED

3.5 Complexity

How efficiently can we implement Huffman coding?

- Maintain leaf set \( L \) as priority queue keyed on frequency.

15
To find two least frequent leaves in $L$, do two extractMin ops.

Then insert new megaleaf back into $L$.

What does this cost?

- Each linking phase does two extractions and one insertion.
- Each of these three ops is $O(\log n)$ w/binary heap.
- There are only $n - 1$ phases, so total cost is $O(n \log n)$.

4 The Cable Guy Problem

- You are an installer for Charter Cable.
- Your job is to do installations by appointment.
- Appointments have a fixed length (30 minutes) and must start on a 30-minute boundary.
- Each new customer specifies an earliest and latest possible start time for her appointment.
- **Goal**: schedule as many appointments as possible in a day.

More formally...

- Given set $P$ of unit-duration jobs that must be scheduled at integer times (“time slots”) 0, 1, 2, . . .
- Only one job may be scheduled per slot.
- Job $i$ may start in any slot from $s_i$ to $e_i$, inclusive.
- **Goal**: find a schedule that maximizes number of jobs scheduled.

4.1 A greedy algorithm

OK, here’s a simple algorithm. (Note: this is symmetric to the one we worked out in class – it works backward from the latest job, not forward from the earliest job.)

- Let $P$ be input set of jobs.
• Let $t$ be latest free slot in which some job from $P$ may be scheduled.
• Let $\Sigma \subset P$ be set of all jobs in $P$ that can be scheduled at time $t$.
• Choose job $i \in \Sigma$ with latest start time $s_i$.
• Schedule job $i$ at time $t$.
• Recur on remaining job set $P - \{i\}$, until this set is empty or no jobs can run.

Example:

4.2 Optimality
Let’s implement our three-part proof...

• **Greedy Choice Property**: let $\hat{i}$ be the job chosen first by greedy algo, and let $\hat{t}$ be the time at which algo scheduled it. Then . . .

  . . . there exists an optimal solution that makes the greedy choice, that is, a soln that schedules $\hat{i}$ at time $\hat{t}$.

  • (Note: it is not enough to find opt soln that uses $\hat{i}$ – it must be placed at time $\hat{t}$, since this is what greedy choice does!)

OK, on with the proof!

• **Pf**: Let $\Pi$ be any optimal solution.
• If $\Pi$ schedules $\hat{i}$ at time $\hat{t}$, great.
• Otherwise, we have two cases.
  • **Case 1**: Suppose $\Pi$ doesn’t schedule $\hat{i}$ at all.
    • If slot $\hat{t}$ is empty in $\Pi$, can simply add $\hat{i}$ for better soln.
• If slot \( \hat{t} \) is occupied by job \( j \), throw out \( j \) and put \( \hat{i} \) there. Soln has same size as before.

• \textit{Case 2}: Suppose \( \Pi \) schedules \( \hat{i} \) at time \( t \neq \hat{t} \)

• If slot \( \hat{t} \) is empty in \( \Pi \), move \( \hat{i} \) to \( \hat{t} \).

• Otherwise, some job \( j \) occupies slot \( \hat{t} \).

• Observe that \( t < \hat{t} \), since by choice of \( \hat{t} \), no job in \( P \) can be scheduled after time \( \hat{t} \).

• Observe also that by choice of job \( \hat{i} \),

\[ s_{\hat{i}} \geq s_j. \]

• Since \( t \geq s_{\hat{i}} \), job \( j \) can run at time \( t \).

• Hence, we simply exchange slots of jobs \( \hat{i} \) and \( j \).

• In all cases, new soln has size at least that of old and so is optimal.

QED

One down, two to go.

• \textbf{Inductive Structure}: after making greedy choice, we are left with smaller instance of scheduling problem with no external constraints.

• \textbf{Pf}: Let \( P' = P - \{\hat{i}\} \).

\( P' \) is clearly a smaller set of jobs to be scheduled.

But what about constraints?

• First time greedy schedules a job, it can use latest slot that any job can take.

• For subsequent choices, this property does not hold! Consider our simple example.

• It seems we are more constrained on recursive calls than on original call! This breaks inductive structure.

• \textit{Two solutions here}. Either extend the problem, or more carefully define the recursive call.
• **Problem extension**: input to problem includes a list of blocked slots. On recursion, also block \( \hat{t} \).

• Check that my algo description and proof of GCP need not change under this extension!

• (This is nice because you can allow the cable guy to take a lunch break.)

• **Recursion Defn**: as it happens, times are filled in from latest to earliest.

• So, trim ends of all unscheduled jobs back to at most \( t - 1 \).

• Also, remove any jobs \( j \) for which \( s_j \geq t \).

• For remaining jobs, algo never looks beyond \( t - 1 \), so does not see additional constraints.

• (Might be able to simplify GCP proof a bit, but forbids lunch breaks.)

• With one of the above two hacks, adding \( \hat{i} \) in slot \( \hat{t} \) will be possible no matter how the subproblem is solved, so subsolution plus greedy choice is a feasible solution.

Two down, one to go.

• **Optimal Substructure**: let \( \Pi' \) be opt solution to subproblem \( P' \). Then \( \Pi' \) together with \( \hat{i} \) at time \( \hat{t} \) is opt solution to \( P \).

• Value of \( \Pi \) is value of \( \Pi' \), plus one.

• Apply usual contradiction argument. QED

**Moral**: be careful what your subproblem is. You may need to generalize your problem defn to make an inductive proof go through.

### 4.3 A fast implementation

How can we implement this algorithm efficiently?

• Let \( t \) be latest free slot.

• Observe that \( t \) decreases monotonically – we always schedule something in latest free slot for any job.
• Each time $t$ decreases, it may pass $e_i$ for one or more jobs $i$. These jobs join eligible set $\Sigma$.

• $t$ may also pass $s_i$ for one or more unscheduled jobs. These jobs are deleted from $\Sigma$ and are unschedulable.

• At any time, we want the job in $\Sigma$ with largest $s_i$.

These observations suggest the right data structure.

- Make sorted list $L$ of jobs in $P$ in decreasing order by $e_i$.
- $t \leftarrow \infty$
- Let $Q$ be a max-first priority queue, keyed by job start times.
- Repeat following loop until $t = 0$ or $L$ is empty:
  - If $Q$ is empty, $t \leftarrow$ ending time of next job in $L$.
  - Otherwise, $t \leftarrow t - 1$.
  - Move all jobs $j$ with $e_j = t$ from $L$ to $Q$.
- $i \leftarrow Q$.ExtractMax()
- Schedule job $i$ at time $t$
- Extract and discard any jobs with start time $t$ from $Q$.

Running time?

- Let $n$ be number of jobs in $P$.
- Each job is added to $Q$ exactly once and removed exactly once.
- Total cost: $O(n \log n)$ for binary heap.
- $t$ is decremented only as many times as a job is scheduled.
- Total cost: $O(n)$.
- List must initially be sorted.
- Total cost: $O(n \log n)$.
- Conclude that overall cost is $O(n \log n)$. 