

Greedy Algorithms

CS5800 Algorithms Lecture Notes - Consolidated with Codex

We begin with two ideas from divide-and-conquer: a structural check on the solution, and a pipeline for computing it. These also organize our discussion of greedy algorithms and dynamic programming (DP).

1 The Shared Framework: Structure First, Choice Second

Step 1 check for divide-and-conquer: solution structure.

Write PB for the problem, SPB_i for its subproblems, and Sol for a solution. The first check is:

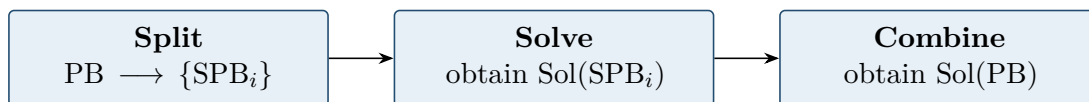
$$\boxed{\text{Sol}(\text{PB}) = \text{Combination}\{\text{Sol}(\text{SPB}_i)\}.$$

For this analysis, suppose the solution is given. Check that it is structurally composed of, or obtainable by combining, solutions to the proposed subproblems. We are examining the shape of a solution, not assuming the algorithm already knows it.

What are the subproblems? What information must each retain? How do their solutions fit together? The combination may include the decision that created the subproblems. The displayed equation expresses composition, not necessarily numerical addition.

The divide-and-conquer pipeline: split, solve, combine.

Once the structural check is justified, the computational pipeline is:



The check and the pipeline serve different roles. The check asks whether solutions have the required structure. The pipeline describes how to compute a solution: split the problem, solve the subproblems, and combine their solutions into the whole solution. Smaller instances use the same procedure until reaching directly solvable base cases.

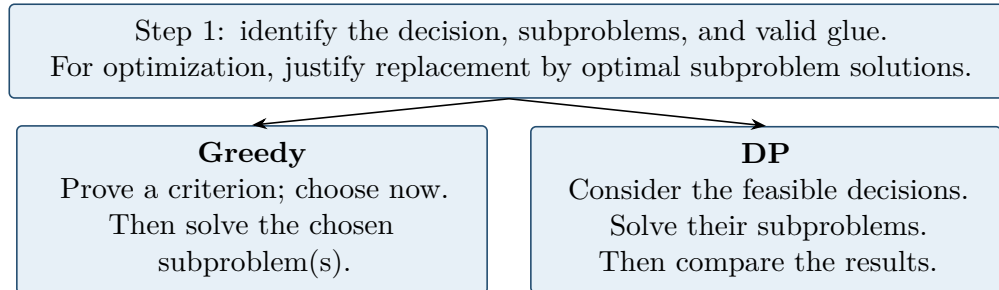
Example: MergeSort. Split the input into two halves, sort each half, then merge the two sorted sequences. The structural check is that a sorted version of the whole input can be obtained by merging sorted versions of those halves.

Key point: Keep both ideas in view: $\text{Sol}(\text{PB})$ is a combination of subproblem solutions; the algorithm follows *split* $\rightarrow$ *solve* $\rightarrow$ *combine*. Greedy and DP differ in when they can settle on the decision or split.

Core examples. Our backbone is fractional knapsack, maximum-count activity selection, and coins with a greedy denomination set. Huffman coding extends the same framework to a different, more mathematical setting: weighted trees.

1.1 From solution structure to an early decision

Greedy and DP both have subproblems, explicit or implicit. For optimization, Step 1 includes a **replacement check**: could a better solution of a residual subproblem improve the whole solution while preserving feasibility? This justifies using optimal subproblem solutions, but does not yet tell us which decision to make.



Greedy: commit before solving. Prove that *some* optimal solution agrees with the proposed choice. Then solve only the chosen continuation, without solving competing alternatives. Step 1 and induction complete the correctness argument.

DP: compare after solving. Consider the relevant feasible choices, solve their required subproblems, and select a best combined result. No greedy-choice proof is needed, but the decomposition must still be correct and cover an optimal solution. Keeping alternatives often means solving many more subproblems.

Conventions. “Optimal” means best for the stated objective among feasible solutions. Arithmetic and comparisons take constant time unless noted otherwise; $\lg n = \log_2 n$.

2 Fractional Knapsack: Exchange Equal Weight

There are n divisible goods. Good i has available weight $w_i > 0$ and total value $v_i \geq 0$. Its value per unit of weight, or *quality*, is

$$q_i = \frac{v_i}{w_i}.$$

With capacity $W \geq 0$, choose weights x_i to maximize

$$\sum_{i=1}^n q_i x_i \quad \text{subject to} \quad 0 \leq x_i \leq w_i, \quad \sum_i x_i \leq W.$$

An x_i is a weight, not a fraction; the fraction of good i taken is x_i/w_i .

Good	Available weight w_i	Total value v_i	Quality q_i
Coffee	25	30	$6/5 = 1.2$
Tea	50	40	$4/5 = 0.8$
Flour	20	15	$3/4 = 0.75$
Maize	70	10	$1/7$

Let $W = 50$. Taking the largest total value first fills the bag with tea and earns 40. Taking the smallest available weight first gives 20 flour, 25 coffee, and 5 tea, earning $15 + 30 + 4 = 49$. Taking the highest quality first gives 25 coffee and 25 tea, earning

$$25 \cdot \frac{6}{5} + 25 \cdot \frac{4}{5} = 50.$$

The last rule is correct; the other two examples merely show plausible rules can fail.

Step 1: what remains after fixing an amount?

Suppose a given optimal solution takes x_g units of good g . Once this entire amount of g is fixed, the remaining goods must optimally fill capacity $W - x_g$. A better feasible allocation among those goods could replace the old one and improve the total value. The glue is simply to add x_g units of g and its value $q_g x_g$.

This structural fact does not choose g or x_g . It only tells us how a fixed decision connects the original problem to a smaller problem.

Step 2: take as much as possible of highest quality.

Order the goods by nonincreasing quality, $q_1 \geq \dots \geq q_n$, fixing one common order for ties. The greedy-choice solution (GCS) takes as much as possible of each good in that order. Write g_i for its quantity of good i :

$$g_i = \min \left\{ w_i, W - \sum_{h < i} g_h \right\}.$$

Compare the whole solutions: first difference and exchange.

The main argument is a **proof by contradiction**, not a formal induction proof. We first develop the strict-improvement case; the questions below ask students to complete the remaining cases.

- 1. Assume GCS is not optimal.** Then there is a best solution OPT with strictly greater value. Write o_i for its quantity of good i . Compare both solutions in the *same quality order*, including zero quantities for unused goods.
- 2. Find the first difference.** Let k be the first index with $g_k \neq o_k$. Thus $k - 1$ ends the initial block of agreement (the block is empty if $k = 1$). All earlier quantities are equal and full:

$$g_i = o_i = w_i \quad (i < k).$$

Indeed, if greedy had already stopped short of a full quantity earlier, it would have exhausted the capacity; an allocation agreeing through that point could not differ later. At k , both solutions have the same remaining capacity, and greedy takes the maximum permitted quantity. Hence

$$g_k > o_k.$$

First difference: $g_k > o_k$

	1	...	$k-1$	k	$k+1$	
GCS	w_1	...	w_{k-1}	g_k	g_{k+1}	...
OPT	w_1	...	w_{k-1}	o_k	o_{k+1}	...

Equal quantities; each full



In OPT: move ε weight to good k

3. Exchange inside OPT. For the main argument, assume OPT uses item $k+1$ and that $q_k > q_{k+1}$. Take a sufficiently small quantity $\varepsilon > 0$ out of o_{k+1} and add it to o_k :

$$o'_k = o_k + \varepsilon, \quad o'_{k+1} = o_{k+1} - \varepsilon.$$

Total weight is unchanged. Choose ε small enough to preserve nonnegative quantities and the supply limit of item k ; justifying this choice is left to the questions below. The feasible modified allocation has value increase

$$V(\text{OPT}') - V(\text{OPT}) = \varepsilon q_k - \varepsilon q_{k+1} = \varepsilon(q_k - q_{k+1}) > 0.$$

Contradiction: OPT was a best solution, but we have constructed a strictly better feasible one.

Complete the argument: questions for students.

1. Why can we choose $\varepsilon > 0$? State the conditions it must satisfy and justify that a positive quantity is possible.
2. What if item $k+1$ does not exist in OPT, or its quantity is zero? How should the argument handle the missing next item, including unused capacity and zero-quality goods?
3. What if $q_k = q_{k+1}$? The exchange then does not strictly improve OPT. How can you finish the argument without claiming a false strict inequality?

These cases are part of completing the proof, not assumptions that may be silently omitted. A formal induction presentation is left to the homework; the lecture's organizing argument compares complete solutions at their first difference.

Implementation and running time.

Algorithm: FractionalKnapsack(w, v, W)

```
1: compute  $q_i = v_i/w_i$  and order goods by nonincreasing  $q_i$ 
2: set every  $x_i \leftarrow 0$ ;  $R \leftarrow W$ 
3: for each good  $i$  in that order do
4:    $x_i \leftarrow \min\{w_i, R\}$ ;  $R \leftarrow R - x_i$ 
5:   if  $R = 0$  then
6:     break
7:   end if
8: end for
9: return  $(x_1, \dots, x_n)$  in the original input order
```

Sorting costs $O(n \lg n)$, and the scan costs $O(n)$. If qualities are already sorted, the scan alone is linear. At most one good is partially taken in this implementation. Outputting the allocation uses $\Theta(n)$ space.

2.1 Why the fractional assumption matters

In 0/1 knapsack, each good must be taken whole or not at all. The equal-weight exchange may no longer be legal. For capacity 50 and goods

$$(w, v) = (10, 60), (20, 100), (30, 120),$$

decreasing-quality order takes the first two, worth 160. The last two fit exactly and are worth 220. The subproblem structure survives, but the greedy-choice proof does not.

Homework connection. One assigned 0/1 variant promises that lighter items are also at least as valuable. Such an extra assumption can make a greedy approach possible even though the general problem defeats it. Formulating the algorithm and proving its correctness under that promise are left to HW4.

3 Activity Selection: Leave the Best Continuation

Activity i occupies $[s_i, f_i)$, with $s_i < f_i$. Select a **maximum number** of pairwise nonoverlapping activities. The objective is not total occupied time or total profit. A *maximal* schedule merely cannot be extended; it need not have maximum cardinality.

Step 1: the remaining activities. Suppose a given optimal schedule begins with activity a . Define

$$S_a = \{i \in S : s_i \geq f_a\}.$$

Its continuation must be optimal for S_a : a larger compatible continuation could be appended to a and improve the original schedule. The glue is to prepend a . Formally, for nonempty S ,

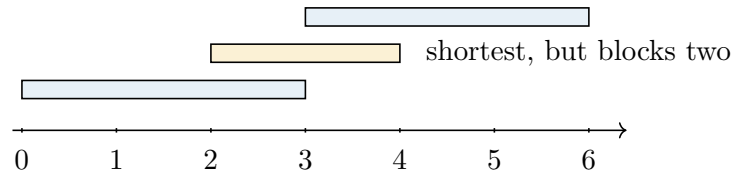
$$\text{OPT}(S) = \max_{a \in S} \{1 + \text{OPT}(S_a)\}, \quad \text{OPT}(\emptyset) = 0.$$

This is the structural “try all first activities” view. A greedy proof will let us replace that search by one first activity.

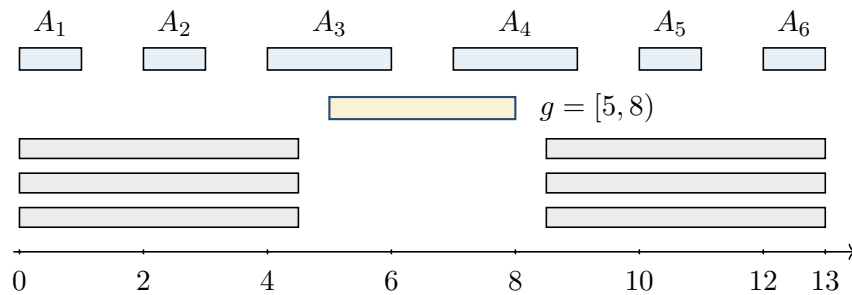
Plausible rules that fail.

Earliest start. On $[0, 10), [1, 2), [2, 3), [3, 4)$, the first-starting activity blocks the other three.

Shortest duration. On $[0, 3), [2, 4), [3, 6)$, the unique shortest activity $[2, 4)$ blocks both others, although $[0, 3)$ and $[3, 6)$ are compatible.



Fewest overlaps. Even minimizing the number of currently conflicting activities can fail. The following explicit version of the lecture's counterexample avoids tie-breaking ambiguity. The six blue activities are mutually compatible. There are three distinct activities with interval $[0, 4.5)$ and three with interval $[8.5, 13)$; distinct activities may have identical endpoints.



The activity g overlaps only A_3, A_4 , so its conflict count is 2. Each of A_1, A_2, A_5, A_6 conflicts with three gray activities; A_3, A_4 each conflict with three gray activities and g . Each gray activity conflicts with three blue activities and its two gray copies. Thus g is the unique minimum-conflict choice.

After choosing g , at most two activities can be selected to its left and two to its right: on either side, choosing a gray interval excludes both surviving blue intervals. Any schedule containing g therefore has at most 5 activities, whereas the six blue activities give a feasible schedule of size 6.

Step 2: choose the earliest finish.

Let g be an activity with minimum finish time. Take an optimal schedule, and let a be its first activity. Because $f_g \leq f_a$, every later activity in that schedule starts at or after f_a , hence also at or after f_g . Replacing a by g keeps the schedule feasible and preserves its cardinality.

Thus **some optimal schedule begins with g** . By Step 1, what follows g is optimal for S_g . Apply the same rule to that smaller instance. Induction on the number of available activities proves optimality, with the empty instance as the base case.

The important comparison is not that g overlaps fewer activities. It is that any continuation available after a remains available after g :

$$f_g \leq f_a \implies S_a \subseteq S_g.$$

3.1 A second proof: greedy stays ahead

The exchange above repairs an optimal schedule. Here is a different comparison: after the same number of activities, greedy has finished no later, so it has at least as much room for the continuation.

Let the greedy schedule be $g_1, \dots, g_r$, and let an optimal schedule be $o_1, \dots, o_m$, each in chronological order. We claim that for every $k \leq m$, g_k exists and

$$f_{g_k} \leq f_{o_k}.$$

For $k = 1$, the earliest-finishing activity finishes no later than o_1 . If the claim holds at $k < m$, then

$$s_{o_{k+1}} \geq f_{o_k} \geq f_{g_k}.$$

So there is at least one next candidate for greedy, namely o_{k+1} . Greedy chooses an earliest-finishing candidate, giving $f_{g_{k+1}} \leq f_{o_{k+1}}$. This proves both existence and the inequality for the next step.

Consequently $r \geq m$: greedy cannot run out of compatible activities before OPT does. But the greedy schedule is feasible and m is optimal, so $r \leq m$. Therefore $r = m$.

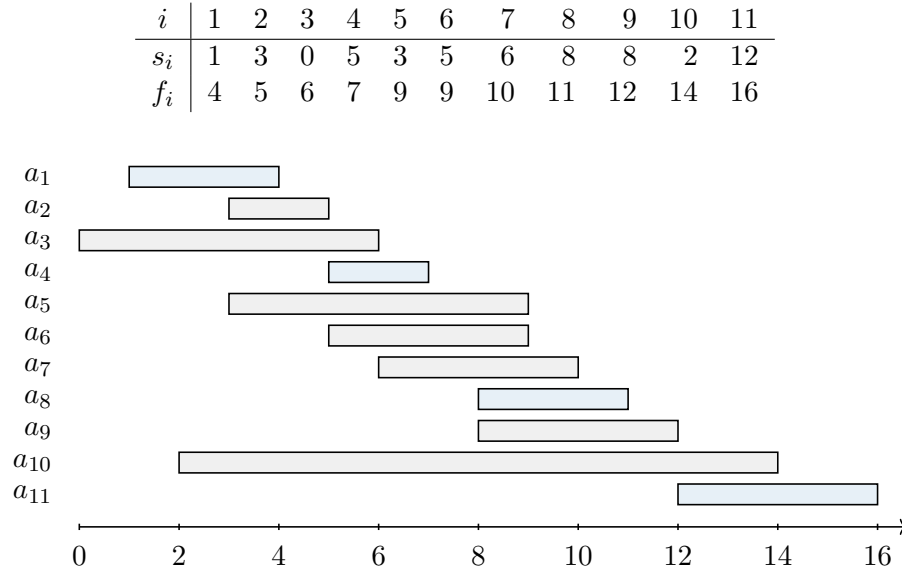
A sorted scan, not repeated searching.

Algorithm: ActivitySelection(S)

```
1: sort activities so that  $f_1 \leq f_2 \leq \dots \leq f_n$ 
2:  $A \leftarrow [ ]$ ;  $t \leftarrow -\infty$ 
3: for  $i = 1, 2, \dots, n$  do
4:   if  $s_i \geq t$  then
5:     append activity  $i$  to  $A$ ;  $t \leftarrow f_i$ 
6:   end if
7: end for
8: return  $A$ 
```

Sorting costs $O(n \lg n)$; the scan costs $O(n)$. With finish-sorted input, the algorithm is linear. Other than sorting and the output, the scan stores only its current finish time and index. Equal finish times do not invalidate the proof.

The eleven-activity example.



The selected sequence is a_1, a_4, a_8, a_{11} , with finish times 4, 7, 11, 16. In the scan, skip a_2, a_3 after choosing a_1 ; skip a_5, a_6, a_7 after choosing a_4 ; skip a_9, a_{10} after choosing a_8 .

3.2 If the objective changes, revisit the proof

Suppose activities have profits v_i . Replacing the first activity by an earlier-finishing one may reduce total profit, so the cardinality exchange proof no longer establishes optimality. For example, $[0, 2)$ of profit 1 and $[1, 3)$ of profit 100 defeat earliest finish.

After sorting by finish time, let $p(j)$ be the largest index $i < j$ with $f_i \leq s_j$, or 0 if none exists. A structural split according to whether activity j is selected gives

$$M(0) = 0, \quad M(j) = \max\{M(j-1), v_j + M(p(j))\}.$$

The first branch excludes j . The second includes j and uses an optimal compatible prefix. These cases cover every solution; both constructions are feasible. That is the recurrence justification. DP compares the resulting values instead of claiming that either branch is locally safe. Maximizing total occupied time is this weighted problem with $v_j = f_j - s_j$, not the original cardinality problem.

4 Coin Change: Structure Does Not Guarantee a Greedy Rule

Let D be a finite set of positive integer denominations, with an unlimited supply of each. For a nonnegative integer amount A , minimize the number of coins whose values sum to A .

Step 1 works for arbitrary denominations. Suppose an optimal solution contains a coin of value d . Removing that coin leaves an optimal solution for $A - d$: otherwise replace the remainder by a solution using fewer coins. Consequently

$$C(0) = 0, \quad C(A) = 1 + \min_{\substack{d \in D \\ d \leq A}} C(A - d) \quad (A > 0).$$

Use $+\infty$ for an unreachable amount and for the minimum of an empty set. This recurrence expresses the structural alternatives; its validity does not assume that the largest coin is safe. Implementing the comparison of alternatives belongs to the DP discussion.

A proposed greedy rule: largest coin first.

The natural rule is to take the largest denomination not exceeding the remaining amount, then repeat on the remainder. For example, with denominations $\{1, 5, 10, 25\}$, this rule constructs $30 = 25 + 5$ and $22 = 10 + 10 + 1 + 1$.

The rule works for some denomination systems, but not for arbitrary ones. Successful traces are not a proof of optimality, and the optimal-remainder property does not establish that the largest first coin is safe. Constructing a specific failure example is part of HW4, so none is supplied here.

Key point: The optimal remainder property is true here, but the largest-coin choice is false in general. Step 1 and a greedy-choice proof are genuinely different requirements.

An explicit greedy-choice proof for a greedy set.

For denominations including 1, call D a **greedy set** if largest-coin-first minimizes the number of coins for *every* nonnegative integer amount. This property of the denomination set, not merely a successful run, is the guarantee we need.

Worked lecture example: $D = \{1, 3, 6\}$. We prove the claim for every amount A , comparing complete solutions rather than giving a formal induction proof. Let an optimal solution use o_1, o_3, o_6 coins of the respective denominations.

1. **Rule out wasteful groups by exchange.** If $o_1 \geq 3$, replace three 1-coins by one 3-coin. The amount is unchanged, but the number of coins decreases by two, contradicting optimality. If $o_3 \geq 2$, replace two 3-coins by one 6-coin, reducing the count by one. Hence every optimum satisfies

$$0 \leq o_1 \leq 2, \quad 0 \leq o_3 \leq 1.$$

2. **Force the largest-coin choice.** The total value supplied by smaller coins is at most $2 + 3 = 5$. Therefore

$$A = 6o_6 + (3o_3 + o_1), \quad 0 \leq 3o_3 + o_1 < 6,$$

so $o_6 = \lfloor A/6 \rfloor$, exactly the number of 6-coins taken by greedy.

3. **Force the remaining choices.** Let $R = A - 6\lfloor A/6 \rfloor$. Since $R = 3o_3 + o_1$ and $0 \leq o_1 < 3$,

$$o_3 = \lfloor R/3 \rfloor, \quad o_1 = R - 3\lfloor R/3 \rfloor.$$

These too are exactly greedy's quantities. Thus an optimal solution has precisely the greedy coin counts: largest-coin-first is optimal for this set.

The exchanges do more than suggest that large coins are attractive. They constrain *every* optimum until the greedy quantities are forced. The argument depends on these denominations; it is not a proof for an arbitrary coin set.

Other denomination sets: questions for HW4.

Two useful settings to investigate are the lecture's $\{1, 5, 10, 25\}$ system and denominations that are powers of a fixed integer base. The broad candidate is the same in both settings: largest coin first. What changes is the justification for trusting that choice.

The proofs for these systems, and a counterexample for a different system, are assigned in HW4. We leave those arguments to the assignment. The lecture takeaway is the distinction between a structural recurrence valid for all denomination sets and an early commitment that needs a system-specific guarantee.

5 General Ideas for Greedy-Choice Proof Techniques

We have now seen the techniques inside fractional knapsack, activity selection, and coins. They are complementary tools, not mutually exclusive proof templates. In every case, first identify the objective, feasibility constraints, and the Step 1 structure; then justify the early commitment.

Contradiction: what exactly is contradicted?

Assume the greedy solution is worse than an optimum, or assume an optimum violates a proposed necessary property. Derive an impossibility. In fractional knapsack's strict-quality case, the exchange produces a feasible solution of greater value than OPT. In the coin proof, forbidden groups produce an equal amount using fewer coins.

A modification that merely ties OPT does *not* contradict optimality. State precisely whether the objective improves strictly or an optimal solution is only being rearranged. This distinction matters in the knapsack tie exercise.

Exchange: change a solution without breaking it.

Choose a small part of an optimal solution and replace it. Explicitly check three things: what is removed and added, why feasibility is preserved, and how the objective changes. Exchange equal weight in fractional knapsack, one activity for one activity in scheduling, or equal coin value in coin change. An exchange can prove strict improvement by contradiction, or show that some optimum agrees with a greedy choice without changing its value.

Induction: formalize repeated safe choices.

A formal induction proof needs a stated parameter, base case, hypothesis, and step. One useful statement is: *after the first k greedy choices, there exists an optimal solution agreeing with those choices*. In the step, keep the agreed prefix fixed, justify the next choice on the residual problem, and obtain agreement through $k + 1$. At termination, explain why this establishes optimality of the complete greedy solution.

Another formulation inducts on the size of the residual instance, using Step 1 to combine a safe choice with an optimal smaller solution. The knapsack lecture proof above is organized as a whole-solution contradiction; writing its formal induction version is separate work.

Staying ahead: compare progress, then the final objective.

Choose a measure of progress after the same number of choices, and show that greedy is never behind a comparison solution. The measure must preserve useful future options. For activities it is finish time: $f_{g_k} \leq f_{o_k}$. The proof must also show that g_k exists whenever o_k exists, then conclude that greedy selects at least as many activities as OPT. “Earlier is better” alone does not establish that conclusion. This argument can itself use induction to maintain its comparison.

Other useful views: forced choices and optimality certificates.

Sometimes an action is unavoidable in every feasible completion, or exchange restrictions force the greedy counts, as in our coin example. Another possibility is to derive a bound that applies to *all* feasible solutions and show that greedy attains it: an upper bound for maximization, or a lower bound for minimization. Such a bound is an optimality certificate only when both its validity and equality for the greedy solution are established.

Key point: Do not choose a proof label first. Identify what protects the objective, prove that fact, and then use the appropriate combination of exchange, contradiction, induction, or a progress invariant to connect local choices to the complete solution.

6 The Two Smallest Elements: A Greedy Split with Memory

Given $n \geq 2$ elements, return the smallest and second-smallest. This is the problem from HW3, CLRS Exercise 9.1-1. We develop the lecture's divide-and-conquer construction with a greedy split, aiming for at most

$$n + \lceil \lg n \rceil - 2 \text{ key comparisons.}$$

Assume distinct keys and power-of-two n ; other cases are exercises below. Return *element identities*: the combination must recover the minimum's earlier opponent.

The proposed split: winners form the subproblem.

Compare arbitrary pairs. Call each smaller element the *winner* and the larger one its *opponent*. The $n/2$ winners form S : the subproblem is to find its two smallest elements.

Keep the information used by the combination. The split returns

$$\text{Split}(A) = (S, P), \quad P[\text{winner}] = \text{its opponent in this split.}$$

Recurse only on S ; retain P in the current call for combination.

Use parallel arrays: $S[j]$ stores a winner and $L[j]$ its opponent. Recursion returns the positions of its answers in S , allowing constant-time opponent lookup in L . Each call retains its own pairings; deeper calls must not overwrite them.

Step 1: inspect the desired solution. Let m and s be the smallest and second-smallest elements of A . The minimum m wins its pair, so $m \in S$. There are two possibilities for s :

- If s wins its pair, it belongs to S and is the second-smallest element of S .
- If s loses its pair, its opponent must be m : no other element is smaller than s .

Thus, if recursion returns (m_S, s_S) , the combination is

$$\text{Sol}(A) = \text{Combine}((m_S, s_S), P) = (m_S, \min\{s_S, P[m_S]\}).$$

Step 2: the pairwise choice is safe before recursion. For a losing element x whose winner w is not m , we have $m < w < x$. Two elements already precede x , so x cannot be second-smallest. We can commit to the winners as the recursive input immediately. We retain every winner's opponent because we do not yet know which winner will become m .

Key point: The greedy split selects the smaller element of each pair for recursion. The combination still needs information from that decision: $\text{Split}(A) = (S, P)$, then solve S , then combine its solution with the saved P .

Worked example: the pairings belong to their recursion level.

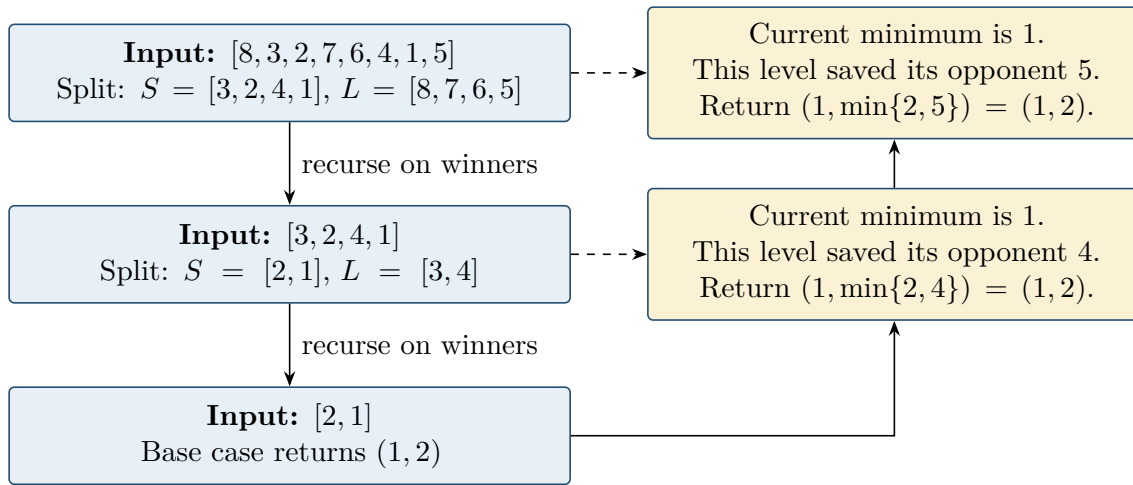
Take

$$A = [8, 3, 2, 7, 6, 4, 1, 5].$$

The first four comparisons give the pairs and retained records below. The columns stay associated: a winner and the entry beneath it came from the same pair.

Pair	(8, 3)	(2, 7)	(6, 4)	(1, 5)
Winners S	3	2	4	1
Saved opponents L	8	7	6	5

Recurse on $[3, 2, 4, 1]$. Its own split compares $(3, 2)$ and $(4, 1)$, creating winners $[2, 1]$ and saved opponents $[3, 4]$. The two-element base case compares 2 with 1 and returns minimum 1, second minimum 2.



Why a single overwritten partner field would fail. Element 1 faces 5 in the outer split, 4 in the next split, and 2 in the base case. These are different comparisons serving different calls. When unwinding, the outer call needs 5, regardless of the pairings made deeper in recursion. Local arrays or separate records for each level preserve that information.

Comparison count for this input.

$$\underbrace{4 + 2}_{\text{pair splits}} + \underbrace{1}_{\text{base case}} + \underbrace{1 + 1}_{\text{combinations}} = 9 = 8 + \lg 8 - 2.$$

The second-smallest must be among the opponents that lost directly to the minimum. Here they are 2, 4, 5; the combination steps find their minimum as recursion returns. This is the tournament interpretation of the same construction.

Implementation outline: preserve the pairs.

Keep winners in S and their positions in the current input A in W . Store the corresponding opponents' positions in L . Each call retains its own arrays. Recursion returns positions in S ; translate them back using W .

Algorithm outline: Two-Min-Indices(A)

Distinct keys; power-of-two input size. Return two indices into A .

- 1: Handle the smallest valid input directly. ▷ TODO: base case
 - 2: Create local arrays S, W, L .
 - 3: **for** each pair of input elements **do**
 - 4: Compare the pair; append the winner to S .
 - 5: Save the winner's and opponent's positions in matching entries of W, L .
 - 6: **end for**
 - 7: $(u, v) \leftarrow \text{TWO-MIN-INDICES}(S)$ ▷ minimum and second minimum in S
 - 8: Recover their positions in A using W . ▷ TODO: index translation
 - 9: Retrieve the minimum's opponent using this call's L .
 - 10: Compare that opponent with the recursive second minimum.
 - 11: Return the minimum and the smaller of those two candidates, as indices into A .
-

Implementation exercise. Complete the base case and index bookkeeping. Recover the saved opponent in constant time, without searching. Check the array accesses against the worked example.

Comparison count: complete the induction.

Let $T(n)$ count key comparisons. For power-of-two $n \geq 4$,

$$T(n) = \underbrace{n/2}_{\text{pair comparisons}} + \underbrace{T(n/2)}_{\text{recursion}} + \underbrace{1}_{\text{combination}}.$$

The induction step to establish is

$$T(n/2) \leq n/2 + \lg(n/2) - 2 \implies T(n) \leq n + \lg n - 2.$$

Exercise: complete the proof, including the base case. Substitute the induction hypothesis into the recurrence and justify the implication. Choose the base case to match the algorithm and verify its exact comparison count.

Key point: This is an exact bound, not an asymptote. $T(n) = \Theta(n)$ does not establish $n + \lg n - 2$: the base case is needed to justify the constants.

Further questions.

1. Does odd n require special treatment? Specify what happens to an unpaired element, revisit the combination, and prove the bound $n + \lceil \lg n \rceil - 2$ for arbitrary $n \geq 2$.
2. What changes if keys can tie? Distinguish the two smallest *elements* from the two smallest *distinct values*; specify how identities are preserved.
3. Complete the correctness induction. Account separately for array construction and storage.

7 Moving on a Line

Frog jumps: go as far as the current jump permits.

The allowed landing positions are

$$0 = x_0 < x_1 < \cdots < x_n = L.$$

A forward jump has length at most $R > 0$. Find a path from 0 to L minimizing the number of jumps, or report that none exists. Both endpoints are allowed landing positions.

Step 1. After fixing the first landing position x_j , the remaining jumps must form a minimum-jump path from x_j to L . A better continuation would improve the whole path. The state needs the current location; there is no hidden dependence on the earlier path because the jump limit is the same everywhere.

Greedy decision. Jump to the farthest allowed position within distance R . This is based on positions alone, without solving a shortest-path subproblem for every possible landing point.

Broad intuition. Moving farther on the current jump appears to leave less distance for the continuation. That intuition suggests the rule; it is not by itself a correctness proof. The connection to our framework is the timing: select a landing position first, rather than solve every possible continuation and compare their lengths.

Homework connection. The water-stop problem has the same broad idea: travel with limited range between replenishments and aim to minimize stops. Because that problem is assigned in HW4, its precise algorithm, implementation analysis, and correctness proof are left to the homework. Different refill costs or location-dependent ranges would require a fresh analysis.

8 Huffman Coding: A Greedy Merge Creates a Subproblem

This is a different style of example from our three core problems. The solution is a tree, the objective is a weighted sum of depths, and the structural step contracts a sibling pair. The extra mathematical setup supports the same Step 1 / greedy-choice framework.

Symbols $c_1, \dots, c_n$ have positive probabilities $p_1, \dots, p_n$ summing to 1. A binary *prefix-free* code gives each symbol a bit string such that no codeword is a prefix of another. This permits immediate decoding of a stream: reach a leaf, output its symbol, then restart at the root.

Represent a code by a binary tree with symbols at leaves and edge labels 0 and 1. The codeword length is the leaf depth ℓ_i . Minimize

$$B(T) = \sum_i p_i \ell_i,$$

the expected bits per symbol. Frequencies may be used instead of probabilities; scaling all weights by the same positive constant does not change the optimum. We assume $n \geq 2$; a single-symbol alphabet can be handled separately by an agreed encoding convention.

An optimal tree can be taken to be *full*: every internal node has two children. If an internal node has only one child, bypassing it shortens the codewords below it while preserving the prefix property. With positive weights this improves cost.

Step 1: fix a sibling pair and contract it.

Suppose two symbols a, b are siblings in a given optimal full tree. Contract their parent into a single pseudo-symbol z with weight

$$p_z = p_a + p_b.$$

If T' is the contracted tree, then

$$B(T) = B(T') + p_a + p_b.$$

All other symbol depths are unchanged; a, b each have depth one larger than the pseudo-symbol. The contracted tree must be optimal for the smaller alphabet. Otherwise expand z in a better smaller tree and obtain a better original code.

The glue is the reverse operation: replace the leaf z by an internal node with children a, b . This proves the structure *conditional on a sibling pair*. We still need to choose a safe pair without solving all candidate smaller coding problems.

Step 2: the two least weights can be siblings.

Let a, b be two minimum-weight symbols. Start with an optimal full tree. Choose a leaf at maximum depth. Its sibling must also be a leaf: an internal sibling would have still deeper descendants. Thus there is a sibling pair at maximum depth h .

Move a into one of these positions by exchanging its label with the label x currently there, if necessary. If a already occupies either position, keep it there and use the other for b . In the nontrivial exchange, $p_a \leq p_x$ and the old depth $d_a \leq h$, so the change in cost is

$$\Delta B = p_a h + p_x d_a - (p_a d_a + p_x h) = (p_a - p_x)(h - d_a) \leq 0.$$

Next move b into the other position, without disturbing a . The same inequality holds because b has minimum weight among the remaining labels. These exchanges do not alter the tree or prefix-freeness. Starting from an optimum, they produce another optimum in which a, b are deepest siblings.

Finish the proof. Merge a, b into weight $p_a + p_b$. By Step 1, the corresponding smaller coding problem has an optimal solution whose expansion is optimal for the original alphabet. Induction on the alphabet size proves that repeatedly merging the two minimum weights is optimal. With two symbols, the single merge is the base case.

Key point: The greedy decision is a *bottom-up merge of two least weights*. The remaining instance contains their combined weight as a new symbol. We do not choose the entire left/right split at the root by solving competing coding subproblems.

Priority-queue implementation.

Algorithm: Huffman($p_1, \dots, p_n$)

```

1: create one leaf for each symbol; build a min-heap  $Q$  by weight
2: while  $|Q| > 1$  do
3:    $x \leftarrow \text{EXTRACTMIN}(Q)$ ;  $y \leftarrow \text{EXTRACTMIN}(Q)$ 
4:   create a parent  $z$  with children  $x, y$  and weight  $w_z = w_x + w_y$ 
5:    $\text{INSERT}(Q, z)$ 
6: end while
7: return the remaining tree root

```

There are $n - 1$ merges. Heap construction costs $O(n)$, and each merge costs $O(\lg n)$, giving $O(n \lg n)$ time and $O(n)$ tree/heap storage. A traversal assigns edge bits and recovers the codewords; writing all codewords explicitly also costs their total written length. Swapping left and right children or resolving equal-weight ties differently can change codewords without changing optimality.

The seven-symbol lecture example.

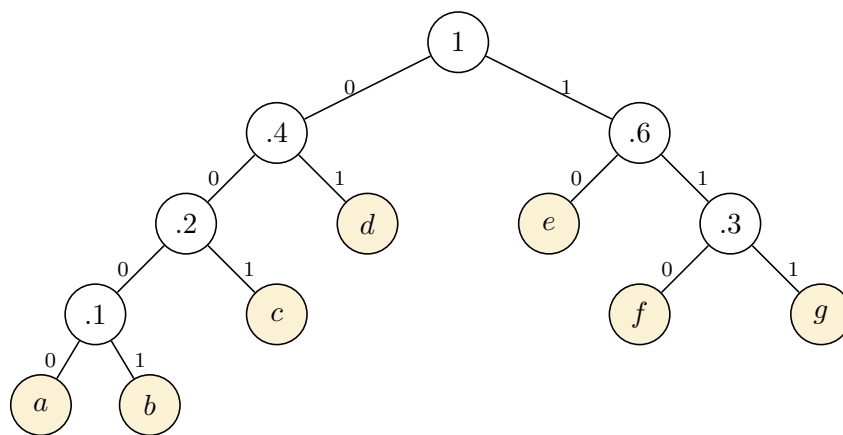
The probabilities are

symbol	a	b	c	d	e	f	g
p	.05	.05	.10	.20	.30	.20	.10

One valid sequence of minimum-weight merges, choosing among ties as needed, is

$$\begin{aligned}
 a + b &\longrightarrow ab \quad (.10), \\
 ab + c &\longrightarrow abc \quad (.20), \\
 g + f &\longrightarrow fg \quad (.30), \\
 abc + d &\longrightarrow abcd \quad (.40), \\
 e + fg &\longrightarrow efg \quad (.60), \\
 abcd + efg &\longrightarrow \text{root} \quad (1.00).
 \end{aligned}$$

The queue includes combined nodes, not only original symbols. The resulting tree can be drawn as follows; put 0 on every left edge and 1 on every right edge.



Symbol	a	b	c	d	e	f	g
Code	0000	0001	001	01	10	110	111
Length	4	4	3	2	2	3	3

Thus

$$B(T) = .05(4) + .05(4) + .10(3) + .20(2) + .30(2) + .20(3) + .10(3) = 2.6.$$

As a useful independent check, the sum of the newly formed weights is also

$$.10 + .20 + .30 + .40 + .60 + 1.00 = 2.6.$$

Each merge increases the depth of all original symbols below it by one, so the sum of merge weights always equals the weighted external path length.

Second lecture example. For probabilities $1/2, 1/4, 1/8, 1/16, 1/16$, one optimal code is 0, 10, 110, 1110, 1111. Its average length is $15/8$ bits per symbol. The highly unbalanced tree is intentional: a balanced tree would ignore the unequal frequencies.

9 Further Greedy Experiments: What Exactly Is Guaranteed?

9.1 Egyptian fractions: a terminating construction

An Egyptian-fraction representation writes a positive proper rational number as a sum of distinct unit fractions, terms of the form $1/m$. For a remainder a/b with integers $0 < a < b$, choose the largest unit fraction that does not exceed it:

$$\frac{1}{m} \leq \frac{a}{b}, \quad m = \left\lceil \frac{b}{a} \right\rceil.$$

Subtract this term and repeat on the remainder. The structural decomposition is simply

$$\frac{a}{b} = \frac{1}{m} + \left(\frac{a}{b} - \frac{1}{m} \right).$$

For example,

$$\frac{5}{6} = \frac{1}{2} + \frac{1}{3}, \quad \frac{7}{15} = \frac{1}{3} + \frac{2}{15} = \frac{1}{3} + \frac{1}{8} + \frac{1}{120}.$$

Why it terminates. The new remainder is

$$\frac{am - b}{bm}.$$

Since $m - 1 < b/a \leq m$, we get $0 \leq am - b < a$. If the numerator is zero, we stop. Otherwise, after reducing the fraction to lowest terms, its positive numerator is still strictly smaller than a . A strictly decreasing sequence of positive integers cannot continue forever.

Why denominators are distinct. From $a/b < 1/(m - 1)$,

$$0 \leq \frac{a}{b} - \frac{1}{m} < \frac{1}{m - 1} - \frac{1}{m} = \frac{1}{m(m - 1)} \leq \frac{1}{m}.$$

If the remainder is nonzero, its next greedy unit fraction is strictly smaller than $1/m$, so the next denominator is larger than m . Repeat this argument. For an improper positive rational, first separate its integer part; the assertion here concerns the proper fractional part.

What this does not prove. Termination and distinctness do not imply a minimum number of terms. The lecture example

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

has a three-term representation, whereas greedy starts with denominators 25, 757, 763309 and requires two more terms. Its later denominators are enormous. The algorithm is a successful representation procedure, but not a minimum-term algorithm. Nor is its bit complexity captured by merely counting the iterations: exact rational numerators and denominators can grow rapidly.

If we require the fewest fractions. This becomes a *discrete number-theoretic optimization problem*: minimize k subject to

$$\frac{a}{b} = \sum_{i=1}^k \frac{1}{d_i}, \quad d_1 < d_2 < \cdots < d_k, \quad d_i \in \mathbb{Z}_{>0}.$$

Its decision version asks whether a representation with at most k terms exists. A finite exact search is possible: if a positive remainder r uses at most k terms, its first (smallest) denominator satisfies

$$\lceil 1/r \rceil \leq d_1 \leq \lfloor k/r \rfloor.$$

Try those denominators and recurse on the remainder, requiring subsequent denominators to increase. Testing successive term limits finds a shortest representation; the greedy construction supplies a finite upper limit on the number of terms.

This search can be expensive, with large denominators and many branches. It is not a polynomial-time guarantee in the binary input length, nor is it an NP-hardness proof. Finding *some* representation and proving *minimum length* are very different tasks; the latter is beyond the greedy construction studied here. See Eppstein's short-sequence search for an exact-search approach.

9.2 Equalizing three stacks: remove from a largest sum

Each of three stacks contains positive integers. A move removes the top element of one stack. Find the largest common sum that can remain in all three stacks; empty stacks of sum zero are allowed. Positivity is important because every removal strictly decreases the corresponding sum.

Let the current sums be S_1, S_2, S_3 . If they are equal, return that value. Otherwise pop the top of any stack having maximum current sum and update that sum. Repeat.

Step 1. Once a top element is removed, the remaining three suffixes form another instance of exactly the same problem. Any solution reachable after that removal can be obtained in the original instance by performing the removal first.

Why the choice is safe. Suppose the sums are not all equal. Any achievable common sum H must satisfy

$$H \leq \min\{S_1, S_2, S_3\}.$$

A largest-sum stack has sum strictly above this minimum. It cannot be left untouched in any equal-sum solution: its top element must eventually be removed, and no deeper element can be removed first. Removing its top now therefore preserves *every* achievable common sum. The maximum achievable common sum is invariant throughout the algorithm. When the sums become

equal, that common sum is itself achievable and is the largest possible in the current state, hence also in the original state.

Lecture example, top to bottom:

$$A = [3, 2, 1, 1, 1], \quad B = [4, 3, 2], \quad C = [2, 5, 4, 1].$$

The sums and removals are

$$(8, 9, 12) \xrightarrow{C:2} (8, 9, 10) \xrightarrow{C:5} (8, 9, 5) \xrightarrow{B:4} (8, 5, 5) \xrightarrow{A:3} (5, 5, 5).$$

Compute initial sums once and keep one top index per stack. Each element is popped at most once, giving $O(n_1 + n_2 + n_3)$ time and $O(1)$ extra state beyond the input. If negative entries were allowed, sums could increase when popping and this proof would fail.

9.3 Cutting a rectangle into squares: a heuristic, not a theorem

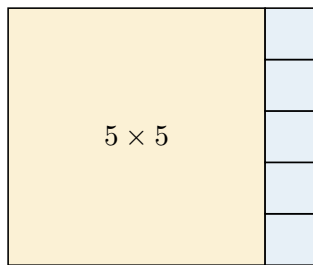
For a rectangle of side lengths $a \leq b$, repeatedly remove a largest axis-aligned square, of side a , and continue on the remaining rectangle. Taking $q = \lfloor b/a \rfloor$ such squares leaves dimensions a by $b \bmod a$. This is the Euclidean-algorithm pattern.

For the lecture's 13×29 rectangle:

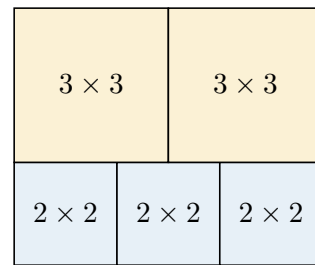
$$29 = 2 \cdot 13 + 3, \quad 13 = 4 \cdot 3 + 1, \quad 3 = 3 \cdot 1 + 0.$$

The construction uses two 13×13 squares, four 3×3 squares, and three 1×1 squares: nine squares in all. This is a valid construction. The calculation alone does *not* prove a minimum number of squares.

In fact, largest-square-first is not generally minimum-cardinality. On a 5×6 rectangle, it produces one 5×5 square and five 1×1 squares: six squares. The alternative below uses only five.



Largest first: six squares



A better tiling: five squares

The greedy cut gives a subproblem, but we have not proved that some optimal tiling contains that cut. This is another failure at Step 2. If we later use DP over full horizontal or vertical cuts, that recurrence must be analyzed for the *guillotine-cut* model; arbitrary square tilings need not admit such a first cut. The decomposition itself must match the allowed solutions.

10 Design Checklist and the Transition to DP

For a new optimization problem, work through the following questions in order.

1. **State the objective and feasible solutions.** Is the goal cardinality, value, total length, number of moves, or merely constructing a representation?
2. **Step 1: inspect a given optimal solution.** Identify a decision and its residual subproblem(s). State the glue. Explain why replacing a residual solution by a better one would improve the whole without violating constraints.
3. **Propose a criterion.** Does it choose a decision using the current instance, without solving the competing subproblems? Specify ties and edge cases.
4. **Try to break it.** Small counterexamples often reveal whether the criterion optimizes the wrong quantity. One counterexample refutes a universal rule; many successful examples do not prove it.
5. **Prove safety.** Give an exchange, a staying-ahead invariant, or another precise argument showing that an optimal solution remains possible after the choice. Then finish with the structural argument and induction.
6. **If the choice is not justified, keep the alternatives.** A correct structural recurrence may lead to DP. The state and coverage still need justification; failure to find a greedy proof is not a theorem that no greedy algorithm exists.
7. **Analyze the implementation.** Sorting, scanning, selection, or a heap can make a large difference even when the mathematical criterion is unchanged. Return the requested solution as well as its value when needed.

One view of the main examples.

Problem	Decision and residual subproblem	Why commit, or why not?
Fractional knapsack	Fix an amount; fill residual capacity from the other goods.	Maximum quality is safe by equal-weight exchange.
Activity selection	Choose the first activity; schedule after its finish.	Earliest finish preserves every continuation of the replaced activity.
Frog jumps	Choose next landing; finish the trip from there.	Farthest reachable is the broad greedy idea; the related water-stop proof is left to HW4.
Coin change	Choose a coin; make the remaining amount.	Largest coin needs a denomination-specific proof; otherwise compare alternatives.
Huffman coding	Merge siblings; solve the contracted alphabet.	Two least weights can be placed as deepest siblings without increasing cost.

10.1 A bridge exercise: longest increasing subsequence

An increasing subsequence must preserve input order; it need not occupy consecutive positions. The objective is to find one of maximum length.

For HW4, investigate whether you can justify a greedy approach using the framework from this lecture. If you cannot find a correct greedy solution, explain intuitively why and leave the DP construction for HW5. Specific proposed rules, counterexamples, and the explanation are left to the assignment.

10.2 Looking ahead: minimum spanning trees

The older notes finish by introducing a minimum spanning tree: for a connected undirected graph with edge weights, choose edges that connect all vertices, contain no cycle, and have minimum total weight. This is a preview of later graph algorithms. A rule such as choosing a light edge will need a precise safety condition; the word “light” alone is not a proof. The same discipline applies: identify the preserved structure, then prove the greedy choice.

Key point: First check that the solution is structurally obtainable from subproblem solutions. Then ask whether a criterion lets us decide before solving them. Greedy earns that early commitment with a proof; DP retains the feasible alternatives and compares their solved values.