Chapter 1: Decision Theory

Algorithmic Game Theory

Expected utility | Utility scales | Risk attitudes | Decision criteria under uncertainty | Bayesian decisions | Decision trees | Value of information

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/agt/slides/Chapter_01_Decision_Theory.html

Warm-up Game: Bayesian Decision Lab

The rules

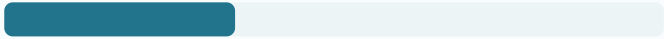
You know the prior of a hidden state, see a noisy signal and choose one of two actions; then the state is revealed and you receive the payoff in the table.

Think about it

- A forecast that is right 90% of the time says "rain". What is the probability of rain?
- If a decision leads to a bad outcome, was the decision bad?

	Rain	Dry
Outdoors	−6	8
Indoors	4	3

Prior: P(rain) = 0.35



Signal: forecast, 90% accurate (Level 1)

Forecast: "rain"

Bayesian inference →

Learning Outcomes

Expected utility and risk

Compute the expected value and expected utility of a lottery; interpret risk attitudes via certainty equivalents and risk premia; apply the maximin, Laplace and Savage criteria when probabilities are unknown.

Utility scales

Explain the vNM axioms and the Allais and Ellsberg choices; show which transformations of utility numbers preserve decisions.

Bayesian decisions

Update a prior with a test result; show base-rate neglect in a numerical example; choose the action with the smallest expected loss.

Decision trees

Find the optimal strategy by backward induction; compute the value of perfect and sample information.

Decision Making: Three Kinds

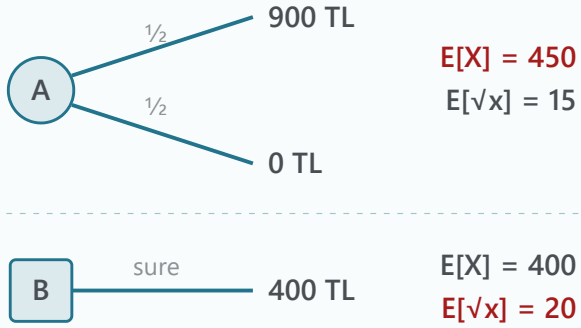
Decision making is the process of **choosing** among options by evaluating their consequences. Three kinds, by what drives the outcome:

Kind	Outcome driven by	Tools	Example
Individual, under risk	Nature / chance (probabilities known or estimated)	Expected utility, Bayes, decision trees	Investment, insurance, medical tests
Strategic	Choices of other rational agents	Best response, Nash equilibrium	Price competition, bargaining, sports
Collective	Joint choice and interaction of a group	Voting, common goods	Policy, corporate strategy

In an individual decision the decision maker plays only against nature (chance). Expected utility, Bayes' rule and decision trees are also the building blocks of strategic and collective decisions.

Rational Choice: Expected Value or Expected Utility?

- **Lottery**: a set of outcomes with probabilities. A = 900 TL with probability 50%, 0 TL with probability 50%.
- **Expected value**: $E[X] = \Sigma\, p_i\, x_i$. $E[A] = ½\cdot 900 + ½\cdot 0 =$ **450** TL > a sure B = 400 TL.
- **Expected utility** (von Neumann–Morgenstern): $E[u] = \Sigma\, p_i\, u(x_i)$. A rational decision maker maximizes expected *utility*.
- With $u(x) = \sqrt{x}$: $E[u(A)] = ½\cdot 30 + ½\cdot 0 =$ **15**; $u(B) = \sqrt{400} =$ **20** ⇒ B is chosen.



Red: each criterion's choice. Expected value picks A, expected utility (√x) picks B.

Expected monetary value and expected utility **may disagree**; the gap is driven by the curvature of the utility function, i.e. the risk attitude.

St. Petersburg paradox: nobody pays much for a game with infinite expected value; this is why Daniel Bernoulli (1738) proposed utility (a similar idea: Cramer, 1728).

Foundations of Expected Utility and the Allais Paradox

- **von Neumann–Morgenstern**: if preferences satisfy four axioms, some u exists and the decision maker maximizes expected utility. $A \succ B$: A preferred to B.
- **Completeness**: any two options are comparable. **Transitivity**: $A \succ B$ and $B \succ C$ imply $A \succ C$. **Continuity**: a middle option equals some mix of the extremes.
- **Independence**: mixing the same component (same probability) into two lotteries keeps their ranking.
- **Allais (1953)**: most people pick A in choice 1, D in choice 2.

	Option	1 million	5 million	0
Choice 1	A	100%	—	—
	B	89%	10%	1%
Choice 2	C	11%	—	89%
	D	—	10%	90%

Swap the common 89% "1 million" in A and B for 89% "0": C and D.

For every u, $E[u(A)] - E[u(B)] = 0.11 \cdot u(1) - 0.10 \cdot u(5) - 0.01 \cdot u(0) = E[u(C)] - E[u(D)]$. So $A \succ B$ forces $C \succ D$; "A and D" fits **no** utility function. Overweighting certainty (the certainty effect) is a starting point of prospect theory.

What Do Utility Numbers Measure? Uniqueness of the Scale

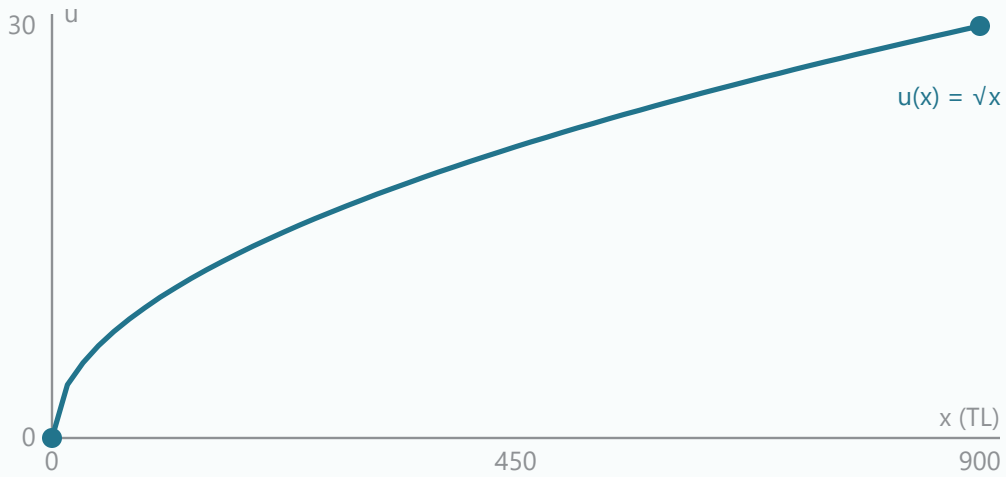
- Utility is like a thermometer: its zero and its unit are arbitrary. Celsius and Fahrenheit rank temperatures the same way.
- **Uniqueness** (vNM): a transformation $v = a \cdot u + b$ ($a > 0$) is called an *affine transformation*: it rescales and shifts, but changes no decision between lotteries. Other (non-affine) transformations can change decisions.
- **Link to game theory**: multiplying all of a player's payoffs by 2 and adding 5 changes none of its decisions.

Utility	lottery A	sure B	choice
$u = \sqrt{x}$	15	20	B
$v = 10\sqrt{x} + 5$	155	205	B
$w = (\sqrt{x})^2 = x$	450	400	A

A: 900 TL with probability ½, 0 TL with probability ½; B: 400 TL for sure. The affine v preserves the decision; the increasing but non-affine w reverses it.

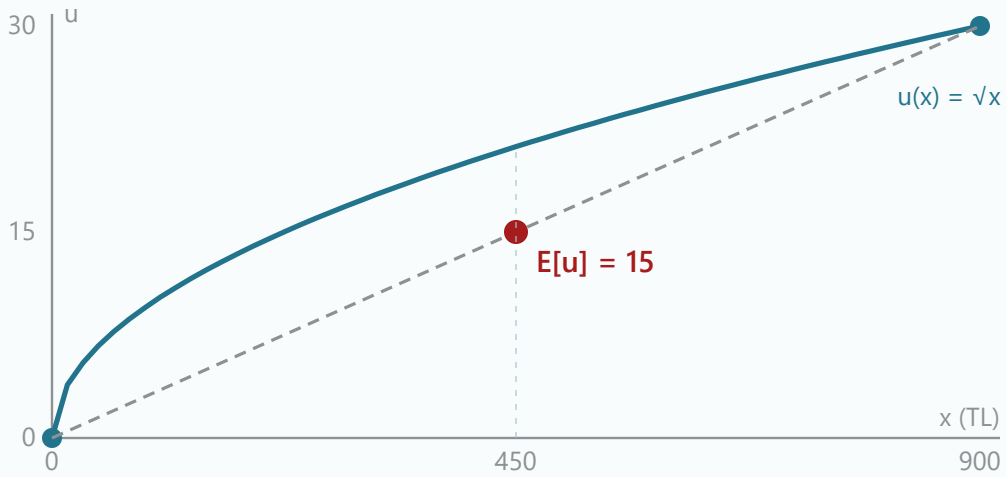
Common mistake: saying "u(B) − u(A) = 5 units of happiness" or "Ayşe's utility is 20 and Bora's 15, so Ayşe is happier". The size of differences and comparisons across people depend on the choice of scale; what is meaningful is one person's ranking of options.

Certainty Equivalent and Risk Premium, Step by Step



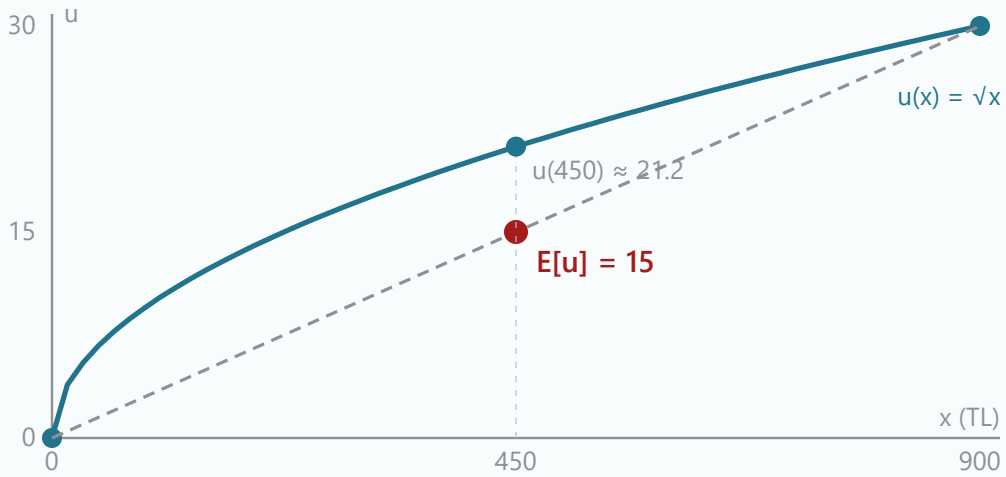
Concave utility u(x) = √x: each extra lira adds less utility than the one before (diminishing marginal utility). Endpoints: u(0) = 0, u(900) = 30.

Certainty Equivalent and Risk Premium, Step by Step



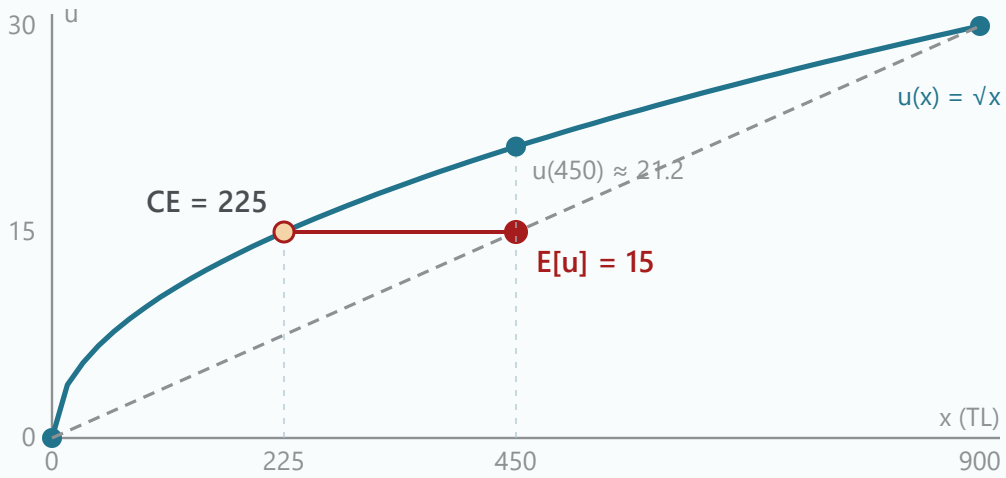
Chord: the line joining the two ends of the curve. Lottery A's expected utility is the chord's midpoint: E[u] = ½·0 + ½·30 = **15** (at x = 450).

Certainty Equivalent and Risk Premium, Step by Step



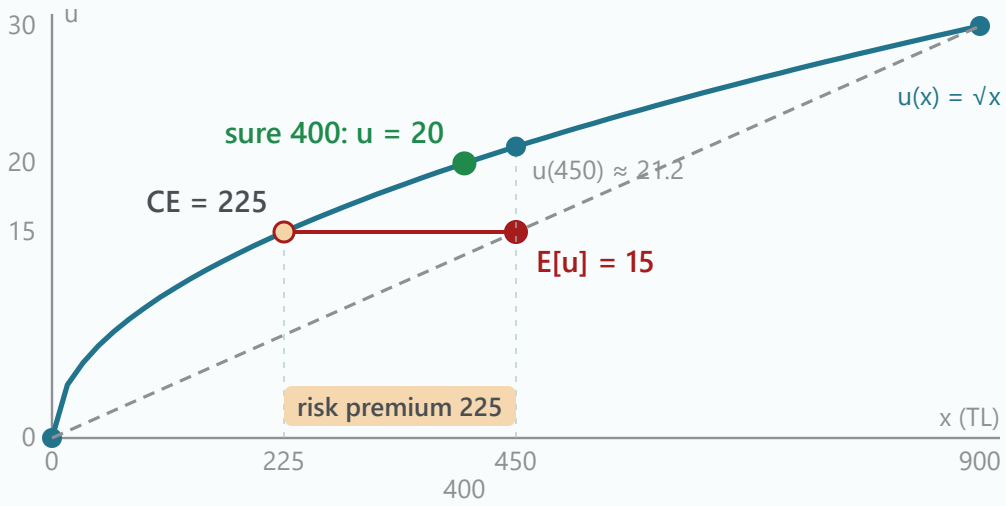
The same 450 TL for sure gives u(450) ≈ 21.2 > 15. On a concave curve the chord lies below the curve, hence E[u(X)] < u(E[X]) (Jensen's inequality).

Certainty Equivalent and Risk Premium, Step by Step



Certainty equivalent (CE): the sure amount with utility 15. √x = 15 ⇒ **CE = 225 TL**. This person would sell the lottery for 225 TL.

Certainty Equivalent and Risk Premium, Step by Step



Risk premium = E[X] − CE = 450 − 225 = **225 TL**. A sure 400 TL has utility 20 > 15 ⇒ a risk-averse person chooses **B**.

Three Risk Attitudes: Same Lottery, Different Decisions

Attitude	Utility	E[u(A)]	u(400)	Choice	CE (TL)	Risk premium
Risk-averse	$u(x) = \sqrt{x}$ (concave)	15	20	B	225	225
Risk-neutral	$u(x) = x$ (linear)	450	400	A	450	0
Risk-seeking	$u(x) = x^2/900$ (convex)	450	177.8	A	636.4	−186.4

Concave → averse
CE < E[X], premium > 0: demands a reward for bearing risk. Most people, for large amounts.

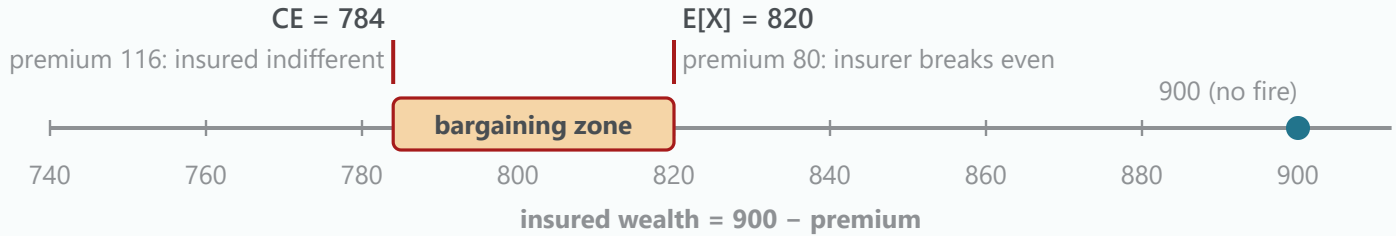
Linear → neutral
CE = E[X]: looks only at expected value. Large firms, small amounts.

Convex → seeking
CE > E[X], premium < 0: pays for risk. Someone buying a lottery ticket.

Lottery A: 900 TL with probability 50%, 0 TL with probability 50% (E[X] = 450). B: 400 TL for sure. CE = $u^{-1}(E[u(A)])$: u^{-1} inverts u; for $\sqrt{x}$, $u^{-1}(15) = 15^2 = 225$. Risk premium = E[X] − CE.

Application: Why Does Insurance Exist?

- Wealth 900 (thousand TL); a loss of 800 (fire) with probability 10%. Utility $u(x) = \sqrt{x}$.
- Uninsured: $E[u] = 0.9 \cdot \sqrt{900} + 0.1 \cdot \sqrt{100} = 27 + 1 = \mathbf{28}$.
- Certainty equivalent: $28^2 = \mathbf{784}$. The person accepts a sure 784 in place of a risky 900.
- Maximum premium = $900 - 784 = \mathbf{116}$; the expected loss is only $0.1 \cdot 800 = 80$.



Insured: paying a premium ≤ 116 to shed the risk raises utility. Insurer: with many independent policies it is nearly risk-neutral; any premium above 80 is profit.

Any premium between 80 and 116 **benefits both sides**: risk moves to the party best able to bear it.

Known Risk or Unknown Probabilities? The Ellsberg Urn

- Under risk the probabilities are known (dice, roulette); under uncertainty they are not. People treat the two differently.
- **Urn**: 90 balls; 30 are red, the other 60 are black or yellow, but how many are black is unknown. Drawing the winning color pays 100 TL.
- **Typical choices**: "red" in bet 1, "black or yellow" in bet 2; in both, the option with a *known* winning probability.

Bet	Option	winning balls
1	red	30 (known)
	black	b (0–60, unknown)
2	red or yellow	30 + (60 − b)
	black or yellow	60 (known)

b: number of black balls in the urn. Orange: the majority choice.

Choice 1 means 30 > b, choice 2 means 60 > 90 − b, i.e. b > 30: **no probability** explains both choices (Ellsberg, 1961). Avoiding options with unknown probabilities is called **ambiguity aversion**; when probabilities are entirely unknown, choosing to make the worst case as good as possible (maximin) is its extreme form.

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad
Big investment	200	50	−120
Small investment	90	40	−20
No investment	0	0	0

Probabilities are **unknown** (uncertainty). Rows are actions, columns are states; each criterion reflects a different degree of "pessimism".

Step 1 / 9

```
1  def criteria(U):
▶      A, m = range(len(U)), len(U[0])
3      mm = max(A, key=lambda i: min(U[i]))      # maximin
4      mx = max(A, key=lambda i: max(U[i]))      # maximax
5      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6      best = [max(r[j] for r in U) for j in range(m)]
7      reg = [max(best[j] - r[j] for j in range(m))
8             for r in U]            # max regret
9      sv = reg.index(min(reg))                 # Savage
10     return mm, mx, lp, sv
```

VARIABLES

`U = 3×3 table` `m = 3`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min
Big investment	200	50	−120	−120
Small investment	90	40	−20	−20
No investment	0	0	0	0

Maximin (Wald, plan for the worst): row minima −120, −20, 0 → the largest is 0: **No investment**.

Step 2 / 9

```
1  def criteria(U):
2      A, m = range(len(U)), len(U[0])
▶      mm = max(A, key=lambda i: min(U[i]))      # maximin
4      mx = max(A, key=lambda i: max(U[i]))      # maximax
5      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6      best = [max(r[j] for r in U) for j in range(m)]
7      reg = [max(best[j] - r[j] for j in range(m))
8             for r in U]            # max regret
9      sv = reg.index(min(reg))                  # Savage
10     return mm, mx, lp, sv
```

VARIABLES

U = 3×3 table

mm = 2 (None)

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max
Big investment	200	50	−120	−120	200
Small investment	90	40	−20	−20	90
No investment	0	0	0	0	0

Maximax (optimistic): row maxima 200, 90, 0 → **Big investment**.

Step 3 / 9

```
1  def criteria(U):
2      A, m = range(len(U)), len(U[0])
3      mm = max(A, key=lambda i: min(U[i]))      # maximin
▶      mx = max(A, key=lambda i: max(U[i]))      # maximax
5      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6      best = [max(r[j] for r in U) for j in range(m)]
7      reg = [max(best[j] - r[j] for j in range(m))
8             for r in U]            # max regret
9      sv = reg.index(min(reg))                  # Savage
10     return mm, mx, lp, sv
```

VARIABLES

`U = 3×3 table` `mm = 2` `mx = 0 (Big)`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.
Big investment	200	50	−120	−120	200	43.3
Small investment	90	40	−20	−20	90	36.7
No investment	0	0	0	0	0	0

Laplace (treat states as equally likely): averages 43.3, 36.7, 0.0 → **Big investment**.

Step 4 / 9

```
1  def criteria(U):
2      A, m = range(len(U)), len(U[0])
3      mm = max(A, key=lambda i: min(U[i]))      # maximin
4      mx = max(A, key=lambda i: max(U[i]))      # maximax
▶      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6      best = [max(r[j] for r in U) for j in range(m)]
7      reg = [max(best[j] - r[j] for j in range(m))
8             for r in U]              # max regret
9      sv = reg.index(min(reg))                  # Savage
10     return mm, mx, lp, sv
```

VARIABLES

`U = 3×3 table` `mm = 2` `mx = 0` `lp = 0 (Big)`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.
Big investment	200	50	−120	−120	200	43.3
Small investment	90	40	−20	−20	90	36.7
No investment	0	0	0	0	0	0
column best:	**200**	**50**	**0**			

Best achievable result in each state: best = [200, 50, 0]. Regret = best − achieved.

Step 5 / 9

```
1  def criteria(U):
2      A, m = range(len(U)), len(U[0])
3      mm = max(A, key=lambda i: min(U[i]))      # maximin
4      mx = max(A, key=lambda i: max(U[i]))      # maximax
5      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
▶      best = [max(r[j] for r in U) for j in range(m)]
7      reg = [max(best[j] - r[j] for j in range(m))
8             for r in U]             # max regret
9      sv = reg.index(min(reg))                  # Savage
10     return mm, mx, lp, sv
```

VARIABLES

U = 3×3 table | mm = 2 | mx = 0 | lp = 0 | best = [200, 50, 0]

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.	max regret
Big investment	200	50	−120	−120	200	43.3	120
Small investment	90	40	−20	−20	90	36.7	110
No investment	0	0	0	0	0	0	200
column best:	**200**	**50**	**0**				

Largest regret per row: Big (bad state, 0 − (−120)) = 120, Small (good state, 200 − 90) = 110, None (good state, 200) = 200.

Step 6 / 9

```
1  def criteria(U):
2      A, m = range(len(U)), len(U[0])
3      mm = max(A, key=lambda i: min(U[i]))      # maximin
4      mx = max(A, key=lambda i: max(U[i]))      # maximax
5      lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6      best = [max(r[j] for r in U) for j in range(m)]
▶      reg = [max(best[j] - r[j] for j in range(m))
▶             for r in U]            # max regret
9      sv = reg.index(min(reg))                  # Savage
10     return mm, mx, lp, sv
```

VARIABLES

`U = 3×3 table` `mm = 2` `mx = 0` `lp = 0` `best = [200, 50, 0]` `reg = [120, 110, 200]`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.	max regret
Big investment	200	50	−120	−120	200	43.3	120
Small investment	90	40	−20	−20	90	36.7	110
No investment	0	0	0	0	0	0	200
column best:	200	50	0				

Savage (minimax regret): the smallest maximum regret is 110 → **Small investment**.

Step 7 / 9

```python
def criteria(U):
    A, m = range(len(U)), len(U[0])
    mm = max(A, key=lambda i: min(U[i]))      # maximin
    mx = max(A, key=lambda i: max(U[i]))      # maximax
    lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
    best = [max(r[j] for r in U) for j in range(m)]
    reg = [max(best[j] - r[j] for j in range(m))
           for r in U]                        # max regret
    sv = reg.index(min(reg))                  # Savage
    return mm, mx, lp, sv
```

VARIABLES

`U = 3×3 table` `mm = 2` `mx = 0` `lp = 0` `reg = [120, 110, 200]` `sv = 1 (Small)`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.	max regret
Big investment	200	50	−120	−120	200	43.3	120
Small investment	90	40	−20	−20	90	36.7	110
No investment	0	0	0	0	0	0	200
column best:	**200**	**50**	**0**				

return (2, 0, 0, 1). Returned indices (0: Big, 1: Small, 2: None): maximin → None, maximax and Laplace → Big, Savage → Small. Three different decisions from one table.


```
1 def criteria(U):
2     A, m = range(len(U)), len(U[0])
3     mm = max(A, key=lambda i: min(U[i]))      # maximin
4     mx = max(A, key=lambda i: max(U[i]))      # maximax
5     lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6     best = [max(r[j] for r in U) for j in range(m)]
7     reg = [max(best[j] - r[j] for j in range(m))
8            for r in U]            # max regret
9     sv = reg.index(min(reg))                  # Savage
▶     return mm, mx, lp, sv
```

VARIABLES

`result = (2, 0, 0, 1)`

Unknown Probabilities: Decisions under Uncertainty

action \ state	Good	Fair	Bad	min	max	avg.	max regret
Big investment	200	50	−120	−120	200	43.3	120
Small investment	90	40	−20	−20	90	36.7	110
No investment	0	0	0	0	0	0	200
column best:	**200**	**50**	**0**				

One more criterion, not in the code: **Hurwicz** sits between the extremes: α·(row max) + (1 − α)·(row min); with optimism coefficient α = 0.5: 40, 35, 0 → Big. If probabilities can be estimated, return to expected utility; if not, the choice of criterion reflects the decision maker's attitude.

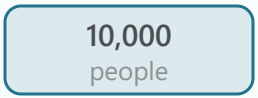
Step 9 / 9

```
1 def criteria(U):
2     A, m = range(len(U)), len(U[0])
3     mm = max(A, key=lambda i: min(U[i]))      # maximin
4     mx = max(A, key=lambda i: max(U[i]))      # maximax
5     lp = max(A, key=lambda i: sum(U[i]) / m)  # Laplace
6     best = [max(r[j] for r in U) for j in range(m)]
7     reg = [max(best[j] - r[j] for j in range(m))
8            for r in U]            # max regret
9     sv = reg.index(min(reg))                  # Savage
▶     return mm, mx, lp, sv
```

VARIABLES

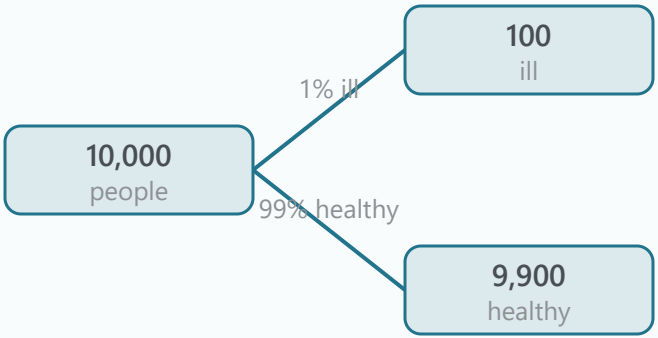
`result = (2, 0, 0, 1)`

Medical Test: What Does a Positive Result Mean?



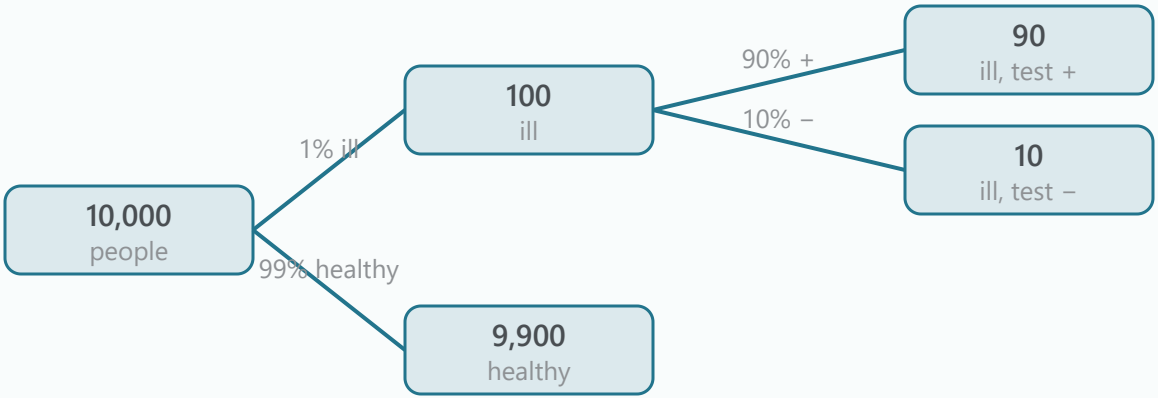
Picture a population of 10,000. Using **natural frequencies** (head counts) instead of probabilities makes Bayesian reasoning intuitive.

Medical Test: What Does a Positive Result Mean?



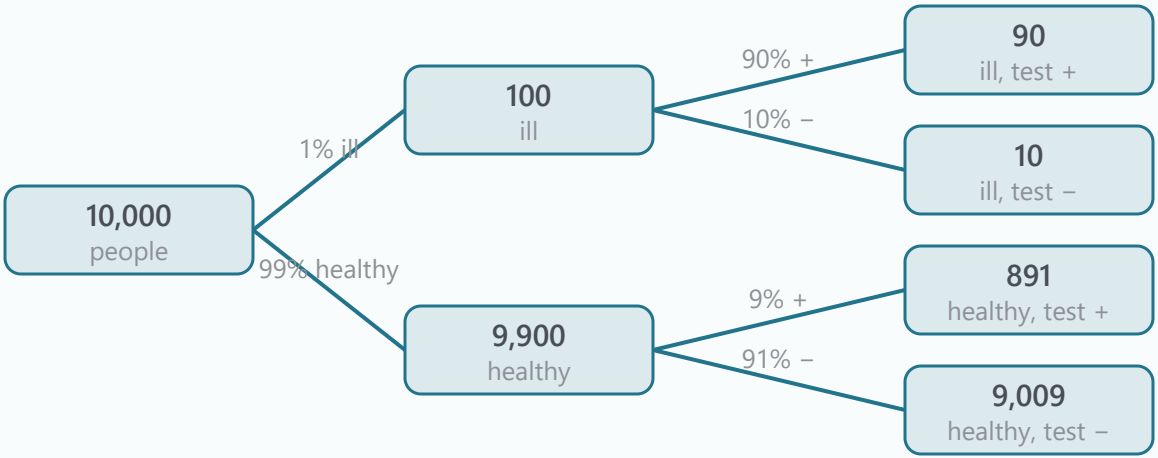
Prior: prevalence 1% ⇒ 100 ill, 9,900 healthy.

Medical Test: What Does a Positive Result Mean?



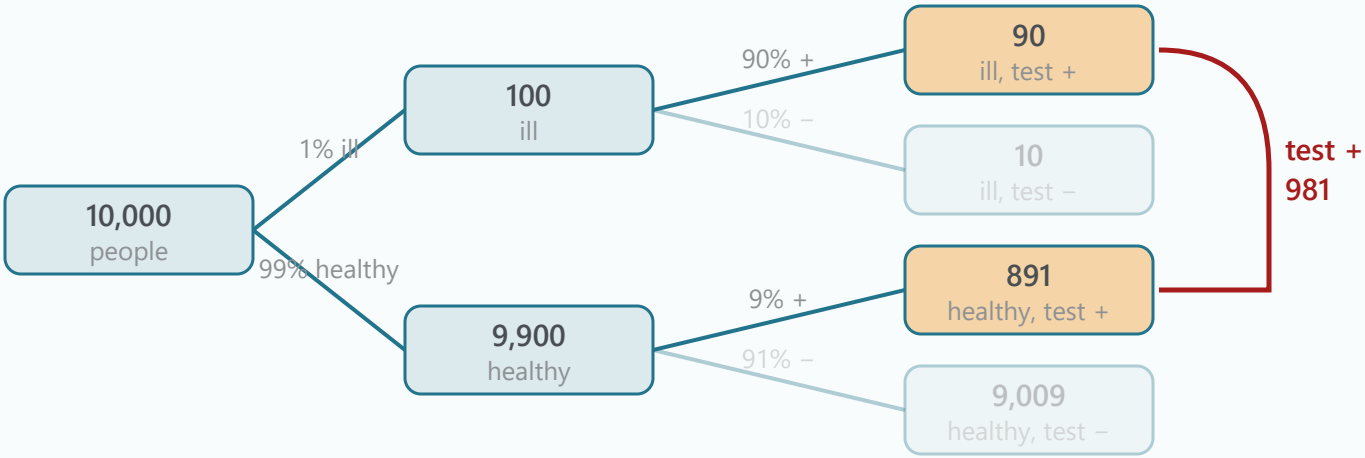
Sensitivity 90%: 90 of the 100 ill people test positive, 10 negative.

Medical Test: What Does a Positive Result Mean?



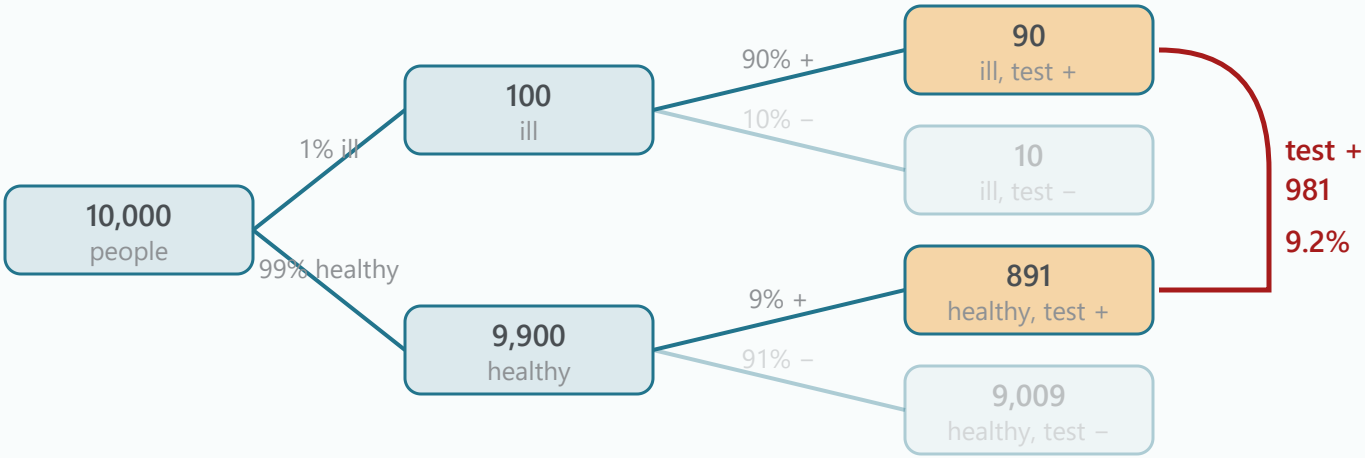
False positive rate 9%: 891 of the 9,900 healthy people also test positive! Healthy people are so numerous that this number is large.

Medical Test: What Does a Positive Result Mean?



Everyone testing positive: 90 + 891 = **981** people. Only 90 of them are actually ill.

Medical Test: What Does a Positive Result Mean?



Posterior: P(ill | +) = 90 / 981 = **9.2%**. Even with 90% sensitivity, a positive result is mostly a false alarm (base-rate neglect).

Code Trace: `bayes(p, sens, fp, tests)`



Call: `bayes(0.01, 0.9, 0.09, ["+", "+", "−"])`. p starts at the prior (1%); the three test results are processed in turn.

Step 1 / 14

```
▶ def bayes(p, sens, fp, tests):
2   for res in tests:
3     if res == "+":
4       num = sens * p
5       den = num + fp * (1 - p)
6     elif res == "-":
7       num = (1 - sens) * p
8       den = num + (1 - fp) * (1 - p)
9     else: raise ValueError("result must be + or -")
10    if den == 0: raise ValueError("impossible result")
11    p = num / den
12    print(res, round(p, 4))
13  return p
```

VARIABLES

`p = 0.0100` `sens = 0.9` `fp = 0.09` `tests = [+, +, −]`

OUTPUT

Code Trace: `bayes(p, sens, fp, tests)`



Test 1 result: +. The current belief p = 0.0100 is this test's prior.

Step 2 / 14

```
1  def bayes(p, sens, fp, tests):
▶    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0100` `sens = 0.9` `fp = 0.09` `res = +`

OUTPUT

Code Trace: `bayes(p, sens, fp, tests)`



res == "+" ? → **True**: use the likelihoods of a positive result. Likelihood: the probability of this result given the disease status, e.g. P(+ | ill) = 0.9.

Step 3 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
▶      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0100` `sens = 0.9`

`fp = 0.09` `res = +`

OUTPUT

Code Trace: `bayes(p, sens, fp, tests)`



num = P(+|ill)·p = 0.9 × 0.0100 = 0.0090; den = num + P(+|healthy)·(1 − p) = 0.0981 (total probability of a positive).

Step 4 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
▶        num = sens * p
▶        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0100` `sens = 0.9` `fp = 0.09` `res = +` `num = 0.0090` `den = 0.0981`

OUTPUT

Code Trace: `bayes(p, sens, fp, tests)`



p = num / den = **0.0917**, printed. Despite the positive test, the probability of disease is still below 10%.

Step 5 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
▶      if den == 0: raise ValueError("impossible result")
▶      p = num / den
▶      print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0917` `sens = 0.9` `fp = 0.09` `res = +` `num = 0.0090` `den = 0.0981`

OUTPUT

```
+ 0.0917
```

Code Trace: `bayes(p, sens, fp, tests)`



Test 2 result: +. The current belief p = 0.0917 is this test's prior.

Step 6 / 14

```
1  def bayes(p, sens, fp, tests):
▶    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0917` `sens = 0.9` `fp = 0.09` `res = +`

OUTPUT

```
+ 0.0917
```

Code Trace: `bayes(p, sens, fp, tests)`



res == "+" ? → **True**: use the likelihoods of a positive result.

Step 7 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
▶      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

VARIABLES

`p = 0.0917` `sens = 0.9` `fp = 0.09` `res = +`

OUTPUT

```
+ 0.0917
```

Code Trace: `bayes(p, sens, fp, tests)`



num = P(+|ill)·p = 0.9 × 0.0917 = 0.0826; den = num + P(+|healthy)·(1 − p) = 0.1643 (total probability of a positive).

Step 8 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
▶        num = sens * p
▶        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

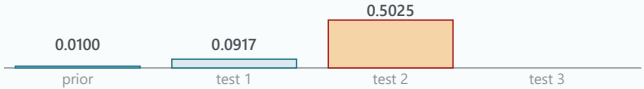
VARIABLES

`p = 0.0917` `sens = 0.9` `fp = 0.09` `res = +` `num = 0.0826` `den = 0.1643`

OUTPUT

```
+ 0.0917
```

Code Trace: `bayes(p, sens, fp, tests)`



p = num / den = **0.5025**, printed. Two positives, independent given the disease status, raised the probability to ~50%.

Step 9 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
▶      if den == 0: raise ValueError("impossible result")
▶      p = num / den
▶      print(res, round(p, 4))
13   return p
```

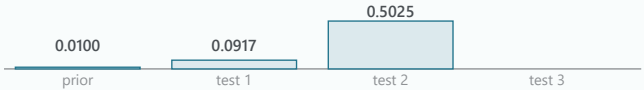
VARIABLES

`p = 0.5025` `sens = 0.9` `fp = 0.09` `res = +` `num = 0.0826` `den = 0.1643`

OUTPUT

```
+ 0.0917
+ 0.5025
```

Code Trace: `bayes(p, sens, fp, tests)`



Test 3 result: **-**. The current belief p = 0.5025 is this test's prior.

Step 10 / 14

```
1  def bayes(p, sens, fp, tests):
▶    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

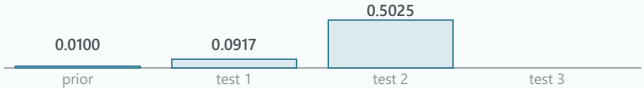
VARIABLES

`p = 0.5025` `sens = 0.9` `fp = 0.09` `res = -`

OUTPUT

```
+ 0.0917
+ 0.5025
```

Code Trace: `bayes(p, sens, fp, tests)`



res == "+" ? → **False**: use the likelihoods of a negative result.

Step 11 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
▶      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
▶      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

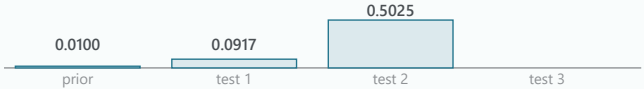
VARIABLES

`p = 0.5025` `sens = 0.9` `fp = 0.09` `res = -`

OUTPUT

```
+ 0.0917
+ 0.5025
```

Code Trace: `bayes(p, sens, fp, tests)`



num = P(−|ill)·p = 0.1 × 0.5025 = 0.0503; den = num + P(−|healthy)·(1 − p) = 0.91 × 0.4975 + num = 0.5030.

Step 12 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
▶        num = (1 - sens) * p
▶        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
13   return p
```

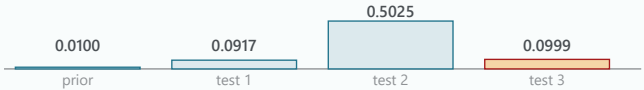
VARIABLES

`p = 0.5025` `sens = 0.9` `fp = 0.09` `res = -` `num = 0.0503` `den = 0.5030`

OUTPUT

```
+ 0.0917
+ 0.5025
```

Code Trace: `bayes(p, sens, fp, tests)`



p = num / den = **0.0999**, printed. The negative result brought the belief back down to 10%.


```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
▶      if den == 0: raise ValueError("impossible result")
▶      p = num / den
▶      print(res, round(p, 4))
13   return p
```

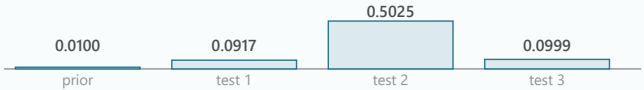
VARIABLES

`p = 0.0999` `sens = 0.9` `fp = 0.09` `res = -` `num = 0.0503` `den = 0.5030`

OUTPUT

```
+ 0.0917
+ 0.5025
- 0.0999
```

Code Trace: `bayes(p, sens, fp, tests)`



The loop ends; the function returns p = 0.0999. Each test uses the previous posterior as its prior (sequential updating).

Step 14 / 14

```
1  def bayes(p, sens, fp, tests):
2    for res in tests:
3      if res == "+":
4        num = sens * p
5        den = num + fp * (1 - p)
6      elif res == "-":
7        num = (1 - sens) * p
8        den = num + (1 - fp) * (1 - p)
9      else: raise ValueError("result must be + or -")
10     if den == 0: raise ValueError("impossible result")
11     p = num / den
12     print(res, round(p, 4))
▶    return p
```

VARIABLES

`p = 0.0999` `result = 0.0999`

OUTPUT

```
+ 0.0917
+ 0.5025
- 0.0999
```

Bayesian Decision Rule: Treat or Wait?

- Update the prior with the evidence; then compute each action's **expected loss** under the posterior. Choose the action with the lowest expected loss.
- Losses: needless treatment of a healthy patient 30, not treating an ill one 200.
- E[loss | treat] = (1 − p)·30; E[loss | wait] = p·200. Treat if it loses less: (1 − p)·30 < p·200 ⇒ 30 < 230p ⇒ p > 30/230 = **3/23 ≈ 13.0%**.

State	p	treat	wait	decision
1 positive	0.0917	27.2	18.3	wait, retest
2 positives	0.5025	14.9	100.5	treat

Expected losses (orange = smallest)

The decision depends not only on the probability but on the **ratio of the losses**: the costlier a missed disease, the lower the threshold.

Back to the Game: Bayesian Decision Lab

Theory's answer: 82.9%

P(rain | "rain") = 0.9·0.35 / (0.9·0.35 + 0.1·0.65) = 0.315 / 0.38 ≈ 0.829.

Information	P(rain)	EU(out)	EU(in)	Decision
prior only	0.35	3.10	3.35	indoors
"rain"	0.829	−3.61	3.83	indoors
"dry"	0.056	7.21	3.06	outdoors

EU: expected utility. Outdoors is chosen only when P(rain) < 1/3. At Level 3 (78% accuracy) the signal's value of information is 1.765; it exceeds the signal's price (1), so buying the signal pays off.

In practice: base-rate neglect

In a classic study, physicians and medical students were asked about a test with prevalence 1/1000 and a 5% false positive rate: only 11 of 60 reached the right answer (about 2%); the most common answer was 95% (Casscells, Schoenberger and Graboys, 1978).

The game's "EU-optimal" counter counts not lucky outcomes but decisions that maximize expected utility given the information at hand: a good decision can lead to a bad outcome.

Bridge to Bayesian Games: Updating the Opponent's Type

- With incomplete information the unknown "state" is the opponent's *type*: the belief is the prior, the move is the evidence.
- **Example** (poker): the opponent is a bluffer with probability 20%, honest with 80%. A bluffer raises 90% of the time whatever the hand; an honest player raises only with a strong hand (30%).
- The opponent raised: P(bluffer | raise) = 0.2·0.9 / (0.2·0.9 + 0.8·0.3) = 0.18 / 0.42 ≈ **0.429**.
- Calling earns +30 against a bluffer, −30 against an honest player: E[call] = 0.429·30 − 0.571·30 ≈ **−4.3** < 0 ⇒ fold. After a second raise the posterior is 0.692.

Observation	P(bluffer)	E[call]	decision
prior only	0.200	−18.0	fold
1 raise	0.429	−4.3	fold
2 raises	0.692	+11.5	call

Calling pays only if P(bluffer) > 0.5 (equal gain and loss).

Common mistake: confusing P(raise | bluffer) = 0.9 with P(bluffer | raise). Honest players are more common, so most raises (0.24 / 0.42 ≈ 57%) come from them. A bluff tries to steer this update.

Decision Tree: Product Launch and Market Research

- A firm must decide **whether to launch** a new product. If the market is good (probability 2/5) it earns +500, if bad (3/5) −200 thousand TL; no launch gives 0.
- It can first buy **market research** for 20 thousand TL. The report is imperfect: P(positive | good) = 0.8, P(positive | bad) = 0.1.
- By Bayes: P(positive) = 0.4·0.8 + 0.6·0.1 = 0.38; P(good | positive) = 0.32/0.38 = 16/19; P(good | negative) = 0.08/0.62 = 4/31.

□ Decision node

The decision maker chooses → the largest value among the branches.

○ Chance node

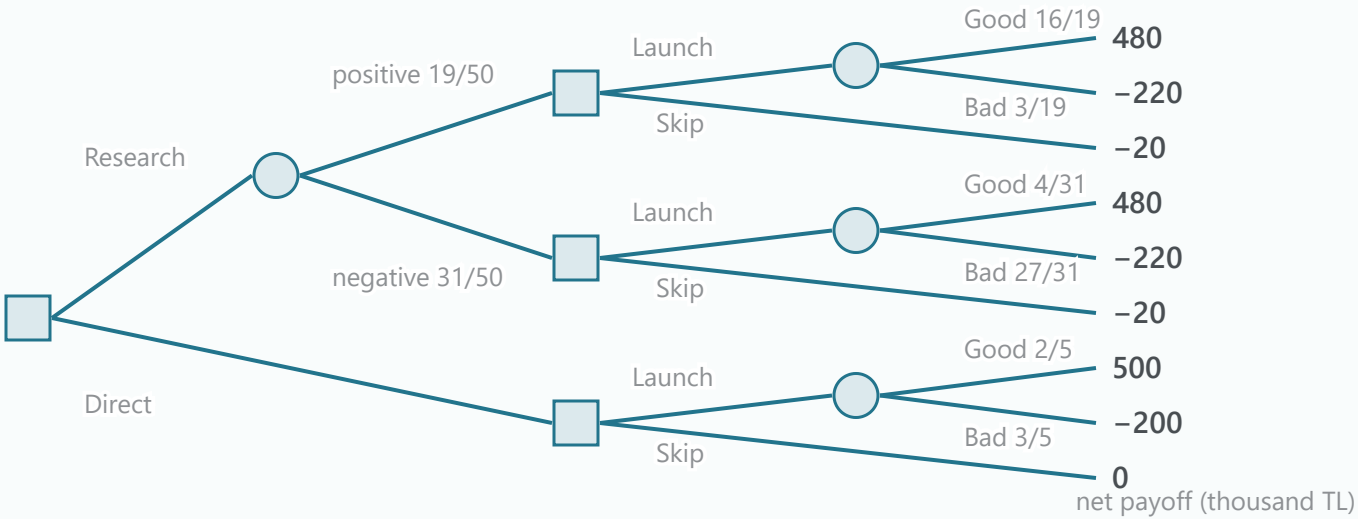
Nature chooses → probability-weighted average.

Leaf

The net result of a path.

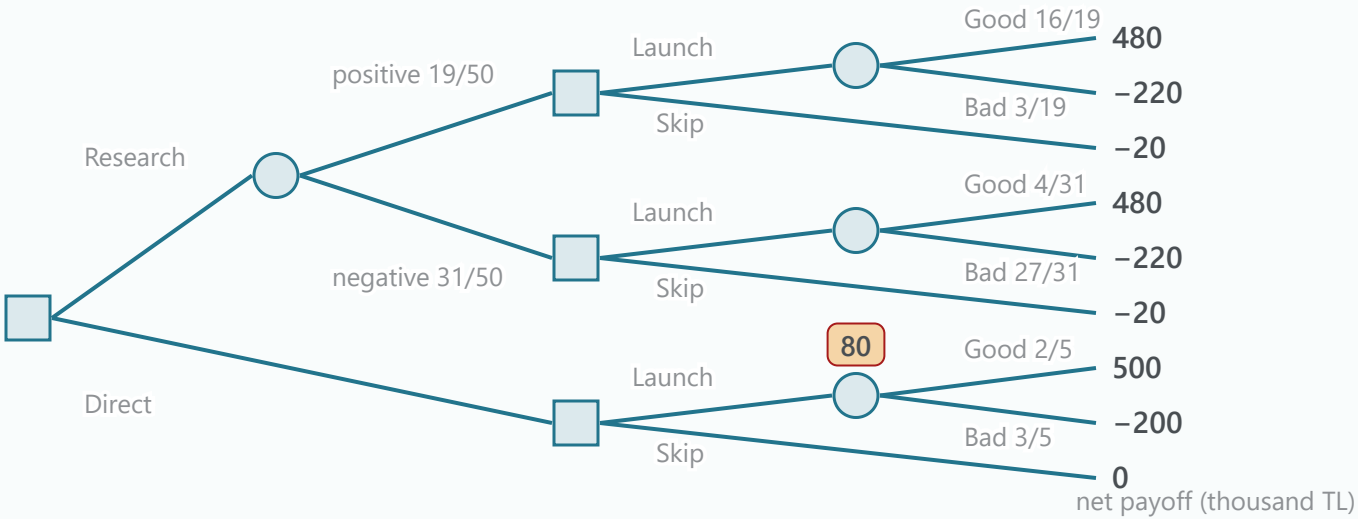
A decision tree is a game tree in which nature also moves (chance nodes); it is solved from the leaves to the root by **backward induction**.

Backward Induction in a Decision Tree



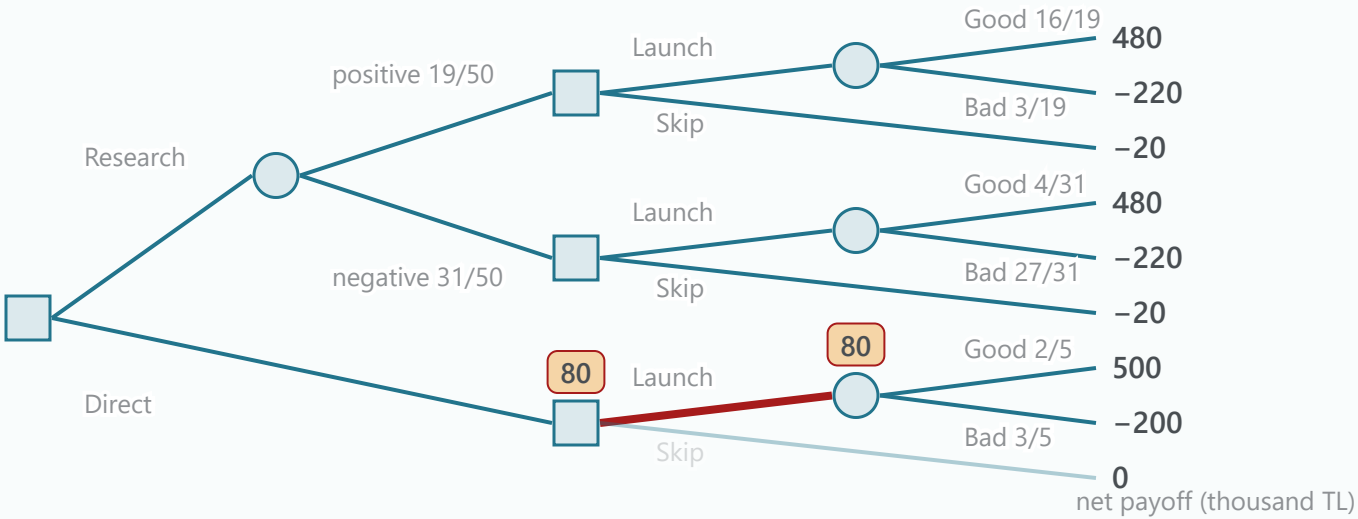
Square = decision node, circle = chance node. Leaves show net payoffs (20 deducted if research is bought). The two root branches: Research and Direct (decide without research). Backward induction: **expected value** at chance nodes, **maximum** at decision nodes.

Backward Induction in a Decision Tree



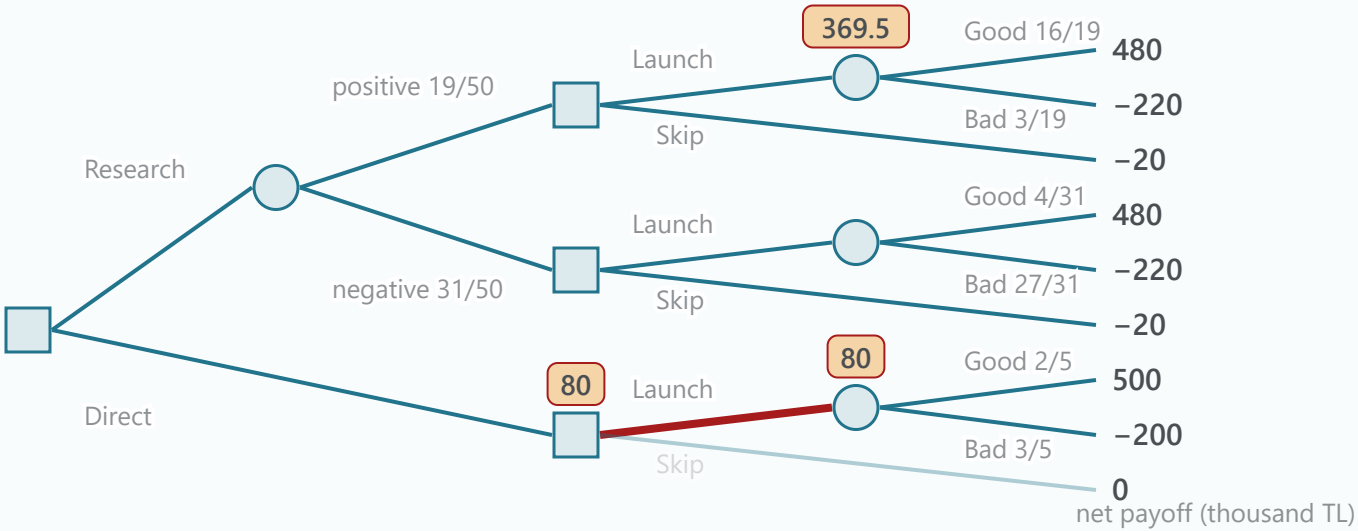
Launching without research: $2/5 \cdot 500 + 3/5 \cdot (-200) = 200 - 120 = \mathbf{80}$.

Backward Induction in a Decision Tree



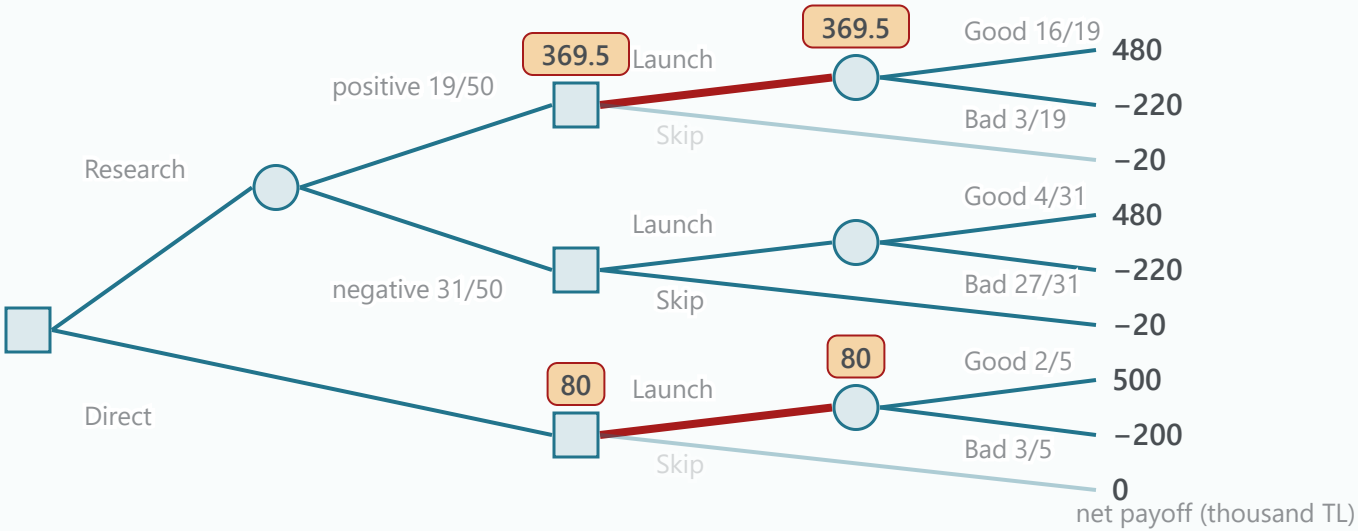
Decision: max(Launch 80, Skip 0) = 80 ⇒ **Launch**. The Skip branch is pruned.

Backward Induction in a Decision Tree



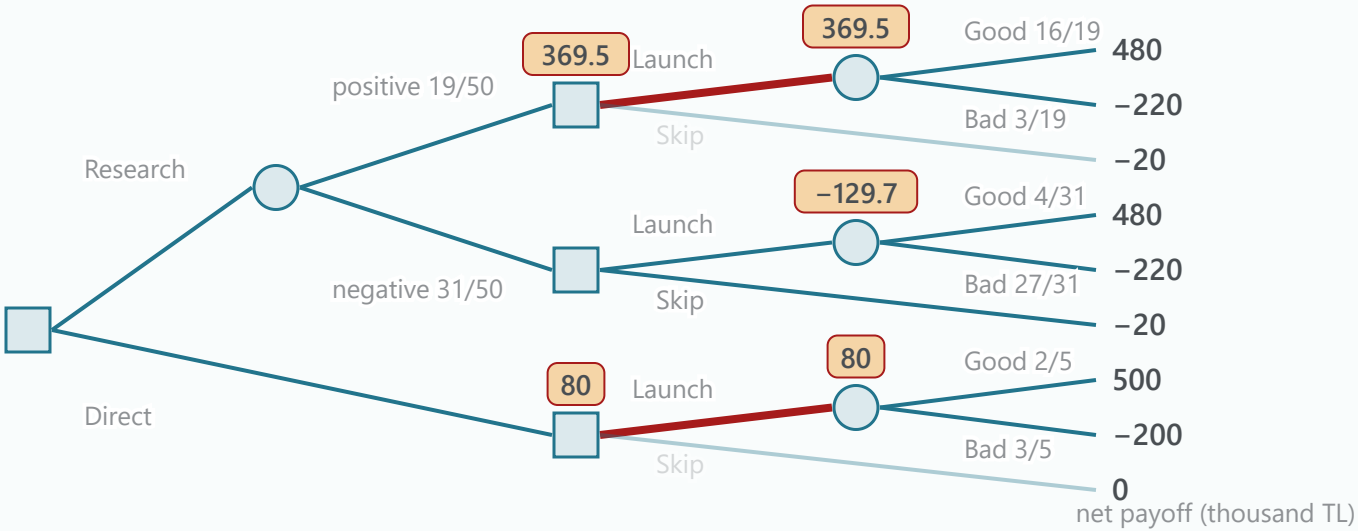
After a positive report, P(Good | positive) = 0.32/0.38 = 16/19 (Bayes). Expected: 16/19·480 + 3/19·(−220) ≈ **369.5**.

Backward Induction in a Decision Tree



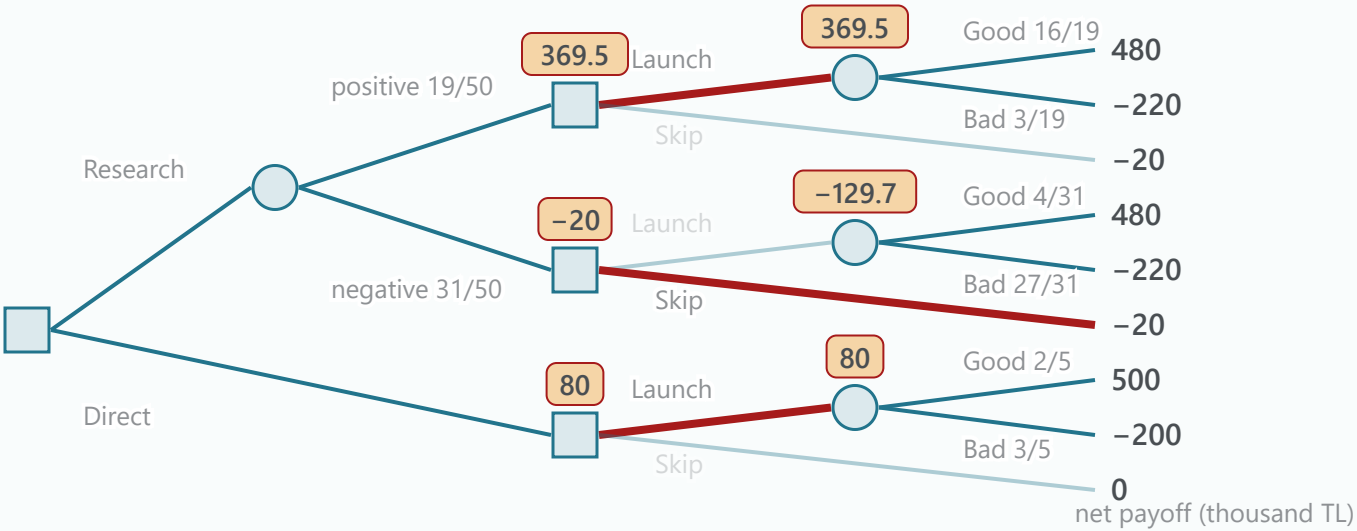
Decision: max(369.5, −20) ⇒ **Launch** after a positive report.

Backward Induction in a Decision Tree



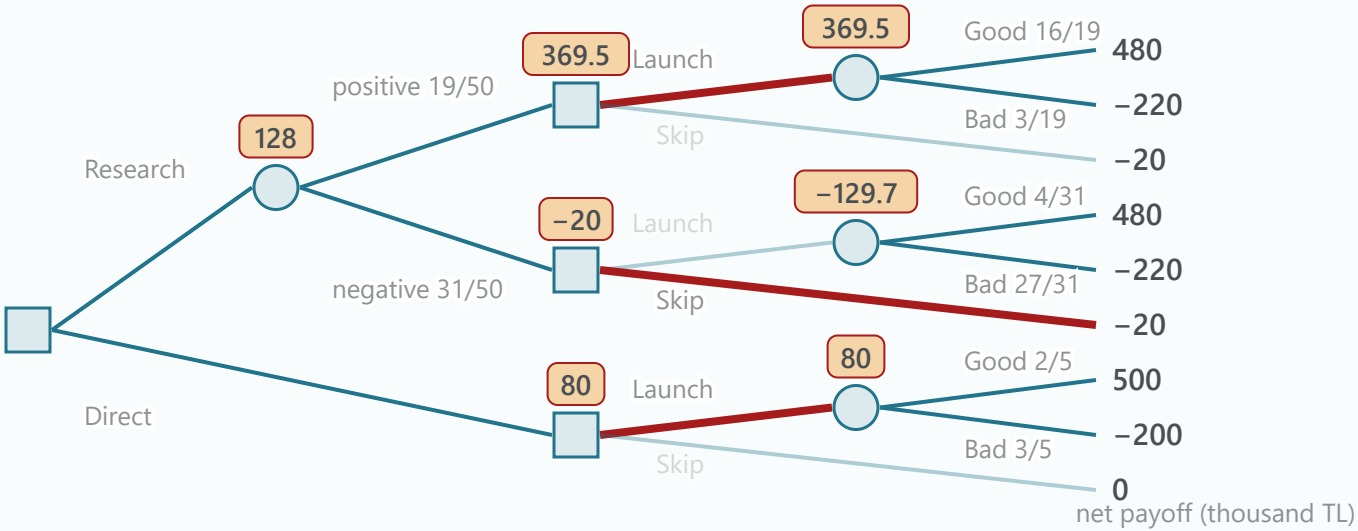
After a negative report, P(Good | negative) = 0.08/0.62 = 4/31. Expected: 4/31·480 + 27/31·(−220) ≈ **−129.7**.

Backward Induction in a Decision Tree



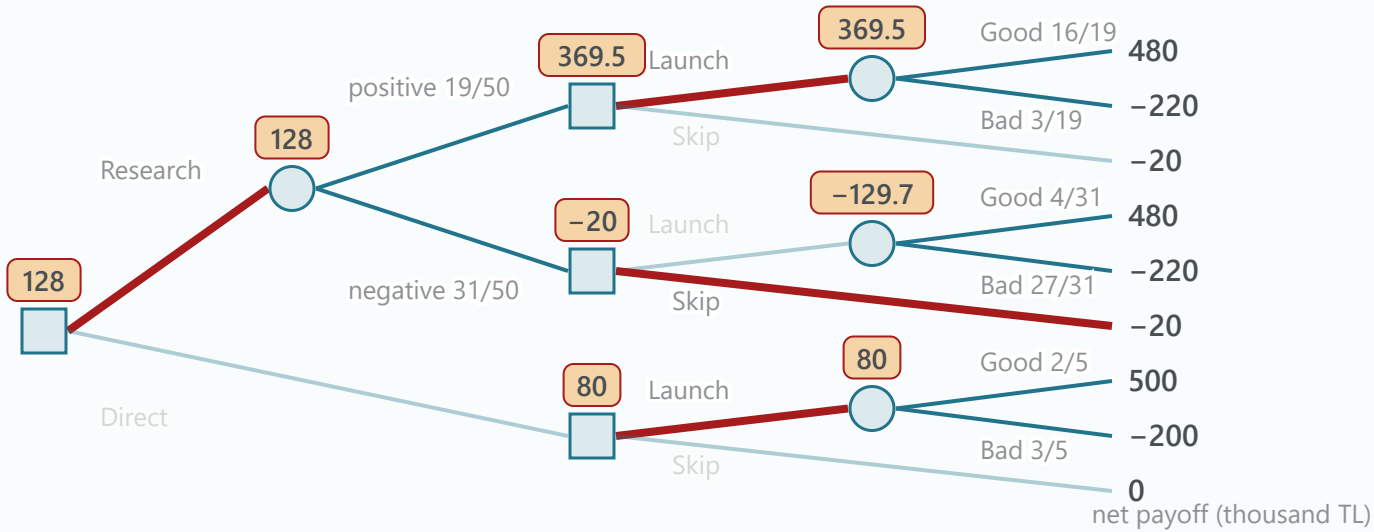
Decision: max(−129.7, −20) = −20 ⇒ **Skip** after a negative report (only the research cost is lost).

Backward Induction in a Decision Tree



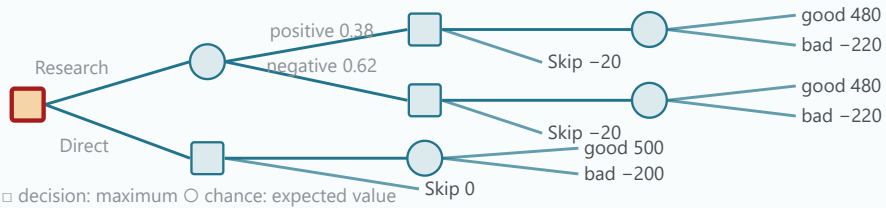
Report chance node: 0.38·369.5 + 0.62·(−20) = **128**.

Backward Induction in a Decision Tree



Root: max(Research 128, Direct 80) ⇒ **Research**. Strategy: do research; launch if positive, skip if negative.

Code Trace: Recursive Backward Induction



`value(root)` is called. The tree is nested tuples: `root = ("D", [("Research", ("C", [(0.38, …), (0.62, …)])), ("Direct", ("D", […]))])` ; "D" is a decision node, "C" a chance node, a leaf is a number.

Step 1 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶     kind, branches = n
▶     if kind == "C":                 # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

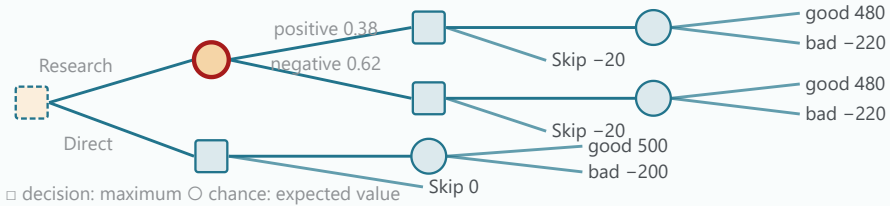
VARIABLES

`n = root`

CALL STACK

`value(root)`

Code Trace: Recursive Backward Induction



`value(Research)` is called: a chance node; its children are evaluated in turn.

Step 2 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶    kind, branches = n
▶    if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

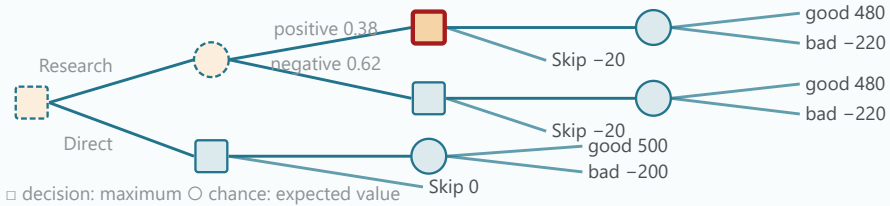
`n = Research`

CALL STACK

`value(Research)`

`value(root)`

Code Trace: Recursive Backward Induction



`value(positive)` is called: a decision node; its children are evaluated in turn.

Step 3 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶     kind, branches = n
▶     if kind == "C":                 # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

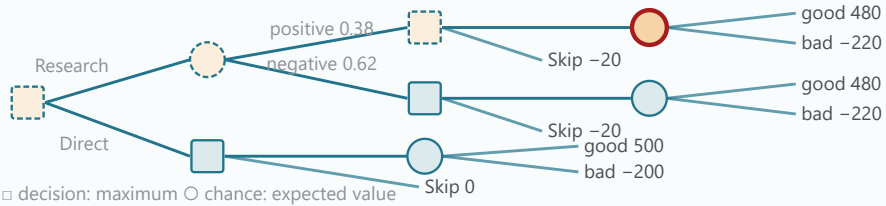
VARIABLES

`n = positive`

CALL STACK

- `value(positive)`
- `value(Research)`
- `value(root)`

Code Trace: Recursive Backward Induction



`value(positive·Launch)` is called: a chance node; its children are evaluated in turn.

Step 4 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶    kind, branches = n
▶    if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

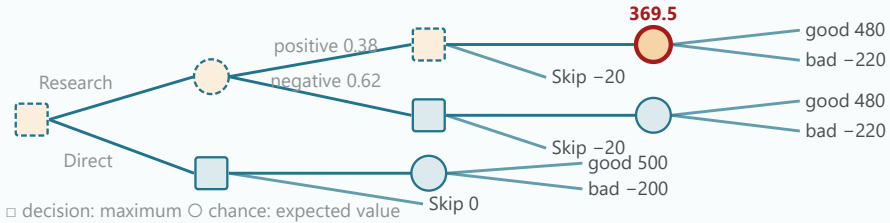
VARIABLES

`n = positive·Launch`

CALL STACK

- `value(positive·Launch)`
- `value(positive)`
- `value(Research)`
- `value(root)`

Code Trace: Recursive Backward Induction



positive·Launch (chance): 0.842·480 + 0.158·−220 = **369.5**.

Step 5 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                  # chance node
▶         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

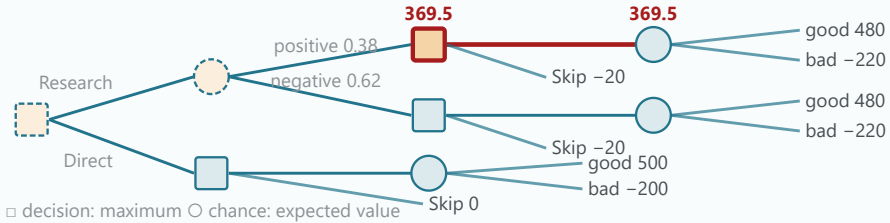
`n = positive·Launch`

`return = 369.47`

CALL STACK

- **value(positive·Launch)**
- value(positive)
- value(Research)
- value(root)

Code Trace: Recursive Backward Induction



positive (decision): max(Launch 369.5, Skip −20.0) = **369.5** → **Launch**.

Step 6 / 16

```
1  def value(n):
2      if not isinstance(n, tuple):    # leaf
3          return n
4      kind, branches = n
5      if kind == "C":                  # chance node
6          return sum(p * value(c) for p, c in branches)
▶      return max(value(c) for name, c in branches)  # decision
```

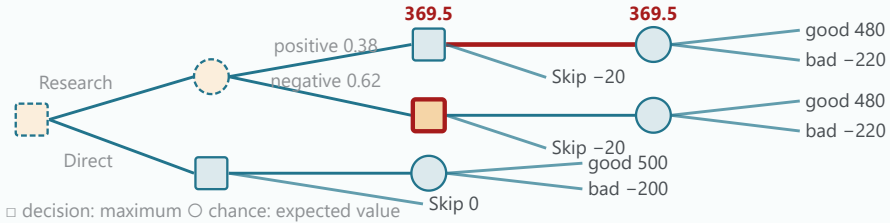
VARIABLES

`n = positive` `return = 369.47`

CALL STACK

- **value(positive)**
- value(Research)
- value(root)

Code Trace: Recursive Backward Induction



`value(negative)` is called: a decision node; its children are evaluated in turn.

Step 7 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶     kind, branches = n
▶     if kind == "C":                 # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

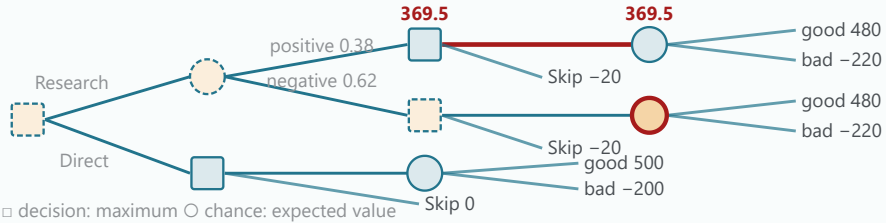
VARIABLES

`n = negative`

CALL STACK

- `value(negative)`
- `value(Research)`
- `value(root)`

Code Trace: Recursive Backward Induction



`value(negative·Launch)` is called: a chance node; its children are evaluated in turn.

Step 8 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶    kind, branches = n
▶    if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

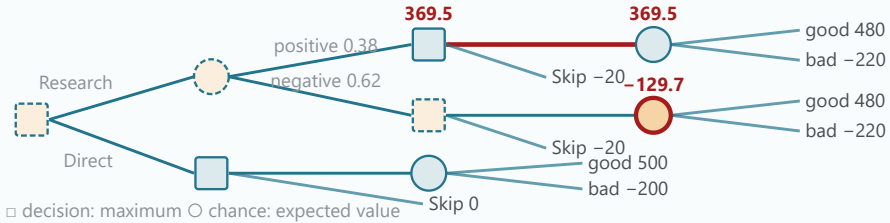
VARIABLES

`n = negative·Launch`

CALL STACK

- `value(negative·Launch)`
- `value(negative)`
- `value(Research)`
- `value(root)`

Code Trace: Recursive Backward Induction



negative·Launch (chance): 0.129·480 + 0.871·−220 = **−129.7**.

Step 9 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                 # chance node
▶         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

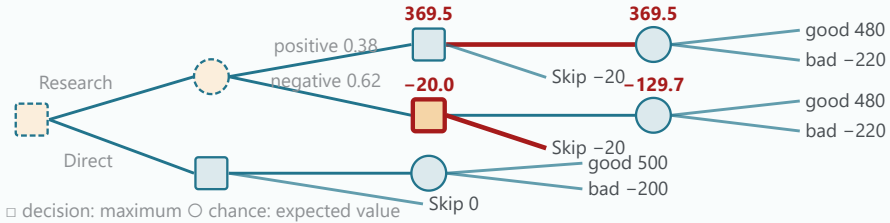
`n = negative·Launch`

`return = -129.68`

CALL STACK

- **value(negative·Launch)**
- value(negative)
- value(Research)
- value(root)

Code Trace: Recursive Backward Induction



negative (decision): max(Launch −129.7, Skip −20.0) = **−20.0** → **Skip**.

Step 10 / 16

```
1  def value(n):
2      if not isinstance(n, tuple):    # leaf
3          return n
4      kind, branches = n
5      if kind == "C":                  # chance node
6          return sum(p * value(c) for p, c in branches)
▶      return max(value(c) for name, c in branches)  # decision
```

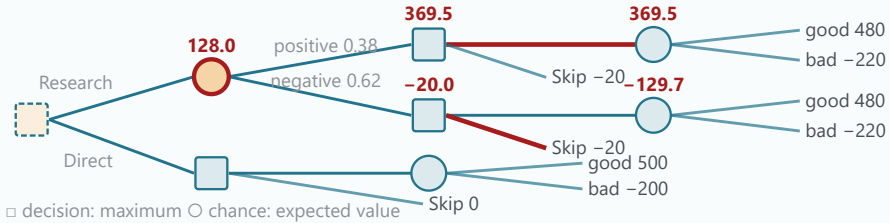
VARIABLES

`n = negative` `return = -20`

CALL STACK

- **value(negative)**
- value(Research)
- value(root)

Code Trace: Recursive Backward Induction



Research (chance): 0.38·369.5 + 0.62·−20 = **128.0**.

Step 11 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                  # chance node
▶         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

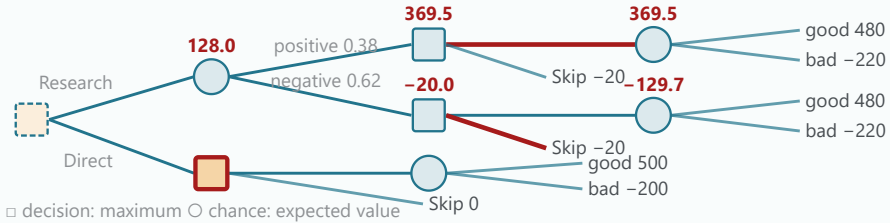
`n = Research` `return = 128.0`

CALL STACK

`value(Research)`

`value(root)`

Code Trace: Recursive Backward Induction



`value(Direct)` is called: a decision node; its children are evaluated in turn.

Step 12 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶     kind, branches = n
▶     if kind == "C":                 # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

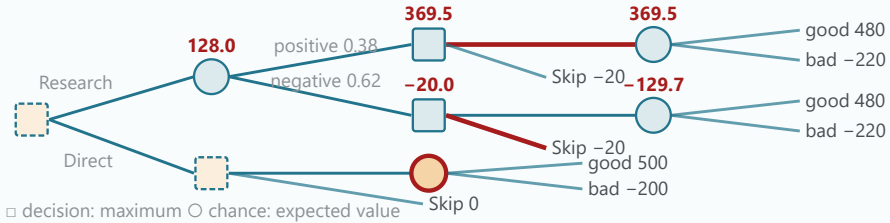
`n = Direct`

CALL STACK

`value(Direct)`

`value(root)`

Code Trace: Recursive Backward Induction



`value(Direct·Launch)` is called: a chance node; its children are evaluated in turn.

Step 13 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
▶    kind, branches = n
▶    if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

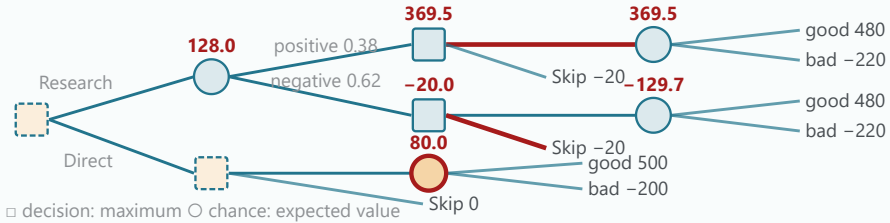
VARIABLES

`n = Direct·Launch`

CALL STACK

- `value(Direct·Launch)`
- `value(Direct)`
- `value(root)`

Code Trace: Recursive Backward Induction



Direct·Launch (chance): 0.4·500 + 0.6·−200 = **80.0**.

Step 14 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                  # chance node
▶         return sum(p * value(c) for p, c in branches)
7     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

n = **Direct·Launch**

return = **80.0**

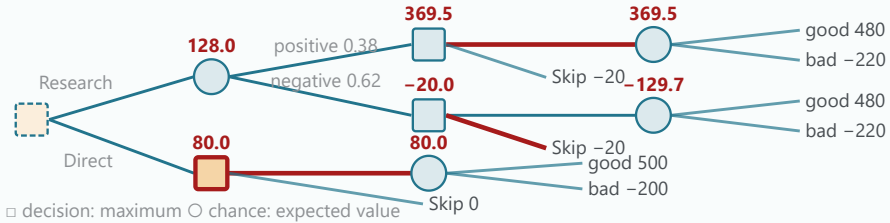
CALL STACK

value(Direct·Launch)

value(Direct)

value(root)

Code Trace: Recursive Backward Induction



Direct (decision): max(Launch 80.0, Skip 0.0) = **80.0** → **Launch**.

Step 15 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

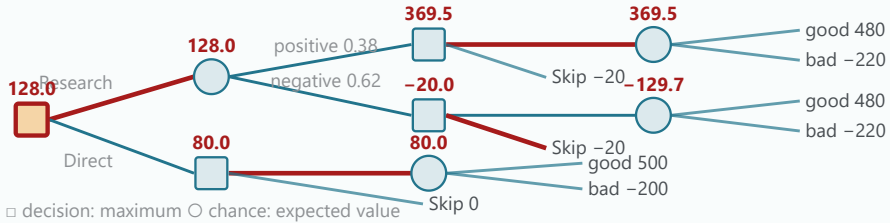
`n = Direct` `return = 80.0`

CALL STACK

`value(Direct)`

`value(root)`

Code Trace: Recursive Backward Induction



root (decision): max(Research 128.0, Direct 80.0) = **128.0** → **Research**. Strategy: do research; launch if positive, skip if negative.

Step 16 / 16

```
1 def value(n):
2     if not isinstance(n, tuple):    # leaf
3         return n
4     kind, branches = n
5     if kind == "C":                  # chance node
6         return sum(p * value(c) for p, c in branches)
▶     return max(value(c) for name, c in branches)  # decision
```

VARIABLES

`n = root` `return = 128.0`

CALL STACK

`value(root)`

Expected Value of Information: EVPI and EVSI

State	Computation	Value
No information (decide now)	Launch: 0.4·500 + 0.6·(−200)	80
.	.	.
.	.	.
.	.	.
.	.	.

Without information the best action is to launch: expected value 80.

Step 1 / 5

Information has value only if it can change the decision: a negative report leads to "skip", so the research has positive value.

Expected Value of Information: EVPI and EVSI

State	Computation	Value
No information (decide now)	Launch: 0.4·500 + 0.6·(−200)	80
Perfect information (oracle)	good → launch 500, bad → skip 0: 0.4·500 + 0.6·0	200
.	.	.
.	.	.
.	.	.

If an oracle revealed the market in advance, the firm would launch in a good market (500) and skip in a bad one (0). Expected 200.

Information has value only if it can change the decision: a negative report leads to "skip", so the research has positive value.

Expected Value of Information: EVPI and EVSI

State	Computation	Value
No information (decide now)	Launch: 0.4·500 + 0.6·(−200)	80
Perfect information (oracle)	good → launch 500, bad → skip 0: 0.4·500 + 0.6·0	200
EVPI = expected value of perfect information	200 − 80	**120**
.	.	.
.	.	.

EVPI = 200 − 80 = **120**: no information source is worth more than this (an upper bound).

Information has value only if it can change the decision: a negative report leads to "skip", so the research has positive value.

Expected Value of Information: EVPI and EVSI

State	Computation	Value
No information (decide now)	Launch: 0.4·500 + 0.6·(−200)	80
Perfect information (oracle)	good → launch 500, bad → skip 0: 0.4·500 + 0.6·0	200
EVPI = expected value of perfect information	200 − 80	**120**
Sample information (research, free)	128 from the tree + research cost 20	148
.	.	.

Gross value of the imperfect research: add its cost back to the tree's 128 ⇒ 148.

Information has value only if it can change the decision: a negative report leads to "skip", so the research has positive value.

Expected Value of Information: EVPI and EVSI

State	Computation	Value
No information (decide now)	Launch: 0.4·500 + 0.6·(−200)	80
Perfect information (oracle)	good → launch 500, bad → skip 0: 0.4·500 + 0.6·0	200
EVPI = expected value of perfect information	200 − 80	**120**
Sample information (research, free)	128 from the tree + research cost 20	148
EVSI = expected value of sample information	148 − 80	**68**

EVSI = 148 − 80 = **68** > cost 20 ⇒ do the research (net gain 48). Efficiency EVSI/EVPI = 57%.

Information has value only if it can change the decision: a negative report leads to "skip", so the research has positive value.

Summary

Expected utility

$E[u] = \Sigma\, p_i\, u(x_i)$. Lottery A: $E[X] = 450$, but with $\sqrt{x}$, $E[u] = 15 < u(400) = 20$.

Utility scales

The transformation $a \cdot u + b$ ($a > 0$) preserves decisions; the Allais and Ellsberg choices deviate systematically from expected utility.

Risk attitude

Concave → averse (CE 225, premium 225); linear → neutral; convex → seeking.

Uncertainty criteria

On the same table, maximin picks No investment, maximax and Laplace Big, Savage Small.

Bayesian decisions

Prevalence 1%, sensitivity 90%, false positives 9% ⇒ $P(\text{ill} \mid +) \approx 9.2\%$; treat only if $p > 3/23$.

Decision trees

Average at chance nodes, maximize at decisions. Research (128) > no research (80); EVPI = 120, EVSI = 68.

SUMMARY

Better Decisions under Risk and Uncertainty

Value information

Current data improves decisions; gathering information pays only if EVSI exceeds its cost.

Scenarios and decision trees

Once the possible states and their probabilities are written down, the tree can be solved backward.

Account for the prior

Base rates enter the posterior through Bayes' rule; thinking in natural frequencies reduces errors.

Opponents and diversification

Game theory and simulation model opponents' behavior; portfolio diversification reduces risk.

SUMMARY

Practice Problems

1. Lottery: 2,500 TL with probability 20%, otherwise 100 TL. Expected value? Certainty equivalent and risk premium for $u(x) = \sqrt{x}$?
2. In the insurance example, what would the maximum premium be if the loss probability were 20%?
3. In the medical test, what would $P(\text{ill} \mid +)$ be with prevalence 10% (sensitivity 90%, false positives 9%)?
4. In the decision tree, what would the best strategy be if research cost 70 thousand TL?

▶ **Show answers**

References

- J. von Neumann, O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, 1944.
- S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Pearson, 2020 (Chapters 12–16: uncertainty, Bayes, decision theory).
- H. Raiffa, *Decision Analysis*, 1968; D. Bernoulli, “Specimen theoriae novae de mensura sortis”, 1738.
- M. Allais, “Le comportement de l'homme rationnel devant le risque”, *Econometrica* 21, 1953; L. J. Savage, “The theory of statistical decision”, *JASA* 46, 1951; D. Ellsberg, *Quarterly J. Economics* 75, 1961.
- G. Gigerenzer, U. Hoffrage, "How to improve Bayesian reasoning without instruction: Frequency formats", *Psychological Review* 102(4), 1995.
- W. Casscells, A. Schoenberger, T. B. Graboys, “Interpretation by physicians of clinical laboratory results”, *NEJM* 299, 1978.
- D. Easley, J. Kleinberg, ***Networks, Crowds, and Markets***, Cambridge University Press, 2010 (Chapter 16: information cascades and Bayes' rule). Tool: **Gambit** (draws trees with chance nodes and solves them backward).

The End

Chapter 1: Decision Theory

Previous: Chapter 0: Preliminaries
Next: Chapter 2: Games and Representations