Chapter 2: Classical Optimization Foundations

Before we can understand quantum optimization, we must understand optimization itself.

Quantum optimization does not replace the ordinary language of optimization. It builds on it. Later, when we talk about QUBO models, Ising Hamiltonians, annealing schedules, cost Hamiltonians, and variational quantum algorithms, we will still be asking the same classical question:

> What are the possible choices, and which choice has the best score?

This chapter gives us the vocabulary needed to ask that question precisely.

We will begin with variables, objective functions, and constraints. Then we will study feasible regions, local and global optima, convex and nonconvex problems, and the special difficulty of combinatorial optimization.

The goal is not to memorize many definitions. The goal is to learn how to look at a real-world decision problem and see its mathematical shape.

2.1 Optimization as choosing with a score

An optimization problem is a problem in which we choose values for some quantities in order to make a score as good as possible.

The quantities we can choose are called variables.

The score we want to improve is called the objective function.

The rules that limit our choices are called constraints.

Let us start with a small example.

A bakery makes loaves of bread and cakes. Each loaf gives a profit of \$3. Each cake gives a profit of \$8. The bakery must decide how many loaves and cakes to make today.

Let

$$x = \text{number of loaves},$$

and

$$y = \text{number of cakes.}$$

Here x and y are variables. They are the choices the bakery controls.

If the bakery makes x loaves and y cakes, the profit is

$$3x + 8y.$$

This expression is the objective function. It gives a score for each possible choice (x,y).

If the bakery wants the largest possible profit, we write the problem as

$$\text{maximize } 3x + 8y.$$

The word maximize means “make as large as possible.”

Sometimes we want to make something small instead. For example, if a delivery company wants the shortest route, it may write

$$\text{minimize total distance.}$$

The word minimize means “make as small as possible.”

Mathematically, maximization and minimization are closely related. Maximizing f(x) is the same as minimizing -f(x), because the largest value of f(x) corresponds to the smallest value of -f(x). For example, maximizing profit is equivalent to minimizing negative profit. This simple conversion is common in optimization, where many textbooks present most theory in minimization form (Nocedal and Wright, 2006).

So the first basic form is:

$$\text{minimize } f(x),$$

where x represents the variables and f(x) is the objective function.

If there are several variables, we often write them as a vector:

$$x = (x_1, x_2, \ldots, x_n).$$

A vector here simply means an ordered list of numbers. If we have five decision quantities, then x may be a list of five values.

For example,

$$x = (2, 5, 0, 7)$$

could mean:

- produce 2 units of product 1,
- produce 5 units of product 2,
- produce 0 units of product 3,
- produce 7 units of product 4.

This vector notation will become very useful later, especially when we encode optimization problems into binary variables for quantum algorithms.

2.2 Variables: what we are allowed to choose

A variable is a mathematical placeholder for a value that can change.

In optimization, variables represent decisions.

Different problems allow different kinds of variables.

A continuous variable can take any real value in some range. For example, a chemical plant might choose a temperature T between 300 K and 500 K:

$$300 \leq T \leq 500.$$

The value might be 350, or 350.7, or 350.7341. It is not restricted to whole numbers.

An integer variable must take a whole-number value. For example, a warehouse cannot ship 3.6 trucks. It may ship 3 trucks or 4 trucks, but not 3.6 trucks.

A binary variable can take only two values, usually 0 or 1:

x□ ∈ \0,1\.

Binary variables are especially important in quantum optimization. A binary variable often represents a yes-or-no decision:

$$x_i = \begin{cases} 1, & \text{if we choose option } i, \\ 0, & \text{if we do not choose option } i. \end{cases}$$

For example, suppose we are choosing which projects to fund. Let

$$x_i = 1$$

mean project i is funded, and

$$x_i = 0$$

mean project i is not funded.

If there are 10 possible projects, then a solution is a binary vector like

$$x = (1, 0, 0, 1, 1, 0, 0, 0, 1, 0).$$

This means we choose projects 1, 4, 5, and 9.

Binary variables may look simple, but they create enormous solution spaces. With n binary variables, there are

$$2^n$$

possible assignments. If n = 10, there are $2^{10} = 1024$ assignments. If n = 100, there are

$$2^{100} \approx 1.27 \times 10^{30}$$

assignments. That number is far too large for direct checking one by one.

This exponential growth is one reason binary optimization is central to both classical and quantum optimization.

2.3 Objective functions: how we score a choice

An objective function is a rule that assigns a number to each possible solution.

The number is the score we are trying to improve.

If lower is better, we minimize the objective. If higher is better, we maximize it.

For example, suppose a delivery driver must visit several locations. A route has a total distance. The objective function might be

$$f(\text{route}) = \text{total distance of the route.}$$

The optimization problem is

$$\text{minimize } f(\text{route}).$$

Now suppose a company chooses how much to spend on online advertisements. Let x be the amount of money spent. The company may estimate revenue as a function R(x). If profit is revenue minus cost, then

$$f(x) = R(x) - x$$

could be the profit objective, and the company may want to maximize f(x).

Objective functions can be simple or complicated.

A linear objective function is one where variables are multiplied only by constants and then added.

For example,

$$f(x_1, x_2, x_3) = 4x_1 + 7x_2 - 2x_3$$

is linear.

A quadratic objective function may include products of two variables, such as

$$f(x_1, x_2) = 3x_1 + 5x_2 + 2x_1x_2.$$

The term $2x_1x_2$ is quadratic because it multiplies two variables.

Quadratic objective functions will matter greatly later. In Chapter 9, we will study QUBO, which stands for Quadratic Unconstrained Binary Optimization. A QUBO problem has binary variables and a quadratic objective function. Many quantum optimization workflows begin by transforming a practical problem into QUBO or an equivalent Ising model.

For now, the important idea is simple:

> The objective function is the mathematical expression of what we care about.

If we choose the wrong objective, we solve the wrong problem.

For example, suppose a hospital scheduling system only minimizes labor cost. It may produce a cheap schedule that exhausts nurses, violates fairness expectations, or creates unsafe staffing patterns. A better model might include cost, rest requirements, skill coverage, fairness, and legal rules. Some of these may appear as constraints, and some may appear as terms in the objective.

Optimization is powerful, but it is not morally automatic. The model reflects what we decide to measure.

2.4 Constraints: the rules of the problem

A constraint is a rule that a solution must satisfy.

Constraints turn “any choice” into “allowed choices.”

Return to the bakery example. The bakery makes loaves and cakes. Let

$$x = \text{number of loaves},$$

and

$$y = \text{number of cakes}.$$

Maybe the bakery has only 40 units of flour. Each loaf uses 1 unit of flour, and each cake uses 4 units. Then the flour constraint is

$$x + 4y \leq 40.$$

This means the bakery cannot use more than 40 units of flour.

Maybe the bakery also has only 30 units of labor time. Each loaf uses 2 units of labor and each cake uses 3 units. Then

$$2x + 3y \leq 30.$$

Also, the bakery cannot make a negative number of loaves or cakes:

$$x \geq 0,$$

$$y \geq 0.$$

If loaves and cakes must be whole objects, we also require

$$x, y \in \{0, 1, 2, \ldots\}.$$

A complete version of the bakery problem might be:

$$\text{maximize } 3x + 8y$$

subject to

$$x + 4y \leq 40,$$

$$2x + 3y \leq 30,$$

$$x \geq 0, \quad y \geq 0,$$

$$x, y \in \{0, 1, 2, \ldots\}.$$

The phrase subject to means “while obeying these constraints.”

There are several common kinds of constraints.

An inequality constraint uses symbols such as ≤, ≥, <, or >. For example,

$$x + 4y \leq 40.$$

An equality constraint requires exact equality. For example, if exactly 12 workers must be assigned to a shift, we might write

$$x_1 + x_2 + \cdots + x_n = 12,$$

where xᵢ = 1 means worker i is assigned, and xᵢ = 0 means worker i is not assigned.

A domain constraint says what type of values a variable may take. For example,

x□ ∈ \0,1\

says xᵢ is binary, while

$$x_i \in \mathbb{Z}$$

says xᵢ is an integer. The symbol mathbbZ means the set of all integers:

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

In real applications, constraints are often the hardest part of modeling. A weak constraint may allow unrealistic solutions. A too-strict constraint may make the problem impossible. A good modeler learns to ask:

- What choices are physically possible?
- What choices are legally allowed?
- What choices are operationally realistic?

- What choices are desirable but not absolutely required?

This last question matters because not every rule must be a hard constraint. Sometimes a violation is allowed but costly. In that case, we may add a penalty to the objective instead of forbidding the violation completely. Penalty modeling will become especially important when we encode constraints into QUBO and Ising forms.

2.5 Feasible solutions and the feasible region

A feasible solution is a choice of variables that satisfies all constraints.

The feasible region is the set of all feasible solutions.

For example, suppose we have the constraints

$$x \geq 0,$$

$$y \geq 0,$$

$$x + y \leq 5.$$

The point (x,y) = (2,2) is feasible because

$$2 \geq 0, \quad 2 \geq 0, \quad 2 + 2 = 4 \leq 5.$$

The point (x,y) = (4,3) is not feasible because

$$4 + 3 = 7 > 5.$$

So (4,3) violates the constraint x+y ≤ 5.

When variables are continuous, the feasible region may look like a geometric shape: a line, a polygon, a curved area, or a higher-dimensional body. In two dimensions, we can often draw it.

When variables are binary, the feasible region is a set of separate points. For example, with three binary variables,

$$x_1, x_2, x_3 \in \{0, 1\},$$

there are eight possible assignments:

$$(0,0,0),\ (0,0,1),\ (0,1,0),\ (0,1,1),$$

$$(1,0,0),\ (1,0,1),\ (1,1,0),\ (1,1,1).$$

If we add the constraint

$$x_1 + x_2 + x_3 = 2,$$

then only the assignments with exactly two 1s are feasible:

$$(1,1,0),\ (1,0,1),\ (0,1,1).$$

The feasible region is therefore the set

$$\{(1,1,0), (1,0,1), (0,1,1)\}.$$

This set is small enough to list. But if we had 100 binary variables and required exactly 50 of them to be 1, the number of feasible assignments would be enormous. It would be

$$\binom{100}{50},$$

where binom10050 means “100 choose 50,” the number of ways to choose 50 items from 100. This number is about 1.01×10^{29}.

So even after constraints remove many possibilities, the feasible region can still be huge.

2.6 Optimal solutions: best among feasible choices

An optimal solution is a feasible solution with the best objective value.

If we are minimizing, an optimal solution has the smallest objective value among all feasible solutions.

If we are maximizing, an optimal solution has the largest objective value among all feasible solutions.

Let us use a small minimization example. Suppose we have four feasible choices:

Choice	Objective value
A	12
B	7
C	9
D	7

If we minimize, choices B and D are both optimal because they both achieve the smallest value, 7.

This shows an important point:

> An optimization problem can have more than one optimal solution.

The optimal value is the best objective value itself. In the table above, the optimal value is 7. The optimal solutions are B and D.

So we should distinguish:

- the optimal solution, meaning the variable assignment,
- the optimal value, meaning the objective score achieved by that assignment.

For example, in the problem

$$\text{minimize } f(x) = (x-3)^2,$$

over all real numbers x, the optimal solution is

$$x = 3,$$

and the optimal value is

$$f(3) = 0.$$

The solution is the input x=3. The value is the score 0.

This distinction will matter later when we discuss sampling from quantum devices. A quantum optimizer may return many candidate solutions. We then evaluate their objective values and ask how close they are to the best known or best possible value.

2.7 Local optima and global optima

A global optimum is the best solution in the entire feasible region.

A local optimum is a solution that is best only compared with nearby solutions.

To understand this, imagine walking on a landscape of hills and valleys. Your height represents the objective value. If you are minimizing, you want to reach the lowest valley.

A valley may be lower than all nearby points, but not the lowest valley in the whole landscape. That valley is a local minimum but not a global minimum.

A local minimum is a point whose objective value is no larger than the objective values of nearby feasible points.

A global minimum is a point whose objective value is no larger than the objective values of all feasible points.

Consider the function

$$f(x) = x^4 - 3x^2 + 2.$$

This function has more than one valley-like region. A point can be locally good without being the best possible over the whole line.

For a simpler discrete example, suppose we have five feasible solutions arranged in a path:

$$A - B - C - D - E.$$

Their objective values are:

Solution	Objective value
A	10
B	6
C	8
D	3
E	5

Suppose “nearby” means connected by one step along the path.

Choice B is a local minimum because its neighbors A and C have objective values 10 and 8, both worse than 6.

But B is not a global minimum, because D has value 3.

Choice D is both a local minimum and a global minimum.

This idea is central to optimization algorithms. Many algorithms improve a solution by making small local changes. Such algorithms can get stuck at a local optimum. They cannot easily tell whether a better solution exists far away.

Classical numerical optimization studies many methods for moving through such landscapes, including gradient-based methods, Newton-type methods, and trust-region methods; these methods are especially important for continuous optimization problems (Nocedal and Wright, 2006). Combinatorial optimization has its own local search methods, where “nearby” may mean flipping one bit, swapping two items, or moving one job to another machine (Papadimitriou and Steiglitz, 1998).

Quantum optimization methods are often motivated by the hope that quantum effects may explore difficult landscapes in useful ways. But to judge that hope properly, we first need to understand what makes a landscape difficult in classical terms.

2.8 Linear optimization: a first structured class

A linear optimization problem has a linear objective function and linear constraints.

A linear expression has variables multiplied by constants and added together, without products of variables or powers like x^2.

For example,

$$4x_1 + 2x_2 - 7x_3$$

is linear.

But

$$x_1x_2 + 3x_3$$

is not linear, because it contains the product x_1x_2.

And

$$x_1^2 + x_2$$

is not linear, because it contains x_1^2.

A standard linear optimization problem may look like this:

$$\text{minimize } c^T x$$

subject to

$$Ax \leq b.$$

This notation may look compact, so let us unpack it.

The vector x contains the variables:

$$x = (x_1, x_2, \ldots, x_n).$$

The vector c contains the coefficients of the objective. The expression $c^{(T)}$ x means

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n.$$

The matrix A and vector b describe the linear constraints. A matrix is a rectangular table of numbers. The expression

$$Ax \leq b$$

means several linear inequalities at once.

For example, the two constraints

$$x + 4y \leq 40,$$

$$2x + 3y \leq 30$$

can be written as

$$\begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 40 \\ 30 \end{bmatrix}.$$

Linear optimization is important because it is both expressive and mathematically well understood. Linear programming can be solved efficiently in a theoretical sense by polynomial-time algorithms, and it is also widely solved in practice by methods such as simplex and interior-point algorithms (Bertsimas and Tsitsiklis, 1997).

However, if we add integer restrictions, the problem changes sharply. A linear problem with integer variables is called an integer linear program, or integer program. Integer programming can model many practical scheduling, routing, assignment, and selection problems, but it is computationally much harder in general than continuous linear programming (Nemhauser and Wolsey, 1988).

This difference is one of the recurring lessons of optimization:

> The type of variables can matter as much as the form of the objective.

A linear problem with continuous variables may be tractable. A similar-looking linear problem with integer variables may become very hard.

2.9 Convex sets: regions without dents or holes

To understand modern optimization, we need the idea of convexity.

A set is convex if, whenever it contains two points, it also contains the entire straight line segment between those points.

That definition is abstract, so let us draw it in words.

Imagine two points inside a circular disk. If you connect them with a straight line, the whole line segment stays inside the disk. Therefore, a disk is convex.

A filled square is also convex. A filled triangle is convex. A solid ball is convex.

Now imagine a crescent shape, like a moon. You can choose two points inside the crescent such that the straight line between them passes outside the crescent. Therefore, the crescent is not convex.

A set with a hole is also not convex, because a line segment between two points may cross the hole.

Mathematically, a set S is convex if for any two points x and y in S, and for any number λ between 0 and 1, the point

$$\lambda x + (1 - \lambda) y$$

is also in S.

The symbol λ is a Greek letter pronounced “lambda.” Here it controls how far we move from y to x. If λ = 0, the expression gives y. If λ = 1, it gives x. If λ = 1/2, it gives the midpoint.

Convex sets matter because they have no separated pockets. In a convex feasible region, moving in a straight line between feasible solutions keeps us feasible.

Linear inequality constraints create convex feasible regions. For example,

$$x + y \leq 5$$

describes one side of a line, which is a convex half-plane. Intersections of convex sets are convex, so a feasible region described by many linear inequalities is convex (Boyd and Vandenberghe, 2004).

This is one reason linear programming has such strong theory.

2.10 Convex functions: bowl-shaped objectives

A convex function is a function shaped so that the line segment between any two points on its graph lies above the graph.

For a one-dimensional example, consider

$$f(x) = x^2.$$

This is the familiar upward-opening parabola. If you choose two points on the curve and draw a straight line between them, the curve stays below that line. This is the typical “bowl” shape of a convex function.

Mathematically, a function f is convex if

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$

for any x, y, and any λ between 0 and 1.

This inequality says:

> The function value at an average point is no worse than the average of the function values.

For example, let

$$f(x) = x^2.$$

Choose x=0, y=2, and λ = 1/2. The midpoint is

$$\frac{1}{2}(0) + \frac{1}{2}(2) = 1.$$

The function value at the midpoint is

$$f(1) = 1.$$

The average function value is

$$\frac{1}{2} f(0) + \frac{1}{2} f(2) = \frac{1}{2}(0) + \frac{1}{2}(4) = 2.$$

Indeed,

$$1 \leq 2.$$

So this example satisfies the convexity inequality.

A function such as

$$f(x) = -x^2$$

is not convex on the real line. It opens downward. For minimization, this is dangerous because the function does not form a simple bowl.

A convex optimization problem is usually a minimization problem where:

1. the objective function is convex,
2. the feasible region is convex.

This combination has a powerful consequence:

> In a convex optimization problem, every local minimum is also a global minimum.

This is one of the central reasons convex optimization is so important (Boyd and Vandenberghe, 2004).

Let us understand why this is true intuitively.

Suppose you are standing in a valley of a perfectly convex bowl. If every nearby direction goes upward, then there cannot be a deeper valley somewhere else. A deeper valley elsewhere would require the shape to bend down again, creating a nonconvex landscape.

This does not mean every convex problem is easy in every practical sense. Very large problems, ill-conditioned problems, or problems with expensive objective evaluations can still be challenging. But convexity removes the specific problem of misleading local minima.

That makes convex optimization a kind of “safe territory” in the optimization world.

2.11 Nonconvex problems: landscapes with traps

A nonconvex problem is a problem that does not satisfy convexity. The feasible region may be nonconvex, the objective may be nonconvex, or both.

Nonconvex problems can have multiple local optima. They can have separated feasible regions. They can have ridges, valleys, plateaus, and traps.

Consider the function

$$f(x) = x^4 - x^2.$$

This function is not convex over the whole real line. It has two symmetric low regions. If an algorithm starts on one side, it may move toward one valley and never explore the other side.

In discrete optimization, nonconvexity appears in another way. Binary variables create separated points rather than a smooth connected region. If a solution is a string of bits, then a local move might flip one bit:

$$(0,1,1,0) \to (0,1,1,1).$$

A solution can be better than all one-bit changes but still far worse than another solution that requires flipping many bits at once.

For example, suppose we minimize a cost over four binary variables. A local search algorithm tries flipping one bit at a time. It may find that every one-bit flip makes the solution worse. But perhaps flipping three bits together would produce a much better solution. The algorithm is stuck because its allowed local moves cannot cross the barrier.

This “barrier” language is not just metaphorical. In optimization, we often picture the objective function as an energy landscape. Low objective values are low-energy states. High objective values are high-energy states. Moving from one good solution to another may require passing through worse intermediate solutions.

This energy-landscape viewpoint will become central in quantum annealing and Ising models. Quantum optimization often tries to use quantum dynamics to search such landscapes differently from ordinary local classical moves. But the landscape itself comes from the classical optimization model.

2.12 Combinatorial optimization: choosing among discrete structures

A combinatorial optimization problem is an optimization problem where the feasible solutions are discrete objects such as subsets, permutations, matchings, paths, schedules, assignments, or graphs.

The word combinatorial comes from combinations: ways of arranging or selecting objects.

Here are some examples.

In a subset selection problem, we choose some items from a larger set. Portfolio selection can have this form: choose which assets to include.

In an assignment problem, we assign items to positions. For example, assign workers to shifts.

In a routing problem, we choose a path or tour through a network. Delivery planning often has this form.

In a scheduling problem, we assign jobs to times and machines.

In a graph problem, we optimize over structures involving nodes and edges. For example, in Max-Cut, we divide the nodes of a graph into two groups and try to maximize the number or weight of edges crossing between the groups.

Combinatorial optimization is a major area of classical optimization and theoretical computer science, with standard problems including traveling salesperson, matching, set cover, satisfiability, graph coloring, and many others (Papadimitriou and Steiglitz, 1998).

The key feature is that the number of possible solutions often grows very quickly.

Suppose we have n yes-or-no decisions. Then there are 2^n possible binary strings.

Suppose we must order n cities in a route. The number of possible orderings is

$$n! = n(n-1)(n-2)\cdots 2\cdot 1.$$

The symbol n! is pronounced “n factorial.”

For n=5,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

For n=20,

$$20! = 2,432,902,008,176,640,000.$$

That is more than two quintillion.

This growth makes direct enumeration impossible for large n. Enumeration means checking every possible solution one by one.

However, we must be careful. A large number of possible solutions does not automatically mean a problem is hard. Some problems have enormous solution spaces but can still be solved efficiently using clever algorithms.

For example, shortest path problems in graphs can be solved efficiently by algorithms such as Dijkstra's algorithm under appropriate nonnegative edge-weight conditions, even though the number of possible paths can be very large. The deeper question is whether there is exploitable structure.

Many important combinatorial optimization problems are difficult because no known algorithm can solve all instances efficiently, and complexity theory gives strong evidence that some of them are fundamentally hard in the worst case (Garey and Johnson, 1979).

2.13 A first look at computational complexity

Computational complexity studies how the resources needed by an algorithm grow as the input size grows.

The most common resource is time: how many elementary computational steps are needed?

Another important resource is memory: how much storage is needed?

The input size is the amount of information needed to describe the problem instance. A problem instance is one specific version of a problem.

For example, "traveling salesperson problem" is a general problem. A specific list of 30 cities and distances between them is an instance.

An algorithm is said to run in polynomial time if its running time is bounded by a polynomial in the input size, such as

$$n^2, \quad n^3, \quad \text{or} \quad n^{10}.$$

An algorithm runs in exponential time if its running time grows like something such as

$$2^n$$

or

$$1.5^n.$$

Polynomial time is usually treated as a rough mathematical model of efficient computation, while exponential time is usually treated as infeasible for large inputs. This distinction is foundational in complexity theory, though real performance also depends on constants, hardware, input structure, and implementation details (Garey and Johnson, 1979).

The class P contains decision problems that can be solved in polynomial time.

A decision problem is a problem with a yes-or-no answer.

For example, instead of asking:

> What is the shortest route?

we may ask:

> Is there a route of length at most 100?

The class NP contains decision problems where, if the answer is yes, a proposed solution can be checked in polynomial time.

For example, if someone gives us a route, we can add its distances and check whether its length is at most 100. Checking a proposed route is much easier than finding the best route from scratch.

A problem is NP-hard if it is at least as hard as every problem in NP, in a precise technical sense based on reductions. A reduction is a method for transforming one problem into another so that solving the second would let us solve the first.

A problem is NP-complete if it is both in NP and NP-hard.

Richard Karp's 1972 paper showed that many natural combinatorial problems are NP-complete by using reductions from one problem to another, helping establish NP-completeness as a central concept in theoretical computer science (Karp, 1972).

For optimization, we often say a problem is NP-hard when its decision version is NP-complete or when solving the optimization problem exactly would allow us to solve an NP-complete decision problem.

This is the key practical meaning:

> If a problem is NP-hard, we do not know any algorithm that solves every instance exactly in polynomial time.

This does not prove that no such algorithm exists. The famous question P = NP asks whether every problem whose solution can be checked quickly can also be solved quickly. This remains open (Garey and Johnson, 1979).

So we must speak carefully:

- NP-hard does not mean "impossible."
- NP-hard does not mean "every instance is difficult."
- NP-hard does not mean "small cases cannot be solved."
- NP-hard does not mean "heuristics are useless."
- NP-hard means that, in the worst case, the problem belongs to a class for which no general efficient exact algorithm is known, and for which there is strong evidence of deep computational difficulty.

This careful language is essential for quantum optimization. Quantum computers do not automatically make NP-hard problems easy. Later, we will examine what kinds of speedups are known, what kinds are hoped for, and what remains unproven.

2.14 Exact algorithms, heuristics, and approximation

Once a problem is hard, what do we do?

We still solve real problems every day. We just need to be clear about what "solve" means.

An exact algorithm guarantees an optimal solution if given enough time and memory.

For example, branch-and-bound methods for integer programming can prove optimality by systematically ruling out parts of the search space. These methods are foundational in integer optimization and are widely discussed in classical treatments of integer and combinatorial optimization (Nemhauser and Wolsey, 1988).

A heuristic is a method designed to find good solutions, but without a guarantee that it will always find the best solution.

For example, a local search heuristic for a scheduling problem might start with a random schedule, then repeatedly make small changes that improve it. This may work well in practice, but it can get stuck in local optima.

An approximation algorithm gives a solution with a provable quality guarantee for a class of problems.

For example, for a minimization problem, an approximation algorithm might guarantee a solution whose cost is at most twice the optimal cost. Approximation algorithms form a major part of combinatorial optimization, especially when exact optimization is too expensive (Papadimitriou and Steiglitz, 1998).

A metaheuristic is a general high-level search strategy that can be adapted to many problems. Examples include simulated annealing, genetic algorithms, tabu search, and other broad search frameworks. These will appear in Chapter 11 when we discuss classical baselines.

For quantum optimization, this distinction matters deeply.

A quantum method may be:

- an exact algorithm for a special problem class,
- a heuristic that often finds good solutions,
- an approximate method with a provable guarantee,
- or a hybrid method combined with classical optimization.

We should not compare these unfairly. A heuristic quantum algorithm should be compared against strong heuristic classical algorithms, not against a weak straw-man baseline. Honest benchmarking will be a major theme later.

2.15 The modeling trade-off: realism versus solvability

Optimization modeling is the art of choosing a mathematical representation that is realistic enough to matter and structured enough to solve.

If the model is too simple, it may ignore important reality.

If the model is too detailed, it may become impossible to solve or too expensive to use.

Suppose we are modeling delivery routing. A simple model may minimize total distance. But real delivery planning may also include:

- vehicle capacity,
- driver shift limits,
- traffic patterns,
- delivery time windows,
- fuel costs,
- customer priority,
- road restrictions,
- uncertain travel times.

Adding these details can make the model more useful. It can also make it harder.

This creates a central modeling question:

> Which details must be included as constraints, which can be included as penalties, and which can be ignored for the current decision?

There is no universal answer. It depends on the purpose of the model.

A model used for rough planning may be simple. A model used for safety-critical scheduling may need stricter constraints. A model used inside a real-time system may need to sacrifice some accuracy for speed.

Quantum optimization does not remove this modeling trade-off. In fact, current quantum hardware often makes it sharper because problems must be encoded into limited numbers of qubits with specific connectivity and noise constraints

Document information

Chapter 2: Classical Optimization Foundations

Project	Quantum Optimization from First Principles
Document	Document 1.6
Author	amnanoor
Verifier	Not verified
Downloaded	July 08, 2026 10:28 KST
Status	Working
Document link	https://theorytrace.com/projects/quantum-optimization-from-first-principles/documents-/chapter-2-classical-optimization-foundations/