How Quantum Computing Helps to Optimize

**Optimization problem: ideal case.** We have an unsorted array of \( n \) elements \( a_1, \ldots, a_n \). We need to find the largest element.

**Optimization problem: ideal case – precise formulation of the problem.**

- We have a black-box algorithm \( a \) that, given a natural number \( i \) from 1 to \( n \), returns the value \( a_i \) of the objective function corresponding to the \( i \)-th alternative.
- We want to find an index \( i \) for which the objective function attains its largest possible value, i.e., for which \( a_i = \max_j a_j \).

**Optimization problem: practical case.** In practical applications, we know that these elements are located in some known interval \( [A, \bar{A}] \). For example, if we compare weights of different people, we know that the weight cannot be smaller than 0 or larger than 500 kg (probably even smaller).

Also, when the difference between the values is very small, smaller than some number \( \varepsilon > 0 \), then we do not distinguish between the values – such small differences can be caused by measurement errors. So, it is sufficient to find the value \( a_i \) for which \( a_i \) is \( \varepsilon \)-close to the maximum, i.e., for which \( |a_i - \max_j a_j| \leq \varepsilon \).

**Optimization problem: practical case – precise formulation of the problem.** We are given:

- a black-box algorithm \( a \) that, given a natural number \( i \) from 1 to \( n \), returns the value \( a_i \) of the objective function corresponding to the \( i \)-th alternative;
- real numbers \( A \) and \( \bar{A} \) for which \( a_i \in [A, \bar{A}] \) for all \( i \), and
- a real number \( \varepsilon > 0 \).

We want to find an index \( i \) for which the value \( a_i \) of the objective function is \( \varepsilon \)-close to the maximum, i.e., for which \( |a_i - \max_j a_j| \leq \varepsilon \).

**What happens in non-quantum computing.** In non-quantum computing, we need at least \( n \) steps to solve this problem.
Indeed, we need to consider all \( n \) elements of the array: if we miss one element, maybe this is the element which is larger than every other one.

**Quantum computing can make it faster.** We can use Grover’s algorithm to find the largest element faster. The corresponding algorithm is iterative.

On each iteration, we generate an interval \([a, \overline{a}]\) that contains the largest value \( \max_j a_j \). On each iteration, the width of this interval will decrease by half.

The main idea behind this algorithm is known as **bisection**.

Let us describe this algorithm.

**How we start.** We start with the interval \([a, \overline{a}] = [A, \overline{A}]\), i.e., with \( a = A \) and \( \overline{a} = \overline{A} \).

**Iteration.** On each iteration:

- first, we compute the midpoint \( m = \frac{a + \overline{a}}{2} \);
- then, we apply Grover’s algorithm to check whether there exists an element \( a_i \) for which \( a_i > m \).

Then:

- if such an element exists, i.e., if \( a_i > m \) for some element \( i \), this implies that \( \max_j a_j > m \); so we can replace the lower bound \( a \) for the maximum with \( m \) and consider a new interval \([m, \overline{a}]\);
- if such an element does not exist, this means that \( a_i \leq m \) for all \( i \); this implies that \( \max_j a_j \leq m \); so we can replace the upper bound \( \overline{a} \) for the maximum with \( m \) and consider a new interval \([a, m]\).

In both cases, we decrease the width of the interval by half.

**What does Grover’s algorithm do: reminder.** Grover’s quantum algorithm finds an element with a given property in an unsorted array of \( n \) elements. It takes \( \sqrt{n} \) steps.

What if there is no such element? Grover’ algorithm always returns something. All we need to do is to check that the element returned by Grover’s algorithm satisfies the given property. If it doesn’t, this means that there in the array, there is no element with this property.

In our new algorithm, we use Grover’s algorithm to find an element \( a_i \) that satisfies the property \( a_i > m \).

**When do we stop iterations and what do we do then: analysis of the problem.** We stop iterations when we get \( \overline{a} - a \leq \varepsilon \).

Each time we applied Grover’s algorithm, if we ended up with a new lower bound \( a \), this means that Grover’s algorithm has found some index \( i \) for which \( a_i \) is larger than this new lower bound.

So, if the final lower bound \( a \) of the final interval is different from the original lower bound \( A \), this means that this value \( a \) was introduced on some iteration,
and thus, on that iteration, we have found an index $i$ for which $a_i > a$. We know that $a - a_i \leq \varepsilon$, thus $a \leq a_i + \varepsilon$. We also know that $\max_j a_j \leq a$, hence
\[
a \leq a_i \leq \max_j a_j \leq a + \varepsilon.
\]
Both $a_i$ and $\max_j a_j$ are therefore located on the interval $[a, a + \varepsilon]$ of width $\varepsilon$, hence
\[
|a_i - \max_j a_j| \leq \varepsilon.
\]
Let us now consider the case when the lower bound $a$ of the final interval is equal to the original lower bound $a = A$. Since $A \leq a_i$ for all $i$, we can thus conclude that in this case, for all $i$, we have
\[
a \leq a_i \leq \max_j a_j \leq a \leq a + \varepsilon.
\]
So, for each index $i$, both $a_i$ and $\max_j a_j$ are therefore located on the interval $[a, a + \varepsilon]$ of width $\varepsilon$. Thus, for any index $i$, the corresponding value $a_i$ satisfies the desired inequality $|a_i - \max_j a_j| \leq \varepsilon$.

**When do we stop iterations and what do we do then: algorithm.** We stop iterations when we get $a - a \leq \varepsilon$; then:

- If $a \neq A$, this means that the lower bound $a$ was introduced on some iteration. It was introduced because on this iteration, Grover’s algorithm returned an index $i$ for which $a_i > a$. This index $i$ – and the corresponding element $a_i$ – are then returned as a solution to our optimization problem.

- If $a = A$, then we return any index $i$ and the corresponding element $a_i$.

**How much time does this algorithm take.** We start with an interval of width $w \equiv A - A$. On each step, the width is decreases in half. So:

- after the first iteration, the width is $w/2$;
- after the second iteration, the width is $w/2^2$;
- \ldots
- after the $k$-th iteration, the width is $w/2^k$.

How many iterations do we need? We need the smallest $k$ for which $w/2^k \leq \varepsilon$, i.e., for which $2^k \geq w/\varepsilon$. By taking the binary logarithm of both sides, we conclude that $k \geq \log_2(w/\varepsilon)$. Since the number of iterations is an integer, $k$ must be the smallest integer which is larger than the number $\log_2(w/\varepsilon)$. Such smallest integer is known as the ceiling of a number and denoted by
\[
k = \lceil \log_2(w/\varepsilon) \rceil.
\]
It is important to notice that this number does not depend on how many elements we have in an array, it is a constant.

For each of these $k$ iterations, we need to apply Grover’s algorithm. Each application requires time $O(\sqrt{n})$, so overall time is constant times $O(\sqrt{n})$, i.e., still $O(\sqrt{n})$.

**Example: description.** Suppose that $A = 0$, $\overline{A} = 16$, and $\varepsilon = 1$.

**How many iterations do we need?** Here,

$$w = \overline{A} - A = 16 - 0 = 16,$$

so $w/\varepsilon = 16$. Here, $16 = 2^4$, so by definition of logarithm,

$$\log_2(w/\varepsilon) = \log_2(16) = 4.$$

Thus, we need $k = 4$ iterations.

**How many times do we need to apply Grover’s algorithm?** In general, we apply it $k$ times, so in this case, we need to apply Grover’s algorithm 4 times.

**Tracing the algorithm: preliminary step.** Let us trace iterations on the example when the largest element is $a_3 = 5.1$. We start with the interval $[a, \overline{a}] = [A, \overline{A}] = [0, 16]$.

**First iteration.** On the first iteration:

- first, we compute the midpoint $m = \frac{a + \overline{a}}{2} = \frac{0 + 16}{2} = 8$;
- then, we apply Grover’s algorithm to check whether there exists an element $a_i$ for which $a_i > m = 8$.

There is no such element, so we replace $a = 16$ with $m = 8$. The resulting interval is now $[a, \overline{a}] = [0, 8]$.

**Second iteration.** On the second iteration:

- first, we compute the midpoint $m = \frac{a + \overline{a}}{2} = \frac{0 + 8}{2} = 4$;
- then, we apply Grover’s algorithm to check whether there exists an element $a_i$ for which $a_i > m = 4$.

Such elements exist — e.g., the element $a_3$, so we replace $a = 0$ with $m = 4$. The resulting interval is now $[a, \overline{a}] = [4, 8]$.

**Third iteration.** On the third iteration:

- first, we compute the midpoint $m = \frac{a + \overline{a}}{2} = \frac{4 + 8}{2} = 6$;
- then, we apply Grover’s algorithm to check whether there exists an element $a_i$ for which $a_i > m = 6$. 


There is no such element, so we replace $\overline{a} = 8$ with $m = 6$. The resulting interval is now $[a, \overline{a}] = [4, 6]$.

**Fourth iteration.** On the fourth iteration:

- first, we compute the midpoint $m = \frac{a + \overline{a}}{2} = \frac{4 + 6}{2} = 5$;
- then, we apply Grover’s algorithm to check whether there exists an element $a_i$ for which $a_i > m = 5$.

Such elements exist – e.g., the element $a_3$, so we replace $a = 4$ with $m = 5$. The resulting interval is now $[a, \overline{a}] = [5, 6]$.

Now, we have $\overline{a} - a = 6 - 5 \leq \varepsilon = 1$, so we stop iterations.

**Final step.** On the fourth iteration, Grover’s algorithm not only concluded that there is an index $i$ for which $a_i > 5$, it also generated one such index. This index $i$ – and the corresponding element $a_i$ – are then returned as the result of the quantum optimization algorithm.