CS 599 P1: Introduction to Quantum Computation — Boston University, Fall 2025
Instructor: Alexander Poremba — **Scribe:** Kabir Peshawaria

LECTURE # 9: SIMON'S ALGORITHM

1 Introduction to Simon's Problem

Last time, we showed that quantum computers can solve *certain* problems faster than classical computers. Specifically, we discussed a model where algorithms were given query access to "black-box oracles" for some Boolean function $f : \{0,1\}^n \to \{0,1\}$, and the task was to determine whether f was constant or balanced. We saw that deterministic classical computers required $2^{n-1}+1$ many oracle queries to find the answer with probability 1. Still, this was slightly unsatisfying: with only a handful of queries, a randomized classical algorithm can determine whether f is balanced or constant with error probability less than 0.01.

In this lecture, we will encounter Simon's algorithm, which will give us a *true* exponential speedup, *even when compared against randomized algorithms that are allowed a small error probability*.

Simon's Problem

Given oracle access to a Boolean function $f : \{0,1\}^n \to \{0,1\}^n$ with the promise that there exists a Boolean string $s \in \{0,1\}^n$ with $s \neq 0^n$ such that, for every pair of inputs $x, y \in \{0,1\}^n$,

$$f(x) = f(y) \iff x \oplus y = s,$$

the task is to output the hidden string s.

Remark 1. *In other words, the function f in an instance of Simon's problem is always 2-to-1 and* periodic *with periodicity $s \in \{0,1\}^n$ since $f(x) = f(x \oplus s)$ for all inputs $x \in \{0,1\}^n$.*

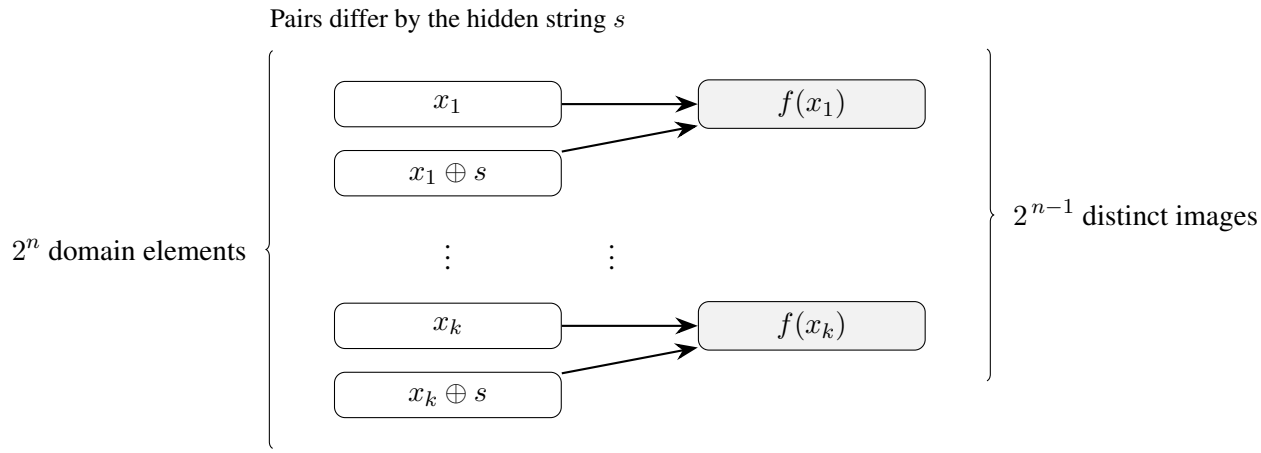

Figure 1: A 2-to-1 function for Simon's problem: each image $f(x)$ has exactly two preimages, x and $x \oplus s$.

Example: 3-bit Simon's Problem.

Consider the Boolean function $f : \{0,1\}^3 \to \{0,1\}^3$ with the following truth table:

x	f(x)
000	101
001	010
010	000
011	110
100	000
101	110
110	101
111	010

To find hidden string $s \in \{0,1\}^n$, let's look for a collision. We see that $f(000) = 101 = f(110)$. Therefore $s = 000 \oplus 110 = 110$. However, we could have just as well used any collision; for example, $f(011) = 110 = f(101)$, so $s = 011 \oplus 101 = 110$.

2 Simon's Problem and Classical Computation

Before we look at quantum algorithms, let's settle how difficult Simon's problem is for a classical computer.

2.1 Upper Bounds

As we just discussed, a simple strategy is to find any collision $f(x) = f(x')$. Given such an x, x' pair, we are done; output $s = x \oplus x'$. So, how difficult is it for a classical algorithm to find a collision in f?

Deterministically, we can find a collision by querying any $2^{n-1}+1$ inputs. By the pigeonhole principle, there must be a collision as the image of f, denoted $\mathsf{Image}(f)$, has size 2^{n-1}. We can do better.

Remark 2. *Advice to reader: if you are not deeply familiar with Big O notation, please refer to the appendix of these lecture notes before continuing.*

Using randomness Using randomness, we can find a collision with probability $\frac{3}{4}$ using only $O(2^{n/2})$ queries! This follows by the *Birthday paradox*, which states that if we pick around $\Omega(\sqrt{N})$ items uniformly at random with replacement from a set of size N, we will pick the same item twice with probability at least $\frac{3}{4}$. In our setting, the set of items is the image of function f (i.e. the set of all 2^{n-1} possible outputs). We pick an item y uniformly at random by picking $x \in_R \{0,1\}^n$ uniformly at random (with replacement) and evaluating $y = f(x)$. After $\Omega(2^{(n-1)/2}) = \Omega(2^{n/2})$ queries, we have seen y twice with high probability, and can take s to be the XOR of the preimages of y.

There is a small technicality in our argument above, as it could be the case that we saw some $y \in \mathsf{Image}(f)$ twice, but *also the same preimage* $x \in \{0,1\}^n$ twice. This is solved by recognizing that it is equally likely to see the same preimage twice as it is to see two different preimages, so if we see k pairs of collisions, with probability $1 - (\frac{1}{2})^k$, we saw a true collision.

Without randomness In fact, we do not even need randomness! There is a deterministic algorithm that makes $O(2^{n/2})$ queries and finds s without any error loss. For simplicity, assume $n = 2m$, i.e. n is even. Then let $S_1 = \{(x||0^m) : x \in \{0,1\}^m\}$ and $S_2 = \{(0^m||y) : y \in \{0,1\}^m\}$, where $||$ denotes string concatenation. Take $S = S_1 \cup S_2$. This set has size $\leq 2 \cdot 2^m = 2^{(n/2)+1}$. For every $s \in \{0,1\}^n$, we can write $s = (s_1||s_2) = (s_1||0^m) \oplus (0^m||s_2)$.

2.2 Lower Bounds

We will now show that any classical algorithm (even randomized) that succeeds in finding $s \in \{0,1\}^n$ with probability at least $\Omega(1)$ requires exponentially many queries to f.

Lemma 2.1. *A (possibly randomized) classical algorithm for Simon's problem with error probability at most $\frac{1}{4}$ requires $\Omega(2^{n/2})$ queries to the black-box oracle for f.*

This proof sketch is a bit involved, so feel free to skip it. The main takeaway is that *classical algorithms require an exponential number of queries to solve Simon's problem.*

Proof. Let's say that a classical algorithm makes q queries, $x_1, \ldots, x_q \in \{0,1\}^n$ to the oracle for function f. The only way the algorithm can learn the hidden string s is by finding a collision (finding two distinct inputs $x_i \neq x_j$ such that $f(x_i) = f(x_j)$. If it does, it found s, since $s = x_i \oplus x_j$.

Let's define the set of all possible collision-producing differences from the queries made:

$$D := \{x_i \oplus x_j \mid 1 \leq i < j \leq q\}.$$

A collision is found if and only if the true hidden string s happens to lie in this set D. The size of this set is at most $|D| \leq \binom{q}{2} < \frac{q^2}{2}$.

Now, assume the hidden string s is chosen uniformly at random from the $2^n - 1$ possible non-zero strings. The probability of finding a collision is the probability that s falls into our set D:

$$P(\text{find collision}) = \frac{|D|}{2^n - 1} \leq \frac{q^2/2}{2^n - 1}.$$

What if the algorithm *does not* find a collision? This happens with high probability if q is small. In this case, the algorithm receives q distinct function values. The key insight is as follows. This set of q query-answer pairs is consistent with every potential period $s' \notin D$. The algorithm has absolutely no information to distinguish the true period s from any other candidate in the set of possibilities $\{0,1\}^n \setminus (D \cup \{0^n\})$.

So, if no collision is found, the algorithm's best chance is to guess one of the remaining possibilities. The number of such possibilities is at least $(2^n - 1) - |D|$. The probability of guessing correctly is therefore:

$$P(\text{succeed without collision}) \leq \frac{1}{(2^n - 1) - |D|}.$$

The total probability of success is bounded by the sum of the probabilities of these two disjoint events (finding a collision, or succeeding by guessing without one):

$$P(\text{Success}) \leq P(\text{find collision}) + P(\text{succeed without collision}) \approx \frac{q^2/2}{2^n} + \frac{1}{2^n - q^2/2}.$$

For the total success probability to be constant, the first term, $\frac{q^2/2}{2^n}$, must be a constant. (The second term is exponentially small and negligible). For this term to be a constant $\Omega(1)$:

$$\frac{q^2}{2^n} = \Omega(1) \implies q^2 = \Omega(2^n) \implies q = \Omega(2^{n/2}).$$

Therefore, any classical algorithm needs $\Omega(2^{n/2})$ queries to succeed with constant probability. □

Remark 3 (Note on Boosting)**.** *The reason it suffices to consider constant success probability is because of a technique called* boosting*. If you have a randomized algorithm that has error probability less than $p \in (0, 1)$ for a problem where you can easily verify if the answer is correct, you can run the algorithm multiple times to decrease the error probability. If you run the algorithm t times, your error probability becomes p^t. So, for any arbitrarily desired small constant error probability, you only need to repeat the algorithm $O(1)$ times to achieve this.*

3 Simon's (Quantum) Algorithm

We are now going to show Simon's algorithm, a classical-quantum hybrid algorithm that only requires $O(n)$ many calls to f to find s with probability at least ≈ 0.288. This is an impressive speedup from the exponential many queries required by a classical computer, and it comes from cleverly exploiting the periodicity and structure promised in the function $f : \{0, 1\}^n \to \{0, 1\}^n$.

Simon's 1994 algorithm directly inspired Peter Shor, who recognized that many number-theoretic problems, including factoring and discrete logarithms, could also be recast as hidden-period problems.

Overview The algorithm proceeds in two phases. In phase one, we run a quantum subroutine $n - 1$ times. Each subroutine will make one call to the oracle U_f. This will output $n-1$ strings $y^{(1)}, \ldots, y^{(n-1)} \in \{0, 1\}^n$ where each $y^{(i)}$ satisfies $\sum_{j=0}^{n-1} y_j^{(i)} \cdot s_j \equiv 0 \pmod 2$. In phase two, we run some classical post-processing. We write an $n - 1 \times n$ dimensional matrix M where the ith row, denoted M_i, is the string $y^{(i)} \in \{0, 1\}^n$

3.1 Phase One: Quantum Subroutine

Recap of U_f unitary and logical registers We remind the reader that U_f is the unitary that implements the Boolean function $f : \{0, 1\}^n \to \{0, 1\}^n$. Specifically, U_f acts on $2n$ qubits in a computational basis state $|x\rangle \otimes |y\rangle$ for $x, y \in \{0, 1\}^n$ like so:

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle .$$

Sometimes, it's nice to logically group qubits into registers. In the above, we call the first n qubits the "input register" and the last n qubits the "output register". This grouping makes sense, since if you wanted to know what $f(x)$ is for a particular $x \in \{0, 1\}^n$, you could prepare the computational basis state $|x\rangle |0\rangle$, apply the U_f unitary, and measure the output register, which will always have amplitude 1 on computational basis state $|f(x)\rangle$.

For illustration, let us suppose the function is of the form $f : \{0, 1\}^3 \to \{0, 1\}^3$. The first three qubits correspond to the input register and the last three to the output register. The value $y \in \{0, 1\}^3$ is the measurement outcome of the input register.

Let us analyze this circuit (generalized to work over $2n$ qubits) step by step.

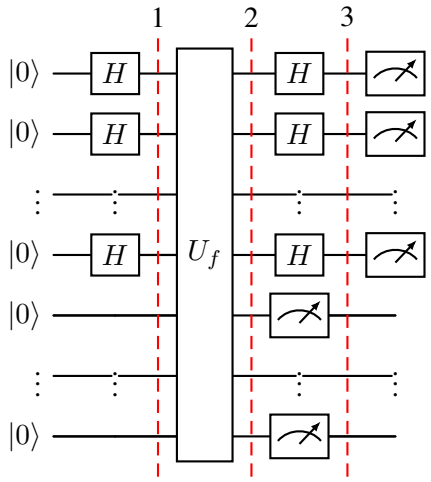

Figure 2: Simon's algorithm.

Step 1 After applying $H^{\otimes n} \otimes I^{\otimes n}$ to the state $|\phi_0\rangle := |0^n\rangle\,|0^n\rangle$, the input register is transformed to the uniform superposition.

$$|\phi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle_I \otimes |0^n\rangle_O \,.$$

Step 2 After applying U_f to the state $|\phi_1\rangle$, we get the following:

$$|\phi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle_I \otimes |f(x)\rangle_O$$

Step 3 We can logically see step three as an measurement on the output register and a unitary operator acting on the input register. Since these actions are local to each register, they can be performed in any order. It makes the analysis a little simpler to consider measuring the output register first, yielding $f(z) \in \{0,1\}^n$ for some $z \in \{0,1\}^n$. By the 2-to-1 property of the periodic function $f : \{0,1\}^n \to \{0,1\}^n$, with probability $\frac{1}{2^n}$ we will get the post-measurement state of

$$\frac{1}{\sqrt{2}} \left(|z\rangle_I + |z \oplus s\rangle\right) \otimes |f(z)\rangle$$

Let's forget about the output register, and let $z \in \{0,1\}^n$ (we have two choices, we can choose arbitrarily) be such that our input register is now in the state $\frac{1}{\sqrt{2}} \left(|z\rangle + |z \oplus s\rangle\right)$.

We now apply $H^{\otimes n}$ to the input register. What is the new state? Let's break this down. First what happens when we apply $H^{\otimes n}$ to the computational basis state $|z\rangle$? Recall by definition of H:

- $H\,|0\rangle = \frac{1}{\sqrt{2}} \sum_{x\in\{0,1\}} (-1)^{x\cdot 0}\,|x\rangle$.
- $H\,|1\rangle = \frac{1}{\sqrt{2}} \sum_{x\in\{0,1\}} (-1)^{x\cdot 1}\,|x\rangle$.

Therefore,

$$H^{\otimes n}|z\rangle = \frac{1}{\sqrt{2^n}} \sum_{y_1\in\{0,1\}} \sum_{y_2\in\{0,1\}} \dots \sum_{y_n\in\{0,1\}} (-1)^{z_1\cdot y_1}(-1)^{z_2\cdot y_2}\dots(-1)^{z_n\cdot y_n}|y_1y_2\cdots y_n\rangle .$$
$$= \frac{1}{\sqrt{2^n}} \sum_{y\in\{0,1\}^n} (-1)^{\langle y,z\rangle}|y\rangle .$$

This means that our state after step 3 is:

$$|\phi_3\rangle = \frac{1}{\sqrt{2}}\cdot\frac{1}{\sqrt{2^n}}\left(\sum_{y\in\{0,1\}^n}(-1)^{\langle y,z\rangle}|y\rangle + (-1)^{\langle y,z\oplus s\rangle}|y\rangle\right)$$
$$= \frac{1}{\sqrt{2^{n+1}}}\left(\sum_{y\in\{0,1\}^n}(-1)^{\langle y,z\rangle}(1+(-1)^{\langle y,s\rangle})|y\rangle\right)$$

Step 4 Now we measure the state; $\forall y \in \{0,1\}^n$ such that $\langle y, s\rangle \equiv 1 \pmod 2$, the state $|\phi_4\rangle$ has amplitude 0 on basis state $|y\rangle$. For every other $y \in \{0,1\}^n$, the amplitude is $(-1)^{\langle y,z\rangle}\cdot 2\cdot\frac{1}{\sqrt{2^{n+1}}} = (-1)^{\langle y,z\rangle}\cdot\frac{1}{\sqrt{2^{(n-1)}}}$.

Conclusion of Quantum Subroutine For every $y \in \{0,1\}^n$ such that $\langle y, z\rangle \equiv 0 \pmod 2$, it is measured with probability $\frac{1}{2^{n-1}}$. Note that there are exactly 2^{n-1} such y's, so they are all occurring with equal probability.

3.2 Phase Two

At this point, we have run the quantum subroutine $n-1$ times and received $n-1$ strings $y^{(1)},\dots,y^{(n-1)} \in \{0,1\}^n$ such that $\langle y^{(i)}, s\rangle \equiv 0 \pmod 2$ for all i.

Detour: Linear Algebra over $\mathbb{F}_2$ To fully understand this part, we need to depart from our nice linear algebra over Hilbert spaces and think about the *binary field*. The set $\{0,1\}$ equipped with addition $\pmod 2$ and multiplication $\pmod 2$ forms a *field*, denoted $\mathbb{F}_2$. For ease of notation, we will now just call this field $\mathbb{F}$. The familiar n dimensional vector space over $\mathbb{F}$ is called $\mathbb{F}^n$. If we view our vectors $y^{(1)},\dots,y^{(n-1)}, s \in \mathbb{F}^n$ as elements (read: column vectors) of the vector space $\mathbb{F}^n$, then we can say that that $y^\top s = 0$ for every $y \in \{y^{(1)},\dots,y^{(n-1)}\}$, where this is an $\mathbb{F}$-matrix multiplication.

Description of the phase We will construct a matrix $M \in \mathbb{F}^{(n-1)\times n}$ on $n-1$ rows and n columns with entries living in $\mathbb{F}$. Specifically, the ith row of M is the string $y^{(i)}$. Using Gaussian Elimination, we will find a nonzero solution $x \in \{0,1\}^n$ to the linear system of $n-1$ equations $Mx = \mathbf{0}$, where $\mathbf{0}$ is the all zero vector in $\mathbb{F}^{n-1}$. We will output this value.[1]

Detour: The Rank-Nullity Theorem Recall that for a matrix $A \in \mathbb{F}^{k\times n}$ with rank $r = \text{rank}(A)$, its nullspace (or kernel), $\ker(A) = \{v \in \mathbb{F}^n : Av = \mathbf{0}\}$, is a subspace of $\mathbb{F}^n$. The Rank-Nullity theorem states that $\text{rank}(A) + \dim(\ker(A)) = n$. Therefore, the dimension of the nullspace is $n - r$.

[1] If we are using boosting, we may want to check whether $f(0) = f(x)$ to verify whether this iteration of the entire algorithm got the correct answer.

Intuition for the Analysis. We claim that if the vectors $y^{(1)}, \ldots, y^{(n-1)}$ are linearly independent over $\mathbb{F}_2$, then our algorithm is guaranteed to output s. Let's see why.

If the $y^{(i)}$ vectors (the rows of M) are linearly independent, then the matrix M has full rank, i.e., $\text{rank}(M) = n - 1$. By the Rank-Nullity theorem, the dimension of its nullspace is:

$$\dim(\ker(M)) = n - \text{rank}(M) = n - (n-1) = 1$$

A one-dimensional vector space over the field $\mathbb{F}$ contains exactly $|\mathbb{F}|^1 = 2^1 = 2$ vectors. These vectors are the trivial solution, $\mathbf{0} \in \mathbb{F}^n$, and one other unique non-zero vector: the hidden string $s \in \{0,1\}^n$.

3.3 Analysis of Success Probability

The final question to ask is: what is the probability that the $n-1$ vectors we sample are linearly independent?

We are sampling vectors y uniformly at random subject to the constraint that $y^\top s = 0$. This means we are sampling from the $(n-1)$-dimensional subspace of $\mathbb{F}^n$ orthogonal to the one-dimensional subspace $\{0, s\}$ that is spanned by s. For notational convience, let's introduce the integer

$$m := n - 1.$$

Our task is to find the probability that m vectors chosen uniformly at random from an m-dimensional vector space over $\mathbb{F}$ are linearly independent.

We can calculate this by finding the probability that each new vector is linearly independent to the preceding ones.

- The first vector, $y^{(1)}$, is linearly independent as long as it's not the zero vector. Since there are 2^m total vectors in our sampling space, the probability of this is:

$$P(y^{(1)} \neq \mathbf{0}) = \frac{2^m - 1}{2^m}$$

- Given that $y^{(1)}, \ldots, y^{(i)}$ are linearly independent, they span an i-dimensional subspace which contains 2^i vectors. For the next vector, $y^{(i+1)}$, to be linearly independent from the previous ones, it must not lie in this span. There are $2^m - 2^i$ such vectors.

  The conditional probability is therefore:

$$P(y^{(i+1)} \notin \text{span}\{y^{(1)}, \ldots, y^{(i)}\}) = \frac{2^m - 2^i}{2^m}$$

The total probability of finding a linearly independent set of m vectors is the product of these probabili-

ties for i from 0 to $m-1$:

$$\begin{aligned}
P(\text{all } m \text{ are linearly independent}) &= \prod_{i=0}^{m-1} \frac{2^m - 2^i}{2^m} \\
&= \left(\frac{2^m-1}{2^m}\right)\left(\frac{2^m-2}{2^m}\right)\left(\frac{2^m-4}{2^m}\right)\cdots\left(\frac{2^m-2^{m-1}}{2^m}\right) \\
&= \prod_{i=0}^{m-1}\left(1-\frac{2^i}{2^m}\right) \\
&= \prod_{k=1}^{m}\left(1-\frac{1}{2^k}\right) \quad \text{(re-indexing with } k = m-i\text{)} \\
&\geq \prod_{k=1}^{\infty}\left(1-\frac{1}{2^k}\right) \\
&\approx 0.2887.
\end{aligned}$$

So, one iteration of Simon's algorithm succeeds with probability at least 0.28, which (for any $\epsilon > 0$) can be boosted to success probability $1-\epsilon$ by repeating the algorithm $O(1)$ times.

Appendix: Big O Notation

For readers who might not be familiar with it, this section is a quick refresher on Big O notation. In short, it's a way to describe the long-term behavior of functions, which is useful for analyzing algorithms.

Big O: Asymptotic Upper Bound

We use **Big O** notation to describe an **upper bound** on a function's growth rate. If we say $f(n) = O(g(n))$, we mean that for all sufficiently large n, $f(n)$ is "overpowered" by some constant multiple of $g(n)$.

Definition 3.1. *For functions $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, we have*

$$f(n) \leq c \cdot g(n).$$

For example, when analyzing polynomials, only the highest-degree term matters asymptotically.

- $3n = O(n^2)$
- $3n^2 + 500n + 312 = O(n^2)$
- $\log n = O(n)$

Big Omega (Ω): Asymptotic Lower Bound

Big Omega provides a **lower bound**. If $f(n) = \Omega(g(n))$, it means that $f(n)$ grows at least as fast as some constant multiple of $g(n)$ for all large n.

Definition 3.2. *For functions $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$, we say $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, we have*

$$f(n) \geq c \cdot g(n).$$

For example:

- $3n^2 = \Omega(n^2)$
- $3n^2 + 500n + 312 = \Omega(n \log^{999} n)$
- $n^{0.00001} = \Omega(\log^{99999}(n) + 999999999)$

Big Theta (Θ): Asymptotic Tight Bound

Big Theta gives us a **tight bound**. If $f(n) = \Theta(g(n))$, it means $f(n)$ and $g(n)$ grow at the same rate, up to constant factors. It's often the most informative of the three.

It's defined simply as: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

This essentially "sandwiches" $f(n)$ between two different constant multiples of $g(n)$.

For example:

- $3n^2 + 500n + 312 = \Theta(n^2)$
- $n \neq \Theta(n^2)$, because while $n = O(n^2)$, it is not $\Omega(n^2)$.