Quantum Query Complexity of Dyck Languages with Bounded Height

Kamil Khadiev\textsuperscript{1} and Yixin Shen\textsuperscript{2}

\textsuperscript{1}Kazan Federal University, Kazan, Russia
\textsuperscript{2}Universit\'e de Paris, IRIF, CNRS, F-75013 Paris, France

Abstract

We consider the problem of determining if a sequence of parentheses is well parenthesized, with a depth of at most $h$. We denote this language as $\text{Dyck}_h$. We study the quantum query complexity of this problem for different $h$ as function of the length $n$ of the word. It has been known from a recent paper by Aaronson et al. that, for any constant $h$, since $\text{Dyck}_h$ is star-free, it has quantum query complexity $\tilde{\Theta}(\sqrt{n})$, where the hidden logarithm factors in $\tilde{\Theta}$ depend on $h$. Their proof does not give rise to an algorithm. When $h$ is not a constant, $\text{Dyck}_h$ is not even context-free. We give an algorithm with $O(\sqrt{n} \log(n) / (h-1))$ quantum queries for $\text{Dyck}_h$ for all $h$. This is better than the trivial upper bound $n$ when $h = \omega(\log(n))$. We also obtain lower bounds: we show that for every $0 < \epsilon \leq 0.37$, there exists $c > 0$ such that $Q(\text{Dyck}_{c \log(n)}(n)) = \Omega(n^{1-\epsilon})$. When $h = \omega(\log(n))$, the quantum query complexity is close to $n$, i.e. $Q(\text{Dyck}_h(n)) = \omega(n^{1-\epsilon})$ for all $\epsilon > 0$. Furthermore when $h = \Omega(n^{\epsilon})$ for some $\epsilon > 0$, $Q(\text{Dyck}_h(n)) = \Theta(n)$.

1 Introduction

Formal languages have a long history of study in classical theoretical computer science, starting with the study of regular languages back to Kleene in the 1950s \cite{12}. Roughly speaking, a formal language consists of an alphabet of letters, and a set of rules for generating words from those letters. Chomsky’s hierarchy is an early attempt to answer the following question: “Given more complex rules, what kinds of languages can we generate?”. The most well-known types of languages in that hierarchy are the regular and context-free languages. Modern computational complexity theory is still defined in terms of languages: complexity classes are defined as the sets of the formal languages that can be parsed by machines with certain computational powers.

The relationship between the Chomsky hierarchy and other models of computations has been studied extensively in many models, including Turing machines, probabilistic machines \cite{13}, quantum finite automata \cite{14}, streaming algorithms \cite{16, 15} and query complexity \cite{2}. Query complexity is also known as the ‘black box model’, in this setting we only count the number of times that we need to query (i.e. access) the input in order to carry out our computation. It has been observed that quantum models of computation allow for significant improvements in the query complexity, when the quantum oracle access to the input bits is available \cite{9}. We assume the reader is familiar with the basis of quantum computing. One may refer to \cite{17} for a more detailed introduction to this topic.

The recent work by Scott Aaronson, Daniel Grier, and Luke Shaheffer \cite{11} is the first to study the relationship between the regular languages and quantum query complexity. They give a full characterization of regular languages in the quantum query complexity model. More precisely, they show that every regular language naturally falls into one of three categories:

- ‘Trivial’ languages, for which membership can be decided by the first and last characters of the input string. For instance, the language describing all binary representations of even numbers is
trivial.

- Star-free languages, a variant of regular languages where complement is allowed (\(\overline{A}\) — i.e. ‘something not in \(A\)'), but the Kleene star is not. The quantum query complexity of these languages is \(\tilde{\Theta}(\sqrt{n})\).

- All the rest, which have quantum query complexity \(\Theta(n)\).

The proof uses the algebraic definitions of regular languages (i.e. in terms of monoids). Starting from an aperiodic monoid, Schützenberger constructs a star-free language recursively based on the “rank” of the monoid elements involved. \(\Pi\) uses this decomposition of star-free language of higher rank into star-free languages of smaller rank to show by induction that any star-free languages has \(\tilde{\Theta}(\sqrt{n})\) quantum query complexity. However their proof does not immediately give rise to an algorithm.

One of the star-free language mentioned in \(\Pi\) is the Dyck language (with one type of parenthesis) with a constant bounded height. The Dyck language is the set of balanced strings of brackets ”(“ and ”)”. When at any point the number of opening parentheses exceeds the number of closing parentheses by at most \(h\), we denote the language as \(\text{Dyck}_h\).

The Dyck language is a fundamental example of a context-free language that is not regular. When more types of parenthesis are allowed, the famous Chomsky–Schützenberger representation theorem shows that any context-free language is the homomorphic image of the intersection of Dyck language and a regular language.

**Contributions** We give an explicit algorithm (see Theorem [15]) for the decision problem of \(\text{Dyck}_h\) with \(O(\sqrt{n} \log(n)^h)\) quantum queries. The algorithm also works when \(h\) is not a constant and is better than the trivial upper bound \(n\) when \(h = o\left(\frac{\log(n)}{\log \log n}\right)\). We note that when \(h\) is not a constant, that is, if the height is allowed to depend on the length of the word, \(\text{Dyck}_h\) is not context-free anymore, therefore previous results do not apply. We also obtain lower bounds on the quantum query complexity. We show (Theorem [21]) that for every \(0 < \epsilon \leq 0.37\), there exists \(c > 0\) such that \(Q(\text{Dyck}_c \cdot \log(n)(n)) = \Omega(n^{1-\epsilon})\). When \(h = \omega(\log(n))\), the quantum query complexity is close to \(n\), i.e. \(Q(\text{Dyck}_h(n)) = \omega(n^{1-\epsilon})\) for all \(\epsilon > 0\), see Theorem [20]. Furthermore when \(h = \Omega(n^\epsilon)\) for some \(\epsilon > 0\), we show (Theorem [19]) that \(Q(\text{Dyck}_h(n)) = \Theta(n)\). Similar lower bounds were recently independently proven by Ambainis, Balodis, Iraids, Prūsis, and Smotrovs \(\hat{\Pi}\), and Buhrman, Patro and Speelman \(\hat{\Pi}\).

**Notation** The Dyck language is the set of balanced strings of brackets ”(“ and ”)”). When at any point the number of opening parentheses exceeds the number of closing parentheses by at most \(h\), we denote the language as \(\text{Dyck}_h\). \(\text{Dyck}_h(n)\) is the set of words of length \(n\) in \(\text{Dyck}_h\); \(h\) can be a function of \(n\). For readability reason, we define \(a = ”(“ \text{ and } b = ”)”). The alphabet is thus \(\Sigma = \{a, b\}\). For all \(u \in \Sigma^*\) where \(u = u_0 u_1 \cdots u_{n-1}\), we define \(l(u) = n\) as the length of \(u\), and \(f(u) = |u_a| - |u_b|\). We call \(f\) the balance. For all \(0 \leq i \leq n-1\), we define \(w[i, j] = u_i u_{i+1} \cdots u_j\). Finally, we define \(b(u) = \max_{0 \leq i \leq n-1} f(u[0, i])\) and \(h^-(u) = \min_{0 \leq i \leq n-1} f(u[0, i])\). We also define the function \(\text{sign}\) such that \(\text{sign}(x) = 1\) if \(x > 0\), and \(\text{sign}(x) = -1\) if \(x < 0\), \(\text{sign}(x) = 0\) if \(x = 0\).

**Structure of the paper** In the next section, we give an algorithm of quantum query complexity \(O(\sqrt{n} \log(n)^h)\) for \(\text{Dyck}_h\). In the following section, we show some lower bounds when \(h\) is \(\Omega(\log(n))\).

## 2 An algorithm for \(\text{Dyck}_h\)

### 2.1 \(\pm k\)-Substring Search algorithm

The goal of this section is to describe quantum algorithm for finding a substring \(u[i, j]\) that has a balance \(f(u[i, j]) \in \{+k, -k\}\) for some integer \(k\). This algorithm is the basis of our algorithms for Dyck languages.
We describe a ±2-substring Search algorithm in Section 2.1.1, a ±3-substring Search algorithm in Section 2.1.2, a ±4-substring Search algorithm in Section 2.1.3 and then we will finish with an algorithm for the general case in Section 2.1.4.

2.1.1 ±2-Substring Search Algorithm

The simplest case is an algorithm that searches for a substring $u[i, j]$ that has a balance $f(u[i, j]) \in \{+2, -2\}$. The algorithm looks for two sequential equal symbols using Grover’s Search Algorithm [10, 17]. Formally, it is a procedure that accepts the following parameters as inputs and outputs:

- Inputs:
  - an integer $l \in \{0, \ldots, n\}$ which is a left border for the substring to be searched. Here $n = l(u)$.
  - an integer $r \in \{l, \ldots, n\}$ which is a right border for the substring to be searched.
  - a set $s \subset \{+1, -1\}$ which represents the sign of the balance $f$ for a substring to be searched.

- Outputs:
  - a triple $(i, j, \sigma)$ where $i$ and $j$ are the left and right border of the found substring and where $\sigma$ is the sign of $f(u[i, j])$, i.e. $\sigma = \text{sign}(f(u[i, j]))$. If there is no such substring, then the algorithm returns NULL. Furthermore, when there is a satisfying substring, the result is such that $l \leq i \leq j \leq r$.

The algorithm searches for a substring $u[i, j]$ such that $f(u[i, j]) \in \{+2, -2\}$ and $\text{sign}(f(u[i, j])) \in s$.

We use a Grover’s Search Algorithm as a subroutine $\text{Grover}(l, r, F)$ that takes as inputs $l$ and $r$ as left and right borders of the search space and some function $F : \{l, \ldots, r\} \rightarrow \{0, 1\}$. We search for any index $i$, where $l \leq i \leq r$, such that $F(i) = 1$. The result of the function $\text{Grover}(l, r, F)$ is either some index $i$ or $-1$ if it has not found the required $i$.

In Algorithm 1 we use Grover’s search on function $F_s : \{0, \ldots, n-1\} \rightarrow \{0, 1\}$ defined by

\[F_s(i) = 1 \iff (u_i = u_{i+1} \text{ or } u_i = u_{i-1})\]

and the following conditions hold:

- if $s = \{+1\}$ then $u_i = 'a'$.
- if $s = \{-1\}$ then $u_i = 'b'$.
- if $s = \{+1, -1\}$ then $u_i = 'a'$ or $u_i = 'b'$.

Algorithm 1: Substring2-Any($\ell, r, s$). The Quantum Algorithm for searching any ±2 substring.

\[
\begin{align*}
& i \leftarrow \text{Grover}(\ell, r, F_s) \\
& \text{if } i = -1 \text{ then} \\
& \quad \text{return NULL} \\
& \text{end if} \\
& \text{if } i \neq -1 \text{ then} \\
& \quad \text{if } u_i = u_{i+1} \text{ then} \\
& \quad \quad \text{return } (i, i+1, f(u[i, i])) \\
& \quad \text{end if} \\
& \quad \text{if } u_i = u_{i-1} \text{ then} \\
& \quad \quad \text{return } (i-1, i, f(u[i, i])) \\
& \quad \text{end if} \\
& \text{end if}
\end{align*}
\]
Lemma 1. The running time of Algorithm 1 is $O(\sqrt{n})$. The error probability is $O\left(\frac{1}{n}\right)$.

Proof. The main part of the algorithm is the Grover’s Search algorithm that has $O(\sqrt{n})$ running time and $O\left(\frac{1}{n}\right)$ error probability.

It will be useful to consider a modification of the algorithm that finds not just any ±2 substring, but the closest to the left border or to the right border. In that case, we use a subroutine \textsc{Grover First One}, with parameters $(l, r, F, \text{direction})$ that accepts $l$ and $r$ as left and right borders of the search space, a function $F$ and $\text{direction} \in \{\text{left}, \text{right}\}$.

- If $\text{direction} = \text{left}$, then we search for the maximal index $i$ such that $F(i) = 1$ where $l \leq i \leq r$.
- If $\text{direction} = \text{right}$, then we search for the minimal index $i$ such that $F(i) = 1$ where $l \leq i \leq r$.

The result of the function \textsc{Grover First One}$(l, r, F, \text{direction})$ is either $i$ or $-1$ if it has not found the required $i$. See [13, 14, 15] on how to implement such a function.

Algorithm 2 implements the \textsc{Substring2 First} subroutine. It has the same input and output parameters as \textsc{Substring2 Any} and an extra input $\text{direction} \in \{\text{left}, \text{right}\}$.

Algorithm 2 \textsc{Substring2 First}$(l, r, s, \text{direction})$. The Quantum Algorithm for searching the first ±2 substring.

\begin{verbatim}
\textbf{i ← Grover First One}(l, r, F, \text{direction})} \triangleright \text{Invoke the Grover search if } i = -1 \text{ then return } NULL 
\text{end if if } i \neq -1 \text{ then return } (i, i + 1, f(u[i, i])) 
\text{end if}
\end{verbatim}

Lemma 2. If the required segment exists, the expected running time of Algorithm 2 is $O(\sqrt{j})$, where $j$ is the furthest border of the searching segment. Otherwise, the running time is $O(\sqrt{r - l})$. The error probability is at most 0.1.

Proof. The main part of the algorithm is the \textsc{Grover First One} algorithm [13, 14, 15] that has $O(\sqrt{j})$ expected running time and at most 0.1 error probability. The running time is $O(\sqrt{r - l})$ if there is no required segment.

2.1.2 ±3-Substring Search Algorithm

We now discuss an algorithm that searches for a substring $u[i, j]$ that has a balance $f(u[i, j]) \in \{+3, -3\}$. The algorithm searches for two ±2-substrings $u[l_1, r_1]$ and $u[l_2, r_2]$ such that there are no ±2-substrings between them. If both substrings $u[l_1, r_1]$ and $u[l_2, r_2]$ are +2-substring, then we get +3-substring in total. If both substrings are −2-substring, then we get −3-substring in total.

Firstly, we discuss a basic procedure for the algorithm that can fail. To search it we do the following steps for some integer $d \leq n$. Assume that the procedure searches for a substring in the segment $[l, r]$, where $0 \leq l \leq r \leq n - 1$.

Step 1. Randomly pick a position $t$.

Step 2. Search for the first ±2-substring on the right at distance at most $d$, i.e. in the segment $[t, \min(t + d - 1, r)]$. If the algorithm does not find it, then it fails. Otherwise the segment is the $u[i_1, j_1]$ substring.
Step 3. Search for the first ±2-substring on the left from \(i_1\) at distance at most \(\bar{d} = d - (j_1 - i_1 + 1)\), i.e. in the segment \([\max(l, i_1 - \bar{d}), i_1 - 1]\). If the algorithm finds the substring \(u[i', j']\) and it has the same balance as the first substring, i.e. \(f(u[i_1, j_1]) = f(u[i', j'])\), then we assign \(i_2 \leftarrow i'\) and \(j_2 \leftarrow j'\) and go to Step 5. Otherwise, the algorithm goes to Step 4.

Step 4. If we have not found a ±2-substring on the left, then we search for the first ±2-substring to the right from \(j_1\) at distance at most \(\bar{d} = d - (j_1 - i_1 + 1)\), i.e. in the segment \([j_1 + 1, \min(r, j_1 + \bar{d})]\). If we do not find it, then the algorithm fails. Otherwise it outputs the ±2-substring.

Step 5. If the algorithm finds the substrings \(u[i_1, j_1]\) and \(u[i_2, j_2]\) such that \(f(u[i_1, j_1]) = f(u[i_2, j_2])\) and sign\((f(u[i_1, j_1])) \in s\), then \([\min(i_1, i_2), \max(j_1, j_2)]\) is the answer, otherwise the algorithm fails.

Algorithm 3 implements this procedure and accepts as inputs:

- the borders \(l\) and \(r\), where \(l\) and \(r\) are integers such that \(0 \leq l \leq r \leq n - 1\).
- the position \(t \in \{l, \ldots, r\}\).
- the maximal distance \(d\), where \(d\) is an integer such that \(0 < d \leq (r - l)\).
- the sign of balance of borders \(s \subset \{+1, -1\}\). +1 is used for searching +3-substring, –1 is used for searching –3-substring, \{+1, –1\} is used for searching both.

\[
\text{Algorithm 3 } \text{SUBSTRING}_{3, \text{BASE}}(l, r, t, d, s). \text{ The basic quantum algorithm for searching any ±3-substring.}
\]

\[
v_1 = (i_1, j_1, \sigma_1) \leftarrow \text{SUBSTRING}_{2, \text{FIRST}}(t, \min(t + d, r), \{+1, -1\}, \text{right}) \quad \triangleright \text{It is the first invocation (Step 2). The algorithm searches for a ±2-substring to the right}
\]

if \(v_1 = \text{NULL}\) then
    return NULL
end if

\[
d \leftarrow d - (j_1 - i_1 + 1)
\]

\[
v_2 = (i', j', \sigma') \leftarrow \text{SUBSTRING}_{2, \text{FIRST}}(\max(i_1 - d, l), i_1 - 1, \{+1, -1\}, \text{left}) \quad \triangleright \text{It is the second invocation (Step 3). The algorithm searches for a ±2-substring to the left}
\]

if \(v_2 \neq \text{NULL}\) and \(\sigma' = \sigma_1\) then
    \[
i_2 \leftarrow i'
\]
    \[
j_2 \leftarrow j'
\]
    \[
\sigma_2 \leftarrow \sigma'
\]
end if
if \(v_2 = \text{NULL}\) or \(\sigma' \neq \sigma_1\) then
    \[
v_3 = (i_2, j_2, \sigma_2) \leftarrow \text{SUBSTRING}_{2, \text{FIRST}}(j_1 + 1, \min(r, j_1 + \bar{d}), \{+1, -1\}, \text{left}) \quad \triangleright \text{It is the third invocation (Step 4). The algorithm searches for a ±2-substring to the right}
\]

if \(v_3 = \text{NULL}\) then
    return NULL
end if
if \(\sigma_1 = \sigma_2\) and \(\sigma_1 \in s\) then
    return \((\min(i_1, i_2), \max(j_1, j_2), \sigma_1)\)
end if

**Lemma 3.** If Algorithm 3 picks the starting point inside the search substring, then it will find it. The expected running time of Algorithm 3 is \(O(2 \cdot \sqrt{d})\). The probability of success is at least \(\frac{\sqrt{2}}{\sqrt{4}}\), where \(g\) is a length of ±3-substring.
Proof. Let us prove the correctness of the algorithm in the case of picking a point inside the search substring. Let us consider the case of +3-substring, the second case is similar. Assume that the substring to be searched is $u[i, j]$. There are two cases:

1. Assume that $g \geq 4$. This means that $u_i = u_{i+1}$ and $u_{j-1} = u_j$.
   - If $t \in \{j-1, j\}$, then the first invocation of SUBSTRING$_2$-FIRST procedure finds $u[j-1, j]$ and the second invocation of SUBSTRING$_2$-FIRST finds $u[i, i+1]$ in the case of $g = j - i + 1 \leq d$.
   - If $t \in \{i, i+1\}$, then the first invocation of SUBSTRING$_2$-FIRST procedure finds $u[i, i+1]$ and the third invocation of SUBSTRING$_2$-FIRST finds $u[j-1, j]$ in the case of $g = j - i + 1 \leq d$.
   - If $i+1 < t < j-1$, then the first invocation of SUBSTRING$_2$-FIRST procedure finds $u[j-1, j]$ and the second invocation of SUBSTRING$_2$-FIRST finds $u[i, i+1]$ in the case of $g = j - i + 1 \leq d$.

2. Assume that $g = 3$. This means that $u_i = u_{i+1} = u_{i+2}$ and $j = i + 2$.
   - If $t \in \{i+1, i+2\}$, then the first invocation of SUBSTRING$_2$-FIRST procedure finds $u[i+1, i+2]$ and the second invocation of SUBSTRING$_2$-FIRST finds $u[i, i+1]$ in the case of $3 \leq d$.
   - If $t = i$, then the first invocation of SUBSTRING$_2$-FIRST procedure finds $u[i, i+1]$ and the third invocation of SUBSTRING$_2$-FIRST finds $u[i+1, i+2]$ in the case of $3 \leq d$.

Due to Lemma 2.1.1, the running time of each SUBSTRING$_2$-FIRST invocation is $O(\sqrt{d})$.

The probability of picking $t$ inside the required segment is the length of the segment over the length of searching space, i.e. $\frac{g}{d}$. \qed

We now provide an algorithm to search for any ±3-substring of fixed length $d$ with high probability. Algorithm 3 succeeds with probability $p_{success} = \frac{g}{d}$. We can use the amplitude amplification algorithm 4, which is a generalization of Grover’s search algorithm 10, and boost the success probability to a higher probability. We should invoke the base algorithm $\frac{1}{p_{success}}$ times, but we do not know $p_{success}$ that depends on the unknown $g$. Therefore we invoke it $\frac{\sqrt{d}}{g}$ times. Let us call this procedure SUBSTRING$_3$-FOR _A FIXED _LENGTH. It accepts the same parameters as SUBSTRING$_3$-BASE except for the extra position $t$.

We can now write an algorithm to search for any ±3-substring. We choose the length $d$ as a power of 2 and search for ±3-substrings of such length. We start with $d = 4$ since the minimal length of ±3-substrings is 3 and $d = 4$ is the smallest power of 2 that is greater than 3. Algorithm 4 accepts the following parameters:

- the borders $l$ and $r$, where $l$ and $r$ are integers such that $0 \leq l \leq r \leq n-1$.
- the sign of balance of borders $s \subseteq \{+1, -1\}$. +1 is used for searching +3-substring, −1 is used for searching −3-substring, {+1, −1} is used for searching both.

Lemma 4. Algorithm 4 finds some ±3-substring with probability at least 0.5 and running time $O(\sqrt{r - l} \log g) = O(\sqrt{r - l} \log(r - l))$, where $g$ is the length of the shortest ±3-substring.

Proof. Assume that shortest ±3-substring $u[i, j]$ is such that the length $g = j - i + 1$. The first invocation of SUBSTRING$_3$-FOR _A FIXED _LENGTH that finds the substring will be in the case $d = d_0 = g^{\log_2 g} \leq 2g$. The working time of SUBSTRING$_3$-FOR _A FIXED _LENGTH is $O \left(\sqrt{\frac{r - l}{d_0} \sqrt{d_0}}\right)$ due to the complexity of Amplitude amplification and Lemma 3.

So, $O(\sqrt{\frac{r - l}{d_0} \sqrt{d_0}}) = O(\sqrt{r - l})$. The total running time is $O(\sqrt{r - l} \log g)$ because before reaching $d \geq 2g$ the algorithm will do $O(\log g)$ steps of the loop.
Algorithm 4 Substring\_3\_Any\((l, r, s)\). The algorithm for searching any ±3-substring.

\[
\begin{align*}
d & \leftarrow 4 \\
isFound & \leftarrow False \\
\text{while } & \text{isFound = False and } d \leq (r - l) \text{ do} \\
& \quad v = (i, j, \sigma) \leftarrow \text{Substring\_3\_For\_A\_Fixed\_Length}(l, r, d, s) \\
& \quad \text{if } v \neq \text{NULL then} \\
& \quad \quad \text{isFound} \leftarrow \text{True} \\
& \quad \text{end if} \\
& \quad d \leftarrow d \cdot 2 \\
\text{end while} \\
& \text{if isFound = False then} \\
& \quad \text{return NULL} \\
& \text{end if} \\
& \text{if isFound = True then} \\
& \quad \text{return } v \\
& \text{end if}
\end{align*}
\]

We can now estimate the success probability. The number of steps of Amplitude Amplification algorithm in a case of \(d = d_0\) is at most \(\sqrt{2}\) more that it is required because the highest probability is for \(\frac{\pi}{4}(\frac{r - l}{g})\) steps. That is why we get 0.5 as a probability of success.

Now consider the algorithm that finds the first ±3-substring. The idea of the algorithm is similar to the first one search algorithm from [13, 14, 15]. We search a ±3-substring in the segment of length \(w\) that are power of 2. Assume that the answer is \(u[i, j]\) and we search it on the left in the segment \([l, r]\), then the first time when we find the substring is the case \(w = 2^{\lceil \log_2(l - j) \rceil} \leq 2(l - j)\).

Algorithm 5 implements this procedure Substring\_3\_First with the same arguments as a procedure Substring\_4\_First in Algorithm 2.

Algorithm 5 Substring\_3\_First\((l, r, s, direction)\). The algorithm for searching the first ±3-substring.

\[
\begin{align*}
w & \leftarrow 4 \\
isFound & \leftarrow False \\
\text{while } & \text{isFound = False and } w \leq 2(r - l) \text{ do} \\
& \quad \text{if direction = right then} \\
& \quad \quad v = (i, j, \sigma) \leftarrow \text{Substring\_3\_Any}(l, min\,(r, l + w - 1), s) \\
& \quad \text{end if} \\
& \quad \text{if direction = right then} \\
& \quad \quad v = (i, j, \sigma) \leftarrow \text{Substring\_3\_Any}(max\,(l, r - w + 1), r, s) \\
& \quad \text{end if} \\
& \quad \text{if } v \neq \text{NULL then} \\
& \quad \quad \text{isFound} \leftarrow \text{True} \\
& \quad \text{end if} \\
& \quad w \leftarrow w \cdot 2 \\
\text{end while} \\
& \text{if isFound = False then} \\
& \quad \text{return NULL} \\
& \text{end if} \\
& \text{if isFound = True then} \\
& \quad \text{return } v \\
& \text{end if}
\end{align*}
\]

Algorithm 5 has the following property.
Lemma 5. The expected running time of Algorithm 3 is \( O(\sqrt{z}\log g) = O(\sqrt{z}\log z) \), where \( z \) is the most far border of the searching segment or the running time is \( O(\sqrt{r-1}\log(r-l)) \) if there is no required segment. The error probability is at most 0.1.

Proof. We can show the properties similarly to [13, 14, 15].

2.1.3 ±4-Substring Search Algorithm

Let us now discuss an algorithm that searches for a substring \( u[i, j] \) that has a balance \( f(u[i, j]) \in \{+4, −4\} \). The algorithm searches for two ±3-substrings \( u[l_1, r_1] \) and \( u[l_2, r_2] \) such that there are no ±3-substrings between them. If both substrings \( u[l_1, r_1] \) and \( u[l_2, r_2] \) are ±3-substrings, then we get ±4-substring in total. If both substrings are ±3-substring, then we get −4-substring in total.

The scheme of the algorithm is similar to the ±3-substring search algorithm. We briefly review the common parts of the two algorithms.

The basic procedure for the algorithm that can fail is searching a substring of length at most \( d \) in the segment \([l, r]\), where \( 0 \leq l \leq r \leq n-1 \).

Step 1. Randomly pick a position \( t \).

Step 2. Search for the first ±3-substring on the right at distance at most \( d \), i.e. in \([t, \min(t+d-1, r)]\). The result is \( u[i_1, j_1] \) or fail.

Step 3. Search for the first ±3-substring on the left from \( i_1 \) at distance at most \( \tilde{d} = d - (j_1 - i_1 + 1) \), i.e. in \([\max(l, i_1 - \tilde{d}), i_1 - 1]\). The result is \( u[i’, j’] \). If \( f(u[i_1, j_1]) = f(u[i’, j’]) \), then \( i_2 \leftarrow i’ \) and \( j_2 \leftarrow j’ \) and go to Step 5. Otherwise, the algorithm goes to Step 4.

Step 4. Search for the first ±3-substring on the right from \( j_1 \) at distance at most \( \tilde{d} = d - (j_1 - i_1 + 1) \), i.e. in \([j_1 + 1, \min(r, j_1 + \tilde{d})]\). The result is \( u[i_2, j_2] \) substring or procedure fails.

Step 5. If the algorithm finds the substrings \( u[i_1, j_1] \) and \( u[i_2, j_2] \) such that \( f(u[i_1, j_1]) = f(u[i_2, j_2]) \) and \( \text{sign}(f(u[i_1, j_1])) \in s \), then \( \min(i_1, i_2), \max(j_1, j_2) \] is the answer, otherwise the algorithm fails.

Algorithm 6 implements this procedure and its input parameters are same as for Algorithm 3.

Let us discuss the property of the algorithm.

Lemma 6. If Algorithm 6 picks the starting point inside the searching substring, then it will find it. The expected running time of Algorithm 6 is \( O(2\cdot \sqrt{d}(\log n)) \). The probability of success is at least \( \frac{1}{\sqrt{d}} \), where \( g \) is a length of ±4-substring.

Proof. Let us prove the correctness of the algorithm in the case of picking a point inside the search substring. There are different cases when searching for a ±4-substring \( u[i, j] \).

1. Assume that there is \( j_1 \) and \( i_2 \) such that \( i < j_1 < i_2 < j \) and \( f(u[i, j_1]) = f(u[i_2, j]) \in s \).
   
   - If \( t \in \{i_2, \ldots, j\} \), then the first invocation of SUBSTRING3-FIRST procedure finds \( u[i_2, j] \) and the second invocation of SUBSTRING3-FIRST finds \( u[i, j_1] \) in the case of \( g = j - i + 1 \leq d \).
   
   - If \( t \in \{i, \ldots, j_1\} \), then the first invocation of SUBSTRING3-FIRST procedure finds \( u[i, j_1] \) and the third invocation of SUBSTRING3-FIRST finds \( u[i_2, j] \) in the case of \( g = j - i + 1 \leq d \).
   
   - If \( j_1 < t < i_2 \), then the first invocation of SUBSTRING3-FIRST procedure finds \( u[i_2, j] \) and the second invocation of SUBSTRING3-FIRST finds \( u[i, j_1] \) in the case of \( g = j - i + 1 \leq d \).

2. Assume that there is \( j_1 \) and \( i_2 \) such that \( i < i_2 < j_1 < j \) and \( f(u[i, j_1]) = f(u[i_2, j]) \in s \).
   
   - If \( t \in \{j_1, \ldots, i_2\} \), then the first invocation of SUBSTRING3-FIRST procedure finds \( u[i_2, j] \) and the second invocation of SUBSTRING3-FIRST finds \( u[i, j_1] \) in the case of \( g = j - i + 1 \leq d \).
Algorithm 6 \textsc{Substring}_1\textsc{Base}(l,r,t,d,s). The basic quantum algorithm for searching any \pm 4-substring.

\begin{verbatim}
v_1 = (i_1,j_1,\sigma_1) \leftarrow \textsc{Substring}_3\textsc{First}(t,\min(t+d,r),\{+1,-1\},\text{right}) \triangleright \text{It is the first invocation (Step 2). The algorithm searches for a} \pm 3\text{-substring to the right}
if v_1 = \text{NULL then}
    \text{return NULL}
end if
\begin{itemize}
    \item \text{if } d \leftarrow d - (j_1 - i_1 + 1)\\
        v_2 = (i',j',\sigma') \leftarrow \textsc{Substring}_3\textsc{First}(\max(i_1 - d, l), i_1 - 1, \{+1,-1\}, \text{left}) \triangleright \text{It is the second invocation (Step 3). The algorithm searches for a} \pm 3\text{-substring to the left}
    \begin{itemize}
        \item if v_2 \neq \text{NULL and } \sigma' = \sigma_1 then
            i_2 \leftarrow i'\\
            j_2 \leftarrow j'\\
            \sigma_2 \leftarrow \sigma'
        end if
    end if
    if v_2 = \text{NULL or } \sigma' \neq \sigma_1 then
        v_3 = (i_2,j_2,\sigma_2) \leftarrow \textsc{Substring}_3\textsc{First}(j_1 + 1, \min(r, j_1 + d), \{+1,-1\}, \text{left}) \triangleright \text{It is the third invocation (Step 4). The algorithm searches for a} \pm 3\text{-substring to the right}
        if v_3 = \text{NULL then}
            \text{return NULL}
        end if
    end if
    if \sigma_1 = \sigma_2 and \sigma_1 \in s then
        \text{return } (\min(i_1, i_2), \max(j_1, j_2), \sigma_1)
    end if
\end{itemize}
end if
\end{verbatim}

\begin{itemize}
    \item If \( t \in \{i,\ldots,j\} \), then the first invocation of \textsc{Substring}_3\textsc{First} procedure finds \( u[i,j] \) and the \textbf{third} invocation of \textsc{Substring}_3\textsc{First} finds \( u[i_2,j] \) in the case of \( g = j - i + 1 \leq d \).
\end{itemize}

Due to Lemma 2.1.2, the running time of each \textsc{Substring}_3\textsc{First} invocation is \( O(\sqrt{d}\log n) \).

The probability of picking \( t \) inside the required segment is the length of the segment over the length of searching space, i.e. \( \frac{g}{r-t} \).

We now provide an algorithm to search for any \pm 4-substring of a fixed length \( d \) with high probability. Algorithm 6 succeeds with probability \( p_{\text{success}} = \frac{2}{r-t} \). As for \pm 3-substring, we use the amplitude amplification algorithm 7. We should invoke the base algorithm \( \sqrt{\frac{1}{p_{\text{success}}}} \) times, at the same time we do not know \( g \) before. That is why we invoke it \( \sqrt{\frac{r-t}{d}} \) times. Let us call this procedure \textsc{Substring}_4\textsc{For},\textsc{Fixed}_\textsc{Length}. It accepts the same parameters as \textsc{Substring}_4\textsc{Base} except the position \( t \).

Finally, we build an algorithm to search for any \pm 4-substring. We choose powers of 2 as a lengths \( d \) and searches \pm 4-substrings of such length. We starts with \( d = 4 \) because the minimal length of \pm 4-substrings is 4. Algorithm 7 accepts the same parameters as Algorithm 6.

Let us discuss the property of Algorithm 7.

\textbf{Lemma 7.} Algorithm 7 finds some \pm 4-substring with probability at least 0.5 and running time \( O(\sqrt{r-t}\log g) = O(\sqrt{r-t}\log(r - l)) \), where \( g \) is the length of the shortest \pm 4-substring.

\textit{Proof.} Assume that shortest \pm 4-substring \( u[i,j] \) is such that the length \( g = j - i + 1 \). The first invocation of \textsc{Substring}_4\textsc{For},\textsc{Fixed}_\textsc{Length} that finds the substring will be in the case \( d = d_0 = 2^{\lceil \log_2 g \rceil} \leq 2g \). The working time of \textsc{Substring}_4\textsc{For},\textsc{Fixed}_\textsc{Length} is \( O \left( \sqrt{\frac{r-t}{d_0}} \log n \cdot \sqrt{d_0} \right) \) due to the complexity of Amplitude amplification and Lemma 6.
Algorithm 7 Substring\_4\_Any(l, r, s). The algorithm for searching any ±4-substring.

```
Algorithm 7 Substring\_4\_Any(l, r, s). The algorithm for searching any ±4-substring.
```

```
d ← 4
isFound ← False
while isFound = False and d ≤ (r - l) do
    v = (i, j, σ) ← Substring\_4\_For\_A\_Fixed\_Length(l, r, d, s)
    if v ≠ NULL then
        isFound ← True
    end if
    d ← d · 2
end while
if isFound = False then
    return NULL
else
    return v
end if
```

So, \( O(\sqrt{\frac{d_{0}}{d_{0} + \log n} \cdot \sqrt{\log n}}) = O(\sqrt{\frac{r - l}{\log n}}) \). The total running time is \( O(\sqrt{r - l} \log n \log g) = O(\sqrt{r - l} (\log n)^2) \) because before reaching \( d \geq 2g \) the algorithm will do \( O(\log g) \) steps of the loop.

Let us estimate the success probability. The number of steps of Amplitude Amplification algorithm in a case of \( d = d_0 \) is at most \( \sqrt{2} \) more that it is required because the highest probability is for \( \frac{1}{4}(\sqrt{\frac{r - l}{\log n}}) \) steps. That is why we get 0.5 as a probability of success.

Let us consider the algorithm that finds the first ±4-substring. The idea of the algorithm is similar to the first ±3-substring and the first one search algorithm from [13, 14, 15]. We search for a ±4-substring in the segment of length \( w \) that are power of 2. Assume that the answer is \( u[i, j] \) and we search it on the left in the segment \([l, r]\), then the first time when we find the substring is the case \( w = 2^{\left\lceil \log_2(l-j) \right\rceil} \leq 2(l-j) \).

Procedure Substring\_4\_First in Algorithm 8 implements this idea and takes the same arguments as procedure Substring\_3\_First in Algorithm 5.

Algorithm 8 has the following property.

**Lemma 8.** The expected running time of Algorithm 8 is \( O(\sqrt{\frac{\log g}{\log z}})^2 = O(\sqrt{\frac{\log z}{\log r - l}})^2 \), where \( z \) is the most far border of the searching segment or the running time is \( O(\sqrt{\frac{r - l}{\log(r - l)}})^2 \) if there is no required segment. The error probability is at most 0.1.

**Proof.** We can show the properties similar to [13, 14, 15].

### 2.1.4 ±k-Substring Search Algorithm

We now discuss an algorithm that searches for a substring \( u[i, j] \) that has a balance \( f(u[i, j]) \in \{+k, -k\} \). The algorithm searches for two ±\((k-1)\)-substrings \( u[l_1, r_1] \) and \( u[l_2, r_2] \) such that there are no ±\((k-1)\)-substrings between them. If both substrings \( u[l_1, r_1] \) and \( u[l_2, r_2] \) are +\((k-1)\)-substring, then we get +\((k-1)\)-substring in total. If both substrings are -\((k-1)\)-substring, then we get -\((k-1)\)-substring in total.

The scheme of the algorithm is similar to the ±4-substring search algorithm so we review the common parts of the algorithms briefly.

The basic procedure for the algorithm that can fail is searching a substring of length at most \( d \) in the segment \([l, r]\), where \( 0 \leq l \leq r \leq n - 1 \).

Step 1. Randomly pick a position \( t \).
Algorithm 8 Substring\_first(l, r, s, direction). The algorithm for searching the first ±4-substring.

```
\begin{algorithm}
   w \leftarrow 4
   isFound \leftarrow False
   \textbf{while} isFound = False and \( w \leq 2(r-l) \) \textbf{do}
      \textbf{if} direction = right \textbf{then}
         v = (i, j, \sigma) \leftarrow \text{Substring\_any}(l, \min(r, l+w-1), s)
      \textbf{end if}
      \textbf{if} direction = right \textbf{then}
         v = (i, j, \sigma) \leftarrow \text{Substring\_any}(\max(l, r-w+1), r, s)
      \textbf{end if}
      \textbf{if} v \neq \text{NULL} \textbf{then}
         isFound \leftarrow True
      \textbf{end if}
      w \leftarrow w \cdot 2
   \textbf{end while}
   \textbf{if} isFound = False \textbf{then}
      \textbf{return} \text{NULL}
   \textbf{end if}
   \textbf{if} isFound = True \textbf{then}
      \textbf{return} v
   \textbf{end if}
\end{algorithm}
```

Step 2. Search for the first ±(k−1)-substring on the right on distance at most d, i.e. in [t, min(t+d−1, r)]. The result is \( u[i_1, j_1] \) or fail.

Step 3. Search for the first ±(k−1)-substring on the left from \( i_1 \) at distance at most \( \tilde{d} = d - (j_1 - i_1 + 1) \), i.e. in \([\max(l, i_1 - \tilde{d}), i_1 - 1]\). The result is \( u[i', j'] \). If \( f(u[i_1, j_1]) = f(u[i', j']) \), then \( i_2 \leftarrow i' \) and \( j_2 \leftarrow j' \) and go to Step 5. Otherwise, the algorithm goes to Step 4.

Step 4. Search for the first ±(k−1)-substring on the right from \( j_1 \) at distance at most \( \tilde{d} = d - (j_1 - i_1 + 1) \), i.e. in \([j_1 + 1, \min(r, j_1 + \tilde{d})]\). The result is \( u[i_2, j_2] \) substring or procedure fails.

Step 5. If the algorithm finds the substrings \( u[i_1, j_1] \) and \( u[i_2, j_2] \) such that \( f(u[i_1, j_1]) = f(u[i_2, j_2]) \) and \( \text{sign}(f(u[i_1, j_1])) \in s \), then \( \min(i_1, i_2), \max(j_1, j_2) \) is the answer, otherwise the algorithm fails.

Algorithm \( \Box \) implements this procedure and its input parameters are same as for Algorithm \( \Box \).

Let us discuss the property of the algorithm.

**Lemma 9.** If Algorithm \( \Box \) picks the starting point inside the searching substring, then it will find it. The expected running time of Algorithm \( \Box \) is \( O(2 \cdot \sqrt{d} (\log n)^{k-3}) \). The probability of success is at least \( \frac{a}{r-l} \), where \( g \) is a length of ±k-substring.

**Proof.** Let us prove the correctness of the algorithm in the case of picking a point inside the search substring. There are different cases to consider when searching for a +k-substring \( u[i, j] \).

1. Assume that there is \( j_1 \) and \( i_2 \) such that \( i < j_1 < i_2 < j \) and \( f(u[i, j_1]) = f(u[i_2, j]) \in s \).
   - If \( t \in \{i, \ldots, j_1\} \), then the first invocation of Substring\( (k-1) \)\_first procedure finds \( u[i_2, j] \) and the second invocation of Substring\( (k-1) \)\_first finds \( u[i, j_1] \) in the case of \( g = j - i + 1 \leq d \).
   - If \( t \in \{i, \ldots, j_1\} \), then the first invocation of Substring\( (k-1) \)\_first procedure finds \( u[i, j_1] \) and the third invocation of Substring\( (k-1) \)\_first finds \( u[i_2, j] \) in the case of \( g = j - i + 1 \leq d \).
Algorithm 9 \textsc{Substring}_{k\text{-base}}(l, r, t, d, s). The basic quantum algorithm for searching any $\pm k$-substring.

\begin{verbatim}
\begin{algorithm}
\label{alg:substring-base}
\begin{algorithmic}
\State $v_1 = (i_1, j_1, \sigma_1) \leftarrow \textsc{Substring}_{(k-1)\text{-first}}(t, \min(t + d, r), \{+1, -1\}, \text{right})$ \Comment{It is the first invocation (Step 2). The algorithm searches for a $\pm (k-1)$-substring to the right}
\If{$v_1 = \text{NULL}$}
\State return \text{NULL}
\EndIf
\State $d \leftarrow d - (j_1 - i_1 + 1)$
\State $v_2 = (i', j', \sigma') \leftarrow \textsc{Substring}_{(k-1)\text{-first}}(\max(i_1 - \tilde{d}, l), i_1 - 1, \{+1, -1\}, \text{left})$ \Comment{It is the second invocation (Step 3). The algorithm searches for a $\pm (k-1)$-substring to the left}
\If{$v_2 \neq \text{NULL}$ and $\sigma' = \sigma_1$}
\State $i_2 \leftarrow i'$
\State $j_2 \leftarrow j'$
\State $\sigma_2 \leftarrow \sigma'$
\EndIf
\If{$v_2 = \text{NULL}$ or $\sigma' \neq \sigma_1$}
\State $v_3 = (i_2, j_2, \sigma_2) \leftarrow \textsc{Substring}_{(k-1)\text{-first}}(j_1 + 1, \min(r, j_1 + \tilde{d}), \{+1, -1\}, \text{left})$ \Comment{It is the third invocation (Step 4). The algorithm searches for a $\pm (k-1)$-substring to the right}
\If{$v_3 = \text{NULL}$}
\State return \text{NULL}
\EndIf
\If{$\sigma_1 = \sigma_2$ and $\sigma_1 \in s$}
\State return $(\min(i_1, i_2), \max(j_1, j_2), \sigma_1)$
\EndIf
\EndIf
\end{algorithmic}
\end{algorithm}
\end{verbatim}

- If $j_1 < t < i_2$, then the first invocation of \textsc{Substring}_{(k-1)\text{-first}} procedure finds $u[i_2, j]$ and the second invocation of \textsc{Substring}_{(k-1)\text{-first}} finds $u[i, j_1]$ in the case of $g = j - i + 1 \leq d$.

2. Assume that there is $j_1$ and $i_2$ such that $i < i_2 < j_1 < j$ and $f(u[i, j_1]) = f(u[i_2, j]) \in s$.

- If $t \in \{j_1, \ldots, i_2\}$, then the first invocation of \textsc{Substring}_{(k-1)\text{-first}} procedure finds $u[i_2, j]$ and the second invocation of \textsc{Substring}_{(k-1)\text{-first}} finds $u[i, j_1]$ in the case of $g = j - i + 1 \leq d$.

- If $t \in \{i_1, \ldots, j_1\}$, then the first invocation of \textsc{Substring}_{(k-1)\text{-first}} procedure finds $u[i, j_1]$ and the third invocation of \textsc{Substring}_{(k-1)\text{-first}} finds $u[i_2, j]$ in the case of $g = j - i + 1 \leq d$.

Due to Lemma 2.1.2, the running time of each \textsc{Substring}_{(k-1)\text{-first}} invocation is $O(\sqrt{d} (\log n)^{k-3})$. The probability of picking $t$ inside the required segment is the length of the segment over the length of searching space, i.e. $\frac{d}{\sqrt{d}}$.

Let us provide the algorithm for searching any $\pm k$-substring of a fixed length $d$ with high probability. The Algorithm 8 succeeds with probability $p_{\text{success}} = \frac{d}{\sqrt{d}}$. As for $\pm (k-1)$-substring, we use the amplitude amplification algorithm [7]. We should invoke the base algorithm $\frac{1}{p_{\text{success}}}$ times, but we do not know $p_{\text{success}}$ that depends on the unknown $g$. Therefore we invoke it $\sqrt{\frac{d}{a}}$ times. Let us call this procedure \textsc{Substring}_{k\text{-for-fixed-length}}. It accepts the same parameters as \textsc{Substring}_{k\text{-base}} except for the extra position $t$.

Finally, we can write an algorithm to search for any $\pm k$-substring. We choose powers of 2 as a length $d$ and search $\pm k$-substrings of such length. We start with $d = 2^{\lceil \log_2 k \rceil}$ that is the smallest power of 2 greater than $d$. Algorithm 10 accepts the same parameters as Algorithm 9.
The working time of Substring Algorithm in a case of $d$ algorithm for Dyck language checking. We can show the properties similar to [13, 14, 15].

**Lemma 10.** Algorithm $\text{Substring}_k$ finds some $\pm k$-substring with probability at least 0.5 and has running time $O(\sqrt{r - l}(\log g)^{k-2}) = O(\sqrt{r - l}(\log(r - l))^{k-2})$, where $g$ is the length of the shortest $\pm k$-substring.

**Proof.** Assume that the shortest $\pm k$-substring $u[i, j]$ is of length $g = j - i + 1$. The first invocation of $\text{Substring}_{k, \text{ForA, Fixed-Length}}$ that finds the substring will be in the case $d = d_0 = 2^{\lfloor \log_2 g \rfloor} \leq 2g$.

The working time of $\text{Substring}_{3, \text{ForA, Fixed-Length}}$ is $O\left(\frac{\sqrt{r - l}}{d_0}(\log n)^{k-3} \cdot \sqrt{d_0}\right)$ due to the complexity of Amplitude amplification and Lemma 9.

So, $O\left(\frac{\sqrt{r - l}}{d_0}(\log n)^{k-3} \cdot \sqrt{d_0}\right) = O(\sqrt{r - l}(\log n)^{k-3} \log g) = O(\sqrt{r - l}(\log n)^{k-2})$ because before reaching $d \geq 2g$ the algorithm will do $O(\log g)$ steps of the loop.

We now estimate the success probability. The number of steps of Amplitude Amplification algorithm in a case of $d = d_0$ is at most $\sqrt{2}$ more that it is required because the highest probability is for $\frac{\sqrt{r - l}}{g} \cdot (\sqrt{\frac{r - l}{g}})$ steps. That is why we get 0.5 as a probability of success. 

Let us consider the algorithm that finds the first $\pm k$-substring. The idea of the algorithm is similar to the first $\pm 3$-substring and the first one search algorithm from [13, 14, 15]. We search a $\pm k$-substring in the segment of length $w$ that is a power of 2. Assume that the answer is $u[i, j]$ and we search it on the left in the segment $[l, r]$, then the first time when we find the substring is the case $v = 2^{\lfloor \log_2 (l - j) \rfloor} \leq 2(l - j)$.

Procedure $\text{Substring}_{k, \text{First}}$ in Algorithm 11 implements this idea. It has the same arguments as a procedure $\text{Substring}_{3, \text{First}}$ in Algorithm 6. It has the following property.

**Lemma 11.** The expected running time of Algorithm $\text{Substring}_k$ is $O(\sqrt{\tau}(\log g)^{k-2}) = O(\sqrt{\tau}(\log z)^{k-2})$, where $z$ is the most far border of the searching segment or the running time is $O(\sqrt{r - l}(\log(r - l))^{k-2})$ if there is no required segment. The error probability is at most 0.1.

**Proof.** We can show the properties similar to [13, 14, 15].

### 2.2 Dyck Language Checking Algorithm

Using the algorithms from Section 2.1 we will construct algorithms for the Dyck Language Checking Problem by the same scheme. Section 2.2.1 contains an algorithm for Dyck$_1$, Section 2.2.2 contains an algorithm for Dyck$_2$, Section 2.2.3 contains an algorithm for Dyck$_3$ and Section 2.2.4 contains an algorithm for Dyck$_h$. 

---

**Algorithm 10** $\text{Substring}_k$-any($l, r, s$). The algorithm for searching any $\pm k$-substring.

$d \leftarrow 2^{\lfloor \log_2 k \rfloor}$

isFound $\leftarrow$ False

while isFound = False and $d \leq (r - l)$ do

$v = (i, j, \sigma) \leftarrow \text{Substring}_k$-ForA, Fixed-Length($l, r, d, s$)

if $v \neq \text{NULL}$ then

isFound $\leftarrow$ True

end if

d $\leftarrow d \cdot 2$

end while

if isFound = False then

return NULL

end if

if isFound = True then

return $v$

end if
Algorithm 11. Substring$^k$First($l, r, s, \text{direction}$). The algorithm for searching the first $\pm k$-substring.

\begin{algorithm}
\begin{algorithmic}
\State $w \leftarrow 2^{\lceil \log_2 k \rceil}$
\State isFound $\leftarrow$ False
\While {isFound = False and $w \leq 2(r - l)$} 
\If {\text{direction} = \text{right}} 
\State $v = (i, j, \sigma) \leftarrow \text{Substring}^k_{-1}\text{Any}(l, \min(r, l + w - 1), s)$
\EndIf
\If {\text{direction} = \text{right}} 
\State $v = (i, j, \sigma) \leftarrow \text{Substring}^k_{-1}\text{Any}(\max(l, r - w + 1), r, s)$
\EndIf
\If {\text{$v \neq \text{NULL}$}} 
\State isFound $\leftarrow$ True
\EndIf
\State $w \leftarrow w \cdot 2$
\EndWhile
\If {isFound = False} 
\State return NULL
\EndIf
\If {isFound = True} 
\State return $v$
\EndIf
\end{algorithmic}
\end{algorithm}

2.2.1 Dyck\(_1\)
We search a substring that breaks the rule of Dyck\(_1\). There are two types of substrings:
\begin{itemize}
\item a substring $u[l, r]$ with balance $f(u[l, r]) = 2$.
\item a substring $u[0, r]$ (prefix) with balance $f(u[0, r]) = -1$.
\end{itemize}
Additionally, the number of $a$ and $b$ symbols should be equal, i.e. $f(u[0, n - 1]) = 0$.
To search for a substring $u[l, r]$ with balance $f(u[l, r]) = 2$, we use Substring$^k_{-1}$\text{Any} with parameter $s = \{+2\}$ from Algorithm 1. To find a substring $u[0, r]$ (prefix) with balance $f(u[l, r]) = -1$, then we should check the following cases:
\begin{itemize}
\item whether symbol $u[0]$ is $b$,
\item whether there is a prefix $u[0, l]$ and substring $u[l, r]$ such that $f(u[0, l]) = 1$ and $f(u[l, r]) = -2$.
\end{itemize}
If we already have checked that there are no $+2$-substring, then it is enough to check whether symbol $u[0]$ is $b$ and whether there is a $-2$-substring using Substring$^k_{-1}$\text{Any} with parameter $s = \{-2\}$.
If there is no substrings of the previous types that breaks Dyck(1) then the balance can be only $f(u[0, n - 1]) = 0$ or $f(u[0, n - 1]) = 1$. Additionally, we can say that $f(u[0, i]) \in \{0, 1\}$ for any $i \in \{0, n - 1\}$. That is why if $f(u[0, n - 1]) = 0$, then $f(u[0, n - 2]) = 1$ and $u[n - 1] = b$. If $f(u[0, n - 1]) = 1$, then $f(u[0, n - 2]) = 0$ and $u[n - 1] = a$. Therefore, the third condition is equivalent to $u[n - 1] = b$.
Finally, we obtain the following Algorithm 12 that returns true if the string $u$ belongs to Dyck\(_1\) language and false otherwise.

Lemma 12. The expected running time of Algorithm 12 is $O(\sqrt{n})$. The error probability is at most $O(\frac{1}{n})$.

Proof. The main part of the algorithm is Algorithm 1 that has $O(\sqrt{n})$ expected running time and the error probability $O(\frac{1}{n})$. The correctness of the algorithm is proven in the above discussion. \qed
Any substring. We use $u$ want to find a substring false language and in Algorithm 2 from Section 2.1.1 to search for the leftmost or rightmost Dyck 2.2.2 Algorithm 12 Dyck 2.1.2. Searching for following two paragraphs.

Additionally, the number of a and b symbols should be equal, i.e. $f(u[0, n - 1]) = 0$.

We search +3-substring using SUBSTRING3_ANY procedure with $s = \{+1\}$ that is described in Algorithm 1 from Section 2.1.2 Searching for $-1$-prefix and checking 0-balance are discussed in the following two paragraphs.

**No $-1$-prefix Checking** Assume that we already checked that there are no +3-substrings. If we want to find a substring $u[0, r]$ (prefix) with balance $f(u[0, r]) = -1$, then there are three cases:

- the symbol $u[0]$ is $b$.
- There is a prefix $u[0, k]$ and a substring $u[k, t]$ such that $f(u[0, k]) \leq 1$ and $f(u[k, t]) = -2$. The condition is right if and only if there are two most left substrings $u[l, r -]$ and $u[l, r +]$ such that $f(u[l, r -]) = -2$, $f(u[l, r +]) = +2$ and $u[l, r -]$ is on the left of $u[l, r +]$. In other words $l_{-} \leq r_{-} \leq l_{+} \leq r_{+}$. We can use the subroutine from Section 2.1.1 for searching $\pm 2$-substring.
- There is a $-3$-substring $u[l, r]$. We can use the subroutine from Section 2.1.2 to search for it.

**0-Balance Checking** Assume, that there are no +3-substrings and $-1$-prefixes in the string. We want to check that $f(u[0, n - 1]) = 0$. We should check one of those two conditions:

- There is a prefix $u[0, k]$ such that $f(u[0, k]) = 2$ and there is a suffix $u[k, n - 1]$ such that $f(u[k, n - 1]) = -2$. This condition is true if and only if there is a $+2$-substring and the rightmost $-2$-substring $u[l, r]$ is such that $r = n - 1$.
- There are no $+2$-substrings, therefore any prefix $u[0, k]$ is such that $f(u[0, k]) \in \{0, 1\}$. In that case $u[n - 1] = b$ as it was for Dyck 2 language in Section 2.2.1

**The Algorithm** We use SUBSTRING3_ANY in Algorithm 1 from Section 2.1.2 to search for a $\pm 3$-substring. We use $s = \{+1\}$ for a $+3$-substring and $s = \{-1\}$ for a $-3$-substring. We use SUBSTRING2_FIRST in Algorithm 2 from Section 2.1.1 to search for the leftmost or rightmost $\pm 2$-substring.

Finally, we obtain the following Algorithm 13 that returns true if the string $u$ belongs to Dyck 2 language and false otherwise.

---

**Algorithm 12** Dyck \(_1\)

- $v = (i, j) \leftarrow$ SUBSTRING\(_2\)_ANY(0, $n - 1, \{+1, -1\}$)  
  \(\triangleright\) Invoke the $\pm 2$ substring search algorithm
- $result \leftarrow$ True
- if $v \neq$ NULL then  
  \(\triangleright\) $\pm 2$ substring existence checking
  - $result \leftarrow$ False
- if $u[0] = b$ then  
  \(\triangleright\) $f(u[0, i]) = -1$ prefix substring existence checking
  - $result \leftarrow$ False
- if $u[n - 1] \neq b$ then  
  \(\triangleright\) $f(u[0, n - 1]) \neq 0$ condition checking
  - $result \leftarrow$ False
- return result

---

**2.2.2 Dyck\(_2\)**

We search a substring that breaks the rule of Dyck\(_2\). There are two types of such substrings:

- a substring $u[l, r]$ with balance $f(u[l, r]) = 3$.
- a substring $u[0, r]$ (prefix) with balance $f(u[l, r]) = -1$.

Additionally, the number of $a$ and $b$ symbols should be equal, i.e. $f(u[0, n - 1]) = 0$.

We search +3-substring using SUBSTRING3_ANY procedure with $s = \{+1\}$ that is described in Algorithm 1 from Section 2.1.2 Searching for $-1$-prefix and checking 0-balance are discussed in the following two paragraphs.

**No $-1$-prefix Checking** Assume that we already checked that there are no +3-substrings. If we want to find a substring $u[0, r]$ (prefix) with balance $f(u[0, r]) = -1$, then there are three cases:

- the symbol $u[0]$ is $b$.
- There is a prefix $u[0, k]$ and a substring $u[k, t]$ such that $f(u[0, k]) \leq 1$ and $f(u[k, t]) = -2$. The condition is right if and only if there are two most left substrings $u[l_{-}, r_{-}]$ and $u[l_{+}, r_{+}]$ such that $f(u[l_{-}, r_{-}]) = -2$, $f(u[l_{+}, r_{+}]) = +2$ and $u[l_{-}, r_{-}]$ is on the left of $u[l_{+}, r_{+}]$. In other words $l_{-} \leq r_{-} \leq l_{+} \leq r_{+}$. We can use the subroutine from Section 2.1.1 for searching $\pm 2$-substring.
- There is a $-3$-substring $u[l, r]$. We can use the subroutine from Section 2.1.2 to search for it.

**0-Balance Checking** Assume, that there are no +3-substrings and $-1$-prefixes in the string. We want to check that $f(u[0, n - 1]) = 0$. We should check one of those two conditions:

- There is a prefix $u[0, k]$ such that $f(u[0, k]) = 2$ and there is a suffix $u[k, n - 1]$ such that $f(u[k, n - 1]) = -2$. This condition is true if and only if there is a $+2$-substring and the rightmost $-2$-substring $u[l, r]$ is such that $r = n - 1$.
- There are no $+2$-substrings, therefore any prefix $u[0, k]$ is such that $f(u[0, k]) \in \{0, 1\}$. In that case $u[n - 1] = b$ as it was for Dyck 2 language in Section 2.2.1

**The Algorithm** We use SUBSTRING3_ANY in Algorithm 1 from Section 2.1.2 to search for a $\pm 3$-substring. We use $s = \{+1\}$ for a $+3$-substring and $s = \{-1\}$ for a $-3$-substring. We use SUBSTRING2_FIRST in Algorithm 2 from Section 2.1.1 to search for the leftmost or rightmost $\pm 2$-substring.

Finally, we obtain the following Algorithm 13 that returns true if the string $u$ belongs to Dyck 2 language and false otherwise.
Algorithm 13. The Quantum Algorithm for Dyck\(_2\) language.

\begin{algorithm}
result ← True
\textbf{if} \(v \neq \text{NULL} \) then
\textbf{end if}
\textbf{if} \(u[0] = b \) then
\textbf{end if}
\textbf{if} \(v \neq \text{NULL} \) then
\textbf{end if}
\v = (i, j) ← \text{SUBSTRING\(_3\)_ANY}(0, n - 1, \{+1\})
\textbf{if} \(v \neq \text{NULL} \) then
\textbf{end if}
\textbf{if} \(v \neq \text{NULL} \) then
\textbf{end if}
v\_ = (i\_, j\_) ← \text{SUBSTRING\(_2\)_FIRST}(0, n - 1, \{-1\})
\textbf{if} \(v\_ \neq \text{NULL} \) and \((v\_ \neq \text{NULL} \text{ or } j\_ < i\_+ \) then
\textbf{end if}
\v\_ = (i\_, j\_) ← \text{SUBSTRING\(_2\)_FIRST}(0, n - 1, \{-1\}, \text{right})
\v\_ = (i\_, j\_) ← \text{SUBSTRING\(_2\)_FIRST}(0, n - 1, \{-1\}, \text{left})
\textbf{if} \(v\_ = \text{NULL} \) and \(v\_+ = \text{NULL} \) and \(u[n - 1] = b \) then
\textbf{end if}
\textbf{return} result
\end{algorithm}

Lemma 13. The expected running time of Algorithm 13 is \(O(\sqrt{n} \log n)\). The error probability is at most 0.5.

\textit{Proof.} There are two invocation of \text{SUBSTRING\(_3\)_ANY} that has \(O(\sqrt{n} \log n)\) running time and three invocations of \text{SUBSTRING\(_2\)_FIRST} that has \(O(\sqrt{n})\) running time due to Lemma 2.1.2 and Lemma 2.1.1. Therefore, the total running time is \(O(\sqrt{n} \log n)\) and the error probability is at most 0.5. The correctness of the algorithm is proven in the above discussion. \(\square\)

2.2.3 Dyck\(_3\)

We search a substring that breaks the rule of Dyck\(_3\). There are two types of substrings:

- a substring \(u[l, r]\) with balance \(f(u[l, r]) = 4\).
- a substring \(u[0, r]\) (prefix) with balance \(f(u[0, r]) = -1\).

Additionally, the number of \(a\) and \(b\) symbols should be equal, i.e. \(f(u[0, n - 1]) = 0\).
We search +4-substring using \texttt{SUBSTRING}$_4$\texttt{ANY} procedure with \(s = \{+1\}\) that is described in Algorithm \texttt{4} from Section 2.1.3. Searching −1-prefix and Checking 0-balance are discussed in the following two paragraphs.

\textbf{No −1-prefix Checking} Assume that we already checked that there are no +4-substring. If we want to find a substring \(u[0, r]\) (prefix) with balance \(f(u[0, r]) = -1\), then we can have one of three cases:

\begin{itemize}
  \item the symbol \(u[0]\) is \(b\).
  \item There is a prefix \(u[0, k]\) and a substring \(u[k, l]\) such that \(f(u[0, k]) \leq 1\) and \(f(u[k, l]) = -2\). The condition is right if and only if there are two most left substrings \(u[l, r]\) and \(u[l, r]\) such that \(f(u[l, r]) = -2\), \(f(u[l, r]) = +2\) and \(u[l, r]\) is on the left of \(u[l, r]\). In other words \(l \leq r_0 \leq l_4 \leq r_4\). We can use the subroutine from Section 2.1.4 for searching ±2-substring.
  \item There is a suffix \(u[k, l]\) and a substring \(u[k, l]\) such that \(f(u[0, k]) \leq 2\) and \(f(u[k, l]) = -3\). The condition is right if and only if there are two most left substrings \(u[l, r]\) and \(u[l, r]\) such that \(f(u[l, r]) = -3\), \(f(u[l, r]) = +3\) and \(u[l, r]\) is on the left of \(u[l, r]\). We can use the subroutine from Section 2.1.2 for searching ±3-substring.
  \item There is a −4-substring \(u[l, r]\). We can use the subroutine from Section 2.1.3 for searching it.
\end{itemize}

\textbf{0-Balance Checking} Assume, that there are no +4-substring and −1-prefix in the string. Let us check that \(f(u[0, n - 1]) = 0\).

We should check one of the two conditions.

\begin{itemize}
  \item There is a prefix \(u[0, k]\) such that \(f(u[0, k]) = 3\) and there is a postfix \(u[k, n - 1]\) such that \(f(u[k, n - 1]) = -3\). This condition is true iff there is a +3-substring and the most right −3-substring \(u[l, r]\) is such that \(r = n - 1\).
  \item There is a prefix \(u[0, k]\) such that \(f(u[0, k]) = 2\) and there is a postfix \(u[k, n - 1]\) such that \(f(u[k, n - 1]) = -2\). This condition is true iff there are no +3-substring, there is a +2-substring and the most right −2-substring \(u[l, r]\) is such that \(r = n - 1\).
  \item There is no +2-substrings, therefore any prefix \(u[0, k]\) is such that \(f(u[0, k]) \in \{0, 1\}\). In that case \(u[n - 1] = b\) as it was for \texttt{Dyck}_1 language in Section 2.2.1.
\end{itemize}

\textbf{An Algorithm} For searching a ±4-substring, we use \texttt{SUBSTRING}$_4$\texttt{ANY} procedure that is described in Algorithm \texttt{4} from Section 2.1.3. We use \(s = \{+1\}\) for a +4-substring and \(s = \{-1\}\) for a −4-substring. For searching most left or most right ±2-substrings and ±3-substrings we use \texttt{SUBSTRING}$_2$\texttt{FIRST} and \texttt{SUBSTRING}$_3$\texttt{FIRST} procedures that are described in Algorithm \texttt{4} from Section 2.1.1 and Algorithm \texttt{5} from Section 2.1.2.

Finally, we obtain the following Algorithm \texttt{14} that returns \texttt{true} if the string \(u\) belongs to \texttt{Dyck}_3 language and \texttt{false} otherwise.

\textbf{Lemma 14.} The expected running time of Algorithm \texttt{14} is \(O((\sqrt{n}(\log n)^2)^2)\). The error probability is at most 0.5.

\textit{Proof.} There are two invocations of \texttt{SUBSTRING}$_4$\texttt{ANY} that has \(O((\sqrt{n}(\log n)^2)^2)\) running time and three invocations of \texttt{SUBSTRING}$_q$\texttt{FIRST} for \(q \in \{2, 3\}\) that has \(O((\sqrt{n}(\log n)^n)^q)\) running time due to Lemma \texttt{1} and \texttt{12} and Lemma \texttt{11}. Therefore, the total running time is
\[
O(2\sqrt{n}(\log n)^2 + 3\sqrt{n} \log n + 3\sqrt{n}) = O\left(3\sqrt{n} \left(\frac{2}{t=0} (\log n)^{t} + 1\right)\right) = O\left(3\sqrt{n} \frac{(\log n)^{2} + 1}{\log n - 1}\right) = O\left(\sqrt{n}(\log n)^2\right).
\]

The error probability is at most 0.5. The correctness of the algorithm is proven in the above discussion. \qed

17
2.2.4 Dyck$_h$

We search a substring that breaks the rule of Dyck$_h$. There are two types of such substrings:

- a substring $u[l, r]$ with balance $f(u[l, r]) = h + 1$.
- a substring $u[0, r]$ (prefix) with balance $f(u[0, r]) = -1$.

Additionally, the number of $a$ and $b$ symbols should be equal, i.e. $f(u[0, n-1]) = 0$.

We search $(h + 1)$-substring using the Substring$_h$Any procedure with $s = \{+1\}$ that is described in Algorithm 10 from Section 2.1.4. Searching $-1$-prefix and Checking 0-balance are discussed in the following two paragraphs.

No $-1$-prefix Checking Assume that we already checked that there are no $(h + 1)$-substring. If we want to find a substring $u[0, r]$ (prefix) with balance $f(u[0, r]) = -1$, then there are three cases:

- the symbol $u[0]$ is $b$.
- For some $q \in \{2 \ldots h\}$, we have the following condition. There is a prefix $u[0, k]$ and a substring $u[k, t]$ such that $f(u[0, k]) \leq q - 1$ and $f(u[k, t]) = q$. The condition is right if and only if there are two most left substrings $u[l_-, r_-]$ and $u[l_+, r_+]$ such that $f(u[l_-, r_-]) = -q$, $f(u[l_+, r_+]) = +q$ and $u[l_-, r_-]$ is on the left of $u[l_+, r_+]$. In other words $l_- \leq r_- \leq l_+ \leq r_+$. We can use the subroutine from Section 2.1.4 to search for $\pm q$-substrings.
- There is a $-(h + 1)$-substring $u[l, r]$. We can use the subroutine from Section 2.1.4 to search for it.

0-Balance Checking Assume, that there are no $(h + 1)$-substrings and $-1$-prefixes in the string. We want to check that $f(u[0, n-1]) = 0$. We should check the following condition.

- For any $q \in \{1, \ldots, h\}$ we have. If there is a prefix $u[0, k]$ such that $f(u[0, k]) = q$, then it should be a postfix $u[k, n-1]$ such that $f(u[k, n-1]) = -q$. This condition is true if there is a $+q$-substring and the rightmost $-q$-substring $u[l, r]$ is such that $r = n - 1$. To check that $q = 1$, it is enough to check that $u[n-1] = b$. The same logic was presented and proved for the Dyck$_1$ language in Section 2.2.1.

The Algorithm We use the Substring$_{(h+1)}$Any procedure (Algorithm 10 from Section 2.1.4) to search for a $\pm (h + 1)$-substring. We use $s = \{+1\}$ for a $(h + 1)$-substring and $s = \{-1\}$ for a $-(h + 1)$-substring. We also use Substring$_q$First (Algorithm 11 from Section 2.1.4) to search for the leftmost or rightmost $\pm q$-substrings where $q \in \{2 \ldots h\}$.

Finally, we obtain the following Algorithm 15 that returns true if the string $u$ belongs to Dyck$_h$ language and false otherwise.

Theorem 15. The expected running time of Algorithm 15 is $O(\sqrt{n}(\log n)^{h-1})$. The error probability is at most 0.5.

Proof. There are two invocations of Substring$_{(h+1)}$Any that has $O(\sqrt{n}(\log n)^{h-1})$ running time and three invocations of Substring$_q$First for $q \in \{2, \ldots h\}$ that has $O(\sqrt{n}(\log n)^{q-2})$ running time due to Lemma 10 and Lemma 2.1.3. Therefore, the total running time is

$$O\left(2\sqrt{n}(\log n)^{h-1} + \sum_{q=2}^{h} 3\sqrt{n}(\log n)^{q-2}\right) = O\left(3\sqrt{n}\sum_{k=0}^{h-1}(\log n)^k\right) = O\left(3\sqrt{n}\frac{(\log n)^{h-1}}{\log n - 1}\right) = O\left(\sqrt{n}(\log n)^{h-1}\right).$$

The error probability is at most 0.5. The correctness of the algorithm is proven in the above discussion. \qed
3 Lower Bounds for Dyck Languages with Bounded Height

Now let’s show some lower bounds for Dyck languages with bounded height.

Let \( k \in \mathbb{N}^+ \). Let \( M_k^0 = \{ a, b \} \). For all \( i \in \mathbb{N}^+ \), let \( M_k^i = \{ a^k b^k w | w \in (M_k^{i-1})^{2k-1}, f(w) = \pm 1 \} \). Here \( f \) is the balance function that we defined in the notation paragraph.

According to our construction, for all \( m \in \mathbb{N}^+ \), we have \( f(m) = \pm 1 \). All words in \( M_k^i \) have the same length that we define as \( l_k(i) \).

**Lemma 16.** \( l_k(i) \sim 2 \cdot (2k)^i \) as \( k \to \infty \).

**Proof.** We have \( l_k(0) = 1 \). For all \( i \in \mathbb{N}^+ \), \( l_k(i) = 2k + (2k - 1)l_k(i - 1) \).

Thus, \( l_k(i) = (2k - 1)^{i-1}((4k - 1) + \frac{k}{k-1}) - \frac{k}{k-1} \sim 2 \cdot (2k)^i \) as \( k \to \infty \).

Define \( h_k(i) = \max_{m \in M_k(i)} (h(m)) \).

**Lemma 17.** For all \( k \in \mathbb{N}^+ \), \( i \in \mathbb{N}^+ \), \( h_k(i) = (i + 1)k \). Furthermore, for all \( m \in M_k^i \), \( h^- (m) \geq 0 \).

**Proof.** This can be shown easily by induction on \( i \).

By induction, define \( g_k^i = g_k \) and \( g_k^{i+1} = g_k \circ (g_k^i, \cdots, g_k^i) \).

**Lemma 18.** For all \( k \in \mathbb{N}^+ \), \( i \in \mathbb{N}^+ \), \( \text{ADV}^\pm (g_k^i) \geq k^i \).

**Proof.** Inspiring from \([6]\) Prop 3.32, we can show that \( \text{ADV}^\pm (g_k) \geq k \).

Since \( \text{ADV}^\pm \) composes exactly even for partial Boolean functions \( f \) and \( g \), meaning, \( \text{Adv}^\pm (f \circ g) = \text{Adv}^\pm (f) \cdot \text{Adv}^\pm (g) \) \([11]\) Lemma 6], we have \( \text{ADV}^\pm (g_k^i) \geq k^i \).

The reason that we introduced \( g_k^i \) is the following: Let \( m \in M_k^i \), \( mb \in \text{Dyck}_{i+1}(l(i)) \) iff \( g_k^i (m') = 1 \), where \( m' \) is obtained from \( m \) by removing all the \( a^k s' \) and \( b^k s' \) appeared in the construction of each \( M_k^j \) for \( j = 1 \) to \( i \).

Now let’s study the lower bound of Dyck language with bounded height.

**Theorem 19.** For all \( \epsilon > 0 \), \( Q(\text{Dyck}_{\Theta(n^{1/\epsilon})}(n)) = \Theta(n) \)

**Proof.** We know that \( l_k(i) \sim 2 \cdot (2k)^i \). \( h_k(i) = (i + 1)k \). From \([19]\], we have \( Q(g^i) = \text{ADV}^\pm (g_k^i) \). Thus Lemma \([11]\) shows that \( Q(g^i) \geq k^i \).

By taking \( i = \text{constant} \), \( k = \Theta(n^{1/\epsilon}) \), we have \( l_k(i) = n \) and \( Q(g^i) = \Theta(n) \). Furthermore, by the equivalence above, computing \( g_k^n \) corresponds to checking if words of height \( \Theta(n^{1/\epsilon}) \) are in Dyck. Thus \( Q(\text{Dyck}_{\Theta(n^{1/\epsilon})}(n)) = \Theta(n) \). This is true for all \( i \in \mathbb{N}^+ \). Therefore, for all \( \epsilon > 0 \), \( Q(\text{Dyck}_{\Theta(n^{1/\epsilon})}(n)) = \Theta(n) \).

**Theorem 20.** \( Q(\text{Dyck}_{\Theta(n^{1/\epsilon})}(n)) = \Omega(n/2^i) \) for \( i = i(n) \), such that \( i(n) \in [\omega(1), o(\log n)] \) as \( n \to \infty \).

**Proof.** We know that \( l_k(i) \sim 2 \cdot (2k)^i \) when \( k = k(n) = \omega(1) \). \( h_k(i) = (i + 1)k \sim ik \) when \( i = \omega(1) \). \( Q(g^i) = \text{ADV}^\pm (g_k^i) \geq k^i \).

By replacing \( k \) by \( \Theta(n^{1/\epsilon}) \), we obtain \( Q(\text{Dyck}_{\Theta(n^{1/\epsilon})}(n)) = \Omega(n/2^i) \).
Theorem 21. For every $0 < \epsilon \leq 1 - \log_3(2) \approx 0.37$, there exists $c > 0$ such that $Q(Dyck_{c \log(n)}(n)) = \Omega(n^{1-\epsilon})$.

Proof. We know that $l_k(i) \sim 2 \cdot (2k-1)^i$, $h_k(i) = (i+1)k \sim ik$ when $i(n) \to \infty$ and $k$ equals to a constant, $k > 1$. $Q(g^i) = ADV^\pm(g_k) \geq k^i$.

By taking $i$ as $\Theta(\log(n))$, we obtain $h = c \log(n)$ for some $c > 0$, $k^i = 2(2k-1)^{i(1-\epsilon)}$ for $\epsilon = 1 - \log_{2k-1}(k)$. Since $k$ is an integer, $\log_{2k-1}(k) \leq \log_3(2)$ when $k \geq 2$.

For every $0 < \epsilon \leq 1 - \log_3(2) \approx 0.37$, there exists $c > 0$ such that $Q(Dyck_{c \log(n)}(n)) = \Omega(n^{1-\epsilon})$.  

Acknowledgement We thank Frédéric Magniez for introducing us to the problem and for helpful discussions.

References

[1] Scott Aaronson, Daniel Grier, and Luke Schaeffer. A quantum query complexity trichotomy for regular languages. Electronic Colloquium on Computational Complexity (ECCC), 26:61, 2018.

[2] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. SIAM J. Comput., 30(6):1842–1862, December 2001.

[3] Andris Ambainis, Kaspars Balodis, Janis Iraids, Krisjanis Prūsis, and Juris Smotrov. Quantum lower bounds for 2d-grid and dyck language, 2019.

[4] Andris Ambainis and Abuzer Yakaryılmaz. Automata and quantum computing, 2015.

[5] Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. Theoretical Computer Science, 494:13 – 23, 2013. Theory and Applications of Models of Computation (TAMC 2010).

[6] Aleksandrs Belovs. Applications of the Adversary Method in Quantum Query Algorithms. PhD thesis, 2014.

[7] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. Fortschritte der Physik, 46(4-5):493–505, 1998.

[8] Harry Buhrman, Subhasree Patro, and Florian Speelman. The quantum strong exponential-time hypothesis, 2019.

[9] D Deutsch and R Jozsa. Rapid solution of problems by quantum computation. Technical report, Bristol, UK, UK, 1992.

[10] Lov K Grover. A fast quantum mechanical algorithm for database search. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 212–219. ACM, 1996.

[11] Shelby Kimmel. Quantum adversary (upper) bound. In Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I, ICALP’12, pages 557–568, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, Automata Studies, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[13] Robin Kothari. An optimal quantum algorithm for the oracle identification problem. In 31st International Symposium on Theoretical Aspects of Computer Science, page 482, 2014.
[14] C. Y.-Y. Lin and H.-H. Lin. Upper bounds on quantum query complexity inspired by the elitzur-vaidman bomb tester. In 30th Conference on Computational Complexity (CCC 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[15] C. Y.-Y. Lin and H.-H. Lin. Upper bounds on quantum query complexity inspired by the elitzur-vaidman bomb tester. Theory of Computing, 12(18):1–35, 2016.

[16] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. In Proceedings of the Forty-second ACM Symposium on Theory of Computing, STOC ’10, pages 261–270, New York, NY, USA, 2010. ACM.

[17] Michael A. Nielsen and Isaac L. Chuang. Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[18] Michael O. Rabin. Probabilistic automata. Information and Control, 6(3):230 – 245, 1963.

[19] Ben W. Reichardt. Reflections for quantum query algorithms. In Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’11, pages 560–569, Philadelphia, PA, USA, 2011. Society for Industrial and Applied Mathematics.
Algorithm 14 Dyck3(). The Quantum Algorithm for Dyck3 language.

\[ \text{result} \leftarrow \text{True} \]

\[ v = (i, j) \leftarrow \text{Substring}_4 \text{Any}(0, n-1, \{+1\}) \]  
\[ \triangleright \text{Invoke the +4 substring search algorithm} \]

\[ \text{if } v \neq \text{NULL then} \]
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright +4 \text{ substring existence checking} \]

\[ \text{end if} \]

\[ \text{if } u[0] = b \text{ then} \]
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright f(u[0, i]) = -1 \text{ prefix substring existence checking} \]

\[ \text{end if} \]

\[ v = (i, j) \leftarrow \text{Substring}_4 \text{Any}(0, n-1, \{-1\}) \]  
\[ \triangleright \text{Invoke the -4 substring search algorithm} \]

\[ \text{if } v \neq \text{NULL then} \]
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright -4 \text{ substring existence checking} \]

\[ \text{end if} \]

\[ \text{for } q \in \{2, 3\} \text{ do} \]
\[ v_{+q} = (i_{+q}, j_{+q}) \leftarrow \text{Substring}_q \text{First}(0, n-1, \{+1\}, \text{right}) \]  
\[ \triangleright \text{Invoke the first } +q \text{-substring search algorithm} \]

\[ v_{-q} = (i_{-q}, j_{-q}) \leftarrow \text{Substring}_q \text{First}(0, n-1, \{-1\}, \text{right}) \]  
\[ \triangleright \text{Invoke the first } -q \text{-substring search algorithm} \]

\[ \text{if } v_{-q} \neq \text{NULL and}(v_{+q} = \text{NULL or } j_{-q} < i_{+q}) \text{ then} \]  
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright -q \text{-substring exists and it is on the left of } +q \text{-substring} \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{if } \text{result} = \text{True then} \]
\[ \quad \text{for } q \in \{3, \ldots, 2\} \text{ do} \]
\[ v_{+q} \neq \text{NULL then} \]
\[ v = (i, j) \leftarrow \text{Substring}_q \text{First}(0, n-1, \{-1\}, \text{left}) \]  
\[ \triangleright \text{Searching the most right } -q \text{-substring} \]

\[ \text{if } v = \text{NULL or } j \neq n-1 \text{ then} \]
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright \text{If there is no error then we check 0-balance} \]

\[ \text{end if} \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{if } u[n-1] \neq b \text{ then} \]
\[ \quad \text{result} \leftarrow \text{False} \]  
\[ \triangleright \text{The first condition of 0-balance checking} \]

\[ \text{end if} \]

\[ \text{return result} \]
Algorithm 15 $\text{DYCK}_h()$. The Quantum Algorithm for $\text{Dyck}_h$ language.

\begin{algorithm}
\textbf{result} $\leftarrow$ True
\textbf{v} $\leftarrow$ $\text{SUBSTRING}_{h+1}(0, n-1, \{+1\})$ $\triangleright$ Invoke the $(h+1)$ substring search algorithm
\textbf{if} $v \neq \text{NULL}$ then $\triangleright$ $(h+1)$ substring existence checking
\hspace{1em} \textbf{result} $\leftarrow$ False
\textbf{end if}
\textbf{if} $u[0] = b$ then $\triangleright$ $f(u[0, i]) = -1$ prefix substring existence checking
\hspace{1em} \textbf{result} $\leftarrow$ False
\textbf{end if}
\textbf{v} $\leftarrow$ $\text{SUBSTRING}_{h+1}(0, n-1, \{-1\})$ $\triangleright$ Invoke the $-(h+1)$-substring search algorithm
\textbf{if} $v \neq \text{NULL}$ then $\triangleright$ $-(h+1)$-substring existence checking
\hspace{1em} \textbf{result} $\leftarrow$ False
\textbf{end if}
\textbf{for} $q \in \{2, \ldots, h-1\}$ do
\hspace{1em} \textbf{v}_{+q} = $(i_{+q}, j_{+q})$ $\leftarrow$ $\text{SUBSTRING}_{q}\text{FIRST}(0, n-1, \{+1\}, \text{right})$ $\triangleright$ Invoke the first $+q$-substring search algorithm
\hspace{1em} \textbf{v}_{-q} = $(i_{-q}, j_{-q})$ $\leftarrow$ $\text{SUBSTRING}_{q}\text{FIRST}(0, n-1, \{-1\}, \text{right})$ $\triangleright$ Invoke the first $-q$-substring search algorithm
\hspace{1em} \textbf{if} $v_{-q} \neq \text{NULL}$ and ($v_{+q} = \text{NULL}$ or $j_{-q} < i_{+q}$) then $\triangleright$ $-q$-substring exists and it is on the left of $+q$-substring
\hspace{2em} \textbf{result} $\leftarrow$ False
\textbf{end if}
\textbf{end for}
\textbf{if} $\text{result} = True$ then $\triangleright$ If there is no error then we check 0-balance
\textbf{for} $q \in (h-1, \ldots, 2)$ do
\hspace{1em} \textbf{if} $v_{+q} \neq \text{NULL}$ then
\hspace{2em} $v = (i, j) \leftarrow$ $\text{SUBSTRING}_{q}\text{FIRST}(0, n-1, \{-1\}, \text{left})$ $\triangleright$ Searching the most right $-q$-substring
\hspace{2em} \textbf{if} $v = \text{NULL}$ or $j \neq n-1$ then
\hspace{3em} \textbf{result} $\leftarrow$ False
\hspace{2em} \textbf{end if}
\hspace{1em} \textbf{end if}
\textbf{end for}
\textbf{if} $u[n-1] \neq b$ then $\triangleright$ The first condition of 0-balance checking
\hspace{1em} \textbf{result} $\leftarrow$ False
\textbf{end if}
\textbf{return} \textbf{result}
\end{algorithm}