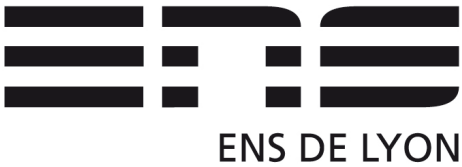
École Normale Supérieure de Lyon
Latvijas Universitāte



UNIVERSITY OF LATVIA
FACULTY OF COMPUTING

M1 Internship Repport

Complexity of recognizing Dyck language of bounded height with quantum query algorithms.

Key words: *Quantum query complexity, Dyck words, Quantum algorithms, regular languages, star free languages, adversary methods.*

Student:
Maxime Cautrès

Supervisor:
Andris Ambainis
Kamil Khadiev

May the 2nd 2022 - July the 22th 2022

Contents

1 Introduction **2**
- 1.1 History of quantum computing . . . 2
- 1.2 The quantum circuit, and quantum query model and complexity . . . 2
- 1.3 Dyck Languages of height k . . . 4
- 1.4 State of the art . . . 4
- 1.5 Goals of the internship . . . 4
- 1.6 Results . . . 5

2 Preliminaries **5**
- 2.1 Quantum query for regular languages. . . . 5
  - 2.1.1 Regular languages . . . 5
  - 2.1.2 Star free languages . . . 6
  - 2.1.3 Trichotomy theorem . . . 6
- 2.2 The bounds for $\textsc{Dyck}_{k,n}$ problem . . . 6
  - 2.2.1 Lower bound on the quantum query complexity of $\textsc{Dyck}_{k,n}$ . . . 7
  - 2.2.2 Best known algorithm to recognize $\textsc{Dyck}_{k,n}$ . . . 7

3 A better algorithm for $\textbf{Dyck}_{k,n}$ **8**
- 3.1 A better Complexity Analysis of the original algorithm . . . 8
- 3.2 A new algorithm for $\textsc{Dyck}_{2,n}$ . . . 9
- 3.3 A new algorithm for k=3 . . . 12
- 3.4 A try for a new algorithm for any k. . . . 12

4 Conclusion **12**

5 Appendix **14**

A The algorithm for $\textbf{Dyck}_{k,n}$ **14**

B The proof of the quantum query complexity for $\textbf{Dyck}_{k,n}$ algorithm's subroutines **14**

1 Introduction

Context of the internship

As part of the first year of Master at the École Normale Supérieure de Lyon, I was able to do a 12 weeks research internship in a laboratory.

My research for an internship in Quantum Algorithmic have brought me to the Faculty of Computing at the University of Latvia and my supervisor Andris Ambainis. My research also brings me to discuss with Kamil Khadiev from Kazan Federal University who has become my co-supervisor. We have discussed by email to find an interesting subject of research on which I have liked to work on. I thank them for their help, their supervision and the time they have given to me during this 12 weeks.

During the internship, I have been integrated to the life of the Center for Quantum Computing Science. I thanks members of the team for the great discussions we had after the seminar.

I also want to thank Omar Fawzi for having introduced me to quantum computing and its fascinating possibilities.

The team's research area is quantum algorithms and complexity theory. More precisely, the team works on establishing new quantum algorithm with better complexity and proving new lower bound to the quantum complexity for many different type of problem belonging from graph theory to cryptography passing by language recognition theory. My work on the recognition of restricted Dyck words integrate itself great in the team work as it has already been studied by the team few years ago [4] and further by Kamil Khadiev [7].

My internship, named "Complexity of recognizing Dyck language with a quantum computer", as for goal to reduce the gap between the lower and the upper bound for a quantum query algorithm that recognizes Dyck words of bounded height. The best lower and upper bound are describe in [4] by Andris Ambainis team.

In the end of the introduction the field of research will be presented more precisely. After that, technical preliminaries, which are useful to understand the current and the new results, ill be detailed. Finally, the last section presents the new results on the problem and the failures.

1.1 History of quantum computing

The history of quantum computing has started in 1980 when Paul Benioff, an american physicist, proposed a quantum mechanical model of the Turing machine [5]. This machine use some properties of the matter that has been discovered by quantum physicist. After that, some computer scientists suggested that the quantum model of turing machine may be more expressive that the classical model. Few years after, the first bricks of the quantum circuit have been introduced by Richard Feymann [6]. The first quantum computers have started to arrived middle of 1990s. During the last 20 years, the founds given to the creation of the first quantum computer have skyrocketed, as the number of start-up and company dedicated to it. This emulation has made from the quantum computer field one of the most active field of research today. On the algorithmic side, the first astonishing result is the algorithm designed by Peter Shor (1994) [9]. The algorithm improves a lot the complexity of factorizing integers, enough to break our cryptographic protocols when quantum computer will be powerful enough. Since 1994, the quantum algorithm area has evolved almost independently from the quantum computers and has developed many beautiful theories and interesting results. But how does a quantum circuit works ?

1.2 The quantum circuit, and quantum query model and complexity

In classical computer science, the piece of information is represented by using 0 and 1. This two states can be easily obtained using electricity with 0 equal to 0V and 1 equal 5V. It is easy to propagate electricity through wires and to stock its level into capacitor. Moreover a little piece

of hardware, named transitor, has allow to do some computations using logical gates which when include in a complex machine create our so-called "computers".

In quantum computer, the story isn't so different. First, the 0 and 1 are now represented using particles like electrons or photons. For example, an electron with a spin of $+\frac{1}{2}$ (note $|1\rangle$) represents a 1 and an other one with a spin of $-\frac{1}{2}$ (note $|0\rangle$) represents a 0. But the use of particles is motivated by their properties and mainly by a quantum property called superposition. A quantum state is not only $|0\rangle$ or $|1\rangle$, but can be $\lambda_0|0\rangle + \lambda_1|1\rangle$ for every $\lambda_{0,1} \in \mathbb{C}$ such that $\lambda_0 + \lambda_1 = 1$ and a quantum bit is now called a qubit. As before, the computations are done by gates, here quantum gates which transform the quantum state of the qubit into another one. At the end, to get the result of a computation, it is mandatory to measure the state of the quantum system, which break the quantum superposition. A quantum state of n qubits can be represented with a length 1 vector in a 2^n dimensional space and a quantum gates by a linear unitary transformation on a 2^n dimensional space. A transformation is said unitary if is preserved the length.

A quantum circuit is a precise configuration of quantum gates on a finite number of qubits. The following Figure 1 represents the quantum circuit that computes a uniform randomize on $\{0,1\}^n$.

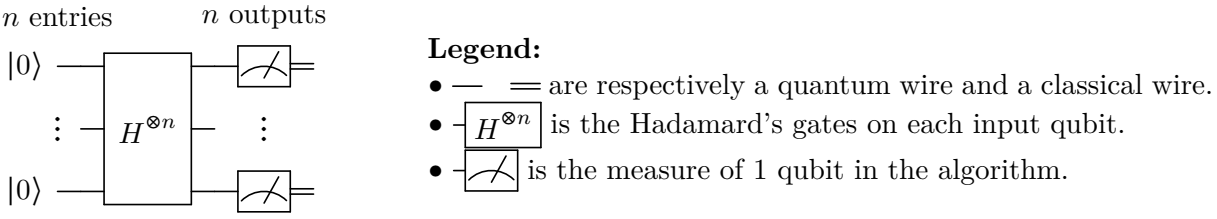


Figure 1: A quantum circuit computing the uniform random on $\{0,1\}^n$.

The quantum query circuit is a quantum circuit used to compute function with an entry space where $x = x_1 \dots x_N$ belongs. The black box model [2] of a quantum query circuit is composed of an input state $|\psi_{start}\rangle$ and a sequence $U_0, Q, U_1, \dots, Q, U_T$ of linear unitary transformations such that $|\psi_{start}\rangle$ and all U_i do not depend on entry x unlike the Q_i which depend on x. The quantum state $|\psi_{start}\rangle$ belong to a d-dimensional space generated by $|1\rangle, |2\rangle, \dots, |d\rangle$. To define Q, the basis vectors first need to be renamed from $|1\rangle, \dots, |d\rangle$ to $|i,j\rangle$ with $i \in [\![0, N]\!]$ and $j \in [\![1, d_i]\!]$ for some d_i such that $d_1 + d_2 + \dots + d_N = d$. Next, Q is define such that

$$Q(|i,j\rangle) := \begin{cases} |0,j\rangle & \text{if } i = 0 \\ |i,j\rangle & \text{if } i > 0 \text{ and } x_i = 0 \\ -|i,j\rangle & \text{if } i > 0 \text{ and } x_i = 1. \end{cases}$$

The gates Q are doing the queries to input x by flipping some of the vectors. Finally, to get the output of the quantum query algorithm it is necessary to measure the output. A quantum query algorithm can the summarized with the following quantum circuit.

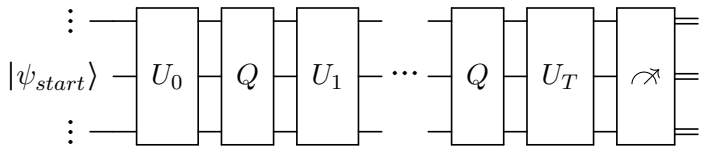


Figure 2: Structure of a quantum query algorithm.

The quantum query complexity of an algorithm corresponds to the number of calls to the Q gate. Often, this number of calls is depending on the size of the entry. The quantum query complexity of a problem corresponds to the higher bound for which it is certain there is no quantum query algorithm with a greater quantum query complexity solving the problem.

1.3 Dyck Languages of height k

First, the Dyck Language corresponds to the set of correct and balanced word of parenthesis. The Dyck language is a context free language as it can easily be recognized using a context free grammar. The work done by Andris Ambainis' team [4] focus on a restriction Dyck language with fixed height k. More precisely, a Dyck word has a height of k if, in every of its prefix, the difference between the number of opening and closing parenthesis does not exceeds k. These restricted Dyck languages are called $\text{DYCK}_{k,n}$ and are interesting because they belong to the already well studied class of star free languages (Detail in subsubsection 2.1.2).

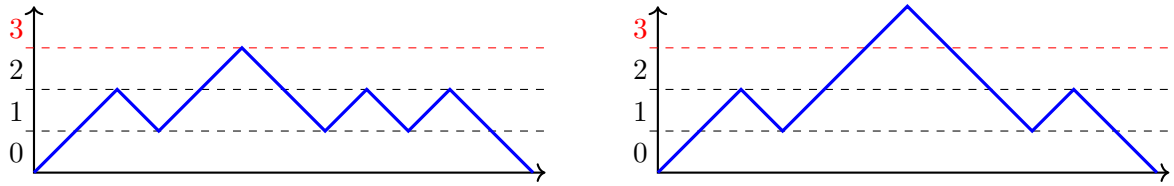


Figure 3: **On the left**, a valid Dyck word of height 3. **On the right**, an invalid Dyck word of height 3.

1.4 State of the art

This state of the art will not be to precise as the understanding of the bibliography has taken almost the first half the internship and will be more detailed in the section 2. First, few years ago Aaronson, Grier and Schaeffer [1, 2019] worked on quantum complexity of recognizing regular languages as they can model a lot of tasks. They concluded that there are 3 different cases depending of the language:

- $O(1)$ if it is sufficient to read constant number of letters at the beginning and the end.
- $\tilde{\Theta}(\sqrt{n})$ if a Grover's search is the best way to recognize the language.
- $\Theta(n)$ if recognizing the language is the same as counting modulo some value that can be computed with $O(n)$.

Further more, it is proved that being in the second classes is equivalent to being a star free language. Andris ambainis' team decided to work on $\text{DYCK}_{k,n}$ as this language is a beautiful example of star free languages. In [4, 2020], the team focus on the power of the logarithm in $\tilde{\Theta}(\sqrt{n})$. They first proved by reduction that the quantum query complexity of $\text{DYCK}_{k,n}$ called $Q(\text{DYCK}_{k,n})$ is in $\Omega\left(c^k\sqrt{n}\right)$ where c is a constant greater than 1 and after gave an algorithm for $\text{DYCK}_{k,n}$ with a quantum query complexity of $O\left(\sqrt{n}(\log(n))^{0.5k}\right)$. Since then, no better lower bound or algorithm have been found.

1.5 Goals of the internship

The problem on the quantum query complexity of $\text{DYCK}_{k,n}$ is still open, my internship has for goal to reduce the gap between the lower bound and the best known algorithm. The researches have been organized on two main axes:

- **Increasing the lower bound.** To do this, It is necessary to understand the bibliography on the adversary method in order to try to find a new adversary with better property.
- **Lowering the upper bound.** It is sufficient to find new algorithms with a quantum query complexity better than the previous best known algorithm's one.

Finally, the overall goal would be to made the two bounds match in order to get the exact quantum query complexity of $\text{DYCK}_{k,n}$. Finally, the problem could also be reformulated for multi-brackets word as done by Kamil Khadiev in [7].

1.6 Results

TODO

2 Preliminaries

In order to understand from where comes the $\tilde{\Theta}(\sqrt{n})$ quantum query complexity, it is mandatory to start by understanding the Trichotomy theorem [1] and its dependency on Regular languages and star free languages. After that, to understand how to find a lower bounds, it is necessary to explore the different adversary methods [10].

2.1 Quantum query for regular languages.

In the article [1], Aaronson, Grier and Schaeffer presented a really interesting algebraic characterization of regular languages based on the recognition by monoids and the syntactic congruence. This totally new to me definition give me some hard time in order to understand it correctly.

2.1.1 Regular languages

Usually, the set of regular languages $\mathcal{R}$ on the alphabet Σ is defined as the smallest fixed point of the function F (the function that computes concatenations, unions, and Kleene stars) including $\{\varnothing\} \cup \{\varepsilon\} \cup (\cup_{l\in\Sigma}\{l\})$ with F equal

$$\begin{aligned} F(X) =& \{AB \ \forall (A,B) \in X^2\} \\ & \cup \{A \cup B \ \forall (A,B) \in X^2\} \\ & \cup \{A \cap B \ \forall (A,B) \in X^2\} \ \# \texttt{ optional} \\ & \cup \{A^* \ \forall A \in X\}. \end{aligned}$$

But here, the more convenient way to characterize regular language is with their algebraic characterization. Indeed, a regular language is always a pre image of a subset of a monoid under monoid homomorphism. A monoid is a 3-tuple of a set M, an internal associative binary operation and finally the identity element associated to the operation. A monoid homomorphism is a map from a monoid to another that preserve the operation and the identity. Usually, to get the monoid representation of a regular language, the first step is to compute the syntactical monoid. The syntactical monoid is obtained by dividing Σ^* by the following equivalence relation called syntactic congruence

$$x \sim_L y \Leftrightarrow \forall (u,v) \in \left(\Sigma^*\right)^2, (uxv \in L \Leftrightarrow uyv \in L).$$

This equivalence relation is a congruence relation as the equivalence class can be multiplied (i.e. if $x \sim_L y$ and $u \sim_L v$ then $xu \sim_L yv$). Now, it is necessary to get the monoid homomorphism. For that it is sufficient to take the homomorphism that map an element to its congruence class.

Inside a congruence class, none or every element of the class is in L. Indeed, if x is in L and $x \sim_L y$ then for all (u, v) in $\left(\Sigma^*\right)^2$, $(uxv$ in $L \Leftrightarrow uyv$ in $L)$ thus x in L implies y in L and finally as x is in L then y is also in L. Now, it is sufficient to prove that the syntactical monoid is of finite size. ma be put the proof if time else say folklore.

Finally, every regular language can be recognized by finite monoid.

2.1.2 Star free languages

The set of star free languages is a really well studied subset of the regular languages. Its definition differs a little from regular languages' one as the Kleene star is replace by the complement operation (note $\overline{L}$). So star free languages are defined as the smallest fixed point of the function F' (the function that computes concatenations, unions and complements) and such that it includes $\{\varnothing\} \cup \{\varepsilon\} \cup (\cup_{l \in \Sigma} \{l\})$. This restriction does not imply that every star free language is finite. Indeed, Σ^* can be written $\overline{\varnothing}$. For example the language on $\Sigma = \{1, 2, 3\}$ described with the regular expression $\Sigma^* 20^* 2\Sigma^*$ can be written as following in the star free way $\overline{\varnothing}2\overline{\overline{\varnothing}\Sigma \setminus \{0\}}\overline{\varnothing}2\overline{\varnothing}$.

As for regular languages, it exists an algebraic characterization for star free languages. Let M be a monoid, M is said to be aperiodic if for every x in m it exits a positive integer n such that $x^n = x^n + 1$. A theorem proved by Schützenberger [8] state that a language is recognized by a finite aperiodic monoid if and only if if is star free.

Good examples of star free languages are the Dyck word languages with bounded heights. It is first easy to have a finite automata that recognize Dyck word of height k by putting one state for each integer from 0 up to k. However, the belonging to the star free regular languages is more delicate to prove. It has been done done by Italian researcher in [11, 1978].

2.1.3 Trichotomy theorem

Theorem 2.1 (Aaronson, Grier and Schaeffer [1])**.** *Every regular language has quantum query complexity* $0, \Theta(1), \tilde{\Theta}(\sqrt{n})$, *or* $\Theta(n)$ *according to the smallest class in the following hierarchy that contains the language.*

- *Degenerate: One of the four languages* $\varnothing, \varepsilon, \Sigma^*$, *or* Σ^+.
- *Trivial: The set of languages which have trivial* [1] *regular expressions.*
- *Star free: The set of languages which have star-free regular expressions.*
- *Regular: The set of languages which have regular expressions.*

This theorem is really important as it give a good idea on the quantum query complexity of many language recognitions. Moreover, the classes are now clearly defined so it is now easier to know where is a problem compared to the classes in the introduction. However, it does not give the exact quantum query complexity because for star free languages the result is given using $\tilde{\Theta}$. The $\tilde{\Theta}(\sqrt{n})$ means that the quantum query complexity of any star free regular language is in $\Theta(\sqrt{n}(\log_2(n))^p)$ for some p a non negative constant. As it is known that $\textsc{Dyck}_k$ languages are star free, it is an interesting problem to find the power of $\log_2(n)$ depending on the value of k.

2.2 The bounds for $\textsc{Dyck}_{k,n}$ problem

The trichotomy theorem state that $\textsc{Dyck}_{k,n}$ language has a quantum query complexity in $\tilde{\Theta}(\sqrt{n})$. The Θ means that the best possible algorithm is both a $\tilde{O}(\sqrt{n})$ and a $\tilde{\Omega}(\sqrt{n})$. So, a common method to find the power of $\log_2(n)$ depending on k is the squeeze theorem. More precisely, the quantum query complexity of $\textsc{Dyck}_{k,n}$ is trapped between a lower and an upper bound for quantum query complexity. So it is possible to deduce the quantum query complexity

[1] A language L is said to be trivial if and only if it exists 2 finite size alphabets L_1 and L_2 such that $L = L_1 \Sigma^* L_2$.

from an increasing sequence of lower bound and a decreasing sequence of upper bound such that their limit is l with l equals to $Q(\textsc{Dyck}_{k,n})$. How to find this sequence ?

- **For the lower bound sequence.** One of the most important tools to compute lower bounds are called **quantum adversary methods** and have been invented by Ambainis [3]. It is a set of tools that allow to compute lower bounds more or less tight. Some of this adversary methods have really useful property such as being compatible with a certain type of composition of problem. This lead to the **reduction methods** that allow to compute a lower bounds with different but easier problem's lower bounds.
- **For the upper bound sequence.** The main method is to **find quantum query algorithms** that are more and more efficient. An other way is also **by reduction** to a more difficult problem with an already known upper bound.
- **For the same limit.** The idea is to do an iterative process where each step improves successively each bound until only one will continue to be improved. Finally, both bounds may end up matching. However, before my internship, both $\textsc{Dyck}_{k,n}$'s lower and upper bounds where stuck to $\Omega\left(c^k\sqrt{n}\right)$ for some constant c greater than 1 and $O\left(\sqrt{n}\left(\log_2(n)\right)^{0.5k}\right)$.

2.2.1 Lower bound on the quantum query complexity of $\textbf{Dyck}_{k,n}$

The Adversary method:

- proof for basic adversary
- presentaton of general adversary
- explain why it is hard to use

The reduction method:

- how does it work
- how does it translate to quantum query complexity
- the result for a lower bound on $\textsc{Dyck}_{k,n}$

2.2.2 Best known algorithm to recognize $\textbf{Dyck}_{k,n}$

How to reject a word from $\textbf{Dyck}_{k,n}$

- recall for the height *height*
- $\pm\ k+1$ strings, minimal ones

explaination why it is hard for $k > 1$

- There is an infinit number of rejecting strings to search for

Andris ambainis team solution

- Give an detailed explaination of the algorithm that are in the annexe
- explain why every part of the algorithm is important.
  - the search on d
  - the log search to have limited error.
  - $\textsc{FindAtLeftMost}_k$
-

3 A better algorithm for $\text{Dyck}_{k,n}$

3.1 A better Complexity Analysis of the original algorithm

In the article [4], Andris Ambainis give us a quantum algorithm to recognize the belonging of a n length bit string in $\text{DYCK}_{k,n}$ using $O(\sqrt{n}(\log_2(n))^{0.5k})$ quantum queries. But the quantum query complexity for $k = 1$ is not as good as a Grover's search which is sufficient. More precisely, for $k = 1$ the algorithm is searching for a minimal ± 2 string in $1x0$ but every minimal ± 2 string is of size 2. So the logarithmic search of the upper bound on the size of the minimal ± 2 string is no more useful and the algorithm can be summarized to applying a Grover search for 2 consecutive 0 or two consecutive 1. This lower the quantum query complexity of the initial case of the function to $O(\sqrt{n})$ instead of $O(\sqrt{n\log_2(n)})$. This give us this following algorithm for FINDANY_k.

Algorithm 1 FINDANY_k(l,r,s)

Require: $0 \leq l < r$ and $s \subseteq \{1, -1\}$
  if $k > 2$ **then**
    Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
      $v_d \leftarrow \text{FINDFIXEDLENGTH}_k(l, r, d, s)$ is **not** NULL
    return v_d or NULL if none
  else
    Find t in $\{l, l+1, \ldots, r\}$ such that
      $v_t \leftarrow \text{FINDATLEFTMOST}_2(l, r, t, 2, s)$ is **not** NULL
    return v_t of NULL if none

The same improvement can be done on FINDFIXEDPOS_k because if $k = 2$ the logarithmic search is useless. So FINDFIXEDPOS_k can be redefined as in ALGORITHM 2. For $k = 2$, the complexity is lowered from $O(\sqrt{\log_2(l-r)})$ to $O(1)$.

Algorithm 2 $\text{FINDFIXEDPOS}_k(l, r, t, s)$

Require: $0 \leq l < r$, $l \leq t \leq r$ and $s \subseteq \{1, -1\}$
  if $k > 2$ **then**
    Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
      $v_d \leftarrow \text{FINDATLEFTMOST}_k(l, r, t, d, s)$ is **not** NULL
    return v_d or NULL if none
  else $v \leftarrow \text{FINDATLEFTMOST}_k(l, r, t, 2, s)$ is **not** NULL
    return v_d or NULL if none

This small improvements on the initial cases will improve the global quantum query complexity of each subroutine and finally the quantum query complexity for $\text{DYCK}_{k,n}$.

Theorem 3.1. Dyck$_{k,n}$'s algorithm correctness *The new definition of* FindAny *and* FindFixedPos *does not change the behavior the original algorithm as other subroutines (Appendix A) stay unchanged.*

Proof Theorem 3.1. The behavior of the Dyck$_{k,n}$ algorithm with the new subroutines is the same than the older one as FindAny (resp. FindFirst) has the same sub-behavior on every entry with its older definition.

Theorem 3.2. Dyck$_{k,n}$'s Subroutines complexity *The subroutines' quantum query complexity for k are the following.*

1. $Q($Dyck$_{k,n}) = O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$ *for* $k \geq 1$
2. $Q($FindAny$_{k+1}(l,r,s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ *for* $k \geq 1$
3. $Q($FindFixedLength$_{k+1}(l,r,d,s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
4. $Q($FindAtLeftMost$_{k+1}(l,r,t,d,s)) = \begin{cases} O\left(\sqrt{d}(\log_2(d))^{0.5(k-2)}\right) & \text{for } k \geq 2 \\ O(1) & \text{for } k = 1 \end{cases}$
5. $Q($FindFirst$_k(l,r,s,left)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
6. $Q($FindFixedPos$_k(l,r,t,s)) = \begin{cases} O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) & \text{for } k \geq 3 \\ O(1) & \text{for } k = 2 \end{cases}$

Proof Theorem 3.2. The idea is that only the $O(\sqrt{n})$ comes from the initial cases for $k = 1$ and for each of the $k-1$ level of the recursion, the quantum query complexity is increased by a $O\left(\sqrt{\log_2(n)}\right)$ factor. The $O\left(\sqrt{\log_2(n)}\right)$ factor is proven by Andris Ambainis' team in [4] while the $O(\sqrt{n})$ for $k = 1$ comes from the new version of FindAny$_k$ (Algorithm 2). The complete proof for the theorem is given in Appendix B.

Unfortunately, the improvements done on the initial cases of some of the subroutines are not sufficient to get a significant improvement for the quantum query complexity of Dyck$_{k,n}$ algorithm. In order to improve more the query complexity, an other algorithm using a different strategy should be found.

3.2 A new algorithm for Dyck$_{2,n}$

First, we would like to find an algorithm with a quantum query complexity near to match the lower bound, $\exists c > 1$ such that $Q\left(\text{Dyck}_{k,n}\right) = \Omega\left(\sqrt{n}c^k\right)$, describes by Andris Ambainis' team in [4]. So the searched algorithm must have a quantum query complexity of $O(\sqrt{n})$.

For $k = 1$, the query complexity comes only from a call to Grover's search because rejecting is easily by finding a 00 or a 11 substrings inside the entry. For $k = 2$ it no more possible as the substrings that reject are of the form 00(10)*0 or of the form 11(01)*1. It implies that the number of calls to Grover's search in the naive approach is in $O(n)$ so the quantum query complexity finally becomes $O(n\sqrt{n})$. In order to keep it in $O(\sqrt{n})$, the algorithm must do a constant number of calls to Grover's search.

For that, we define a new alphabet that can express every even length binary strings and that have convenient property for a Grover's search. Let $\mathcal{A} = \{a, b, c, d\}$ the alphabet where a corresponds to 00, b to 11, c to 01, and d to 10. So every string of size 2 has its letter in $\mathcal{A}$ thus every even length bit string is expressed in $\mathcal{A}^*$. This alphabet allow us to prove the following theorem.

Theorem 3.3. ***Substrings rejection for Dyck word of height at most 2.*** *A word on the alphabet $\mathcal{A}$ embodies a Dyck word of height at most 2 if and only if it does not contain $aa, ac, bb, bd, cb, cd, da$, or dc as substrings.*

Proof Theorem 3.3. First, this alphabet $\mathcal{A}$ is important because each letter has a height variation in $\{-2, 0, 2\}$. Indeed, a has a 2 height variation, b a -2, c a 0, and d a 0. This means that after each letter in a word, the current height will be even. Moreover, for a valid Dyck word of height at most 2, after every letter the height will be 0 or 2 which are respectively the lower and upper bound for the height. It means that no letter can cross a border after its first bit.

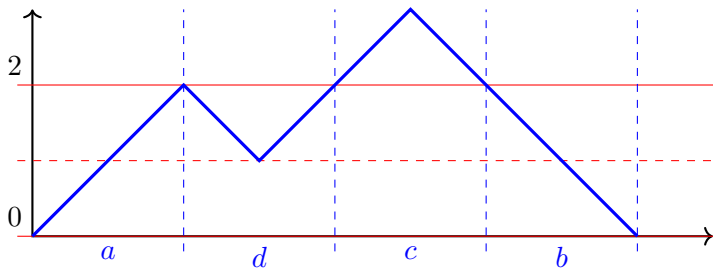


Figure 4: Illustration of the letters of $\mathcal{A}$ using Dyck's representation.

This property is important as it implies that every ± 3 strings uses at least two letters. So by checking if a pair of letter as a substring of a word make it not a Dyck word, $\mathcal{A}^2$ can be split into two sets described in Table 1.

Table 1: Partition of $\mathcal{A}$ into $\mathcal{X}, \mathcal{V}$.

$\mathcal{X}$	aa ac bb bd cb cd da dc
$\mathcal{V}$	ab ad ba bc ca cc db dd

- The set $\mathcal{X}$. First, every couple of letter which contains a ± 3 strings is in $\mathcal{X}$. This first condition explains the belonging of $aa, ac, dc, da, cb, bb, bd$, and cd. Next, cd and dc belong to $\mathcal{X}$ because of the following property: For any valid Dyck word of height at most 2, the current height is bounded between 0 and 2, moreover after each letter the current height is even so both couple cd and dc start and finish on the same bound. Futhermore, cd and dc are going above and below the height at which they start so both are going outside off the bounds, thus a word which contains cd or dc can not be a Dyck Word of height a most 2. The Figure 5 shows each couple of $\mathcal{X}$.

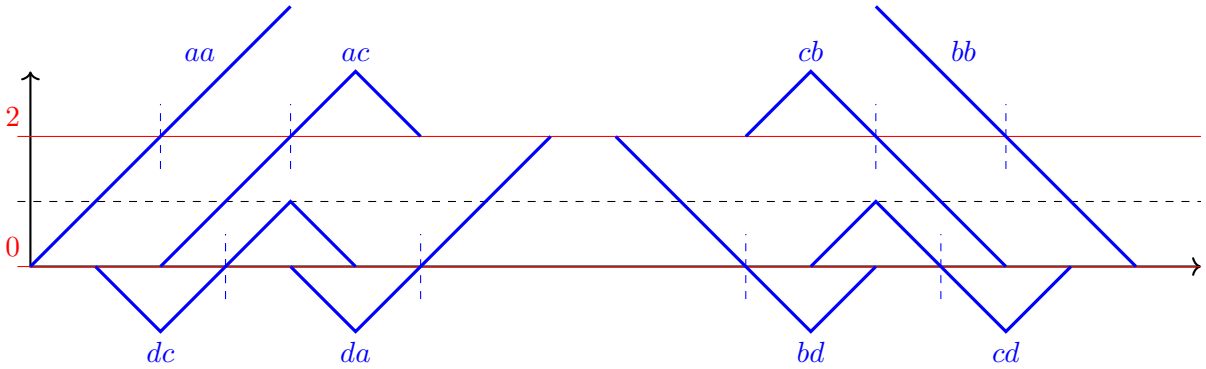


Figure 5: Every 2 letters configuration that implies the word, whom the configuration is a substring, is not a Dyck word of height at most 2.

- The set $\mathcal{V}$. The couples of $\mathcal{A}$ do not imply that the word is not a Dyck word of height at most 2 because each couple can fit inside the height bounds. The Figure 6 shows that every couple not in $\mathcal{X}$ (ie. $ab, ad, ba, bc, ca, cc, db, dd$) fit between height 0 and 2.

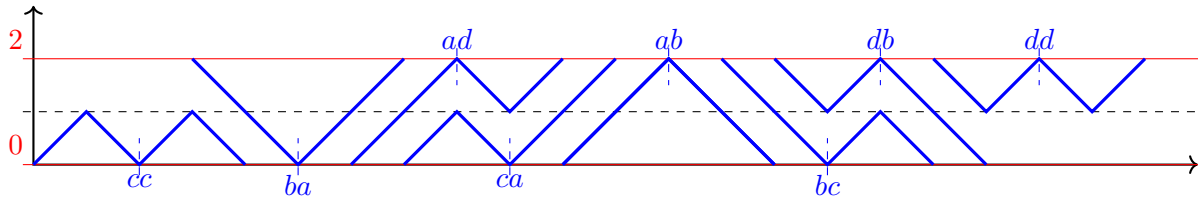


Figure 6: Every 2 letters configuration that can be found in a valid Dyck word of height at most 2.

So a word, whose letter representation has a substring in $\mathcal{X}$, cannot be a Dyck word of height two. But does every non Dyck word of height at most 2 have a substring in $\mathcal{X}$?

A word is not a dyck word of height at most 2 if it include a ± 3 strings. But how are represented ± 3 strings using the letters? There are 8 different cases which are 2 by 2 symmetrical so Figure 7 and Figure 8 show only the cases for $+3$ strings. In Figure 7, every $+3$ string of size 3 is include in aa or ac so it is sufficient to search for this two couple. In Figure 8 every $+3$ strings of length greater than 3 are composed of 2 minimal $+2$ strings. This implies that one must be a a while the other must be da of dc. Because da or dc are rejecting substrings, it is sufficient to search for them.

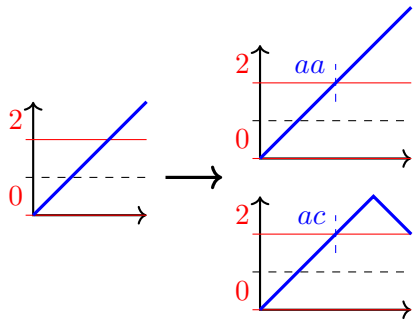


Figure 7: Configuration for a $+3$ strings of size 3.

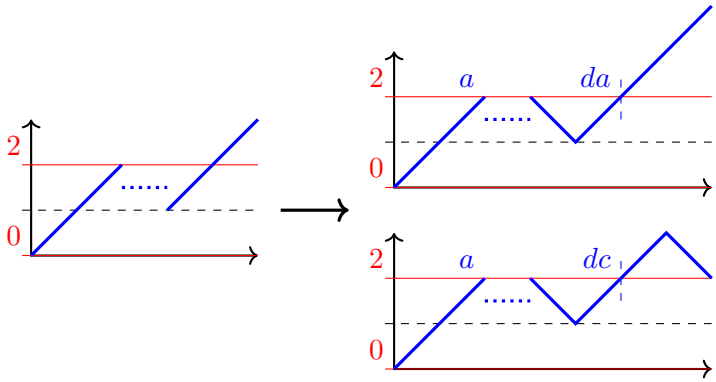


Figure 8: Configurations for a $+3$ string of size greater than 3.

Finally, a word on the alphabet $\mathcal{A}$ embodies a Dyck word of height at most 2 if and only if it does not contain $aa, ac, bb, bd, cb, cd, da, dc$ as substrings. The following ALGORITHM 3 for $\text{DYCK}_{2,n}$ comes from the direct application of the theorem.

Theorem 3.4. ***The quantum query complexity of*** $\textbf{DyckFast}_{2,n}$**.** *The* $\text{DYCKFAST}_{2,n}$ *algorithm has a quantum query complexity of* $O\left(\sqrt{n}\right)$*.*

Proof Theorem 3.4. The algorithm is doing at most 8 Grover's search on the modified input string $11x00$. So the total quantum query complexity is the folling.

$$Q(\text{DYCKFAST}_{2,n}) = 8 \times O\left(\sqrt{n+4}\right) = O\left(\sqrt{n}\right)$$

Algorithm 3 DyckFast$_{2,n}$

Require: $n \geq 0$, x such that $|x| = 2n$
  $x \leftarrow 11x00$
  $\texttt{t} \leftarrow$ Null
  for $\texttt{reject\_symbol} \in \{aa, ac, bb, bd, cb, cd, da, dc\}$ **do**
    if $t ==$ Null **then**
      Find $\texttt{t}$ in $[\![0, n]\!]$ such that
      $x[2\texttt{t}, \ldots, 2\texttt{t} + 3] = \texttt{reject\_symbol}$
  return $\texttt{t} ==$ Null

3.3 A new algorithm for k=3

3.4 A try for a new algorithm for any k.

4 Conclusion

Conclusion here.

References

[1] Scott Aaronson, Daniel Grier, and Luke Schaeffer. A quantum query complexity trichotomy for regular languages, 2018.

[2] Andris Ambainis. Understanding quantum algorithms via query complexity.

[3] Andris Ambainis. Quantum lower bounds by quantum arguments, 2000.

[4] Andris Ambainis, Kaspars Balodis, Jānis Iraids, Kamil Khadiev, Vladislavs Kļevickis, Krišjānis Prūsis, Yixin Shen, Juris Smotrovs, and Jevgēnijs Vihrovs. Quantum lower and upper bounds for 2d-grid and dyck language. *Leibniz International Proceedings in Informatics*, 170, 2020.

[5] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, (22(5):653-591), 1980.

[6] Richard Feymann. Quantum mechanical computers. *Foundamentale Physics*, (16, 507-531), 1986.

[7] Kamil Khadiev and Dmitry Kravchenko. Quantum algorithm for dyck language with multiple types of brackets. In Irina Kostitsyna and Pekka Orponen, editors, *Unconventional Computation and Natural Computation - 19th International Conference, UCNC 2021, Espoo, Finland, October 18-22, 2021, Proceedings*, volume 12984 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2021.

[8] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Inf. Control.*, 8:190–194, 1965.

[9] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[10] Robert Spalek and Mario Szegedy. All quantum adversary methods are equivalent. 2004.

[11] Crespi-Reghizzi Stefano, Guida Giovanni, and Mandrioli Dino. Noncounting context-free languages. 1978.

List of Figures

1 A quantum circuit computing the uniform random on $\{0,1\}^n$. 3
2 Structure of a quantum query algorithm. 3
3 **On the left**, a valid Dyck word of height 3. **On the right**, an invalid Dyck word of height 3. 4
4 Illustration of the letters of $\mathcal{A}$ using Dyck's representation. 10
5 Every 2 letters configuration that implies the word, whom the configuration is a substring, is not a Dyck word of height at most 2. 10
6 Every 2 letters configuration that can be found in a valid Dyck word of height at most 2. 11
7 Configuration for a +3 strings of size 3. 11
8 Configurations for a +3 string of size greater than 3. 11

List of Algorithms

1 $\textsc{FindAny}_k$(l,r,s) 8
2 $\textsc{FindFixedPos}_k(l,r,t,s)$ 8
3 $\textsc{DyckFast}_{2,n}$ 12
4 $\textsc{Dyck}_{k,n}$ 14
5 $\textsc{FindAny}_k(l,r,s)$ 14
6 $\textsc{FindFixedLength}_k(l,r,d,s)$ 14
7 $\textsc{FindAtLeftMost}_k(l,r,d,t,s)$ 15
8 $\textsc{FindFirst}_k(l,r,s,left)$ 15
9 $\textsc{FindFixedPos}_k(l,r,t,s)$ 15

5 Appendix

The frame of the intership

A The algorithm for Dyck$_{k,n}$

All the subroutines' pseudo code can be found from Algorithm 4 to Algorithm 9.

Algorithm 4 Dyck$_{k,n}$

Require: $n \geq 0$ and $k \geq 1$
Ensure: $|x| = n$
  $x \leftarrow 1^k x 0^k$
  $v \leftarrow \text{FindAny}_{k+1}(0, n + 2 * k - 1, \{1, -1\})$
  return v = Null

Algorithm 5 FindAny$_k(l, r, s)$

Require: $0 \leq l < r$ and $s \subseteq \{1, -1\}$
  Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
    $v_d \leftarrow \text{FindFixedLength}_k(l, r, d, s)$ is **not** Null
  return v_d or Null if none

Algorithm 6 FindFixedLength$_k(l, r, d, s)$

Require: $0 \leq l < r$, $1 \leq d \leq r - l$ and $s \subseteq \{1, -1\}$
  Find t in $\{l, l + 1, \ldots, r\}$ such that
    $v_t \leftarrow \text{FindAtLeftMost}_k(l, r, t, d, s)$ is **not** Null
  return v_t of Nullif none

B The proof of the quantum query complexity for Dyck$_{k,n}$ algorithm's subroutines

Theorem B.1. Dyck$_{k,n}$*'s Subroutines complexity* *The subroutines' quantum query complexity for k are the following.*

1. $Q(\text{Dyck}_{k,n}) = O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$ *for* $k \geq 1$
2. $Q(\text{FindAny}_{k+1}(l, r, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ *for* $k \geq 1$
3. $Q(\text{FindFixedLength}_{k+1}(l, r, d, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
4. $Q(\text{FindAtLeftMost}_{k+1}(l, r, t, d, s)) = \begin{cases} O\left(\sqrt{d}(\log_2(d))^{0.5(k-2)}\right) & \text{for } k \geq 2 \\ O(1) & \text{for } k = 1 \end{cases}$
5. $Q(\text{FindFirst}_k(l, r, s, left)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
6. $Q(\text{FindFixedPos}_k(l, r, t, s)) = \begin{cases} O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) & \text{for } k \geq 3 \\ O(1) & \text{for } k = 2 \end{cases}$

Proof Theorem B.1. The proof is done by induction on the height k of the Dyck word.

Algorithm 7 FindAtLeftMost$_k(l, r, d, t, s)$

Require: $0 \le l < r$, $l \le r \le r$, $1 \le d \le r - l$ and $s \subseteq \{1, -1\}$
  $v = (i_1, j_1, \sigma_1) \leftarrow$ FindAtLeftMost$_{k-1}(l, r, t, d-1, \{1, -1\})$
  if $v \neq$ Null **then**
    $v' = (i_2, j_2, \sigma_2) \leftarrow$ FindAtRightMost$_{k-1}(l, r, i_1 - 1, d-1, \{1, -1\})$
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow$ FindFirst$_{k-1}(\max(l, j_1 - d + 1), i_1 - 1, \{1, -1\}, left)$
    if $v' \neq$ Null and $\sigma_2 \neq \sigma_1$ **then** $v' \leftarrow$ Null
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow$ FindAtLeftMost$_{k-1}(l, r, j_1 + 1, d-1, \{1, -1\})$
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow$ FindFirst$_{k-1}(j_1 + 1, \max(r, i_1 + d - 1), \{1, -1\}, right)$
    if $v' =$ Null **then return** Null
  else
    $v = (i_1, j_1, \sigma_1) \leftarrow$ FindFirst$_{k-1}(t, min(t + d - 1, r), \{1, -1\}, right)$
    if v $=$ Null **then return** Null
    $v' = (i_2, j_2, \sigma_2) \leftarrow$ FindFirst$_{k-1}(max(t - d + 1, l), t, \{1, -1\}, left)$
    if $v' =$ Null **then return** Null
  if $\sigma_1 = \sigma_2$ and $\sigma_1 \in s$ and $\max(j_1, j_2) - \min(i_1, i_2) + 1 \le d$ **then**
    return $(\min(i_1, i_2), \max(j_1, j_2), \sigma_1)$
  else return Null

Algorithm 8 FindFirst$_k(l, r, s, left)$

Require: $0 \le l < r$ and $s \subseteq \{1, -1\}$
  $lBorder \leftarrow l, rBorder \leftarrow r, d \leftarrow 1$
  while $lBorder + 1 < rBorder$ **do**
    $mid \leftarrow \lfloor (lBorder + rBorder)/2 \rfloor$
    $v_l \leftarrow$ FindAny$_k(lBorder, mid, s)$
    if $v_l \neq$ Null **then** $rBorder \leftarrow mid$
    else
      $v_{mid} \leftarrow$ FindFixedPos$_k(lBorder, rBorder, mid, s, left)$
      if $v_{mid} \neq$ Null **then return** v_{mid}
      else $lBorder \leftarrow mid + 1$
    $d \leftarrow d + 1$
  return Null

Algorithm 9 FindFixedPos$_k(l, r, t, s)$

Require: $0 \le l < r$, $l \le t \le r$ and $s \subseteq \{1, -1\}$
  Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
    $v_d \leftarrow$ FindAtLeftMost$_k(l, r, t, d, s)$ is **not** Null
  return v_d or Null if none

Initialization: For $k = 1$ and $k = 2$ we have the following initialization.

- For $k = 1$, only $\textsc{FindAtLeftMost}_2$, $\textsc{FindAny}_2$,and $\textsc{Dyck}_{1,n}$ are defined. The $O(1)$ quantum query complexity of $\textsc{FindAtLeftMost}_2$ comes directly from the definition of its initial case, as the $O(\sqrt{r-l})$ quantum query complexity of $\textsc{FindAny}_2$. Then the $O(\sqrt{n})$ quantum query complexity of $\textsc{Dyck}_{1,n}$ comes from the call to $\textsc{FindAny}_2$.
- For $k = 2$, the inductive part of the algorithm start and every subroutines is defined. The $O(1)$ quantum query complexity of $\textsc{FindFixedPos}_2$ comes from the call to $\textsc{FindAtLeftMost}_2$. The $O\left(\sqrt{r-l}\right)$ quantum query complexity of $\textsc{FindFirst}_2$ comes from the dichotomize search using $\textsc{FindAny}_2$ and $\textsc{FindFixedPos}_2$ because $\sum_{u=1}^{log_2(r-l)} 2u\left(O\left(\sqrt{\frac{r-l}{2^{u-1}}}\right)+O(1)\right) = O(\sqrt{r-l})$ (Detailed in the induction). The $O(\sqrt{d})$ quantum query complexity of $\textsc{FindAtLeftMost}_3$ comes from the constant amount of calls to $\textsc{FindFirst}_2$ and $\textsc{FindAtLeftMost}_2$ with entry of size d. The $O(\sqrt{r-l})$ quantum query complexity of $\textsc{FindFixedLength}_3$ comes from the $O\left(\sqrt{\frac{r-l}{d}}\right)$ calls to $\textsc{FindAtLeftMost}_3$. The $O\left(\sqrt{(r-l)\log_2(r-l)}\right)$ quantum query complexity of $\textsc{FindAny}_3$ comes from the $O\left(\sqrt{\log_2(r-l)}\right)$ calls to $\textsc{FindFixedLength}_3$. Finally, the $O\left(\sqrt{(r-l)\log_2(r-l)}\right)$ quantum query complexity of $\textsc{Dyck}_2$ comes from the call to $\textsc{FindAny}_3$.

Induction: Let suppose it exists k such that Theorem B.1 is true for k. Let prove that it is true for $k+1$.

First, the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ quantum query complexity of $\textsc{FindFixedPos}_{k+1}$ comes from the $O\left(\sqrt{\log(r-l)}\right)$ calls to $\textsc{FindAtLeftMost}_{k+1}$.

$$\begin{aligned}
Q(\textsc{FindFixedPos}_{k+1}(l,r,t,s)) &= O(\sqrt{\log(r-l)}) \times O\left(Q(\textsc{FindAtLeftMost}_{k+1}(l,r,t,d,s))\right) \\
&\overset{IH}{=} O\left(\sqrt{\log(r-l)} \times \sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) \\
&= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)
\end{aligned}$$

Thus the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ quantum query complexity of $\textsc{FindFirst}_{k+1}$ comes from the dichotomize search using calls to $\textsc{FindAny}_{k+1}$ and $\textsc{FindFirst}_{k+1}$.

---

[a] $\sum_{u=1}^{+\infty}\left(\frac{\mathrm{d}}{\mathrm{d}x}(x^u)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{u=1}^{+\infty}x^u\right)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{1}{(1-x)^2}\right)\left(\frac{1}{\sqrt{2}}\right) \leq \frac{1}{(1-\frac{1}{\sqrt{2}})^2} \leq \frac{\sqrt{2}^2}{(\sqrt{2}-1)^2}$

$$
\begin{aligned}
Q(\textsc{FindFirst}_{k+1}(l,r,t,d,s)) = \; & \textstyle\sum_{u=1}^{\log_2(r-l)} 2u \times O\left(Q(\textsc{FindAny}_{k+1}(0,\tfrac{r-l}{2^{u-1}},s))\right) \\
& + \textstyle\sum_{u=1}^{\log_2(r-l)} 2u \times O\left(Q(\textsc{FindFixedPos}_{k+1}(0,\tfrac{r-l}{2^{u-1}},\_,s,left))\right) \\
\overset{IH}{=} & \; O\left(\sum_{u=1}^{\log_2(r-l)} 2u \times \sqrt{\frac{r-l}{2^{u-1}}}(\log_2(\frac{r-l}{2^{u-1}}))^{0.5(k-1)}\right) \\
= & \; O\left(\sum_{u=1}^{\log_2(r-l)} 2u \times \sqrt{\frac{r-l}{2^{u-1}}}(\log_2(r-l))^{0.5(k-1)}\right) \\
= & \; O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)} \sum_{u=1}^{\log_2(r-l)} u \times (\frac{1}{\sqrt{2}})^{u-1}\right) \\
=^{a} & \; O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)} \frac{\sqrt{2}^2}{(\sqrt{2}-1)^2}\right) \\
= & \; O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)
\end{aligned}
$$

Next, the $O\left(\sqrt{d}(\log_2(d))^{0.5(k-1)}\right)$ quantum query complexity comes of $\textsc{FindAtLeftMost}_{k+2}$from the constant amount of calls to $\textsc{FindAtLeftMost}_{k+1}$, $\textsc{FindAtRightMost}_{k+1}$,and $\textsc{FindFirst}_{k+1}$.

$$
\begin{aligned}
Q(\textsc{FindAtLeftMost}_{k+2}(l,r,t,d,s)) = \; & 3 \times O\left(Q(\textsc{FindAtLeftMost}_{k+1}(l,r,t,d,\{1,-1\}))\right) \\
& +4 \times O\left(Q(\textsc{FindFirst}_{k+1}(l,r,\{1,-1\},left))\right) \\
\overset{IH}{=} & \; O\left(\sqrt{d}(\log_2(d))^{0.5(k-1)}\right)
\end{aligned}
$$

After that, the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ quantum query complexity of $\textsc{FindFixedLength}_{k+2}$ comes from the $O\left(\sqrt{\frac{r-l}{d}}\right)$ calls to $\textsc{FindAtLeftMost}_{k+2}$.

$$
\begin{aligned}
Q(\textsc{FindFixedLength}_{k+2}(l,r,d,s)) &= O\left(\sqrt{\frac{r-l}{d}}\right) \times O\left(Q(\textsc{FindAtLeftMost}_{k+2}(l,r,t,d,s))\right) \\
&= O\left(\sqrt{\frac{r-l}{d}} \times \sqrt{d}(\log_2(d))^{0.5(k-1)}\right) \\
&= O\left(\sqrt{r-l}(\log_2(d))^{0.5(k-1)}\right) \\
&= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)
\end{aligned}
$$

Hence the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5k}\right)$ quantum query complexity of $\textsc{FindAny}_{k+2}$ comes from the the $O\left(\sqrt{\log_2(r-l)}\right)$ calls to $\textsc{FindFixedLength}_{k+2}$.

$$
\begin{aligned}
Q(\textsc{FindAny}_{k+2}(l,r,s)) &= O\left(\sqrt{\log(r-l)}\right) \times O\left(Q(\textsc{FindFixedLength}_{k+2}(l,r,d,s))\right) \\
&= O\left(\sqrt{\log(r-l)} \times \sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right) \\
&= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5k}\right)
\end{aligned}
$$

Finally, the $O\left(\sqrt{n}(\log_2(n))^{0.5k}\right)$ quantum query complexity of $\textsc{Dyck}_{k+1,n}$ comes from the call to $\textsc{FindAny}_{k+2}$.

$$
\begin{aligned}
Q(\textsc{Dyck}_{k+1,n}) &= O\left(Q(\textsc{FindAny}_{k+2}(0, n + 2k + 1, s))\right) \\
&= O\left(Q(\textsc{FindAny}_{k+2}(0, n, s))\right) \\
&= O\left(\sqrt{n}(\log_2(n))^{0.5k}\right)
\end{aligned}
$$

Conclusion: By the induction principle we get that the Theorem B.1 is true for $k \in \mathbb{N}^*$