Quantum Algorithms for the Most Frequently String Search, Intersection of Two String Sequences and Sorting of Strings Problems

Kamil Khadiev[1,2] and Artem Ilikaev[2]

[1] Smart Quantum Technologies Ltd., Kazan, Russia
[2] Kazan Federal University, Kazan, Russia
kamil.hadiev@kpfu.ru, artemka.tema1998@gmail.com

Abstract. We study algorithms for solving three problems on strings. The first one is the Most Frequently String Search Problem. The problem is the following. Assume that we have a sequence of n strings of length k. The problem is finding the string that occurs in the sequence most often. We propose a quantum algorithm that has a query complexity $\tilde{O}(n\sqrt{k})$. This algorithm shows speed-up comparing with the deterministic algorithm that requires $\Omega(nk)$ queries.
The second one is searching intersection of two sequences of strings. All strings have the same length k. The size of the first set is n and the size of the second set is m. We propose a quantum algorithm that has a query complexity $\tilde{O}((n+m)\sqrt{k})$. This algorithm shows speed-up comparing with the deterministic algorithm that requires $\Omega((n+m)k)$ queries.
The third problem is sorting of n strings of length k. On the one hand, it is known that quantum algorithms cannot sort objects asymptotically faster than classical ones. On the other hand, we focus on sorting strings that are not arbitrary objects. We propose a quantum algorithm that has a query complexity $O(n(\log n)^2\sqrt{k})$. This algorithm shows speed-up comparing with the deterministic algorithm (radix sort) that requires $\Omega((n+d)k)$ queries, where d is a size of the alphabet.
Keywords: quantum computation, quantum models, quantum algorithm, query model, string search, sorting

1 Introduction

Quantum computing [26, 5] is one of the hot topics in computer science of last decades. There are many problems where quantum algorithms outperform the best known classical algorithms [12, 17, 19, 18].

One of these problems are problems for strings. Researchers show the power of quantum algorithms for such problems in [25, 6, 29].

In this paper, we consider three problems:

- ⋆ the Most Frequently String Search problem;
- ⋆ Strings sorting problem;
- ⋆ Intersection of Two String Sequences problem.

Our algorithms use some quantum algorithms as a subroutine, and the rest part is classical. We investigate the problems in terms of query complexity. The query model is one of the most popular in the case of quantum algorithms. Such algorithms can do a query to a black box that has access to the sequence of strings. As a running time of an algorithm, we mean a number of queries to the black box.

The first problem is the following. We have n strings of length k. We can assume that symbols of strings are letters from any finite alphabet, for example, binary, Latin alphabet or Unicode. The problem is finding the string that occurs in the sequence most often. The best known deterministic algorithms require $\Omega(nk)$ queries because an algorithm should at least test all symbols of all strings. The deterministic solution can use the Trie (prefix tree) [11, 7, 9, 21] that allows to achieve the required complexity.

We propose a quantum algorithm that uses a self-balancing binary search tree for storing strings and a quantum algorithm for comparing strings. As a self-balancing binary search tree we can use the AVL tree [2, 10] or the Red-Black tree [14, 10]. As a string comparing algorithm, we propose an algorithm that is based on the first one search problem algorithm from [22–24]. This algorithm is a modification of Grover's search algorithm [13, 8]. Our algorithm for the most frequently string search problem has query complexity $O(n(\log n)^2 \cdot \sqrt{k}) = \tilde{O}(n\sqrt{k})$, where $\tilde{O}$ does not consider a log factors. If $\log_2 n = o(k^{0.25})$, then our algorithm is better than deterministic one. Note, that this setup makes sense in practical cases.

The second problem is String sorting. Assume that we have n strings of length k. It is known [15, 16] that no quantum algorithm can sort arbitrary comparable objects faster than $O(n \log n)$. At the same time, several researchers tried to improve the hidden constant [28, 27]. Other researchers investigated space bounded case [20]. We focus on sorting strings. In a classical case, we can use an algorithm that is better than arbitrary comparable objects sorting algorithms. It is radix sort that has $O((n+d)k)$ query complexity [10], where d is a size of the alphabet. Our quantum algorithm for the string sorting problem has query complexity $O(n(\log n)^2 \cdot \sqrt{k}) = \tilde{O}(n\sqrt{k})$. It is based on standard sorting algorithms like Merge sort [10] or Heapsort [30, 10] and the quantum algorithm for comparing strings.

The third problem is the Intersection of Two String Sequences problem. Assume that we have two sequences of strings of length k. The size of the first set is n and the size of the second one is m. The first sequence is given and the second one is given in online fashion, one by one. After each requested string from the second sequence, we want to check weather this string belongs to the first sequence. We propose two quantum algorithms for the problem. Both algorithms has query complexity $O((n+m) \cdot \log n \cdot \log(n+m)\sqrt{k}) = \tilde{O}(n\sqrt{k})$. The first algorithm uses a self-balancing binary search tree like the solution of the first problem. The second algorithm uses a quantum algorithm for sorting strings and has better big-O hidden constant. At the same time, the best known deterministic algorithm requires $O((n+m)k)$ queries.

The structure of the paper is the following. We present the quantum subroutine that compares two strings in Section 2. Then we discussed three problems: the Most Frequently String Search problem in Section 3, Strings Sorting problem in Section 4 and Intersection of Two String Sequences problem in Section 5.

2 The Quantum Algorithm for Two Strings Comparing

Firstly, we discuss a quantum subroutine that compares two strings of length k. Assume that this subroutine is COMPARE_STRINGS(s, t, k) and it compares s and t in lexicographical order. It returns:

- -1 if $s < t$;
- 0 if $s = t$;
- 1 if $s > t$;

As a base for our algorithm, we will use the algorithm of finding the minimal argument with 1-result of a Boolean-value function. Formally, we have:

Lemma 1. *[22–24] Suppose, we have a function $f : \{1, \ldots, N\} \to \{0, 1\}$ for some integer N. There is a quantum algorithm for finding $j_0 = \min\{j \in \{1, \ldots, N\} : f(j) = 1\}$. The algorithm finds j_0 with expected query complexity $\sqrt{j_0}$ and error probability that is at most $\frac{1}{2}$.*

Let us choose the function $f(j) = (s_j \neq t_j)$. So, we search j_0 that is the index of the first unequal symbol of the strings. Then, we can claim that s precedes t in lexicographical order iff s_{j_0} precedes t_{j_0} in alphabet Σ. If there are no unequal symbols, then the strings are equal.

We use the standard technique of boosting success probability. So, we repeat the algorithm $3 \log_2 n$ times and return the minimal answer, where n is a number of strings in the sequence s. In that case, the error probability is $O\left(\frac{1}{2^{3 \log n}}\right) = \left(\frac{1}{n^3}\right)$.

Let us present the algorithm. We use THE_FIRST_ONE_SEARCH(f, k) as a subroutine from Lemma 1, where $f(j) = (s_j \neq t_j)$. Assume that this subroutine returns $k + 1$ if it does not find any solution.

Let us discuss the property of the algorithm:

Lemma 2. *Algorithm 1 compares two strings of length k in lexicographical order with query complexity $O(\sqrt{k} \log n)$ and error probability $O\left(\frac{1}{n^3}\right)$.*

3 The Most Frequently String Search Problem

Let us formally present the problem.

Problem. For some positive integers n and k, we have the sequence of strings $s = (s^1, \ldots, s^n)$. Each $s^i = (s^i_1, \ldots, s^i_k) \in \Sigma^k$ for some finite size alphabet Σ. Let $\#(s) = |\{i \in \{1, \ldots, m\} : s^i = s\}|$ be a number of occurrences of string s. We search $s = argmax_{s^i \in S} \#(s^i)$.

Algorithm 1 COMPARE_STRINGS(s, t, k). The Quantum Algorithm for Two Strings Comparing.

$j_0 \leftarrow$ THE_FIRST_ONE_SEARCH(f, k) ▷ The initial value
for $i \in \{1, \ldots, 3\log_2 n\}$ **do**
  $j_0 \leftarrow \min(j_0,$ THE_FIRST_ONE_SEARCH$(f, k))$
end for
if $j_0 = k + 1$ **then**
  $result \leftarrow 0$ ▷ The strings are equal.
end if
if $(j_0 \neq k + 1)\&(s_{j_0} < t_{j_0})$ **then**
  $result \leftarrow -1$ ▷ s precedes t.
end if
if $(j_0 \neq k + 1)\&(s_{j_0} > t_{j_0})$ **then**
  $result \leftarrow 1$ ▷ s succeeds t.
end if
return $result$

3.1 The Quantum algorithm

Firstly, we present an idea of the algorithm.

We use the well-known data structure a self-balancing binary search tree. As an implementation of the data structure, we can use the AVL tree [2, 10] or the Red-Black tree [14, 10]. Both data structures allow as to find and add elements in $O(\log N)$ running time, where N is a size of the tree.

The idea of the algorithm is the following. We store pairs (i, c) in vertexes of the tree, where i is an index of a string from s and c is a number of occurrences of the string s^i. We assume that a pair (i, c) is less than a pair (i', c') iff s^i precedes $s^{i'}$ in the lexicographical order. So, we use COMPARE_STRINGS$(s^i, s^{i'}, k)$ subroutine as the compactor of the vertexes. The tree represents a set of unique strings from $(s^1, \ldots, s^n)$ with a number of occurrences.

We consider all strings from s^1 to s^n and check the existence of a string in our tree. If a string exists, then we increase the number of occurrences. If the string does not exist in the tree, then we add it. At the same time, we store $(i_{max}, c_{max}) = argmax_{(i,c)}$ in the tree$c$ and recalculate it in each step.

Let us present the algorithm formally. Let BST be a self-balancing binary search tree such that:

- ⋆ FIND(BST, s^i) finds vertex (i, c) or returns $NULL$ if such vertex does not exist;
- ⋆ ADD(BST, s^i) adds vertex $(i, 0)$ to the tree and returns the vertex as a result;
- ⋆ INIT(BST) initializes an empty tree;

Let us discuss the property of the algorithm.

Theorem 1. *Algorithm 2 finds the most frequently string from $s = (s^1, \ldots, s^n)$ with query complexity $O(n(\log n)^2 \cdot \sqrt{k})$ and error probability $O\left(\frac{1}{n}\right)$.*

Algorithm 2 The Quantum Algorithm for Most Frequently String Problem.

```
INIT(BST)                                          ▷ The initialization of the tree.
c_max ← 1                                          ▷ The maximal number of occurrences.
i_max ← 1                                          ▷ The index of most requently string.
for i ∈ {1, ..., n} do
    v = (i, c) ← FIND(BST, s^i)                    ▷ Searching s^i in the tree.
    if v = NULL then
        v = (i, c) ← ADD(BST, s^i)                 ▷ If there is no s^i, then we add it.
    end if
    c ← c + 1   ▷ Updating the vertex by increasing the number of occurrences.
    if c > c_max then                              ▷ Updating the maximal value.
        c_max ← c
        i_max ← i
    end if
end for
return s^{i_max}
```

Proof. The correctness of the algorithm follows from the description. Let us discuss the query complexity. Each operation $\textsc{Find}(BST, s^i)$ and $\textsc{Add}(BST, s^i)$ requires $O(\log n)$ comparing operations $\textsc{Compare\_strings}(s^i, s^{i'}, k)$. These operations are invoked n times. Therefore we have $O(n \log n)$ comparing operations. Due to Lemma 2, each comparing operation requires $O(\sqrt{k} \log n)$ queries. The total query complexity is $O(n\sqrt{k}(\log n)^2)$.

Let us discuss the error probability. Events of error in the algorithm are independent. So, all events should be correct. Due to Lemma 2, the probability of correctness of one event is $1 - \left(1 - \frac{1}{n^3}\right)$. Hence, the probability of correctness of all $O(n \log n)$ events is at least $1 - \left(1 - \frac{1}{n^3}\right)^{\alpha \cdot n \log n}$ for some constant α.

Note that

$$\lim_{n \to \infty} \frac{1 - \left(1 - \frac{1}{n^3}\right)^{\alpha \cdot n \log n}}{1/n} < 1;$$

Hence, the total error probability is at most $O\left(\frac{1}{n}\right)$.

□

The data structure that we used can be considered as a separated data structure. We call it *"Multi-set of strings with quantum comparator"*. Using this data structure, we can implement

- ⋆ *"Set of strings with quantum comparator"* if always $c = 1$ in pair (i, c) of a vertex;
- ⋆ *"Map with string key and quantum comparator"* if we replace c by any data $r \in \Gamma$ for any set Γ. In that case the data structure implements mapping $\Sigma^k \to \Gamma$.

All of these data structures has $O((\log n)^2\sqrt{k})$ complexity of basic operations (Find, Add, Delete).

3.2 On the Classical Complexity of the Problem

The best known classical algorithm stores string to Trie (prefix tree) [11, 7], [9, 21] and do the similar operations. The running time of such algorithm is $O(nk)$. At the same time, we can show that if the algorithm tested $o(nk)$ variables, then it can return a wrong answer.

Theorem 2. *Any deterministic algorithm for the Most Frequently String Search problem has $\Omega(nk)$ query complexity.*

Proof. Suppose, we have a deterministic algorithm A for the Most Frequently String Search problem that uses $o(nk)$ queries.

Let us consider an adversary that suggest an input. The adversary wants to construct an input such that the algorithm A obtains a wrong answer.

Without loss of generality, we can say that n is even. Suppose, a and b are different symbols from an input alphabet. If the algorithm requests an variable s_j^i for $i \leq n/2$, then the adversary returns a. If the algorithm requests an variable s_j^i for $i > n/2$, then the adversary returns b.

Because of the algorithm A uses $o(nk)$ queries, there are at least one $s_{j'}^{z'}$ and one $s_{j''}^{z''}$ that are not requested, where $z' \leq n/2$, $z'' > n/2$ and $j', j'' \in \{1, \ldots, k\}$.

Let s' be a string such that $s'_j = a$ for all $j \in \{1, \ldots, k\}$. Let s'' be a string such that $s''_j = b$ for all $j \in \{1, \ldots, k\}$.

Assume that A returns s'. Then, the adversary assigns $s_{j'}^{z'} = b$ and assigns $s_j^i = b$ for each $i > n/2, j \in \{1, \ldots, k\}$. Therefore, the right answer should be s''.

Assume that A returns a string $s \neq s'$. Then, the adversary assigns $s_{j''}^{z''} = a$ and assigns $s_j^i = a$ for each $i \leq n/2, j \in \{1, \ldots, k\}$. Therefore, the right answer should be s'.

So, the adversary can construct the input such that A obtains a wrong answer. □

4 Strings Sorting Problem

Let us consider the following problem.

Problem. For some positive integers n and k, we have the sequence of strings $s = (s^1, \ldots, s^n)$. Each $s^i = (s_1^i, \ldots, s_k^i) \in \Sigma^k$ for some finite size alphabet Σ. We search order $ORDER = (i_1, \ldots, i_n)$ such that for any $j \in \{1, \ldots, n-1\}$ we have $s^{i_j} \leq s^{i_{j+1}}$ in lexicographical order.

We use Heap sort algorithm [30, 10] as a base and Quantum algorithm for comparing string from Section 2. We can replace Heap sort algorithm by any other sorting algorithm, for example, Merge sort [10]. In a case of Merge sort, the big-O hidden constant in query complexity will be smaller. At the same time, we need more additional memory.

Let us present Heap sort for completeness of the explanation. We can use Binary Heap [30]. We store indexes of strings in vertexes. As in the previous

section, if we compare vertexes v and v' with corresponding indexes i and i', then $v > v'$ iff $s^i > s^{i'}$ in lexicographical order. We use COMPARE_STRINGS$(s^i, s^{i'}, k)$ for comparing strings. Binary Heap BH has three operations:

- ⋆ GET_MIN_AND_DELETE(BH) returns minimal s^i and removes it from the data structure.
- ⋆ ADD(BH, s^i) adds vertex with value i to the heap;
- ⋆ INIT(BH) initializes an empty heap;

The operations GET_MIN_AND_DELETE and ADD invoke COMPARE_STRINGS subroutine $\log_2 t$ times, where t is the size of the heap.

The algorithm is the following.

Algorithm 3 The Quantum Algorithm for Sorting Problem.

```
INIT(BH)                                          ▷ The initialization of the heap.
for i ∈ {1, ..., n} do
    ADD(BH, s^i)                                  ▷ Adding s^i to the heap.
end for
for i ∈ {1, ..., n} do
    ORDER ← ORDER ∪ GET_MIN_AND_DELETE(BH)        ▷ Getting minimal string.
end for
return ORDER
```

If we implement the sequence s as an array, then we can store the heap in the same array. In this case, we do not need additional memory.

We have the following property of the algorithm that can be proven by the same way as Theorem 1.

Theorem 3. *Algorithm 4 sorts $s = (s^1, \ldots, s^n)$ with query complexity $O(n(\log n)^2 \cdot \sqrt{k})$ and error probability $O\left(\frac{1}{n}\right)$.*

The lower bound for deterministic complexity can be proven by the same way as in Theorem 2.

Theorem 4. *Any deterministic algorithm for Sorting problem has $\Omega(nk)$ query complexity.*

The Radix sort [10] algorithm almost reaches this bound and has $O((n + |\Sigma|)k)$ complexity.

5 Intersection of Two Sequences of Strings Problem

Let us consider the following problem.

Problem. For some positive integers n, m and k, we have the sequence of strings $s = (s^1, \ldots, s^n)$. Each $s^i = (s^i_1, \ldots, s^i_k) \in \Sigma^k$ for some finite size alphabet Σ. Then, we get m requests $t = (t^1 \ldots t^m)$, where $t^i = (t^i_1, \ldots, t^i_k) \in \Sigma^k$. The

answer to a request t^i is 1 iff there is $j \in \{1, \ldots, n\}$ such that $t^i = s^j$. We should answer 0 or 1 to each of m requests.

We have two algorithms. The first one is based on *"Set of strings with quantum comparator"* data structure from Section 3. We store all strings from s to a self-balancing binary search tree BST. Then, we answer each request using $\textsc{Find}(BST, s^i)$ operation. Let us present the Algorithm 4.

Algorithm 4 The Quantum Algorithm for Intersection of Two Sequences of Strings Problem using *"Set of strings with quantum comparator"* .

```
INIT(BST)                                          ▷ The initialization of the tree.
for i ∈ {1, ..., n} do
    ADD(BST, s^i)                                  ▷ We add s^i to the set.
end for
for i ∈ {1, ..., m} do
    v ← FIND(BST, t^i)                             ▷ We search t^i in the set.
    if v = NULL then
        return 0
    end if
    if v ≠ NULL then
        return 1
    end if
end for
```

The second algorithm is based on Sorting algorithm from Section 4. We sort strings from s. Then, we answer to each request using binary search in the sorted sequence of strings [10] and COMPARE_STRINGS subroutine for comparing strings during the binary search. Let us present the Algorithm 5. Assume that the sorting Algorithm 4 is the subroutine $\textsc{Sort\_strings}(s)$ and it returns the order $ORDER = (i_1, \ldots, i_n)$. The binary search algorithm with COMPARE_STRINGS subroutine as comparator is subroutine $\textsc{Binary\_search\_for\_strings}(t, s, OREDER)$ and it searches t in the ordered sequence $(s^{i_1}, \ldots, s^{i_n})$. Suppose that the subroutine BINARY_SEARCH_FOR_STRINGS returns 1 if it finds t and 0 otherwise.

Algorithm 5 The Quantum Algorithm for Intersection of Two Sequences of Strings Problem using sorting algorithm .

```
ORDER ← SORT_STRINGS(s)                            ▷ We sort s = (s^1, ..., s^n).
for i ∈ {1, ..., m} do
    ans ← BINARY_SEARCH_FOR_STRINGS(t, s, OREDER)  ▷ We search t^i in the
ordered sequence.
    return ans
end for
```

The algorithms have the following query complexity.

Theorem 5. *Algorithm 4 and Algorithm 5 solve Intersection of Two Sequences of Strings Problem with query complexity* $O((n+m)\sqrt{k}\cdot\log n\cdot\log(n+m))$ *and error probability* $O\left(\frac{1}{n+m}\right)$.

Proof. The correctness of the algorithms follows from the description. Let us discuss the query complexity of the first algorithm. As in the proof of Theorem 1, we can show that constructing of the search tree requires $O(n\log n)$ comparing operations. Then, the searching of all strings t^i requires $O(m\log n)$ comparing operations. The total number of comparing operations is $O((m+n)\log n)$. We will use little bit modified version of the Algorithm 1 where we run it $3(\log(n+m))$ times. We can prove that comparing operation requires $O(\sqrt{k}\log(n+m))$ queries. The proof is similar to the proof of corresponding claim from the proof of Lemma 2. So, the total complexity is $O((n+m)\sqrt{k}\cdot\log n\cdot\log(n+m))$.

The second algorithm also has the same complexity because it uses $O(n\log n)$ comparing operations for sorting and $O(m\log n)$ comparing operations for all invocations of the binary search algorithm.

Let us discuss the error probability. Events of error in the algorithm are independent. So, all events should be correct. We can prove that the error probability for comparing operation is $O(1/(n+m)^3)$. The proof is like the proof of Lemma 2. So, the probability of correctness of one event is $1-\left(1-\frac{1}{(n+m)^3}\right)$. Hence, the probability of correctness of all $O((n+m)\log n)$ events is at least $1-\left(1-\frac{1}{(n+m)^3}\right)^{\alpha\cdot(n+m)\log n}$ for some constant α.

Note that

$$\lim_{n\to\infty}\frac{1-\left(1-\frac{1}{(n+m)^3}\right)^{\alpha\cdot(n+m)\log n}}{1/(n+m)}<1;$$

Hence, the total error probability is at most $O\left(\frac{1}{n+m}\right)$.

□

Note that Algorithm 5 has a better big-O hidden constant than Algorithm 4, because the Red-Black tree or AVL tree has a height that greats $\log_2 n$ constant times. So, adding elements to the tree and checking existence has bigger big-O hidden constant than sorting and binary search algorithms.

The lower bound for deterministic complexity can be proven by the same way as in Theorem 2.

Theorem 6. *Any deterministic algorithm for Intersection of Two Sequences of Strings Problem has* $\Omega((n+m)k)$ *query complexity.*

This complexity can be reached if we implement the set of strings s using Trie (prefix tree) [11, 7, 9, 21].

Note, that we can use the quantum algorithm for element distinctness [4],[3] for this problem. The algorithm solves a problem of finding two identical elements in the sequence. The query complexity of the algorithm is $O(D^{2/3})$, where D is

a number of elements in the sequence. The complexity is tight because of [1]. The algorithm can be the following. On j-th request, we can add the string t^j to the sequence $s^1, \ldots, s^n$ and invoke the element distinctness algorithm that finds a collision of t^j with other strings. Such approach requires $\Omega(n^{2/3})$ query for each request and $\Omega(mn^{2/3})$ for processing all requests. Note, that the streaming nature of requests does not allow us to access to all $t^1, \ldots, t^m$ by Oracle. So, each request should be processed separately.

6 Conclusion

In the paper we propose a quantum algorithm for comparing strings. Using this algorithm we discussed four data structures: *"Multi-set of strings with quantum comparator"*, *"Set of strings with quantum comparator"*, *"Map with a string key and quantum comparator"* and *"Binary Heap of strings with quantum comparator"*. We show that the first two data structures work faster than the implementation of similar data structures using Trie (prefix tree) in a case of $\log_2 n = o(k^{0.25})$. The trie implementation is the best known classical implementation in terms of complexity of simple operations (add, delete or find). Additionally, we constructed a quantum strings sort algorithm that works faster than the radix sort algorithm that is the best known deterministic algorithm for sorting a sequence of strings.

Using these two groups of results, we propose quantum algorithms for two problems: the Most Frequently String Search and Intersection of Two String Sets. These quantum algorithms are more efficient than deterministic ones.

Acknowledgement

We thank Aliya Khadieva and Farid Ablayev for useful discussions.

The work is performed according to the Russian Government Program of Competitive Growth of Kazan Federal University.

References

1. Aaronson, S., Shi, Y.: Quantum lower bounds for the collision and the element distinctness problems. Journal of the ACM (JACM) 51(4), 595–605 (2004)
2. Adel'son-Vel'skii, G.M., Landis, E.M.: An algorithm for organization of information. In: Doklady Akademii Nauk. vol. 146, pp. 263–266. Russian Academy of Sciences (1962)
3. Ambainis, A.: Quantum walk algorithm for element distinctness. In: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science. pp. 22–31. FOCS '04 (2004)
4. Ambainis, A.: Quantum walk algorithm for element distinctness. SIAM Journal on Computing 37(1), 210–239 (2007)
5. Ambainis, A.: Understanding quantum algorithms via query complexity. arXiv preprint arXiv:1712.06349 (2017)

6. Bennett, C.H., Bernstein, E., Brassard, G., Vazirani, U.: Strengths and weaknesses of quantum computing. SIAM journal on Computing 26(5), 1510–1523 (1997)
7. Black, P.E.: Dictionary of algorithms and data structures— nist. Tech. rep. (1998)
8. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. Fortschritte der Physik 46(4-5), 493–505 (1998)
9. Brass, P.: Advanced data structures, vol. 193. Cambridge University Press Cambridge (2008)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms-Secund Edition. McGraw-Hill (2001)
11. De La Briandais, R.: File searching using variable length keys. In: Papers presented at the the March 3-5, 1959, western joint computer conference. pp. 295–298. ACM (1959)
12. De Wolf, R.: Quantum computing and communication complexity (2001)
13. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 212–219. ACM (1996)
14. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). pp. 8–21. IEEE (1978)
15. Høyer, P., Neerbek, J., Shi, Y.: Quantum complexities of ordered searching, sorting, and element distinctness. In: International Colloquium on Automata, Languages, and Programming. pp. 346–357. Springer (2001)
16. Høyer, P., Neerbek, J., Shi, Y.: Quantum complexities of ordered searching, sorting, and element distinctness. Algorithmica 34(4), 429–448 (2002)
17. Jordan, S.: Bounded error quantum algorithms zoo, https://math.nist.gov/quantum/zoo
18. Khadiev, K., Kravchenko, D., Serov, D.: On the quantum and classical complexity of solving subtraction games. In: Proceedings of CSR 2019. LNCS (2019)
19. Khadiev, K., Safina, L.: Quantum algorithm for dynamic programming approach for dags. applications for zhegalkin polynomial evaluation and some problems on dags. In: Proceedings of Unconventional Computation and Natural Computation 2019, LNCS, vol. 4362 (2019)
20. Klauck, H.: Quantum time-space tradeoffs for sorting. In: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing. pp. 69–76. ACM (2003)
21. Knuth, D.: Searching and sorting, the art of computer programming, vol. 3 (1973)
22. Kothari, R.: An optimal quantum algorithm for the oracle identification problem. In: 31st International Symposium on Theoretical Aspects of Computer Science. p. 482 (2014)
23. Lin, C.Y.Y., Lin, H.H.: 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)
24. Lin, C.Y.Y., Lin, H.H.: Upper bounds on quantum query complexity inspired by the elitzur–vaidman bomb tester. Theory OF Computing 12(18), 1–35 (2016)
25. Montanaro, A.: Quantum pattern matching fast on average. Algorithmica 77(1), 16–39 (2017)
26. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information. Cambridge university press (2010)
27. Odeh, A., Abdelfattah, E.: Quantum sort algorithm based on entanglement qubits {00, 11}. In: 2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT). pp. 1–5. IEEE (2016)

28. Odeh, A., Elleithy, K., Almasri, M., Alajlan, A.: Sorting n elements using quantum entanglement sets. In: Third International Conference on Innovative Computing Technology (INTECH 2013). pp. 213–216. IEEE (2013)
29. Ramesh, H., Vinay, V.: String matching in $o(\sqrt{n}+\sqrt{m})$ quantum time. Journal of Discrete Algorithms 1(1), 103–110 (2003)
30. Williams, J.W.J.: Algorithm 232 - heapsort. Commun. ACM 7(6), 347–349 (Jun 1964)