Understanding The Complexity Of Interpolation Search

Seminar Advanced Algorithms and Data Structures

Simon Yuan

1 Introduction

Let's consider the following problem:
Given an English dictionary and a word starting with the letter "A", we want to find the page, on which the word is listed. It turns out that we have a very high chance of finding that word in the first 26th of the dictionary, and thus skipping 25/26 of all the pages. This is only the case, because the starting letter of a word is approximately uniformly distributed between the 26 letters of the English alphabet.
This searching technique is already well known in the field of computer science. It was first described in [6] in the year 1957 and is today known as interpolation search. Suppose we want to find (by interpolation search) a key in an ordered array of n keys, which were drawn independantly from a uniform distribution. Then the average number of probes (key comparisons) will be in $\mathcal{O}(\log \log n)$, which is superior than the average of binary search in $\mathcal{O}(\log n)$. The $\log \log n$ average behaviour has been proven in multiple ways as in [10] and [4], but neither proof can give us a simple intuition why there is such an behaviour.
In this report we will discuss yet another approach to analyse the average behaviour of a slightly modified version of the conventional interpolation search with the goal to better understand the intuition behind the behaviour. The algorithm is called binary interpolation search, which requires on average less than $2.42 \log \log n$ probes, as introduced in [5] (which we will refer to as *the paper*).
We will first try to comprehend the $\log \log n$ behaviour by exploring the idea that this behaviour may be derived from a quadratic application of the binary search, and then proceed to look at the binary interpolation search algorithm, which implements this idea and analyse its complexity.

2 Binary Search and Interpolation Search

Let's begin by reviewing the binary search and interpolation search algorithm. Given an ordered array A of n keys $x_1 < x_2 < \cdots < x_n$ and a key y, find the target index $i \in \{1, 2, \ldots, n\}$ such that $x_i = y$. We call x_i the target key.

2.1 Binary Search

What binary search essentially does, is probe the middle of the array and find out in which half it has to continue searching. It will first compare y with $x_{\lfloor n/2 \rfloor}$. If they are equal, then it has found the target index. Otherwise it continues to probe the middle of the respective sub-array $x_1, \ldots, x_{\lfloor n/2 \rfloor - 1}$ if $y < x_{\lfloor n/2 \rfloor}$ and $x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n$ if $y > x_{\lfloor n/2 \rfloor}$. If there is no such appropriate sub-array, then y is not in the array.
The number of probes in Binary search (see [8]) in the best case is just 1 and in the average and worst case it is in $\mathcal{O}(\log n)$. In contrast to interpolation search. The behaviour will not be improved even if we assume that the keys of A were independently drawn from a uniform distribution.
We can visualize the $\log n$ behaviour when we view the keys as a tree:

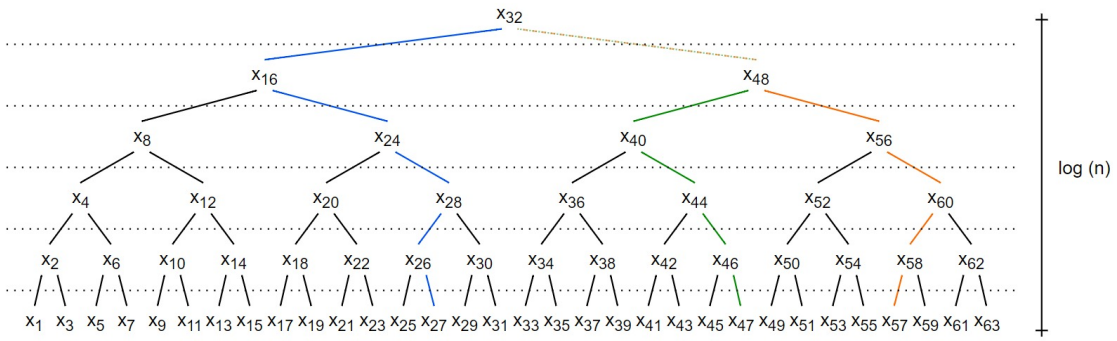

Figure 1: An array with $n = 63$ keys viewed as a tree. The levels are separated by dotted lines.

The first key to be probed is the root of the tree. After each key comparison the next key to be probed is either the left or the right child, meaning it continues one level deeper after each probe. Let's refer to the sequence of keys probed as the search path (see figure 1 for 3 such possible search paths). Binary search terminates the latest, when a leaf or in other words when the bottom of the search path has been reached. Because a complete binary tree with n keys is exactly $\lceil \log(n+1) \rceil$ levels deep, we can conclude that the number of probes required for binary search is in $\mathcal{O}(\log n)$.

2.2 Interpolation Search

We now assume that the keys of A are randomly and independently drawn from a uniform distribution over the range $[x_0, x_{n+1}]$ for some $x_0 \leq x_1$ and $x_{n+1} \geq x_n$. Because the keys are now uniformly distributed, we can expect around $p = (y - x_0)/(x_{n+1} - x_0)$ of the keys are less than y. Thus, interpolation search first compares y with $x_{\lceil pn \rceil}$. If they are equal, it has found the target index, otherwise we apply the same method recursively on the respective sub-array $x_1, \ldots, x_{\lceil pn \rceil - 1}$ if $y < x_{\lceil pn \rceil}$ and $x_{\lceil pn \rceil + 1}, \ldots, x_n$ if $y > x_{\lceil pn \rceil}$. If there is no such sub-array, then y is not in A.
The number of probes in interpolation search (see [9], [10] or [4]) in the best case is also just 1, in the average case it is in $\mathcal{O}(\log \log n)$ and in the worst case it is in $\mathcal{O}(n)$.

Remark: The complexity in the worst case could be improved by running interpolation search and binary search in parallel or alternatingly, such that the worst case is in $\mathcal{O}(\log n)$. However this approach has no real utitlity, since it has been shown in [4] that it is already unlikely that interpolation search will require just a few more than $\log \log n$ probes.

3 The Quadratic Application of Binary Search

We have already seen in the previous chapter that binary search requires about $\log n$ many probes, this suggests that it might be possible to view the $\log \log n$ behaviour of interpolation search as a quadratic application of binary search. We have also seen, that when applying binary search on an array with n keys, we get a search path, whose length is bounded by $\log n$. But what if, instead of going down the search path linearly, we applied binary search on the search path itself? This would then have the desired $\log \log n$ behaviour. Unfortunately, because the search path is not known before having applied binary search on the whole array of n keys, applying binary search on the search path is of course not possible. What is possible, is to apply binary search to the levels of the tree. Meaning, we cut the levels of tree in half after each probe (in figure 1 that would correspond to the cut along the third dotted line). This will lead us with a top half consisting of one sub-tree and a bottom half with several sub-trees. All these sub-trees have a depth of $\frac{1}{2}log(n) = log(\sqrt{n})$, and thus all have around $\sqrt{n}$ keys each. See figure 2 for a visualization.

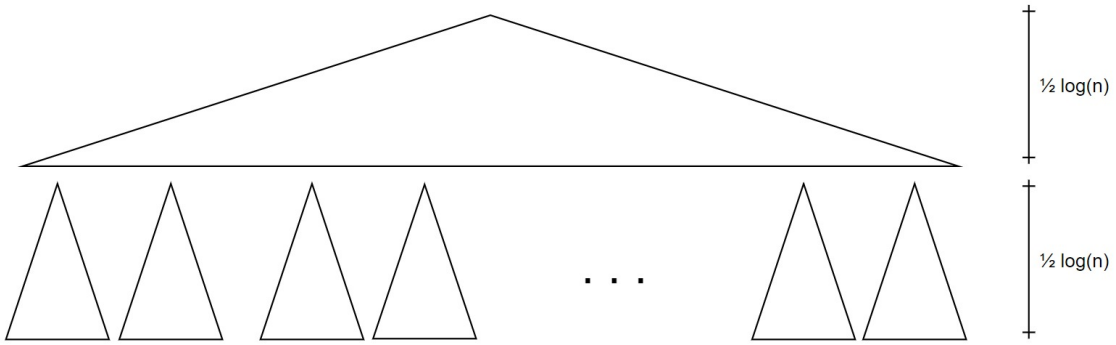

Figure 2: Splitting the tree into half.

Splitting the tree into half does not yet help us locate where the target key is. What we can do is to sequentially probe the keys from the upper sub-tree and determine in which bottom sub-tree we have to continue searching. Of course, if the target key is in the upper half we do not have to continue to search the appropriate next sub-tree in the bottom half. But if not, then after having determined in which bottom sub-tree to continue searching, we can apply the same searching method recursively on the new found sub-tree until the target key has been found or until there are no more sub-trees, in which case there is no target key.

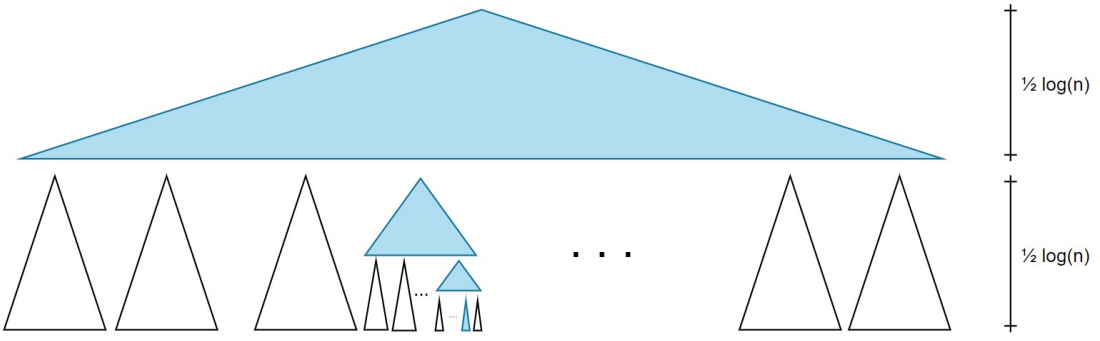

Figure 3: The search path visualized.

Figure 3 shows us a possible search path, where the blue marked sub-trees are the top trees, whose keys are being probed to determine the next sub-tree. In the next chapter we will see an algorithm

which requires only a constant number of probes on average in the process of determining the next sub-tree. Thus, with this quadratic binary search method we have effectively found a way to search for the target key in $\mathcal{O}(\log \log n)$ complexity.

4 Binary Interpolation Search

We will now examine the binary interpolation search algorithm, which uses interpolation search to determine the first key to be probed. For simplicity, we assume that the key to be probed is in the top sub-tree. It then continues to sequentially probe the keys to the left or to the right of the top sub-tree, depending on the result of the first probe.
In the analysis we will see that by using interpolation search for the first probe on each iteration we are on average only a constant number of probes away to determine the next corresponding sub-tree.

4.1 The Algorithm

Algorithm 1
BinaryInterpolation$(A, y, 0, n-1)$ returns the index of y in a 0-based Array A with n keys. If y is not in A, it returns -1. l and r are the left and right bound of the search space.

```
1:  procedure BinaryInterpolation(A, y, l, r)
2:      if l > r || (l == r && A[l] ≠ y) then
3:          return −1
4:      else if l == r && A[l] == y then
5:          return l
6:      p ← (y − A[l])/(A[r] − A[l])
7:      mid ← l + p ∗ (r − l)
8:      i ← 1
9:      if y > A[mid] then
10:         loop forever
11:             next ← mid + i⌈√n⌉
12:             if next > r || y < A[next] then break
13:             if y == A[next] then return next
14:             i ← i + 1
15:         l ← mid + (i − 1)⌈√n⌉ + 1
16:         r ← Min(r, next − 1)
17:         return BinaryInterpolation(A, y, l, r)
18:     else if y < A[mid] then
19:         loop forever
20:             next ← mid − i⌈√n⌉
21:             if next < l || y > A[next] then break
22:             if y == A[next] then return next
23:             i ← i + 1
24:         r ← mid − (i − 1)⌈√n⌉ − 1
25:         l ← Max(l, next + 1)
26:         return BinaryInterpolation(A, y, l, r)
27:     else
28:         return mid
```

The algorithm first covers the base cases in lines 2-5, where there is only one key to be checked. Then it interpolates over the search space to find the first key x_{mid} to be probed (lines 6-7). If $y > x_{mid}$ then it essentially loops over $i \in \{1, 2, ...\}$ in lines 10-14 to find the smallest i, such that

$y < x_{mid+i\lceil\sqrt{n}\rceil}$. When this i has been found, it can continue in line 15-17 to search recursively in the remaining search space between $x_{mid+(i-1)\lceil\sqrt{n}\rceil+1}$ and $x_{mid+i\lceil\sqrt{n}\rceil-1}$ (if it isn't an edge case). This corresponds to the sequential probing of the keys to the right of x_{mid} in the top sub-tree. In other words, $x_{mid+i\lceil\sqrt{n}\rceil}$ corresponds to the i-th key to the right of x_{mid}, because each bottom sub-tree has around $\lceil\sqrt{n}\rceil$ keys.
The case, where $y < x_{mid}$ (lines 18-26), is analogous, in which it finds the smallest i, such that $y > x_{mid-i\lceil\sqrt{n}\rceil}$. In both cases the size of the search space of n keys is at least reduced to $\sqrt{n}$.
Note that this algorithm, does not implicitly go through the keys in the top sub-tree, it can only be viewed as such, if the assumption that x_{mid} actually is in the top sub-tree holds. But even if the assumption does not hold, the algorithm in fact still works with the desired complexity, as it reduces the search space to $\sqrt{n}$ anyway and that on average in a constant time. This will be important in the analysis.

4.2 Analysis

Let's begin with the analysis of the best and worst case complexity. In the best case, only 1 probe is needed (line 27). The number of iterations (variable i) of the loop in line 10 or 19 is bounded by $\sqrt{n}$, because then $mid \pm \sqrt{n}\lceil\sqrt{n}\rceil$ is definitely out bound and we exit the loop in line 12 or 21 respectively. Thus, in the worst case the number of probes is $\sqrt{n} + \sqrt{\sqrt{n}} + \ldots$, which is in $\mathcal{O}(n^{1/2})$. This could technically be improved as described in the remark of section 2.2.
Let's now analyse the average case. Let $T(n)$ be the average number of probes needed to find a key in an Array of size n. To determine that, we must calculate the expected number of probes C, which are used by BinaryInterpolation to reduce the search space of size x to the size $\sqrt{x}$, meaning the number of key comparisons used from line 9 to 16, or line 18 to 25 respectively. By lemma 1 (see below), C is bounded by a constant $\hat{C}$. Therefore we have

$$T(n) \leq \hat{C} + T(\sqrt{n})$$

To eliminate $T(\sqrt{n})$ from the term we need to telescope the equation. Assume that $n = z^{2^k}$ for some $k, z \in \mathbb{N}$ and that $T(z)$ is small. We have

$$T(n) = T(z^{2^k}) \leq \hat{C} + T(z^{2^{k-1}}) \leq \hat{C} + \hat{C} + T(z^{2^{k-2}}) \leq \underbrace{\hat{C} + \hat{C} + \cdots + \hat{C}}_{k \text{ times}} + T(z) = k\hat{C} + T(z)$$

Because $n = z^{2^k}$ and $z \geq 2$ if $n > 1$ we have $log_2(z) \geq 1$ and thus

$$k = log_z(log_2(n)) = \frac{log_2(log_2(n))}{log_2(z)} \leq log_2(log_2(n))$$

Therefore we conclude that

$$T(n) \leq \hat{C} * log(log(n)) + T(z)$$

Finally, after plugging in the constant from lemma 1 and ignoring the small $T(z)$ we have:

$$T(n) \leq 2.42 \cdot log(log(n))$$

Lemma 1. *Assuming that the keys are drawn independently from a uniform distribution, then the expected number of probes C used in* BinaryInterpolation *to determine the next subspace is bound by a constant.*

Proof. Assume without loss of generality that the current search space is from 0 to $n-1$. Because the keys x_i are all independently drawn from a uniform distribution, the indicator that a key is less or equal to y is Bernoulli distributed with probability $p = (y - A[0])/(A[n-1] - A[0])$. Thus, the number of keys X, which are less or equal to y is binomially distributed: The probability of

exactly i keys being less or equal to y is $\binom{n}{i}p^i(1-p)^{n-i}$. Because the distribution of X is binomial, its expected value $\mu = np$ and the variance σ^2 is $np(1-p)$.
We determine C as follows:

$$C = \sum_{i=1}^{\infty} i \cdot \Pr[\text{exactly i probes are used}] = \sum_{i=1}^{\infty} \Pr[\text{at least i probes are used}]$$

Notice that at least two probes are always used, first the probe with *mid* at line 9 or 18 and the second with $mid \pm \lceil\sqrt{n}\rceil$ at line 12 and 21 respectively. And for $i \geq 3$ we have that

$$\Pr[\text{at least i probes are used}] \leq \Pr[|(\text{location of } y) - np| \geq (i-2)\lceil\sqrt{n}\rceil]$$

This probability is bounded above by $\frac{\sigma^2}{((i-2)\lceil\sqrt{n}\rceil)^2}$ by Chebyshev's inequality:

$$\Pr[|X - \mu| \geq t] \leq \frac{\sigma^2}{t^2}$$

for a random variable X with expected value μ and variance σ^2.
By observation 1 (see below) we have $\sigma^2 = p(1-p)n \leq \frac{1}{4}n$ and therefore:

$$C \leq 2 + \sum_{i=3}^{\infty} \frac{\sigma^2}{((i-2)\lceil\sqrt{n}\rceil)^2} \tag{1}$$

$$\leq 2 + \sum_{i=3}^{\infty} \frac{1}{4}\frac{n}{((i-2)\lceil\sqrt{n}\rceil)^2} \tag{2}$$

$$\leq 2 + \sum_{i=3}^{\infty} \frac{1}{4}\frac{1}{(i-2)^2} \tag{3}$$

$$= 2 + \frac{1}{4}\sum_{i=1}^{\infty} \frac{1}{i^2} \tag{4}$$

$$= 2 + \frac{1}{4}\frac{\pi^2}{6} \leq 2.42 \tag{5}$$

□

Remark: In the paper we can find a proof for an even tighter bound for C, which uses the DeMoivre-Laplace limit theorem. The result is:

$$C \leq 2 + \sum_{i \geq 3} \frac{1}{2\sqrt{2\pi}}\frac{1}{i-2}e^{-2(i-2)^2} \approx 2.027029$$

Observation 1. *Let $f(x) = x(1-x)$. Then $f(x) \leq \frac{1}{4}$ holds for all $x \in \mathbb{R}$.*

Proof. $f(x)$ is a quadratic function with the maximum at $x = 1/2$ and value $f(\frac{1}{2}) = \frac{1}{4}$, because

$$\frac{d}{dp}p(1-p) = \frac{d}{dp}p - p^2 = 1 - 2p = 0 \leftrightarrow p = \frac{1}{2} \quad \text{and}$$
$$\frac{d^2}{dp^2}p(1-p) = -2 < 0$$

□

5 Conclusion

In mathematical fields we often encounter proofs, which are completely correct, but are nevertheless non intuitive or hard to comprehend. In every presentation of some mathematical proof, there is the underlying math and the way how it is presented. To illustrate: it is hard to understand why $\sin(45) = \frac{\sqrt{2}}{2}$ if we only know the expanded form of sinus $\sin(x) = \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!}$ and not know that $\sin(x)$ can be seen as the ratio of the length of the opposite side to the length of the hypotenuse. Both views are correct and $\sin(45) = \frac{\sqrt{2}}{2}$ can proven using either definition, but obviously, proving it with the second view is much simpler than proving it using the other. In my own experience I can say, that I learn and understand concepts far easier if they are presented in a clear and intuitive way and thus, such an proof, just like the proof presented by the paper, has high pedagogical value.
The main take away from the paper was not the introduction of another search algorithm with the $\log \log n$ average behaviour or its exact analysis, but rather the fact that the introduction of BinaryInterpolation is helping us to better understand the behaviour of interpolation search.
As this report has come to an end, I would like to recommend some elegant proofs, all presented in a very intuitive and graphical way. The first link [3] is a proof of the basel problem, where one has to show that $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$, which we actually used in the proof of lemma 1 at equation (5). More beautifully explained proofs can be found via [2], [1] or [7].

References

[1] 3Blue1Brown. Pi hiding in prime regularities. `https://www.youtube.com/watch?v=NaL_Cb42WyY`. Accessed: 25.10.2018.

[2] 3Blue1Brown. Why does this product equal $\pi/2$? A new proof of the Wallis formula for π. `https://www.youtube.com/watch?v=8GPy_UMV-08`. Accessed: 25.10.2018.

[3] 3Blue1Brown. Why is pi here? And why is it squared? A geometric answer to the Basel problem. `https://www.youtube.com/watch?v=d-o3eB9sfls`. Accessed: 25.10.2018.

[4] Perl, Y., Itai, A., and Avni, H. Interpolation search - a log log n search. *Communications of the ACM 21*, 7 (1978), 550–553.

[5] Perl, Y., and Reingold, E. M. Understanding the complexity of interpolation search. *Information Processing Letters 6*, 6 (1977), 219–222.

[6] Peterson, W. W. Addressing for random-access storage. *IBM journal of Research and Development 1*, 2 (1957), 130–146.

[7] ThinkTwice. `https://www.youtube.com/channel/UC9yt3wz-6j19RwD5m5f6HSg/videos`. Accessed: 25.10.2018.

[8] Wikipedia contributors. Binary search algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Binary_search`. Accessed: 24.10.2018.

[9] Wikipedia contributors. Interpolation search — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Interpolation_search`. Accessed: 24.10.2018.

[10] Yao, A. C., and Yao, F. F. The complexity of searching an ordered random table. In *Foundations of Computer Science, 1976., 17th Annual Symposium on* (1976), IEEE, pp. 173–177.