**Algorithm Study Template**

**Algorithm**: Jump Search

**aka**: Block Search

**Techniques**: Jumping

**Categories**: Searching

**Problem**: This algorithm is designed to find a particular value (or key) in a sorted list. If the value is found, the algorithm also reports the location of the key inside of the list.

**Applications**: Although the Binary Search is often favored over Jump Search because of its complexity and ease of implementation, the Jump Search can be better in certain circumstances. For instance, jumping to the middle of the list right away causes the Binary Search to have to jump back many times if searching for one of the first items in a very large list. The Jump Search, in contrast, only has to jump back once at most. So, the Jump Search is useful for searching sorted lists, especially when searched keys are frequently near the beginning of the list.

**References**:

* <http://www.stoimen.com/blog/2011/12/12/computer-algorithms-jump-search/>
* <http://xlinux.nist.gov/dads//HTML/jumpsearch.html>
* *Jump Searching: A Fast Sequential Search Technique*, CACM, 21(10):831-834, October 1978.

**Implementation details**:

* **Big Idea**: Search a sorted list by checking every so many items according to a fixed step. Once the right area of the list is found, a sequential search is performed to find the desired key.
* **Description**:

The Algorithm begins by taking in a sorted list and the search key as parameters. Next, the step is calculated by taking the floor of the square root of the length of the list, or floor(sqrt(len)). Then we start checking from the first step and if list[step] is less than the key, we step forward again until list[min(step,len)] is greater than the key we’re searching for. This indicates that we have reached the general area of the key. However, if list[min(step,len)] is never greater than the search key, this will lead to the conclusion that the key is not in the list because the last item in the list, which should be the largest due to the fact that this is a sorted list, is smaller than the search key. Next, if the current list value being compared to the key is not equal, then is it less, so we now perform a sequential search to reach the key. If the sequential search finds that the value being compared is equal to min(step,len), then it concludes that the key is not in the list because we’ve reached the end of the general searching area. Obviously, if the value being compared is equal to the key value, we return the location of that key in the list.

* **Pseudo-code**: (adapted from [Wikipedia](http://en.wikipedia.org/wiki/Jump_search))

**Algorithm** JumpSearch

Input: An ordered list *L*, its length *n* and a search key *s*.

Output: The position of *s* in *L*, or **-1** if *s* is not in *L*.

*a* ← 0

*b* ← ⌊√*n*⌋ //floor(sqrt(n))

**while** *L*min(*b*,*n*)-1 < *s* **do**

*a* ← *b*

*b* ← *b* + ⌊√*n*⌋ //increment by floor(sqrt(n))

**if** *a* ≥ *n* **then**

**return** **-1**

**while** *La* < *s* **do**

*a* ← *a* + 1

**if** *a* = min(*b*,*n*)

**return** **-1**

**if** *La* = *s* **then**

**return** *a*

**else**

**return** **-1**

* **Specific implementation**: (see Searches.java)

**Correctness**:

**Theoretical**: If list[min(step,len)] is never greater than the search key, then we know the key is not in the list because the last item in the list, which should have the largest value because the list is sorted, is less than the search key. If list[step] is greater than the search key, we know that the key is somewhere between the current and previous steps. So, we go back to the previous step and sequentially search forward until the key is found. If the sequential search reaches the step larger than the key, or the end of the list, without finding a match, then we know the key is not in the list because the key was not in the range where it should have been. The key would have to be in that range, if in the list, because the list is sorted.

**Empirical**: In order to show correctness, one of the tests a user can choose to run when the program starts is a correctness test. This test performs two actions to show that the search is operating as it should. First, it generates a list of ten million integers (0 – 9,999,999). Then, it picks a random integer of the range 0 – 9,999,999 and searches for it in the list, where it is inevitably found. Next, it searches for a programmatically specified integer that is known for a fact to be outside the aforementioned range, and thus not in the list. Since the last value in the list is 9,999,999 I chose to search for 10,000,000. The value, of course, is not found. I also searched for values like 50,000,005 and 70,007,001, of course not finding them in the list. Thus, the search finds values that are known to be in the list and fails to find values that are known not to be in the list. A sampling of one of these runs is shown below.

Searching for: 2,121,080

Jump Search found key at index 2,121,080.

Searching for: 70,007,001

Jump Search did not find the key.

**Performance**:

**Theoretical**: Because the Jump Search checks at most sqrt(n) items, it runs in O(sqrt(n)) time. This is better than a Sequential search, but worse than a Binary Search.

**Empirical**: I recorded time measurements (in milliseconds) for the execution of just the search algorithm itself (with no printed output or other “noise” included) with a variety of different datasets. I used a total of twenty five datasets, five sizes, each with five configurations. Each test was run five times. I tested with ten million, one million, one hundred thousand, ten thousand, and one thousand integers. For each of those sizes, I had a list where all values were randomized and then sorted into place, a list where all values were generated sequentially in order (similarly to the aforementioned correctness test), a list where one fourth of all values were identical and the rest random but not equal to the identical value, a list where one half of all values were identical and the rest random but not equal to the identical value, and a list where three fourths of all values were identical and the rest random but not equal to the identical value. The recorded times from all of these tests can be seen in Test Results.xlsx, which also includes tests for the other search algorithms detailed in this same report. Please see that file for the actual data, as none of it is reported in this document. I also wish to note that for the datasets containing set percentages of duplicates I tested by searching for the duplicated value and then any value other than the duplicated value to see if there would be a difference in execution times. A full compare and contrast of all searches in this report is reserved for the final page. The following is a brief discussion of Jump Search’s performance:

Overall, I was impressed by the Jump Search. It was second only to the Binary search in my tests, which surprised me to a degree. I wouldn’t have thought that what is essentially an optimized Sequential Search would do so much better than a regular Sequential Search, but Jump Search seemed better across the board. The only exception to that would be for the tests with a list size of one thousand. There, the Sequential Search tended to outperform Jump Search because the optimizations were likely not needed for that list size. The execution times for Jump Search did decrease noticeably as the list sizes decreased, but not by much more than half a millisecond from one size to the next down. The presence of percentages of duplicate values didn’t affect the time as much as I had imagined they would. Larger datasets seemed to slightly favor searching for duplicates and smaller datasets seemed to slightly favor searching for non-duplicates. I imagine this is because the larger datasets with more duplicate values allow the algorithm to “jump” farther and find them more easily due to them being a fairly large target to land on with a “jump.” With a smaller dataset, the “jumps” could actually get in the way and lead to going forward and backward, rather than just forward with a standard Sequential Search.

**Anecdotes**: Many programmers who favor the Binary Search over the Jump Search miss out on Jump Search’s speed when finding keys toward the beginning of the list.

**History**: It appears that Jump Search was first written about by Ben Shneiderman in *Communications of the ACM* in October 1978.

**Variations**: I was unable to identify any direct variations of Jump Search, but Jump Search itself can be described as a variation of the Sequential Search.

**Alternatives**: Sequential Search, Binary Search, and Interpolation Search can all also be used for finding a search key in a sorted list, though they each go about the task in fairly different ways. The Sequential Search starts at the beginning and checks each item, the Binary Search starts at the middle and recursively divides the list, and the Interpolation Search uses a formula to more effectively divide the list for searching.

**Credits:**

* <http://en.wikipedia.org/wiki/Jump_search>

**Algorithm Study Template**

**Algorithm**: Binary Search

**aka**: Half-Interval Search

**Techniques**: Divide & Conquer

**Categories**: Searching, Dichotomotic Search

**Problem**: This algorithm is designed to find a particular value (or key) in a sorted list. If the value is found, the algorithm also reports the location of the key inside of the list.

**Applications**: Binary Search can be used for virtually any application where searching a sorted list is required. A more specific example would be searching a dictionary for a certain word or a phone directory for a certain number since these are distinct items in sorted lists.

**References**:

* <http://www.stoimen.com/blog/2011/12/26/computer-algorithms-binary-search/>
* <http://codeabbey.com/index/wiki/binary-search>
* <http://xlinux.nist.gov/dads//HTML/binarySearch.html>
* *Algorithms*, 4th ed., Sedgewick & Wayne, pp. 46 – 49
* *The Art of Computer Programming*, Donald Knuth, Vol.3, section 6.2.1
* <http://rosettacode.org/wiki/Binary_search>

**Implementation details**:

* **Big Idea**: Begin by comparing the item in the middle of the list to the search key. If the key is smaller, recursively search the first half of the list. If the key is larger, recursively search the second half of the list.
* **Description**: Begin by ensuring that the first index in the search partition is less than or equal to the last index in the search partition. This is to ensure that the base case for recursion has not been violated. Next, we set the midpoint by taking the sum of the first and last indices of the partition and dividing by two. This number becomes the index of the midpoint. Now, if the search key is less than the middle list item, we recursively search the left half of the list. If the search key is greater than the middle list item, we recursively search the right half of the list. If the key is not found, we eventually return negative one.
* **Pseudo-code**: (adapted from [Code Codex](http://www.codecodex.com/wiki/Binary_search#Pseudocode))

**function** binarySearch(a, value, left, right)

**if** right < left

**return** *-1*

mid := (left + right) >> 1

**if** a[mid] = value

**return** mid

**if** value < a[mid]

**return** binarySearch(a, value, left, mid-1)

**else**

**return** binarySearch(a, value, mid+1, right)

* **Specific implementation**: (see Searches.java)

**Correctness**:

**Theoretical**: If the first index of the search partition is greater than the last index of the search partition, then the key value does not exist in the list because it would have been found by now where the first index was less than or equal to the last index. Now, if the midpoint of the search partition is equal to the search key, the key is found. If the key is less than the midpoint, we can assert than the key is between the first index and the midpoint. If the key is larger than the midpoint, we can assert that the key is between the midpoint and the last index. Then the list can be sorted recursively, according to those assertions, until the key is found or not found.

**Empirical**: In order to show correctness, one of the tests a user can choose to run when the program starts is a correctness test. This test performs two actions to show that the search is operating as it should. First, it generates a list of ten million integers (0 – 9,999,999). Then, it picks a random integer of the range 0 – 9,999,999 and searches for it in the list, where it is inevitably found. Next, it searches for a programmatically specified integer that is known for a fact to be outside the aforementioned range, and thus not in the list. Since the last value in the list is 9,999,999 I chose to search for 10,000,000. The value, of course, is not found. I also searched for values like 50,000,005 and 70,007,001, of course not finding them in the list. Thus, the search finds values that are known to be in the list and fails to find values that are known not to be in the list. A sampling of one of these runs is shown below.

Searching for: 2,121,080

Binary Search found key at index 2,121,080.

Searching for: 70,007,001

Binary Search did not find the key.

**Performance**:

**Theoretical**: The Binary Search is a logarithmic algorithm and executes in O(log n) time. log2(*N*)−1 is the expected number of probes in an average successful search, and the worst case is log2(*N*), just one more probe.

**Empirical**: I recorded time measurements (in milliseconds) for the execution of just the search algorithm itself (with no printed output or other “noise” included) with a variety of different datasets. I used a total of twenty five datasets, five sizes, each with five configurations. Each test was run five times. I tested with ten million, one million, one hundred thousand, ten thousand, and one thousand integers. For each of those sizes, I had a list where all values were randomized and then sorted into place, a list where all values were generated sequentially in order (similarly to the aforementioned correctness test), a list where one fourth of all values were identical and the rest random but not equal to the identical value, a list where one half of all values were identical and the rest random but not equal to the identical value, and a list where three fourths of all values were identical and the rest random but not equal to the identical value. The recorded times from all of these tests can be seen in Test Results.xlsx, which also includes tests for the other search algorithms detailed in this same report. Please see that file for the actual data, as none of it is reported in this document. I also wish to note that for the datasets containing set percentages of duplicates I tested by searching for the duplicated value and then any value other than the duplicated value to see if there would be a difference in execution times. A full compare and contrast of all searches in this report is reserved for the final page. The following is a brief discussion of Binary Search’s performance:

I was pleasantly surprised by the performance of Binary Search. It fully “lived up to the hype” and easily outperformed all other searches tested. Overall, the times to find identical values versus random values mixed in with identical ones were almost the same, with a slight bias in favor of non-identical values. The slowest execution times for the largest dataset were tenths of milliseconds, and all times gradually fell into thousandths of milliseconds for the smallest dataset. There wasn’t much difference between times for the ten thousand values dataset and the one thousand values dataset, but times decreased somewhat uniformly when stepping down through the first three datasets.

**Anecdotes**: Many professional programmers tend to use Binary Search as a “one size fits all” method of searching sorted lists, but there are cases where either the Jump Search or the Interpolation Search performs better. One supposes this potential performance loss is accepted because the Binary Search is viewed as “tested and proven” by many programmers who want something that “just works”.

**History**: The Binary Search was first mentioned in 1946 by John Mauchly. It was published by H. Bottenbruch in 1962.

**Variations**: Variations include the Fibonacci Search, Interpolation Search, Balanced Binary Search, and Noisy Binary Search. They are all variations on one overarching search method.

**Alternatives**: Sequential Search, Jump Search, and Interpolation Search would all be able to accomplish the same task of searching a sorted list. The Sequential Search starts at the beginning and checks each item, the Jump Search uses a calculated step value to perform a hybrid Sequential Search, and the Interpolation Search uses a formula to more effectively divide the list for searching. In addition, there is also the binary search tree.

**Credits:**

* <http://en.wikipedia.org/wiki/Binary_search>
* <http://www.codecodex.com/wiki/Binary_search>

**Algorithm Study Template**

**Algorithm**: Interpolation Search

**aka**: Extrapolation Search

**Techniques**: Divide & Conquer

**Categories**: Searching

**Problem**: This algorithm is designed to find a particular value (or key) in a sorted list. If the value is found, the algorithm also reports the location of the key inside of the list.

**Applications**: In an above template, I mentioned that the Jump Search improves on the Binary Search by being able to more easily find values toward the beginning of a list. The Interpolation Search improves on that by being better able to handle values toward the beginning or the end of the list. This is because the Interpolation Search partitions by a variable amount that allows it to quickly zoom in on a particular area of a list. With that said, perhaps you have an array of employee data records that are sorted by year of birth. If all of the employees were born in the 1980’s and you are searching for somebody born in 1981, the Interpolation Search narrows down the search area to just those employees born in 1981 for us and performs a Binary Search on those employees.

**References**:

* <http://www.stoimen.com/blog/2012/01/02/computer-algorithms-interpolation-search/>
* <http://www.cs.technion.ac.il/~itai/publications/Algorithms/p550-perl.pdf>
* <http://xlinux.nist.gov/dads//HTML/interpolationSearch.html>
* Sedgewick, Robert (1990), *Algorithms in C*, Addison-Wesley
* <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Interpolation_search.html>

**Implementation details**:

* **Big Idea**: The Interpolation Search is essentially a Binary Search that uses k=(x-L)(R-L) instead of ½ for partitioning. This allows us to get closer to the search key in fewer partitions.
* **Description**:

Like the Binary Search, the Interpolation Search begins by ensuring that the first index of the list is less than or equal to the last index of the list as a base for the repetition. Then, if the list elements at the first and last indices (right and left) are equal and the first item in the list is equal to the search key, the first index is returned as the location of the key. If the list elements at the first and last indices are equal but the first item in the list is not equal to the search key, then the key is not in the list and negative one is returned.

Next, the k value is calculated by the formula k=(key-list[left])(list[right]-list[left]). If k is not found (value is less than zero or greater than one), the key is not in the list and negative one is returned. Next, the midpoint is calculated by the formula mid=left + k\*(right - left). This gives us a midpoint that is at least somewhat close to the search key.

The rest of the algorithm is exactly like a Binary Search. If the key is less than the midpoint, we change the right value to be mid-1. If the key is greater than the midpoint, we change the left value to be mid+1. And, of course, if the key is not less than or greater than the midpoint, it must be equal, so we return the mid index as the location of the key.

* **Pseudo-code**: (adapted from my implementation code)

function interpolationSearch(int key, int[] list){

int left = 0

int right = length(list)-1

//list beginning and end could be the same

//if so, that value could be the search item

while(left <= right){

if(list[left] == list[right]){

if(list[left] == key){

return left

}else{

return -1

}

}

//find k, used for finding a midpoint closer to the search value

int k = (x - list[left])/(list[right] - list[left])

//if not found (can't calculate k)

if(k < 0 or k > 1){

return -1

}

//find midpoint for searching

Int mid = left + k \* (right - left)

//probe list and set bounds for next iteration's search

if(key < list[mid]){

right = mid-1

}else if(key > list[mid]){

left = mid+1

}else{

return mid

}

}

return -1

}

* **Specific implementation**: (see Searches.java)

**Correctness**:

**Theoretical**: If the first index is greater than the last index, violating the base case, negative one is returned to indicate that no match is found. If a midpoint cannot be calculated, then the algorithm could not estimate a position for the key. Therefore, the key must not be in the list. Furthermore, once a midpoint is found, if the key is less than the midpoint, then the key must be between left and mid-1. If the key is greater than the midpoint, then it must be between mid+1 and right. By those assumptions, if list[mid] = key, then mid is returned as the location of the key.

**Empirical**: In order to show correctness, one of the tests a user can choose to run when the program starts is a correctness test. This test performs two actions to show that the search is operating as it should. First, it generates a list of ten million integers (0 – 9,999,999). Then, it picks a random integer of the range 0 – 9,999,999 and searches for it in the list, where it is inevitably found. Next, it searches for a programmatically specified integer that is known for a fact to be outside the aforementioned range, and thus not in the list. Since the last value in the list is 9,999,999 I chose to search for 10,000,000. The value, of course, is not found. I also searched for values like 50,000,005 and 70,007,001, of course not finding them in the list. Thus, the search finds values that are known to be in the list and fails to find values that are known not to be in the list. A sampling of one of these runs is shown below.

Searching for: 2,121,080

Interpolation Search found key at index 2,121,080.

Searching for: 70,007,001

Interpolation Search did not find the key.

**Performance**:

**Theoretical**: The performance of Interpolation Search is about O(log log n). As seen [here](http://www.stoimen.com/blog/wp-content/uploads/2012/01/logntologlogn.png), the function grows very slowly. It should be noted, however, that the favorable performance of Interpolation Search depends on the list data being equally dispersed or uniformly distributed. The less uniform the data is, the worse the search will perform.

**Empirical**: I recorded time measurements (in milliseconds) for the execution of just the search algorithm itself (with no printed output or other “noise” included) with a variety of different datasets. I used a total of twenty five datasets, five sizes, each with five configurations. Each test was run five times. I tested with ten million, one million, one hundred thousand, ten thousand, and one thousand integers. For each of those sizes, I had a list where all values were randomized and then sorted into place, a list where all values were generated sequentially in order (similarly to the aforementioned correctness test), a list where one fourth of all values were identical and the rest random but not equal to the identical value, a list where one half of all values were identical and the rest random but not equal to the identical value, and a list where three fourths of all values were identical and the rest random but not equal to the identical value. The recorded times from all of these tests can be seen in Test Results.xlsx, which also includes tests for the other search algorithms detailed in this same report. Please see that file for the actual data, as none of it is reported in this document. I also wish to note that for the datasets containing set percentages of duplicates I tested by searching for the duplicated value and then any value other than the duplicated value to see if there would be a difference in execution times. A full compare and contrast of all searches in this report is reserved for the final page. The following is a brief discussion of Interpolation Search’s performance:

I was very disappointed by Interpolation Search’s performance. Having read that it was created as an improvement of the Binary Search, I expected those two to compete for the best performance. I could not have been more wrong. The Interpolation Search was outperformed, without fail, by the Sequential Search in virtually all tests. I believe this comes from the overhead involved with getting the Interpolation Search into the “general area” of the search key. The calculations involved with that, along with the Binary Search that follows, end up taking much more time than a regular Binary Search. I had also expected to see the greatest effect of the percentages of identical values on the Interpolation Search. This was true to a degree, but not in the way I thought I would see. I imagined that the Interpolation Search would do very well when there were large numbers of duplicates for it to maneuver around and reach “general areas” to perform Binary Searches on, but it didn’t really do very much better or worse compared to random or sequential data. The worst times came from sequential data, most likely because the search’s overhead calculations weren’t needed. Interpolation Search tended to do better with searching for identical values with lists of ten million values and better with searching for non-identical values with lists of one thousand values. This is likely because the larger lists had larger numbers of duplicates, which made for a larger target to hit as a “general area.” Beyond that, the results of having different percentages of identical values were inconclusive, much to my disappointment.

**Anecdotes**: <none>

**History**: The algorithm appears to have been published by G. H. Gonnet in “Interpolation and Interplanation Hash Searching”, University of Waterloo, 1977, as well as by Y. Perl and E. M. Reingold in “Understanding the Complexity of Interpolation Search”, *Information Processing Letters*, December 1977. Those were the earliest publications I could find. I was unable to find out who is attributed with discovering the search technique.

**Variations**: The Interpolation Search itself is a variation of the Binary Search. I was unable to find any direct variations of the Interpolation Search.

**Alternatives**: Sequential Search, Jump Search, and Binary Search would all be able to accomplish the same task of searching a sorted list. The Sequential Search starts at the beginning and checks each item, the Jump Search uses a calculated step value to perform a hybrid Sequential Search, and the Binary Search starts at the middle and recursively divides the list.

**Credits:**

* <http://en.wikipedia.org/wiki/Interpolation_search>

**Searches Compared**

As a baseline for this comparison, I decided to also implement a Sequential Search, but chose not to fully detail it in a template because of its lack of complexity. The fastest search, by far, was the Binary Search, followed by the Jump Search, Sequential Search, and finally, the Interpolation Search. As previously mentioned, I was initially quite surprised to see Interpolation Search come in last place. However, upon further inspection, it actually makes sense to me. The Interpolation Search seems to be intended for much larger datasets with large amounts of duplicate keys, rather than just one. In short, my data was not as “evenly distributed” as the Interpolation Search likes for good performance. Additionally, I believe that the process of calculating a midpoint to search from took longer than the actual searching process per iteration. An example of how the search works would be the way people use a phonebook. When you know the last name of the person you’re looking up starts with a ‘C’, you start searching from the ‘C’s. That’s kind of how the Interpolation Search is supposed to work, by first finding the general area of the key and then performing a Binary Search to find it in that area. My test data really didn’t give it much of an area to find, thus it performed poorly as a Binary Search with extra overhead.

I was also somewhat surprised to see that Jump Search outperformed Sequential search in all but the smallest of tested datasets. Jump Search is essentially just an optimized version of Sequential Search, but apparently it is quite well optimized indeed, to outperform both the Sequential and Interpolation Searches. Although I had read that Jump Search can outperform Binary Search, I did not see that happen in the tests I ran. In order for that to happen, you essentially have to search for a value toward the beginning of a very large list, which forces the Binary Search to start at the middle of the list and do very many backwards movements, whereas the Jump Search finds the value almost immediately with few or no backwards movements needed.

The other aspect of my testing, the lists with set percentages of duplicate values, did not affect execution times nearly as much as I had expected. I had thought that the Jump Search and Binary Search would do better searching for duplicated values, but the results were inconclusive. The Sequential Search did very well searching for non-duplicated values sometimes, but not all the time. The Interpolation Search was sometimes better at finding duplicated values and sometimes better at finding non-duplicated values. I really couldn’t discern a pattern for it. If I were to compare these searches again, I think I would go with all randomly generated and sorted lists and test a greater number of list sizes, since size seems to be the best way to see changes in execution times.

Based on these comparisons, I believe that I will most likely be favoring the Binary Search in practice. Although the Jump Search was neat to look at and performed pretty well, the Binary Search is easier to remember how to implement and performs better anyhow. I don’t think I would bother with an Interpolation Search unless the situation specifically called for it. And, of course, I don’t think I want to go around using Sequential Search all the time either, due to its less than stellar performance.