

# Dafny Jump Search Tutorial

*Teaching you about algorithms using software specifications*



Hassaan Malik

Winter 2018

# Table Of Contents

How to install Dafny	3
How to install on Windows	3
How to install on Mac OS and Linux	3
Running a program in Dafny	4
Assertions in Dafny	6
Loop Invariants in Dafny	7
Jump Search	8
Jump Search Code in Dafny	11
Exercise	14
Application of Jump Search	14
Conclusion	14
References	15

## How to install Dafny

Before starting to learning about Jump Search you need to install Dafny on your computer. There are two ways to run Dafny, one is to run it through the online IDE or install it on your computer. The online IDE is not reliable since the website goes down often, so it is best to install it on your computer.



## How to install on Windows

To install on a windows computer follow the steps in the following link <https://github.com/Microsoft/dafny/wiki/INSTALL> and follow those steps.

## How to install on Mac OS and Linux

To install on a Mac OS and Linux computer follow the steps in the following link <https://github.com/Microsoft/dafny/blob/master/INSTALL.md> and follow those steps. To fully execute your dafny code you will also need wine for Mac OS so download from the following link <https://wiki.winehq.org/Download>

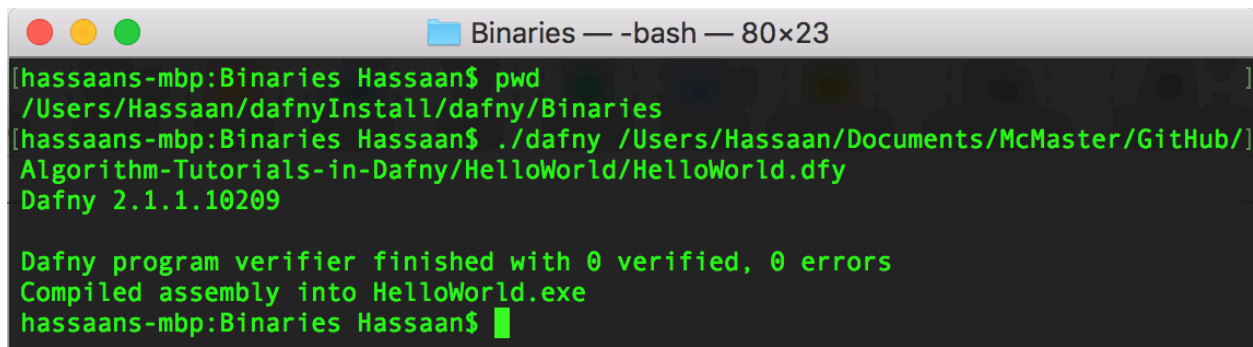
## Running a program in Dafny

Lets create a hello world program in dafny and run the code as well. The code is available at my GitHub: <https://github.com/Hassaanmalik/Algorithm-Tutorials-in-Dafny>

```
method Main() {  
    print "Hello World \n";  
}
```

The code above will just print out the string “hello world” onto your screen. Save this code in a file call HelloWorld.dfy. To run your code now you need to navigate to the directory where you installed dafny and then the binary directory.

In the screenshot bellow it shows the directory you need to be in and how to compile the hello world code.

A terminal window titled "Binaries — -bash — 80x23" showing the following commands and output:

```
[hassaans-mbp:Binaries Hassaan$ pwd  
/Users/Hassaan/dafnyInstall/dafny/Binaries  
[hassaans-mbp:Binaries Hassaan$ ./dafny /Users/Hassaan/Documents/McMaster/GitHub/  
Algorithm-Tutorials-in-Dafny/HelloWorld/HelloWorld.dfy  
Dafny 2.1.1.10209  
  
Dafny program verifier finished with 0 verified, 0 errors  
Compiled assembly into HelloWorld.exe  
hassaans-mbp:Binaries Hassaan$
```

Dafny compiles all of your code into an executable file, these executable files can only be run by the windows operating system unless you have wine installed on any other OS. If you are on a windows computer just open the HelloWorld.exe file in your terminal. If you are on a mac or linux you will need to run this through wine stable. For that double click on the wine stable icon and follow the screenshot bellow.

```
Hassaan — -bash — 80x24
#####
#                               Wine Is Not an Emulator                               #
#####

Welcome to wine-2.0.2.

In order to start a program:
.exe: wine program.exe
.msi: wine msiexec /i program.msi

If you want to configure wine:
winecfg

To get information about app compatibility:
appdb Program Name

hassaans-mbp:~ Hassaan$ wine /Users/Hassaan/Documents/McMaster/GitHub/Algorithm-
Tutorials-in-Dafny/HelloWorld/HelloWorld.exe
```

Once you press enter it may throw some warnings in wine. But at the end of the warning output it will display Hello World.

```
Hassaan — -bash — 80x24
/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleFSCompression/AppleFSCompre
ssion-96.30.2/Libraries/CompressData/CompressData.c:353: Error: Unknown compress
ion scheme encountered for file '/System/Library/CoreServices/CoreTypes.bundle/C
ontents/Resources/Exceptions.plist'
/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleFSCompression/AppleFSCompre
ssion-96.30.2/Common/ChunkCompression.cpp:50: Error: unsupported compressor 8
/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleFSCompression/AppleFSCompre
ssion-96.30.2/Libraries/CompressData/CompressData.c:353: Error: Unknown compress
ion scheme encountered for file '/System/Library/CoreServices/CoreTypes.bundle/C
ontents/Library/AppExceptions.bundle/Exceptions.plist'
/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleFSCompression/AppleFSCompre
ssion-96.30.2/Common/ChunkCompression.cpp:50: Error: unsupported compressor 8
/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleFSCompression/AppleFSCompre
ssion-96.30.2/Libraries/CompressData/CompressData.c:353: Error: Unknown compress
ion scheme encountered for file '/System/Library/CoreServices/CoreTypes.bundle/C
ontents/Library/AppExceptions.bundle/Exceptions.plist'
Hello World
hassaans-mbp:~ Hassaan$
```

## Assertions in Dafny

An assertion in programming is a statement at some point in which a predicate is presumed to hold true. In dafny, if an assertion returns false then the program will throw an error stating the results of the assertion that don't allow the program to continue. We will use assertions in our algorithms to make sure that they will always take the correct inputs and always output the correct solutions.

This is an example of how we can use assertions

```
method Main() {  
  print "Hello World \n";  
  assert 3<2;  
}
```

The code above when compiled will fail. This is because the assertion that we set is incorrect. 3 is not less than 2 so the assertion is return false and the compiler will say there is 1 error in the code. To fix this issue we have to make that statement true and this can be done in any way we want and one way we can do this is by changing the less than symbol to the greater than symbol.

```
method Main() {  
  print "Hello World \n";  
  assert 3>2;  
}
```

This will now return true and the compiler will say there are 0 errors.

## Loop Invariants in Dafny

A loop invariant is an expression that must hold true when in a loop and after every iteration through the loop as well. The reason we need loop invariants in dafny is because dafny does not know how many times to iterate through the loop. So invariants allow us to set conditions so the loop does not go through an infinite amount of times.

This is an example of how we can use invariants

```
method Main() {
  testFunction(4);
}
method testFunction(n:
int)
{
  var i := 0;
  while i < n
    invariant 0 <= i
  {
    i := i + 1;
  }
}
```

In the code above we are specifying that the function will take in an integer and run a while loop n times. We also have an invariant in there, the invariant says that the value of “i” must always be greater than or equal to 0. By setting this invariant any time we change the value of “i” we will always check that it is greater than or equal 0 even if the changed to something else later on. If we change the value of “i” to -1 the invariant will not hold true and compiler will throw an error.

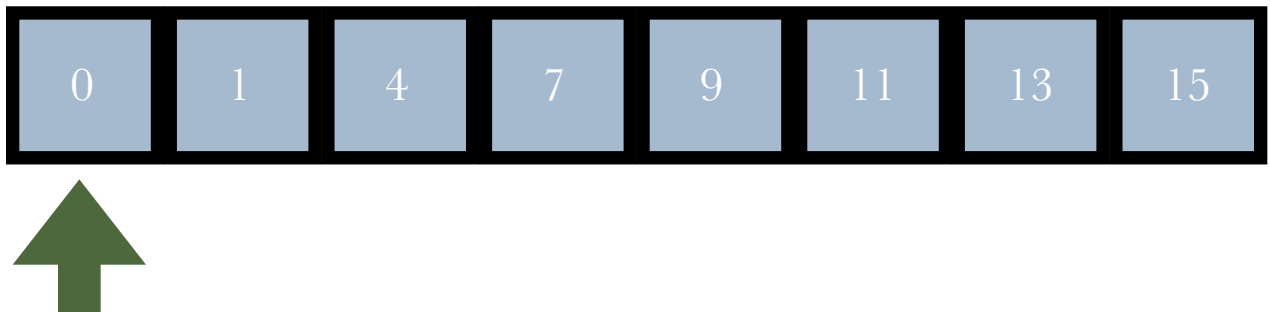
## Jump Search

Jump Search is an algorithm that searches through a sorted array by jumping to the next index by a fixed value to find a value and returns the index that the value is found at. The time complexity of jump search is  $O(\sqrt{n})$ . Lets go through an example of jump search.

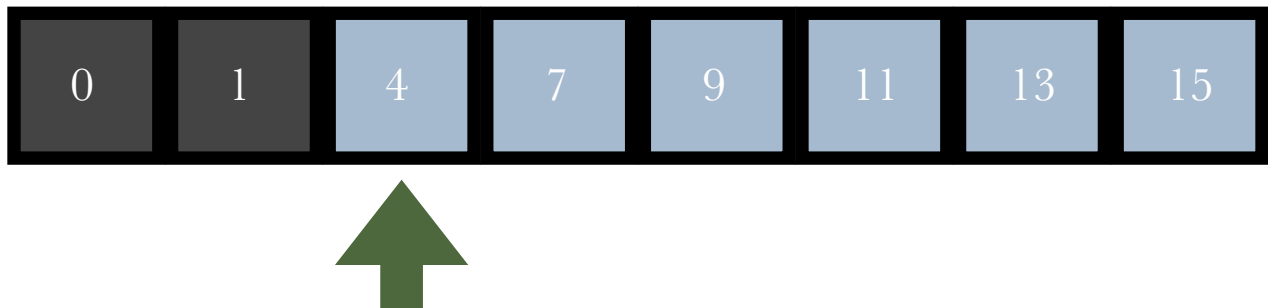


Here we have a sorted array of size 8. Jump search does not work if we do not have a sorted array so that is the first thing we need to check. Lets say we want to find the value 7 in this array.

The first thing the algorithm does is find the amount that we will be need to jump by. This done by finding the  $\sqrt{(\text{length of array})}$  and stored in an int, this allow for the value to be rounded down. In our case the  $\sqrt{8} = 2$  which means we jump 2 indexes every-time. So we first start at index 0.

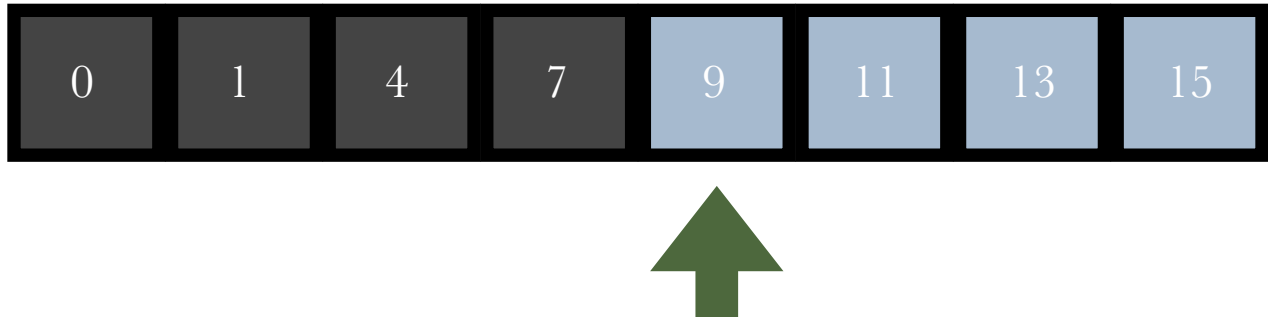


Here we compare if the value we are looking for is less than the value at the current index. In our case  $0 < 7$  so we jump ahead by our jump value to the 2 index.

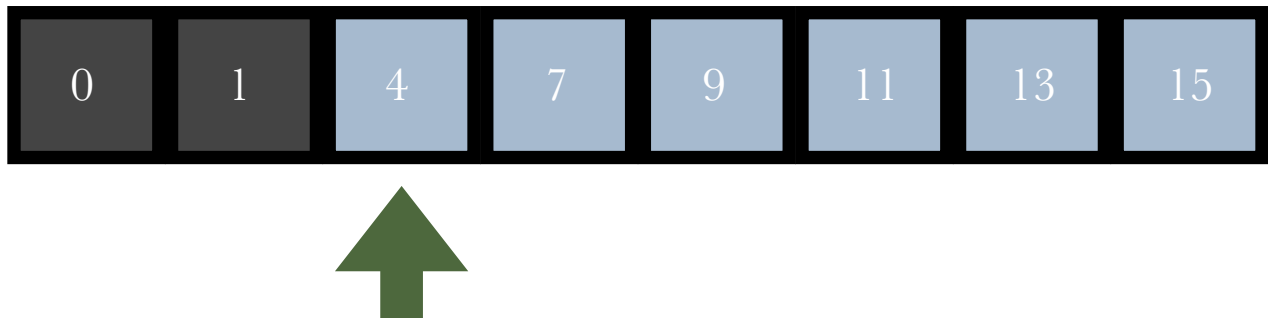




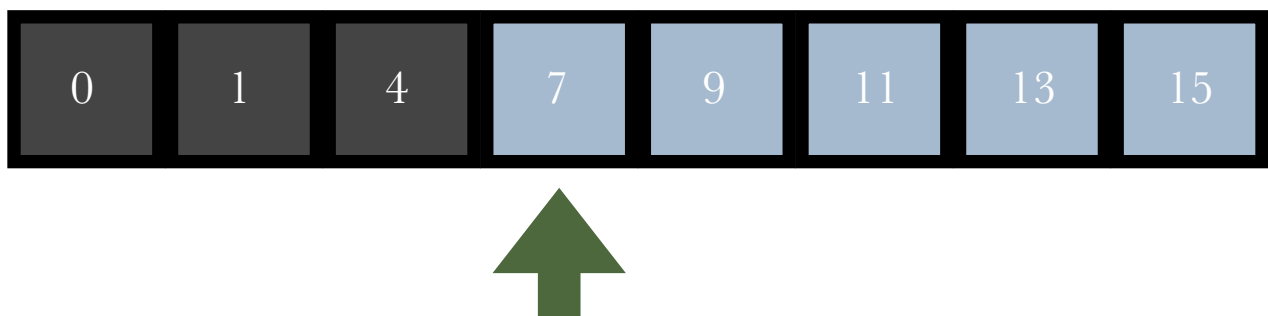
At index 2 we compare the value at index 2 being 4 and the value we are trying to find 7 and conclude that  $4 < 7$  so we continue through our loop. We now jump to the index 4



At index 4 we compare the value at index 4 being 9 and the value we are trying to find 7 and conclude that  $9 > 7$ . So we will now bring out of our initial while loop and now go back to the previous jump which is 2. Now we apply linear search to find out value. So now we will jump to index 2.

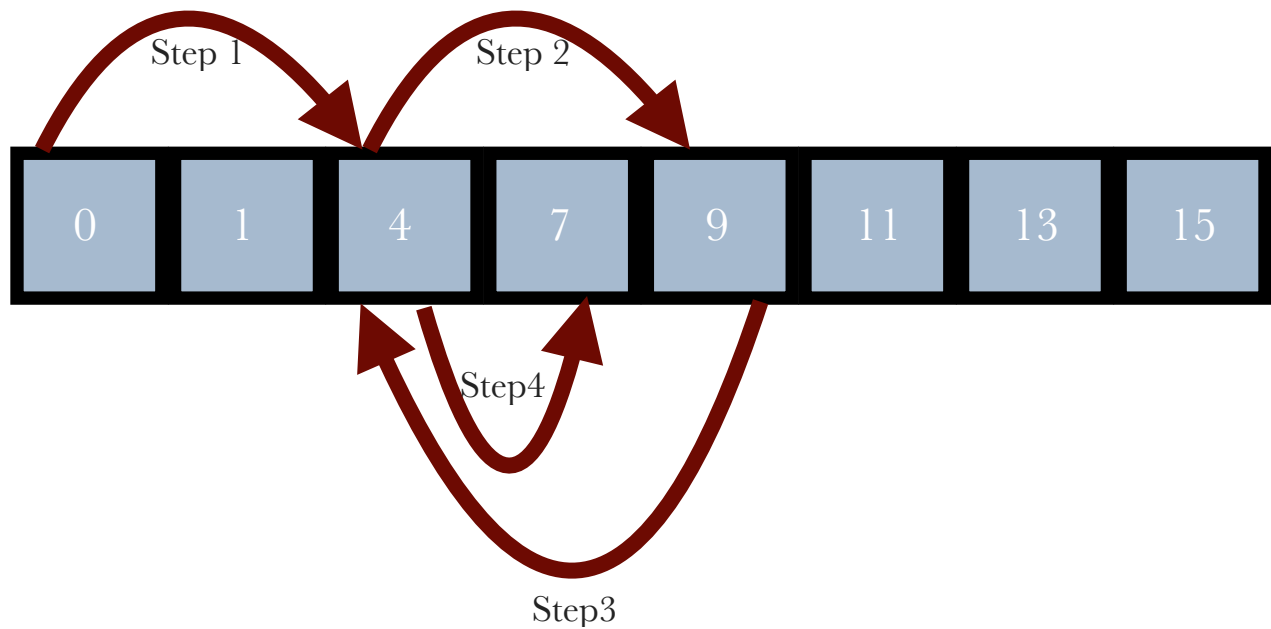


We now check if the value of the index we currently at is less than the value we want to find. In this case we compare  $4 < 7$  which is true. So now we go to the next index which is 3.



Now we break out of the linear search and confirm if the value we are at index we are at is the value we want. So in our example we have now reached the value we want to find so the algorithm will return 3

Here is an example of the steps we have taken for jump search



# Jump Search Code in Dafny

The following Jump search code is in dafny

```
method Main(){
  var a := new int[6];
  a[0], a[1], a[2], a[3], a[4], a[5] := 0,2,5,10,20,30;
  var value := 20;
  var index := jumpSearch(a, value);
  print "The value ",value, " was found at index ",index,"\n";
}

method squareRoot(n: int) returns (r : int)
  requires n >= 0;
  ensures 0 <= r*r && r*r <= n && n < (r+1)*(r+1);
{
  r := 0 ;
  while ((r+1) * (r+1) <= n)
    invariant r*r <= n ;
  {
    r := r+1 ;
  }
}

method min(a: int, b: int) returns (d: int){
  if(a<b){
    return a;
  }
  else{
    return b;
  }
}

method jumpSearch (a: array<int>, value: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == value
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value
{
  var length := a.Length;
  var jump := squareRoot(length);
  var left := 0;
  var right := 0;

  while (left < right && a[left] <=value)
  {
    right := min(length-1, left + jump);

    if (a[left] <= value && a[right] >= value){
      break;
    }
    left := left + jump;
  }

  if (left >= length || a[left] > value){
    return -1;
  }

  right := min(length-1, right);

  var i:= left;
  while(i <= right && a[i] <= value)
    invariant i <= right
  {
    if(a[i] == value){
      return i;
    }
    i := i + 1;
  }
  return -1;
}
```

The code we see above is incomplete, we are missing a few checks in there that you should see right away. But first we need to understand what requires. Requires is a precondition and must hold true for the function to continue, so in this case the array length has to be greater than or equal to 0 otherwise we could get an array out of bounds error. We also have an invariant in the code above as well, this invariant makes sure that the values are always in between the range of our array. So the low value must be less than equal to the high value which is less than equal to the length of the array. This allows the while loop to know what condition must hold true throughout the execution of the loop.

To fix the code above we need to remember the first thing that jump search needs to have in order to run and that is we need a sorted array. In the code above we do not check if the array is sorted. So lets write out a predicate function that is required when function jump search is ran.

```
predicate sorted(a: array<int>)
  reads a
{
  forall i, j :: 0 <= i < j < a.Length ==>
    a[i] <= a[j]
}
```

Lets try and understand what this function is doing. The predicate function will always return true or false if the condition we give it is true or false. It then uses a quantifier to check the indices of the array a. This quantifier is just checking that the each value i is always preceding j and is within the bounds of the array, for each index i and j, the value of a[i] is less than or equal to a[j]. This quantifier makes sure that the values will be sorted. If they are not sorted then our jump search function should not run.

To make sure that the jump search does not run unless it is sorted we need to add a condition in the first requires clause.

```
requires a.Length >= 0 && sorted(a)
```

In the first while loop for jump search we are trying find the range where the number is located. But we are not setting an invariant in that location. So lets try and create one. The first invariant we need is to state the bounds of the two values we want to use. So we know that the left and right variables will need to reside somewhere within the array. So we can state the first invariant like so:

```
invariant 0 <= left <= right <= length
```

The second invariant we need to say what is happening to the i value. In this invariant we need to say that all values that are in the array that are less than left are going to be either less than or equal to the value we want to find. This allows dafny to know what happens when a[i] is not the value we wanted.

```
invariant forall i ::  
0 <= i < left ==> a[i] <= value
```

## Exercise

We are still missing some conditions in the code to fully ensure that jump search will always output the correct value. In the code below some of the conditions are incorrect, all of them need to be there but are incorrect in some way. If you understand jump search works correctly you should be able to solve the errors. (Hint: try and compile the code, it will tell you which lines have the error) The code is too long, please take a look at the file `jumpSearch(incorrect).dfy` on GitHub.

## Application of Jump Search

Jump search does not have many practical applications other than when jumping back in a list takes more time than jumping forward. This is because in some algorithms it may be too expensive to jump back to a previous location. An example can be if you are in a scenario where scanning forward for a value is cheaper than randomly seeking for a value.

## Conclusion

After going through this tutorial, you should gain a good understanding of how Jump Search works in Dafny. Now you should be able to code jump search in any programming language as well as understand each aspect of it. This will allow you to succeed in interviews when they ask you about your knowledge of searching algorithms which will be used often in the software field.

## References

Dafny: A Language and Program Verifier for Functional Correctness. (n.d.).

Retrieved April 1, 2018, from

<https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/>

Ryder, A. (2018, February 21). Jump Search Algorithm. Retrieved April 20, 2018, from <http://iq.opengenus.org/jump-search-algorithm/#applications>

Popov, S. (2011, December 20). Algorithm of the Week: Jump Search - DZone Web Dev. Retrieved April 20, 2018, from <https://dzone.com/articles/algorithm-week-jump-search>

Jump Search. (2017, December 08). Retrieved April 20, 2018, from <https://www.geeksforgeeks.org/jump-search/>