Binary search algorithm

#include <stdio.h>

int iterativeBinarySearch(int array[], int start\_index, int end\_index, int element){

   while (start\_index <= end\_index){

      int middle = start\_index + (end\_index- start\_index )/2;

      if (array[middle] == element)

         return middle;

      if (array[middle] < element)

         start\_index = middle + 1;

      else

         end\_index = middle - 1;

   }

   return -1;

}

int main(void){

   int array[] = {1, 4, 7, 9, 16, 56, 70};

   int n = 7;

   int element = 16;

   int found\_index = iterativeBinarySearch(array, 0, n-1, element);

   if(found\_index == -1 ) {

      printf("Element not found in the array ");

   }

   else {

      printf("Element found at index : %d",found\_index);

   }

   return 0;

}

Output

Element found at index:4

#include <stdio.h>

int recursiveBinarySearch(int array[], int start\_index, int end\_index, int element){

   if (end\_index >= start\_index){

      int middle = start\_index + (end\_index - start\_index )/2;

      if (array[middle] == element)

         return middle;

      if (array[middle] > element)

         return recursiveBinarySearch(array, start\_index, middle-1, element);

      return recursiveBinarySearch(array, middle+1, end\_index, element);

   }

   return -1;

}

int main(void){

   int array[] = {1, 4, 7, 9, 16, 56, 70};

   int n = 7;

   int element = 9;

   int found\_index = recursiveBinarySearch(array, 0, n-1, element);

   if(found\_index == -1 ) {

      printf("Element not found in the array ");

   }

   else {

      printf("Element found at index : %d",found\_index);

   }

   return 0;

}

### Output

Element found at index : 3

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

};

int main()

{

    struct Node\* head = NULL;

    struct Node\* second = NULL;

    struct Node\* third = NULL;

    head = (struct Node\*)malloc(sizeof(struct Node));

    second = (struct Node\*)malloc(sizeof(struct Node));

    third = (struct Node\*)malloc(sizeof(struct Node));

 head->data = 1;

    head->next = second;

  second->data = 2;

    second->next = third;

third->data = 3;

    third->next = NULL;

return 0;

}

**Calculating Big O**

To calculate Big O, you can go through each line of code and establish whether it’s O(1), O(n) etc and then return your calculation at the end. For example it may be O(4 + 5n) where the 4 represents four instances of O(1) and 5n represents five instances of O(n).

There’s an easier way to calculate this however, and that’s done with the following four rules:

1. Assume the worst
2. Remove constants
3. Use different terms for inputs
4. Drop any non dominants

Taking each of these in turn:

1. **Assume the worst**

When calculating Big O, you always think about the worst case. For example, if you were to loop over an array and look for an item, it could be the first item or it could be the last. As we aren’t certain then we must assume O(n) in this instance.

2. **Remove constants**

This is a little bit trickier but bear with me. Consider this code:

We have a function which does several things. Some are O(1) such as line 3, whereas others are O(n) such as line 9.

We could express the Big O notation of this as O(1 + n/2 +100) but it’s really too specific. We can remove our O(1) operations because, as a rule of thumb, they are likely to be insignificant compared to our O(n) operations. If we are providing a million items in our input array, then an extra 100 operations from O(1) aren’t our main concern.

So then we have O(n / 2) — as n gets larger and larger, dividing it by two has a diminishing effect. So ultimately our Big O notation for this function becomes O(n).

1. **Different terms for inputs**

If we were to loop twice of the same array then our Big O is technically O(2n) but let’s drop the constants so we are left with O(n). But now we need to think about different terms for inputs. What does that mean?

If we look at above code, we can see we now have two inputs. One could be one item long, the other could contain a million items. So our function is no longer O(n), it is O(a + b). The *n* we use in our notation is arbitrary, so we need to reflect both inputs in our notation.

In this instance our loops aren’t nested. If they were then our code would be O(n²) which would be a potential target for refactoring. Typically if there are nested loops then you are multiplying rather than adding the variables.

4. **Drop non-dominant terms**

Take a look at this code:

So we have a single loop at the top which is O(n) and then a nested loop which is O(n²). So our total is O(n + n²). But as we saw in rule #2, we want to drop constants. So which of the two terms is more important to us?

In this case we drop O(n) and retain O(n²) as this is more important. It is the dominant term as it has a much heavier impact on performance.

**Conclusion**

So what’s the point?

If we know we will only deal with small inputs then Big O doesn’t really matter too much. But when when writing you code, you should consider its scalability.

By paying attention to Big O you have an eye to the future and you are more likely to write code which can be scaled effectively. You can start looking at the cost of your code and make informed choices in how you write it.

## 1. O(1)

void printFirstElementOfArray(int arr[])

{

printf("First element of array = %d",arr[0]);

}

This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

## 2. O(n)

void printAllElementOfArray(int arr[], int size)

{

for (int i = 0; i < size; i++)

{

printf("%d\n", arr[i]);

}

}

This function runs in O(n) time (or "linear time"), where n is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

## 3. O(n2)

void printAllPossibleOrderedPairs(int arr[], int size)

{

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

printf("%d = %d\n", arr[i], arr[j]);

}

}

}

Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n2 total prints. Thus this function runs in O(n2) time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

## 4. O(2n)

int fibonacci(int num)

{

if (num <= 1) return num;

return fibonacci(num - 2) + fibonacci(num - 1);

}

An example of an O(2n) function is the recursive calculation of Fibonacci numbers. O(2n) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2n) function is exponential - starting off very shallow, then rising meteorically.

## 5. Drop the constants

When you're calculating the big O complexity of something, you just throw out the constants. Like:

void printAllItemsTwice(int arr[], int size)

{

for (int i = 0; i < size; i++)

{

printf("%d\n", arr[i]);

}

for (int i = 0; i < size; i++)

{

printf("%d\n", arr[i]);

}

}

This is O(2n), which we just call O(n).

void printFirstItemThenFirstHalfThenSayHi100Times(int arr[], int size)

{

printf("First element of array = %d\n",arr[0]);

for (int i = 0; i < size/2; i++)

{

printf("%d\n", arr[i]);

}

for (int i = 0; i < 100; i++)

{

printf("Hi\n");

}

}

This is O(1 + n/2 + 100), which we just call O(n).

Why can we get away with this? Remember, for big O notation we're looking at what happens as n gets arbitrarily large. As n gets really big, adding 100 or dividing by 2 has a decreasingly significant effect.

## 6. Drop the less significant terms

void printAllNumbersThenAllPairSums(int arr[], int size)

{

for (int i = 0; i < size; i++)

{

printf("%d\n", arr[i]);

}

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

printf("%d\n", arr[i] + arr[j]);

}

}

}

Here our runtime is O(n + n2), which we just call O(n2).

// C Program for Infix to Postfix conversion

// Array based stack implimentation

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#define MAX 20

char stk[20];

int top = -1;

int isEmpty(){

return top == -1;

}

int isFull(){

return top == MAX - 1;

}

char peek(){

return stk[top];

}

char pop(){

if(isEmpty())

return -1;

char ch = stk[top];

top--;

return(ch);

}

void push(char oper){

if(isFull())

printf("Stack Full!!!!");

else{

top++;

stk[top] = oper;

}

}

// A utility function to check if the given character is operand

int checkIfOperand(char ch)

{

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// Fucntion to compare precedence

// If we return larger value means higher precedence

int precedence(char ch)

{

switch (ch)

{

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The driver function for infix to postfix conversion

int covertInfixToPostfix(char\* expression)

{

int i, j;

for (i = 0, j = -1; expression[i]; ++i)

{

// Here we are checking is the character we scanned is operand or not

// and this adding to to output

if (checkIfOperand(expression[i]))

expression[++j] = expression[i];

// Here, if we scan character ‘(‘, we need push it to the stack.

else if (expression[i] == '(')

push(expression[i]);

// Here, if we scan character is an ‘)’, we need to pop and print from the stack

// do this until an ‘(‘ is encountered in the stack.

else if (expression[i] == ')')

{

while (!isEmpty() && peek() != '(')

expression[++j] = pop();

if (!isEmpty() && peek() != '(')

return -1; // invalid expression

else

pop();

}

else // if an opertor

{

while (!isEmpty() && precedence(expression[i]) <= precedence(peek()))

expression[++j] = pop();

push(expression[i]);

}

}

// Once all inital expression characters are traversed

// adding all left elements from stack to exp

while (!isEmpty())

expression[++j] = pop();

expression[++j] = '\0';

printf( "%s", expression);

}

int main()

{

char expression[] = "((a+(b\*c))-d)";

covertInfixToPostfix(expression);

return 0;

}

#### **Output**

abc\*+d-



