# CptS 223 Homework #1

Please complete the homework problems on the following page. Note that this is an individual assignment and all work must be your own. Be sure to show your work when appropriate. This assignment must be turned in by the due date via Blackboard in *PDF* format. You may use any editor you like (or even print it out, *legibly* write in answers, and scan it in), but convert it to *PDF* for submission. I have provided MS Word (doc) and LibreOffice (ODF) versions for your platform of choice.

**1. [5] Order the following set of functions by their growth rate:**

1. N
2. √N
3. N^1.5
4. N^2
5. N log N
6. N log(log(N))
7. N log^2 N
8. 2/N
9. 2^N
10. 2^(N/2)
11. 37
12. N^2 log(N)
13. N^4

2/N < 37 < sqrt(N) < N < N log(log(N)) < N log N < Nlog^2 N < N^1.5 < N^2

< N^2 log(N) < N^4 < 2^(N/2) < 2^N

**2. [5] A program takes 35 seconds for input size 20 (i.e., n=20). Ignoring the effect of constants, approximately how much time can the same program be expected to take if the input size is increased to 100 given the following run-time complexities?**

1. O(N) linear, 100 = 20 \* 5 so T = T\_orig \* 5 = 175s
2. O(N + log N) same as #1 except log(100 = 2, so 177s
3. O(N^3) cubed, so 5x the input size = 5^3 = 125x the time

time = 35s \* 125 = 4375s = 1.2 hours

1. O(2^N)[[1]](#footnote-2) equivalent to 2^(5N) = (2^N)^5 meaning the original time raised to the fifth = 35s^5 = about 5.25 \* 10^7 s = 1.67 years

**4. [8] Given the following two functions:**

|  |  |
| --- | --- |
| int g(int n)  {  if(n <= 0)  {  return 0;  }  return 1 + g(n - 1);  } | int f(int n)  {  int sum = 0;  for(int i = 0; i < n; i++)  {  sum += 1;  }  return sum;  } |

1. [2] State the runtime complexity of both f() and g()

g: O(N) (N recursions)

f: O(N) (N iterations of loop)

1. [2] State the memory (space) complexity for both f() and g()

g: O(N) (N stack frames)

f: O(1) (one variable being constantly updated)

1. [4] Write another function called "int h(int n)" that does the same thing, but is significantly faster.

int h(int n) {

if(n <= 0) {

return 0;

}

return n;

}

**5. [5] State g(n)'s runtime complexity:**

|  |
| --- |
| int f(int n){  if(n <= 1){  return 1;  }  return 1 + f(n/2);  }  int g(int n){  for(int i = 1; i < n; i \*= 2){  f(i);  }  } |

g iterates on the order of log(N) times, and f recurs on the order of log(N) times

total runtime complexity for g: O(log^2(N))

**7. [5] What is the runtime complexity of Adam's famous string splitter code? Hint: Make sure to look into the source code for string.find() in the C++ std library. I’ve included that code (downloaded from GNU).**

|  |
| --- |
| static vector<string> split(string text, string delimiter)  {  vector<string> pieces;  int location = text.find(delimiter);  int start = 0;  //while we find something interesting  while (location != string::npos){    //build substring  string piece = text.substr(start, location - start);  pieces.push\_back(piece);  start = location + 1;  //find again  location = text.find(delimiter, start);  }  string piece = text.substr(start, location - start);  pieces.push\_back(piece);  return pieces;  } |

**GCC/G++ source downloaded from:** [**http://mirrors.concertpass.com/gcc/releases/gcc-6.3.0/**](http://mirrors.concertpass.com/gcc/releases/gcc-6.3.0/) **Source file: gcc-6.3.0/libstdc++-v3/include/ext/vstring.tcc**

|  |
| --- |
| template<typename \_CharT, typename \_Traits, typename \_Alloc,  template <typename, typename, typename> class \_Base>  typename \_\_versa\_string<\_CharT, \_Traits, \_Alloc, \_Base>::size\_type  \_\_versa\_string<\_CharT, \_Traits, \_Alloc, \_Base>::  find(const \_CharT\* \_\_s, size\_type \_\_pos, size\_type \_\_n) const  {  \_\_glibcxx\_requires\_string\_len(\_\_s, \_\_n);  const size\_type \_\_size = this->size();  const \_CharT\* \_\_data = this->\_M\_data();  if (\_\_n == 0)  return \_\_pos <= \_\_size ? \_\_pos : npos;  if (\_\_n <= \_\_size)  {  for (; \_\_pos <= \_\_size - \_\_n; ++\_\_pos)  if (traits\_type::eq(\_\_data[\_\_pos], \_\_s[0])  && traits\_type::compare(\_\_data + \_\_pos + 1,  \_\_s + 1, \_\_n - 1) == 0)  return \_\_pos;  }  return npos;  } |

We assume a string of size N, with M <= N delims.

We conclude that the find function will iterate through the entire string once throughout the course of the string splitter function, regardless of whether delimiters are found and how many times the while loop executes. Thus, we can conclude the "find" portions of the program will eventually sum up to an O(N) operation.

The while loop inside the split function will trigger once each delim, so on the order of M times. Inside the while we find:

-a substring grabber, with approximately N/M operations, since each substring is approximately N/M characters in length.

-a function to append the new substring to the string vector, with constant operations (assigning pointers)

-an increment, 1 operations

-the find function, which we already analyzed.

Thus the while loop contains approximately M \* (N/M + const) = N + cM operations, not including the find function which is already covered.  
Thus the total number of operations is N + cM + N, or

O(M + N) in the average case, assuming the number of delims is far fewer than the number of chars in the source string.

O(N) in the worst case, assuming every char is a delim.

**6. [10] (adapted from the 2012 ICPC programming competition) Write an algorithm to solve the following problem and specify its runtime complexity using the most relevant terms:**

Given a nonnegative integer, what is the smallest value, k, such that

*n, 2n, 3n, …, kn*

contains all 10 decimal numbers (0 through 9) at least once? For example, given an input of "1", our sequence would be:

and thus k would be 10. Other examples:

|  |  |
| --- | --- |
| Integer Value | K value |
| 10 | 9 |
| 123456789 | 3 |
| 3141592 | 5 |

Have k start at 0. Have the "current number" start at 0, and n is the integer that was passed in. set up a 10 element list containing the elements 0 - 9.

do:

increment k

add n to "current number"

for each entry in list:

if that digit is found in "current" number, remove it from list.

while list is not empty.

return k

consider that any integer n we pick will always have a location with a 0 somewhere to its left. For numbers without any internal zeros, place a 0 next to the leading digit for this thought experiment.

It should be obvious that it will take less than ten iterations of addition for the values to accumulate enough such that 0 becomes a 1. Ten more iterations and that 1 will become a 2, etc... without skipping. Thus we can conclude that the do loop will execute <= 100 times.

Incrementing k and adding n to current number are both O(1) operations in our analysis.

The inner for loop will execute a total of ten or less times for obvious reasons.

However, the searching will need to go through each digit of "current number", which can be up to 100x the original n placed in. In other words, the current number can have up to d + 2 digits, where d is the number of digits in the original n.

We have thus shown that this algorithm is less than 1000(d + 2) operations, or O(d), where d is the number of digits in n.

**7. [18] Provide the algorithmic efficiency for the following tasks. Justify your answer, often with a small piece of pseudocode or a drawing to help with your analysis.**

1. [3] Determining whether a provided number is odd or even

O(1)

bool isOdd(n) { return n & 1; }

bitwise AND is a O(1) operation

1. [3] Determining whether or not a number exists in a list

O(n)

check first element, if match, return true

while(!eoList && !match) {

check next element, if match, return true

}

return false

needs to run through list once, or on average on the order of running through the list once, so O(n)

1. [3] Finding the smallest number in a list

O(n)

set "current smallest" to first number in list

while(!eoList) {

check next number, if it's smaller, update current smallest

}

return current smallest

needs to run through list once, so O(n)

1. [3] Determining whether or not two **unsorted** lists of the same length contain all of the same values (assume no duplicate values)

Make copies of list 1 and list 2.

For each element in list 1,

search through elements in list 2 to see if there is a match. If there is, remove the element from both lists and continue.

If there is no match, break and return false.

If we complete the for loop without returning false, return true.

O(N^2) ... for each of the N elements in list 1, performing an O(N) operation to find matching value in list 2, so total computations are

N \* N. Copying a list is an O(N) operation and inconsequential compared to the O(N^2) matching operation.

1. [3] Determining whether or not two **sorted** lists contain all of the same values (assume no duplicate values)

Let i point to the first element in list 1, and j point to the first element in list 2.

while list1[i] == list2[j] AND we're not at the end of the list,

increment i and j.

When we break out of loop, check if we were at the end of the list. If not, return false. If so, check the last two elements, list1[i = N] and list2[j = N].

If they are the same, return true.

If not, return false.

O(N) - n comparisons, running through both lists once.

1. [3] Determining whether a number is in a BST

start at the root.

1. check if number equal to current node. If so, return true and stop. If not...

2. if number is greater than current node...

a. If the right child doesn't exist, return false.

b. Else, move to right child and go back to step 1.

3. if number is less than current node...

a. If left child doesn't exist, return false.

b. Else, move to left child and go back to step 1.

O(log(N)) where N is the number of elements in the BST... if BST is acceptably balanced the height of the tree should be on the order of log(N) and thus the maximum number of traversals before an answer is obtained should be on the order of log(N).

**8. [6] Fill in what these Linux commands do.**

**For example:**

ls "list" - lists files and directories

cp "copy" - copies files/directories

rm "remove" - removes a file or a directory

mkdir "make directory"

ssh "secure shell" allows for connecting to remote machines

g++ the GNU c++ compiler, for... compiling cpp files.

scp "secure copy" transfer files to/from remote system

**9. [4] How do these variables get set and what do they get set with?**

int main(int argc, char\* argv[]) {

return(0);

}

argc is set automatically at runtime and is set to an int, n + 1, where n is the number of arguments supplied in the command line.

argv is an array of size n + 1, with the program name being argv[0] and the command line parameters making up the rest of the elements.

so in the shell, if foo is the program that is to be run,

> foo moo boo goo

will place moo as argv[1], boo as argv[2], and goo as argv[3]

1. You might need an online calculator with arbitrarily large numbers for this one. Scientific notation and 8 significant figures is just fine. [↑](#footnote-ref-2)