CMPT 125: Introduction to Computing Science and Programming II

Spring 2023

Week 5: Big-O notation, searching algorithms

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Fact of the day

The first hard drive made was in 1979 and could only hold 5MB of data

Recap from Last Lecture

- Measuring performance of algorithms
  - #1 Criteria: It has to be correct
  - #2 Criteria: easy to code & debug, needs less memory, takes less time, …etc.
- Big-O notation
  - A way to compare estimate of critical operations of algorithm to reference functions
  - Typically in terms of the input size

Review from Last Lecture (I)

- What happens to the time complexity if the for loop in Line 5 (and thus Line 6) in Example 3 of the Calculating Time Complexity slides is indented to be part of the for-loop in Line 3?

```
1       count = 0
2       for i in 1…n:
3         for j in 1…n:
4           count = count + 10
5           for k in 1…n:
6             count += 2
```

Recall:
- Count the number of times a critical operation is executed
- Disregard “constants”
- Disregard “lower exponent terms”

- Critical operations?
  - addition/assignments (lines 1,4&6)
- What are the constants?
  - Line 1
- How many addition/assignments that matters are executed?
  - $((n+1)* n) * n = n^3 + n^2$ (line 6 repeated n times in k loop, together with line 4 they repeat n times in j loop, all repeated n times in i loop)
- What is the time complexity?
  - $O(n^3)$

Review from Last Lecture (2)

- This is the pseudo-code for the **Selection Sort**. What is its time complexity?

```
1    for i = 0 to N-1
2      minIndex ← i
3      for j = i+1 to N-1
4        if A[j] < A[minIndex]
5          minIndex ← j
6        end if
7      end for
8      Swap A[i], A[minIndex]
9    end for
```

- Critical operations?
  - assignment, comparison, swap (lines 2, 4, 5, 8)
- What are the constants?
  - No constants
- How many assignments/comparisons/swap that matters are executed?
  - $[1 + (N-1)*2 + C] + [1 + (N-2)*2 + C] + [1 + (N-3)*2 + C] + \ldots + [1 + 0*2 + C]$
    $= 1*N + [(N-1)+(N-2)+(N-3)+\ldots+0] + C*N$
    $= (N+1)*N/2 + CN$
- What is the time complexity?
  - $O(N^2)$

A More Sophisticated Example – Checking Duplicates

outside loop: 3 ops

outer loop: 4*n ops (condition, assignment, increment)

```
//main idea: each round expand the search space by 1, check left
bool check_duplicates(const int* arr, int n) {
  int i=0, j;
  bool found = false;
  while (i<n && !found) {
    j = 0;
    while (j<i && !found) {
      if (arr[i] == arr[j]) {
        found = true;
      }
      j++;
    }
    i++;
  }
  return found;
}
```

inner loop: 4*i ops (condition, increment)

Outside loop: 3
Outer loop: 4n
Inner loop = 4*1 + 4*2 + 4*3+ … + 4*(n-1) = $2n^2$ -2n

Total: ≅ $2n^2 + 2n + 3 = O(n^2)$

Today

- Big-O notation (cont’d)
  - other mathematical variations
  - ways to derive time complexity on recursive algorithms
- Complexity analysis of sorting algorithms
  - Selection Sort
  - Mergesort
- Complexity analysis of searching algorithms
  - Linear Search
  - Binary Search, under one condition

Other Variations of the Big-O Mathematical Definition

- Let f(N) and g(N) are two functions (mathematical functions, not programming functions) with N being integers

The original

f = O(g) if there exists a large enough constant C such that f(N) <= C*g(N) for all sufficiently large N

- This is equivalent to say (this allows us to find C easier in some cases):

f = O(g) if there exists a large enough constant C such that f(N)/g(N) <= C for all sufficiently large N

- Or:

f = O(g) if $\lim_{N \to \infty} \left(\frac{f(N)}{g(N)} \right) < \boldsymbol{infinity}$

Some Simple Rules for Deriving Big-O Mathematically

- For fixed values a & b where $0 < a < b$ (e.g., $a=2$, $b=4$)
  - $N^a = O(N^b)$ (but the reverse is NOT true!)
- $\log(N)$ is smaller than any positive power of N, i.e., $\log(N) = O(N^a)$, even when a is < 1
- If you see $\log_2(N)$, or with any base, you can just write $\log(N)$, because $\log_2(N) = \log_m(N) * \log_2(m)$
- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$
  - Further more, $f + g = O(h)$, because $f + g \le 2\max(f, g) = O(\max(f, g))$
- If $f_1 = O(g)$ and $f_2 = O(g)$, then $f_1 + f_2 = O(g)$

One More Example

- Suppose $f(N) = 10*2^N + 4N^4 + 3$, prove that $f = O(2^N)$
  Use the fact that $N^4 < 2^N$ for all $N>20$

- Prove:
  $f(N) = 10*2^N + 4N^4 + 3 < 10*2^N + 4*2^N + 3*2^N$ (for $N>20$) $= 17*2^N$
  That is, there exists a large enough constant, $C=17$, that results in $f(N) <= C* 2^N$ for all sufficiently large $N>20$.
  Therefore $f = O(2^N)$

Why So Much Math??

- Computer Science and Mathematics have a lot of overlaps
  - You use math to calculate complexity of algorithms so you can tell if which one is good at what situation
  - When you have an algorithm, you can often use math/logic to prove its correctness (e.g., induction)
  - In many cases what your computer is doing is just calculations, for example:
    - Rotating an image (transforming pixel locations)
    - Encrypting/Decrypting data (a series of math operations)
    - Machine learning (extract patterns and calculate the most likely match)
- If you understand math, you can understand CS better

Big-O Notations for Recursive Algorithms

- We can use **algebraic mechanisms** to find the time complexities of recursive algorithms
- For example, the time complexity of our recursive Selection Sort can be expressed as:

$$T(N) = N + C + T(N-1)$$

- where N is the search space (number of elements) and C is a constant including all other operations
- The recursive call works on 1 less element, hence N-1

```c
void selectionSortRecursive(int array[], unsigned size, unsigned int start) {
    //only need to sort when there is more than 1 element (start is at most size-1)
    if (size-start > 1) {
        unsigned int minIndex = start;
        for (int i=start+1; i<size; i++) {
            if (array[i] < array[minIndex]) {
                minIndex = i;
            }
        }

        //swap the smallest value to the front of the search space
        int temp = array[minIndex];
        array[minIndex] = array[start];
        array[start] = temp;
        printf("swapping %d with %d\n", array[minIndex], array[start]);

        //repeat starting from the next element
        selectionSortRecursive(array, size, start+1);
    }
}
```

Big-O Notations for Recursive Algorithms (Cont'd)

$$
\begin{aligned}
T(N) &= N + C + T(N\text{-}1) \\
&= N + C + [(N\text{-}1) + C + T(N\text{-}2)] \\
&= N + C + (N\text{-}1) + C + [(N\text{-}2) + C + T(N\text{-}3)] \\
&= \ldots \\
&= N + C + (N\text{-}1) + C + (N\text{-}2) + C + \ldots + [(N\text{-}(N\text{-}2)) + C + T(N\text{-}(N\text{-}1))] \\
&= N + (N\text{-}2) + (N\text{-}3) + \ldots + 2 + (N\text{-}1)*C + T(1) \\
&= N(N+1)/2 - 1 + CN - C + T(1) \\
&= O(N^2)
\end{aligned}
$$

Drop the multiplicative constant

Drop the lower order terms

Mergesort Time Complexity

- Recall Mergesort is a recursive algorithm

```
mergeSort(array)
    if (array has 2 or more elements):
        sortedLeft = mergeSort(left half of array)
        sortedRight = mergeSort(right half of array)
        result = merge(sortedLeft , sortedRight)
    else:
        result = array  # the array is already sorted, 1 element only
    return result
```

- The merge part takes O(N) of time because it simply scans the 2 smaller sorted arrays and fill up the larger array; and when N is 1, it is the base case where nothing needs to be sorted, thus takes O(1) of time

Mergesort Time Complexity (Cont'd)

- The time complexity of Mergesort can be expressed as

$T(N) = 2T(N/2) + N + C$ (where N is the number of elements in the current call of the function, C is a constant)

$= 2[2T(N/2^2) + N/2 + C] + N + C$

$= 2^2T(N/2^2) + N + 2C + N + C$

$= 2^2[2T(N/2^3) + N/2^2 + C] + N + 2C + N + C$

$= 2^3T(N/2^3) + N + 2^2C + N + 2C + N + C$

$= \ldots$

$= 2^kT(N/2^k) + N + 2^{k-1}C + \ldots + N + 2^2C + N + 2C + N + C$

$= 2^kT(N/2^k) + kN + (2^{k-1} + \ldots + 1)*C$

Mergesort Time Complexity (Cont'd)

- The time complexity of Mergesort can be expressed as

$T(N) = 2T(N/2) + N + C$ (where N is the number of elements in the current call of the function, C is a constant)

$= 2[2T(N/2^2) + N/2 + C] + N + C$

$= \ldots$

$= 2^kT(N/2^k) + N + 2^{k-1}C + \ldots + N + 2^2C + N + 2C + N + C$

$= 2^kT(N/2^k) + kN + (2^{k-1} + \ldots + 1)*C$

Let $N = 2^k \rightarrow \log_2 N = k$, then

$T(N) = NT(N/N) + N\log_2 N + (2^k - 1)C$

$= NT(1) + N\log_2 N + (N - 1)C$

$= N * O(1) + N\log_2 N + (N - 1)C = O(N\log_2 N)$ (or just $O(N\log N)$)

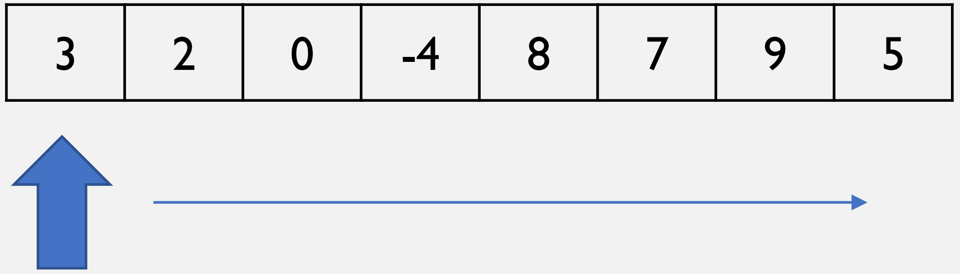
Searching Data in A Data Set

- It is a very common thing to do in computing science when handling a set of data
  - check if a record exists
  - look for duplicates
  - information lookup
  - …etc.
- Data are typically stored in a data structure, which may or may not have some special ordering
  - sorted from small to large or large to small based on a certain attribute
  - store based on time of insert
  - …etc.
- For now we assume data are stored in arrays, and we call the look up value "key"

Linear Search (Boolean)

Main idea: go through the array, upon finding a match with the key, return **true**. If reaches the end of the array, it means there is no match, return **false**.

```
for i = 0 to N-1
  if array[i] == key
    return true
  end if
end for
return false
```



Linear Search (Index)

Main idea: go through the array, upon finding a match with the key, return **index**. If reaches the end of the array, it means there is no match, return **-1**.

```
for i = 0 to N-1
  if array[i] == key
    return i
  end if
end for
return -1
```

Linear Search Time Complexity

- As mentioned in the previous lecture, some algorithms have different time complexities depending on the case
- Best case: every time the **first** array item matches with the key → O(1) (it is equivalent to "return array[0]")
- Worst case: everything the **last** array item matches with the key, or **no item** matches with the key → O(N)

```
for i = 0 to N-1
  if array[i] == key
    return i
  end if
end for
return -1
```

- We typically choose to report the worst case for a more conservative analysis (after all the best case doesn't happen that often, it's more likely that the match is randomly located in the array, so roughly N/2 comparisons)

Linear Search Is Slow

- The Linear Search algorithm is considered as a slow searching algorithm because it essentially is looking at each data once (it's considered the lower bound because one can't do worse than that)

- What if instead of a disorganized data set you have **an array where the items are sorted**?
  - Intuitively it'll be faster because you have a rough idea on which part of the array to look, e.g., if the key is small, the match can't be at the back in an ascendingly sorted array
    - The question is how much "back" you can skip looking for a match

Introducing Binary Search

- Suppose we have a pre-sorted (smallest to largest) array and we want to search with a key = 17

-2	-1	8	14	17	23	29	37	74	75	81	87	95

- Half of the array can be skipped by:
  - Check the middle item (if there are even number of items use the smaller one): 29
  - If the key is smaller than 29, then the item has to be in the left part (let's call this part the "active part")
    - otherwise the key is larger than 29, then the item has to be in the right part; or is a match, then we stop

Introducing Binary Search (Cont'd)

Main idea: Start with the whole array as the active part, look at the middle of the active part, if it is a match, done. Otherwise, decide which half is the new active part by comparing the middle item with the key and continue until there is no more items in the active part (not found).

-2	-1	8	14	17	23	29	37	74	75	81	87	95
-2	-1	8	14	17	23	29	37	74	75	81	87	95
-2	-1	8	14	17	23	29	37	74	75	81	87	95

Pseudo-code for Binary Search

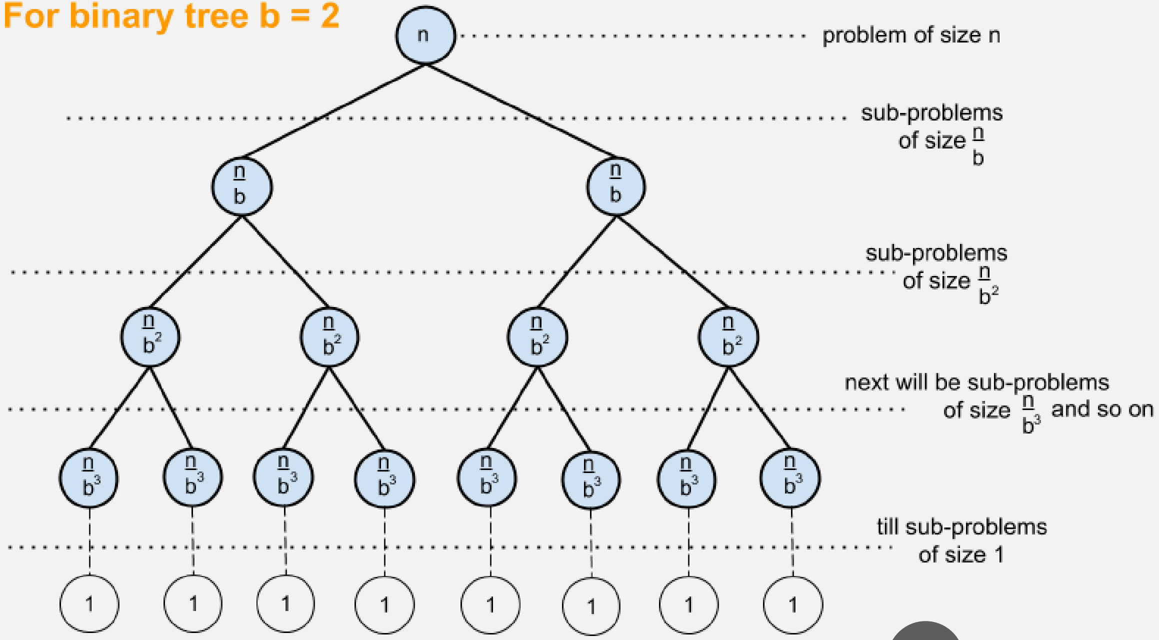
Assume array is ascendingly sorted. Let L and R the left/right boundary indexes of the active part (start as L = 0 and R = N-1)

```
while L <= R
  mid ← (L + R)/2
  if array[mid] == key
    return mid
  else if array[mid] < key
    L ← mid + 1
  else
    R ← mid - 1
  end if
end while
return -1
```

make the right half as the active part

make the left half as the active part

For binary tree b = 2



Binary Search Complexity

- In each iteration, the size of the active part is reduced by roughly half, and the loop ends when there is no more items to compare (unless the key has a match somewhere)
  - There is only a constant amount of critical operations in each iteration (say, C)
- Hence the total number of critical operations can be expressed as: C * number of iterations
  - This is equivalent to ask how many times can we divide the input size N by 2 until it becomes 1 (or 0)

$$\frac{N}{2^x} = 1$$

$$N = 2^x$$

$$x = \log_2 N$$

- Therefore, time complexity of Binary Search = O(logN)

Binary Search Only Works on Sorted Arrays

- Generally speaking, a comparison-based sorting algorithm can only be as fast as O(nlogn). So technically, to perform a binary search on an unsorted array of items the time complexity is O(nlogn) + O(logn) = O(nlogn)
- So why can't we just use the O(n) Linear Search Algorithm?

- Let's say you need to search 10000 times within 256 items
  - then you'll have 1 O(nlogn) sort + 10000 O(logn) searches ≈ 256*8 + 10000*8 ≈ 80k operations
  - instead of 10000 O(n) searches ≈ 10000*256 ≈ 2560k operations
- The efficiency comes from sorting a more stable set of data once and do an efficient search many times

Extra!

- Remember the fib(n) recursive function?
- Time complexity can be expressed by: $T(n) = T(n-1) + T(n-2) + C$

$T(n) = [T(n-2) + T(n-3) + C] + [T(n-3) + T(n-4) + C] + C$ ← this is very tedious ☹

- Let's consider T(n-1) and T(n-2), we can assume that T(n-1) has more operations to do than T(n-2), so

$T(n) > 2T(n-2) + C = 2[2T(n-4) + C] + C = 2^2[2T(n-6) + C] + 2C + C = \ldots = 2^kT(n-2k) + (2^k-1)C = O(2^{n/2})$

- On the other hand, we can also write this:

$T(n) < 2T(n-1) + C = 2[2T(n-1) + C] + C = 2^2[2T(n-2) + C] + 2C + C = \ldots = 2^kT(n-k+1) + (2^k-1)C = O(2^n)$

- We can conclude that $O(2^{n/2}) < T(n) < O(2^n)$, and thus T(n) is $O(2^n)$ (i.e., grows exponentially)

10Min Break And Practice

- Sort the functions in the increasing order:
  - $f_1(N) = N^2 + 100N$
  - $f_2(N) = 2^N + N^6 + 100N$
  - $f_3(N) = N^3 \log(N) + 400N^2$
  - $f_4(N) = 2N^3 + 100N + 10^8$
  - $f_5(N) = (\log(N))^{15}$
  - $f_6(N) = 99N + \log(N) + 4^N$
  - $f_7(N) = N \log(N) + 100N$
  - $f_8(N) = \log(N/2)$

Practice Answers

Sort the functions in the increasing order:

- $f_1(N) = N^2 + 100N$ - $O(N^2)$
- $f_2(N) = 2^N + N^6 + 100N$ - $O(2^N)$
- $f_3(N) = N^3 \log(N) + 400N^2$ - $O(N^3 \log(N))$
- $f_4(N) = 2N^3 + 100N + 10^8$ - $O(N^3)$
- $f_5(N) = (\log(N))^{15}$ - $(\log_{10}(N))^{15} << N \rightarrow O(N)$
- $f_6(N) = 99N + \log(N) + 4^N$ - $O(4^N)$
- $f_7(N) = N \log(N) + 100N$ - $O(N \log(N))$
- $f_8(N) = \log(N/2)$ - $O(\log(N))$

$f_8 < f_5 < f_7 < f_1 < f_4 < f_3 < f_2 < f_6$

Go to https://www.desmos.com/ and draw all these functions

Live Code Demo

- Function pointers (with arrays)
  - Implement the filter function
- File I/O
  - Using fscanf and fgets

Today's Review

- Big-O notation (cont'd)
  - other mathematical variations
  - using algebraic mechanics to derive time complexity of recursive algorithms
- Sorting algorithms
  - an $O(N^2)$ way (Selection Sort)
  - an $O(nlogn)$ way (Mergesort)
- Searching algorithms
  - an $O(n)$ way (Linear Search)
  - an $O(logn)$ way (Binary Search, under one condition)

Homework!

- Linear Search typically returns upon the first match. What if you want a Linear Search function that finds all the matches are? How do you write it in C?
- Suppose instead of ascendingly sorted we have an array of descendingly sorted items, how would you change the Binary Search algorithm to keep it working?
- Binary Search can also be written as a recursive function, try to write the code for it!

- Watch a video for a more detailed proof for time complexity of the Mergesort
  https://stream.sfu.ca/Media/Play/70de71db57474331a42982630605837d1d
- Watch a visualization of the Binary Search algorithm
  https://www.cs.usfca.edu/~galles/visualization/Search.html