SCC120
Fundamentals of Computer Science

Introduction to Algorithms

Rough guide to O()

How do we find O() for some algorithm or code?

- Loops
- Nested Loops
- Sequences
- Branches
- Dominant Term

Loops

```
int sumArray(int[] aiNumbers)
{
      int iSum = 0;
      for (int i=0; i<aiNumbers.length; i++)
            iSum += aiNumbers[i];
      return iSum;
}
```

How many iterations of the for loop?

This loop always executes *aiNumbers.length* times

sumArray function

What is the **Big O complexity class** (of *sumArray*)?

aiNumbers.length is the length of our input (N)

T(N) = c1 * N + c2

Big O complexity class is O(N)

- So the Big O complexity class is O(N)
- But we have done all of that before!

- A faster way to find Big O complexity for general “for” loops is…

General *for* loops

```
for (int i=0; i<N; i++)
      { sequence of statements; }
```

Loop executes N times (in worst case), so the sequence of statements also executes N times

If we assume statements are O(1), the total time of *for* loop is O(N) x O(1), which is O(N) overall

What if statements are O(log N) or O(N)?

while loops

```
int i = 0;
while (i < N) {
      sequence of statements;
      i++;
}
```

Loop executes N times (in worst case), so the sequence of statements also execute N times

What if statements are O(1), O(log N), or O(N)?

Loops: Exercises (few min)

What is the complexity of these loops?

```
i = 0;
while (i < N)
        { /* while body, assume constant */ }
```

what if O(M)?

```
for (int i=N-2; i<N; i++)
        { /* for loop body, assume constant */ }
```

Nested Loops

- Nested loops are dealt with by applying Loops rule from the inside out
- Running time of a group of nested loops is **product of sizes of all loops**

```
count = 0;
for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
            for (int k=0; k<n; k++)
                  count++;
```

- This code's complexity is $O(n^3)$

General Nested Loops

```
for (int i=0; i<N; i++)
      for (int j=0; j<M; j++)
            { sequence of statements; }
```

- How many times does the outer loop executes?
  - N times
- For every outer loop execution, how many times does the inner loop executes?
  - M times
- How many times does the sequence of statements in inner loop execute?
  - N x M times, and assuming sequence of statements is O(1), total complexity is O(N*M)

Nested Loops

```
count = 0;
for (int i=n-10; i<n; i++)
      for (int j=n-10; j<n; j++)
            for (int k=n-10; k<n; k++)
                  count++;
```

- Each loop may not execute n times
- In this case, each loop executes 10 times, which is a constant (the exact number 10 does not matter)
- So complexity is O(1)

Nested Loops

```
for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
            for (int k=0; k<n; k++)
                  { statements; }
```

- Statements may or may not be constant time
- If statements is $O(m)$, overall complexity is $O(n^3 m)$
- Complexity of statements can be any function of n, say f(n), and overall complexity is $O(n^3 \times f(n))$

Nested Loops: Exercise (few min)

```
for (int i=1; i<=n; i++) {
        j = n;
        while (j > 1) {
                j = j/2;
        }
}
```

What is O() complexity?

Answer

- Outer *for* loop executed n times
- Each time *while* loop executes, number of steps is cut in half
- We have seen this before, and number of times *while* loop executes is $\log_2 n$
- So this code is O(n x log n)

Sequences

- Consecutive operations should be added together
- What is complexity of this code?

```
count = 0;
for (int i=0; i<n; i++)
      count++;
for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
            count++;
```

- Complexity is $O(n) + O(n^2) = O(n + n^2) = O(n^2)$

General Sequences

- Sequence of statements

  statement 1; statement 2; … statement k;

- Total time is found by adding the time of each statement
- Total time = time(statement 1) + time(statement 2) + … + time(statement k)
- If each statement involves only basic operations and has constant time, then total time is O(1)

Sequences: Exercise (few min)

```
int aMethod(int[] aiX)
{
      int var = 0;
      for (int i=0; i<aiX.length; i++)
            var += aiX[i];
      for (int i=0; i<aiX.length; i++)
            var -= aiX[i] / 2;
      for (int i=0; i<aiX.length; i++)
            var -= aiX[i] / 2;
      return var;
}
```

- What is the complexity class?

Sequences

- What is the complexity class?
  - Complexity is $O(n + n + n) = O(n)$

Branches

```
int count = 0;
if (n <= 0) {
      count = n;
} else {
      for (int i=0; i<n; i++)
            count += i;
}
```

- What is code complexity?
- “if” part is O(1), “else” part is O(n), and we assume worst-case for overall complexity of O(n)

General Branches

```
if (condition) {
      sequence of statements 1
} else {
      sequence of statements 2
}
```

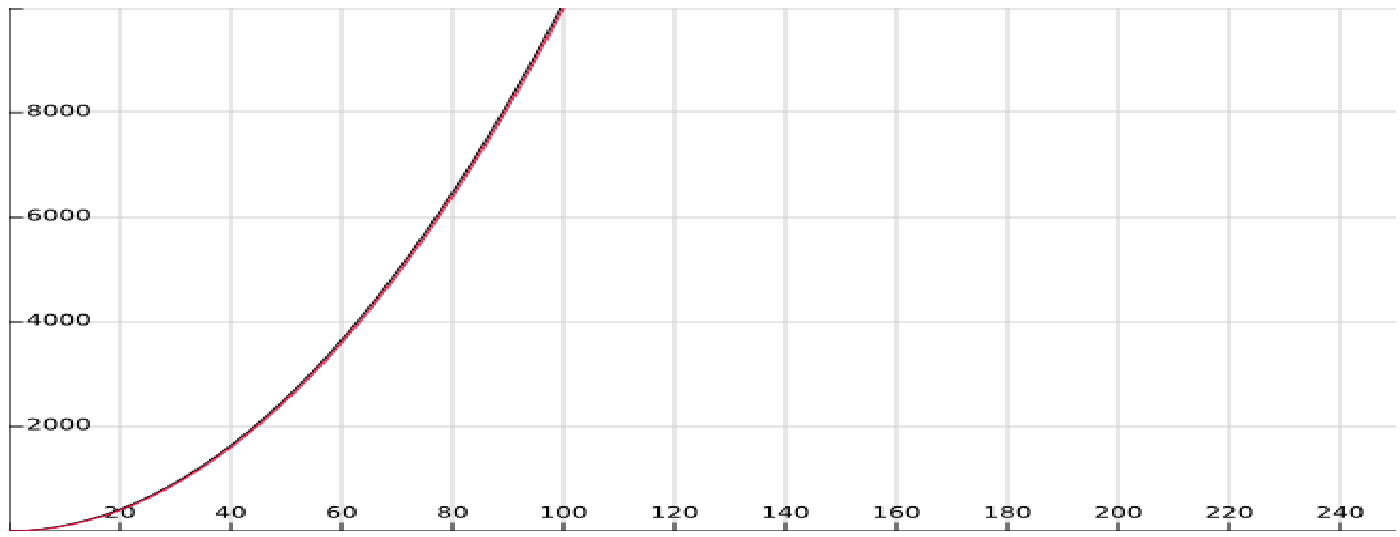
- What is code complexity?
- Either sequence 1 or sequence 2 will execute
- Worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2))
- For example, if sequence 1 is $O(n^2)$, and sequence 2 is $O(n)$, overall complexity for if-then-else is $O(n^2)$

Dominant Term

- Example:

$$O(n + n^2) = O(n^2)$$

- n^2 terms dominate as n gets larger



Dominant Term

- More generally, expressions of the type:

  $$O(n + n^2 + n^3 + \ldots + n^{k-2} + n^{k-1} + n^k) = O(n^k)$$

- The n^k term will dominate for large n

- Many Big O expressions can be reduced this way

SCC120
Fundamentals of Computer Science

Introduction to Algorithms

Overview

- Improving search algorithms
  - Searching an unsorted array using a Sentinel
  - Searching a sorted array
  - Searching a sorted array starting from the middle, or binary search

Linear Search

```
boolean isInArray(int theArray[], int iSearch)
{
      int N = theArray.length;
                        o1    o3
      for (int i = 0; i < N; i++)
                            o2
            if (theArray[i] == iSearch)
                  return true;

      return false;
}
```

Operations o1, o2, o3 executed every time around *for* loop

Sentinel Search

```
boolean isInSentinel(int[] theArray, int iSearch)
{
        int N = theArray.length;
        if (theArray[N-1] == iSearch)
                return true;
        theArray[N-1] = iSearch;
                    no o1!    o3
        for (int i=0; i< n ; i++)
        {
                                  o2
                if (theArray[i] == iSearch)
                        break;
        }
        return (i < (N-1));
}
```

Sentinel Search

Show example

isInSentinel() is a function which takes two arguments:

- An array of integers, and an integer to search the array for
- *isInSentinel* returns true if the number is in the array, false otherwise

Use the fact that the loop stops *whenever* the target is found:

- For this, place the value *iSearch* in the array as the last element and use this comparison (o2) to mark the end of the array
- Avoid using the comparison $i<N$ (o1)

This method is called “Sentinel Search”, because the item acts as a guard, warning the end of the array

Compare Linear vs. Sentinel Search

```
boolean isInArray(int[] theArray,int iSearch)
{
     int N =theArray.length;
     for (int i=0; i<N; i++)
     {
          if (theArray[i] ==iSearch)
               return true;
     }
     return false;
}
```

```
boolean isInSentinel(int[] theArray,int  iSearch)
{
     int N = theArray.length;
     if (theArray[N-1] ==iSearch)
          return true;
     theArray[N-1] =iSearch;
     for (int i=0;    ; i++)
     {
          if (theArray[i] ==iSearch)
               break;
      }
      return i < (N-1);
 }
```

- Time for one iteration of the loop (C_L for linear and C_S for sentinel)

  $C_L > C_S$ (sentinel is better)

- Initialisation times (say, I_L for linear and I_S for sentinel)

  $I_L < I_S$ (sentinel is worse)

- Both O(N) in worst case, but sentinel better for large N

Effect of sorting on search

Show example

isInSorted() is a function which takes two arguments:

- an array of integers in increasing order
- and an integer to search the array for

isInSorted() returns true if the number is in the array, false otherwise

Use idea of Sentinel again

Sentinel Search

```
boolean isInSentinel (int[] theArray, int iSearch)
{
     int N = theArray.length;
     if (theArray[N-1] == iSearch)
          return true;
     theArray[N-1] = iSearch;
     int i;
     for (i=0;    ; i++)
          if (theArray[i] == iSearch)
               break;
     return (i < (N-1));
}
```

If array is sorted in increasing order, what change can make the algorithm more efficient?

Sentinel on Sorted

```
boolean isInSorted(int[] theArray, int iSearch){
     int N = theArray.length;

     if (theArray[N-1] == iSearch)
          return true;
     theArray[N-1] = iSearch;

     int i;
     for (i=0;    ; i++)
          if (theArray[i] >= iSearch)
               break;
     return (i < (N-1)) && (theArray[i] == iSearch);
}
```

Enhanced Sentinel on Sorted

```
boolean isInSortedEnhanced(int[] theArray, int iSearch){
    int N = theArray.length;

    if (theArray[N-1] < iSearch)
        return false; // Because we know array is sorted
    if (theArray[N-1] == iSearch)
        return true;
    theArray[N-1] = iSearch;

    int i;
    for (i=0;    ; i++)
        if (theArray[i] >= iSearch)
            break;
    return (i < (N-1)) && (theArray[i] == iSearch);
}
```

Enhanced Sentinel on Sorted

```
boolean isInSortedSuper(int[] theArray, int iSearch){
     int N = theArray.length;

     if ((theArray[N-1] < iSearch) OR
         (theArray[0] > iSearch))
         return false;  // Because we know array is sorted

     … Rest same as before
```

Effect of sorting on search

Compare this method to Sentinel method on unsorted array

- Loop time is the same as the Sentinel method on unsorted array
- In many cases, we will get an improvement
  - [5,8,9,12,18] and searching for 10
  - [5,8,9,12,18] and searching for 20
- But in the worst case, we won't
  - [5,8,9,12,18] and searching for 14
- Worst Case Time complexity still: O(N)

Overview

- Improving search algorithms
  - Searching an unsorted array using a Sentinel
  - Searching a sorted array
  - Searching a sorted array starting from the middle, or binary search

Binary Search

- *isInBinary()* is a function which takes two arguments: an array of integers in increasing order, and an integer to search the array for
- *isInBinary()* returns true if integer is in array, false otherwise

Use idea of Binary Search (i.e. keep splitting list in half):

- Start by checking middle item in list
- If it is not target and target is smaller than middle item, target must be in first half of list
- If target is larger than middle item, target must be in right half of list
- **One comparison reduces number of items left to check by half**
- Search continues by checking middle item in remaining half of list
- The splitting process continues until target is found, or the remaining list consists of one item only
- If that item is not target, then it's not in the list

Binary Search Function

```
boolean isInBinary(int[] theArray, int iSearch)
{
     int lo = 0;
     int hi = theArray.length - 1;
     int mid = 0;
     while (hi >= lo) {
          mid = (lo + hi)/2; //round to higher integer
          if (theArray[mid] == iSearch)
               return true;
          else if (theArray[mid] < iSearch)
               lo = mid + 1;
          else
               hi = mid - 1;
     }
     return false;
}
```

Binary Search in Action

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	<-	index
3	6	8	12	13	14	17	**21**	23	25	28	31	34	35		

L=0 **M=7** **H=13** **Done!**

Binary Search in Action

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	<- index
3	6	8	12	13	14	17	21	23	25	28	31	34	35	

L=0 M=7 H=13 (iterate)

L=0 M=3 H=6 (iterate)

L =0 M=1 H=2 (iterate)

L=M=H=2 (Done)

4 iterations of the loop when n=14

$\log_2(14)$ = 3.8, which is approx. 4, which is $\log_2(16)$

Worst Case Complexity

What is maximum number of iterations?

- How many times can we divide the array in half before only 1 item is left?
- Let's assume the array has N = 64 items

Initially	list size = 64
After 1st search	list size = 32
After 2nd search	list size = 16
After 3rd search	list size = 8
…	…
After 6th search	list size = 1

So if we have N = 64, then $N = 2^6$

We will have approximately 6 + 1 searches!

Worst case= $(\log_2 n) + 1$ searches = $O(\log n)$

Summary

- Improving search algorithms
  - Searching an unsorted array using a Sentinel O(N)
  - Searching a sorted array O(N)
  - Searching a sorted array starting from the middle, or binary search O(log N)

The Correctness of Algorithms

SCC 120: Fundamentals of Computer Science

Proving Algorithms Correct

Does the algorithm serve its purpose?

- Upon termination, does the algorithm produce the *correct* answer?
  - For insertion sort, correct means the output is a permutation of the input but sorted in increasing order
  - For linear search, correct means returning true *if and only if* number being searched occurs in the array

Establishing Correctness

- State a *loop invariant*
  - A property that holds before any iteration of a loop
- Initialization
  - Prove that invariant holds before first iteration of loop
- Maintenance
  - Prove that if invariant holds before i^{th} iteration (hypothesis), then it holds before $(i+1)^{th}$ iteration
- Termination
  - Identify the conditions under which algorithm terminates
  - For each condition, show that desired outcome is obtained
  - Invariant helps in establishing the desired outcome

Linear Search

Input: an array A of numbers, a number s

Output: true iff $\exists k \; A[k] = s \wedge (0 \leq k < A.length)$

```
for(int i = 0; i < A.length; i++) {
    if(A[i] == s)
        return true
}
return false
```

Correctness of Linear Search

Input: A and s

Output: true iff $\exists k\ A[k] = s \wedge (0 \leq k < A.length)$

- *Loop invariant*
  - Before start of any iteration of loop, $\forall j\ (0 \leq j < i) \rightarrow A[j] \neq s$ holds
- Initialization
  - Before start of first iteration ($i = 0$), invariant vacuously holds.
- Maintenance
  - Prove: If $\forall j\ (0 \leq j < i) \rightarrow A[j] \neq s$ holds before start of i^{th} iteration, then $\forall j\ (0 \leq j < i+1) \rightarrow A[j] \neq s$ holds before start of $i+1^{th}$ iteration
  - Proof: By looking at the body of the loop, if loop goes on to start of $(i+1)^{th}$ iteration, then $A[i] \neq s$. From this fact and hypothesis, $\forall j\ (0 \leq j < i+1) \rightarrow A[j] \neq s$ holds before start of $i+1^{th}$ iteration.
- Termination
  - Loop terminates because $A[i] = s$ for some i such that $0 \leq i < A.length$. We return true in this case.
  - Loop terminates because $i = A.length$. By the loop invariant, this means $\forall j\ (0 \leq j < A.length) \rightarrow A[j] \neq s$ holds, in which case we return false.

Insertion Sort

Input: An array A

```
for (int i = 1; i < A.length; i++) {
    int x = A[i];
    int j;
    for (j = i-1; j >= 0 && A[j] > x; j--) {
        A[j+1] = A[j];
    }
    A[j+1] = x;
}
```

Correctness of Insertion Sort

Input: an array A of integers

- Output
  - A such that $\forall\, j, k\ (0 \leq j, k < A.length) \rightarrow (A[j] < A[k] \rightarrow k > j)$ and the output A is a permutation of the input A
- Loop invariant
  - Before start of an iteration, $\forall\, j, k\ (0 \leq j, k < i) \rightarrow (A[j] < A[k] \rightarrow k > j)$ holds and output subarray $A[0 \ldots i-1]$ is a permutation of input subarray $A[0 \ldots i-1]$.
- Initialization...
- Maintenance...
- Termination ...

Correctness of Binary Search

Input: A and s

Output: true iff $\exists k\ A[k] = s \wedge (0 \leq k < A.length)$

- Loop invariant: At the start of any iteration of the loop,

$$\forall i\ (0 \leq i < lo) \rightarrow A[i] < s$$

and

$$\forall i\ (hi < i < A.length) \rightarrow A[i] > s$$