
MONASH University
BYTE INTO YOUR IT CAREER
11 AUGUST 2016
Employer Exhibition
Seminar Series
25 AUGUST 2016
Mock Interviews
Job Application Checking
Book in
http://it.monash.edu/byte-career

Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 2: Analysis of Algorithms

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Recommended reading

- Basic mathematics used for algorithm analysis: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Math/
- Program verification: http://www.csse.monash.edu.au/courseware/cse2304/2006/03logic.shtml
- For more about Loop invariants: Also read Cormen et al. Introduction to Algorithms, Pages 17-19, Section 2.1: Insertion sort.).

Algorithmic Analysis

In algorithmic analysis, one is interested in (at least) two things:

- An algorithm's correctness.
- The amount of resources used by the algorithm

In this lecture we will explore these two issues.

Proving correctness of algorithms

- Commonly, we write programs and then test them.
- However, testing can only show that a program is **wrong**.
- It can never show that it is **always** correct!
  - It may give correct results for 1 Billion test cases but may still be incorrect …
- [Logic], on the other hand, can prove that a program is always correct. This is usually achieved in two parts:
  1. Show that the program **always** terminates, and
  2. Show that a program is correct when it terminates

Finding minimum value

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]//note: we assume index range is 1 … N
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

Does it always terminate?

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

Correct result at termination?

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

Correctness using Loop Invariant

```
//A Loop Invariant (LI) is a property that is true before and
after each iteration
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 … index - 1]
while index <= N
      //LI: min equals the minimum value in array[1 … index-1]
      if array[index] < min
            min = array[index]
      //min equals the minimum value in array[1 … index]
      index = index + 1
      //LI: min equals the minimum value in array[1 … index-1]

//LI: min equals the minimum value in array[1 … index-1]
// and index = N + 1; thus min is min value in array [1 … N]
return min
```

Proof of Correctness

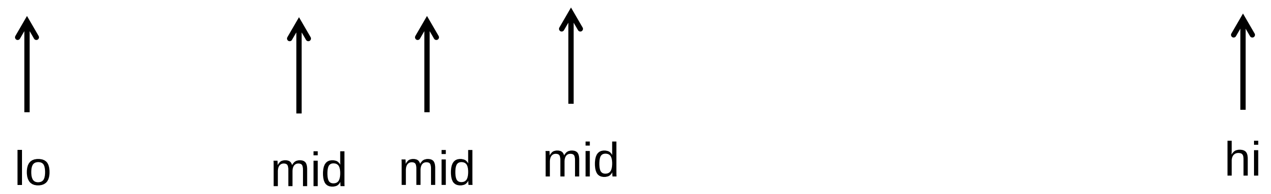
Important: Always prove the correctness of your algorithm.

- It guarantees that your algorithm will always return correct results
- It also helps to identify and fix the bugs (if any)

We will show this using a Binary Search algorithm

Binary Search revisited

5	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
1	2	3	4	5	6	7	8	9	10



Searching 20

- Since 20 < array[mid],
  - Search from lo to mid (e.g., move hi to mid)
- Since 20 > array[mid]
  - Search from mid to hi (e.g., move lo to mid)
- …

Algorithm for Binary Search

```
lo = 1
hi = N

while ( lo < hi )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Is this algorithm correct?

To prove correctness, we need to show that

1. it **always** terminates, and
2. it returns correct result when it terminates

Does this algorithm **always** terminate?

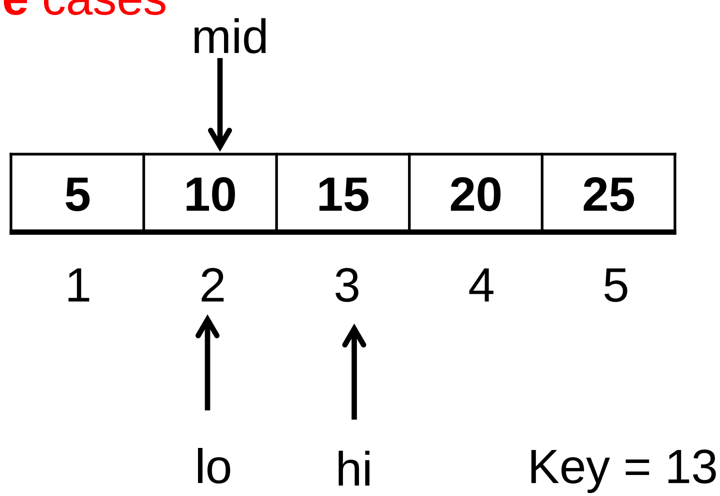
```
lo = 1
hi = N

while ( lo < hi )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Enter your answers at MARS

1. Visit http://mars.mu on your internet enabled device
2. Log in using your Authcate details
3. Touch the + symbol
4. Enter the code for your unit: Y4R44G
5. Answer questions when they pop up

Let us try to fix this

This algorithm may never terminate in **some** cases

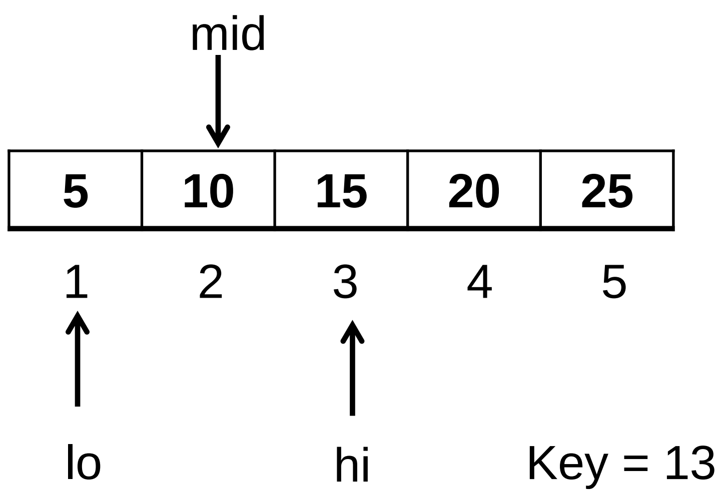


Does this algorithm <u>always</u> terminate?

```
lo = 1
hi = N

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Proof that it always terminates

- lo < hi – 1implies that the difference between lo and hi is always at least 2
- Therefore, lo < mid < hi.
- Hence, the search space always shrinks (e.g., lo and hi get closer after every iteration of the while loop until lo ≥ hi – 1 in which case the algorithm terminates)



Correct result at termination?

```
lo = 1
hi = N

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

The algorithm always terminates. But does it give correct result when it terminates?

Border cases:

What if

- array is empty?

Correct result at termination?

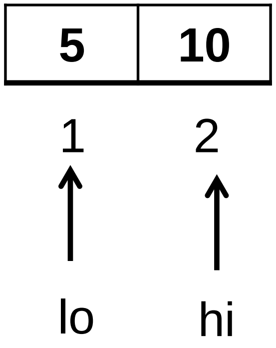
```
lo = 1
hi = N

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

The algorithm always terminates. But does it give correct result when it terminates?

Border cases:

What if

- array is empty?
- array has 1 element?
- array has 2 elements?



Key = 10

Correct result at termination?

```
lo = 1
hi = N + 1

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key > array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

The algorithm always terminates. But does it give correct result when it terminates?

Border cases:

What if

- array is empty?
- array has 1 element?
- array has 2 elements?

Is it possible that the algorithm accesses array out of the range (e.g., array[N+1]?)

No, because;

- it only accesses array[mid] or array[lo], and
- lo < mid < hi

5	10

1 2 3

lo hi

Key = 10

Correct result at termination?

```
lo = 1
hi = N + 1

while ( lo < hi - 1 )
   mid = floor( (lo+hi)/2 )
   if key > array[mid]
      lo=mid
   else
      hi=mid

if N > 0 and array[lo] == key
   print(key found at index lo)
else
   print(key not found)
```

The modified algorithm returns correct results for border cases. Does it return correct results for the general case (i.e., N > 2)?

Observations:

- The algorithm never accesses array[hi], and
- lo < mid < hi

This means if key == array[hi], the algorithm will not find it

Fix: hi = mid only if key < array[mid]

5	**10**	**15**	**20**	**25**
1	2	3	4	5
lo	mid lo	mid hi		hi

Key = 15

Correct result at termination?

```
lo = 1
hi = N + 1

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Hopefully, we have fixed all the bugs!

Since we made several changes, we need to show that the modified algorithm;

1. **always** terminates.
2. returns correct result when terminates.

- Easy to show that it always terminates (as before)
- Easy to show that it is correct for border cases
- Next, we show the correctness for the general case using Loop Invariant

Correctness using Loop Invariant

```
lo = 1
hi = N + 1
// LI: key in array[1 … N] if and only if (iff) key in array[lo … hi - 1]
while ( lo < hi - 1 )
    // LI: key in array[1 … N] iff key in array[lo … hi-1]
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
       // key in array[1 … N] iff key in array[mid … hi-1]
       lo=mid
       // LI: key in array[1 … N] iff key in array[lo … hi-1]
    else
       // key in array[1 … N] iff key in array[lo … mid-1]
       hi=mid
       // LI: key in array[1 … N] iff key in array[lo … hi-1]
// LI: key in array[1 … N] iff key in array[lo … hi-1]
```

Correctness using Loop Invariant

```
// LI: key in array[1 … N] if and only if (iff) key in array[lo … hi - 1]
while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
      lo=mid
    else
      hi=mid
// LI: key in array[1 … N] iff key in array[lo … hi-1]
//  lo ≥ hi – 1 → lo + 1 ≥ hi                           ----     (A)
//  lo < hi → lo + 1 ≤ hi                                ----     (B)
// From (A) and (B): lo + 1 = hi → lo = hi - 1
// Hence,  key in array[1 … N] iff key in array[lo … lo];   (Proof Complete)
 if N > 0 and array[lo] == key
      print(key found at index lo)
    else
      print(key not found)
```

Note: lo < hi when loop terminates, because

- lo < mid < hi in each iteration and
- we update either lo to be mid or hi to be mid

More on Loop Invariants

- Loop Invariants help us to formally prove the correctness of the algorithms
- Loop invariants can also be used to write the algorithms
- Assertions can be used to identify problems

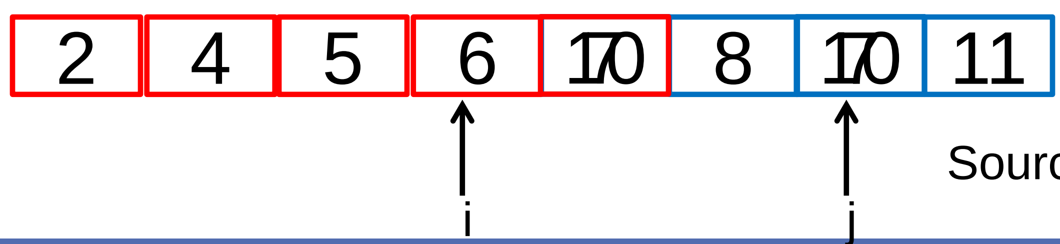
Next, we show how to write two sorting algorithms using Loop Invariants

Writing algorithms using Loop Invariant

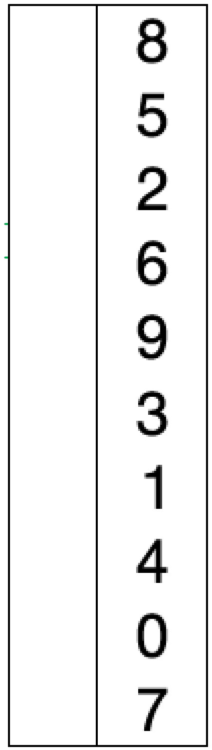
Sort an array (denoted as arr) in ascending order

```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
// LI: arr[1 … i] is sorted AND arr[1 … i] <= arr[i+1 …
}
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```

This is Selection Sort



Source: wikimedia.org



Writing algorithms using Loop Invariant

Sort an array (denoted as arr) in ascending order

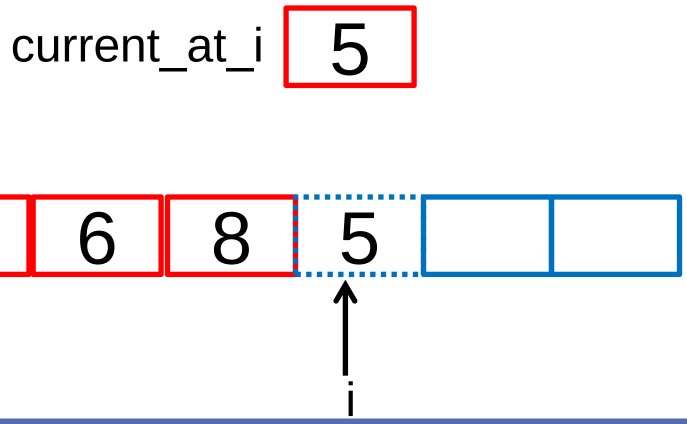
```
for(i = 1; i < N; i++) {
       j = index of minimum element in arr[i … N]
       swap (arr[i],arr[j])
// LI: arr[1 … i] is sorted AND arr[1 … i] <= arr[i+1 … N]
}
// LI: arr[1 … N-1] is sorted AND arr[1 … N-1] <= arr[N]
```

```
Could we use a weaker loop invariant, e.g.,
// LI: arr[1 … i] is sorted
```

Writing algorithms using Loop Invariant

Sort an array (denoted as arr) in ascending order

```
for(i = 1; i ≤ N; i++) {
       current_at_i = arr[i]
       // insert current_at_i in arr[1 … j] in sorted order
       // arr[1 … j] <= current_at_i < arr[j+2 … i]
// LI: arr[1 … i] is sorted
}
// LI: arr[1 … N] is sorted
```



Writing algorithms using Loop Invariant

Sort an array (denoted as arr) in ascending order

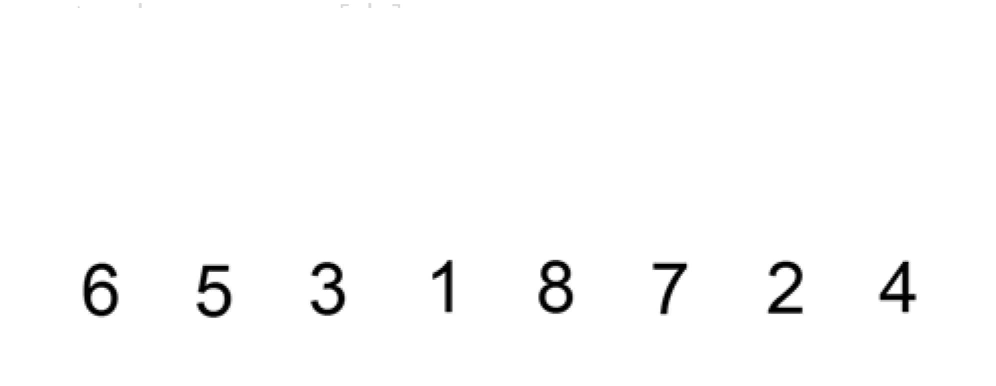
```
for(i = 1; i ≤ N; i++) {
      current_at_i = arr[i]
      j = i - 1
      while(current_at_i < arr[j] and j>0{
            arr[j+1] = arr[j])
            j = j-1
            //LI2: current_at_i < arr[j+2 … i]
      }
      arr[j+1] = current_at_i
      // arr[1 … j] <= current_at_i < arr[j+2 … i]
// LI: arr[1 … i] is sorted
}
// LI: arr[1 … N] is sorted
```

This is Insertion Sort

Writing algorithms using Loop Invariant

Sort an array (denoted as arr) in ascending order

```
for(i = 1; i ≤ N; i++) {
        current
        j = i -
        while(c

        }
        arr[j+1
        // arr[                                   i]
// LI: arr[1 … i] is sorted
}
// LI: arr[1 … N] is sorted
```



Source: wikimedia.org

This is Insertion Sort

Complexity Analysis

- Time/space complexity of an algorithm
  - Amount of time/space taken by an algorithm as a function of the input size
- Worst-case complexity
- Best-case complexity
- Average-case complexity

Let’s analyze the comlexity of the algorithms we studied today

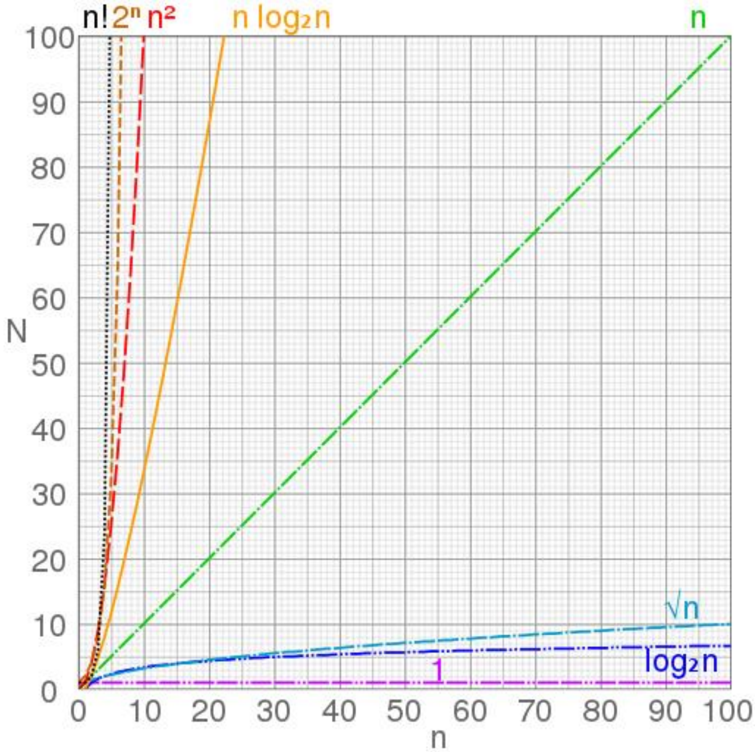


Image Source: By Cmglee - Own work, https://commons.wikimedia.org/w/index.php?curid=50321072

Complexity: Finding minimum value

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

      if array[index] < min
            min = array[index]
      index = index + 1

return min
```

Time Complexity?

- Worst-case
- Best-case
- Average

Space Complexity?

- Worst-case, Best-case, Average

Complexity: Binary Search

```
lo = 1
hi = N + 1

while ( lo < hi - 1 )
    mid = floor( (lo+hi)/2 )
    if key >= array[mid]
        lo=mid
    else
        hi=mid

if N > 0 and array[lo] == key
    print(key found at index lo)
else
    print(key not found)
```

Time Complexity?

- Worst-case
  - Search space at start: N
  - Search space after 1st iteration: N/2
  - Search space after 2nd iteration: N/4
  - …
  - Search space after x-th iteration: 1

  What is x? i.e., how many iterations in total?

  O(log N)
- Best-case
  - Can be improved to O(1) by returning key when key == array[mid]
- Average

Space Complexity?

- Worst-case, Best-case, Average

Complexity: Selection Sort

```
for(i = 1; i < N; i++) {
        j = index of minimum element in arr[i … N]
        swap (arr[i],arr[j])
}
```

Time Complexity?

- Worst-case
  - Complexity of finding minimum element at i-th iteration:
  - Total complexity:
- Best-case
- Average

Space Complexity?

- Worst-case, Best-case, Average

Complexity: Insertion Sort

```
for(i = 1; i ≤ N; i++) {
 current_at_i = arr[i]
 j = i - 1
 while(current_at_i < arr[j] and j>0{
  arr[j+1] = arr[j])
  j = j-1
 }
 arr[j+1] = current_at_i
}
```

Time Complexity?

- Worst-case
  - Complexity of while loop at i-th iteration ;
  - Total complexity:
- Best-case
  - Complexity of while loop at i-th iteration:
  - Total complexity:
- Average

Space Complexity?

- Worst-case, Best-case, Average

Complexity of recursive algorithms

```
// Compute Nth power of x
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N–1)
}
```

Our goal is to reduce this term to T(1)

Time Complexity

Cost when N = 1: T(1) = b (b&c are constant)

Cost for general case: T(N) = T(N-1) + c (A)

Cost for N-1: T(N-1) = T(N-2) + c

Replacing T(N-1) in (A)

T(N) = T(N-2) + c + c = T(N-2) + 2*c (B)

Cost for N-2: T(N-2) = T(N-3) + c

Replacing T(N-2) in (B)

T(N) = T(N-3) + c+c+c = T(N-3) + 3*c

Do you see the pattern?

T(N) = T(N-k) + k*c

Find the value of k such that N-k = 1 → k = N-1

T(N) = T(N-(N-1)) + (N-1)*c = T(1) + (N-1)*c

T(N) = b + (N-1)*c = c*N + b - c

Hence, the complexity is O(N)

Complexity of recursive algorithms

```
// Recursive version
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N–1)
}
// Iterative version
result = 1
for i=1; i<= N; i++{
        result = result * x
}
return result
```

Space Complexity

Total space usage = Space used during the execution of the function + space used by stack to record the recursive calls

= O(1) + number of recursive calls

= O(N)

Note that an iterative version of power uses O(1) space

Complexity of recursive algorithms

```
// Compute Nth  power x
power2(x,N)
{
  if (N==0)
    return 1
  if (N==1)
    return x
  if (N is even)
    return power2( x * x, N/2)
 else
   return power2( x * x, N/2 ) * x
}
```

Time Complexity

Cost when N = 1: T(1) = b (b&c are constant)

Cost for general case: T(N) = T(N/2) + c (A)

Cost for N/2: T(N/2) = T(N/4) + c

Replacing T(N/2) in (A)

T(N) = T(N/4) + c + c = T(N/4) + 2*c (B)

Cost for N/4: T(N/4) = T(N/8) + c

Replacing T(N/4) in (B)

T(N) = T(N/8) + c+c+c = T(N/8) + 3*c

Do you see the pattern?

$T(N) = T(N/2^k) + k*c$

Find the value of k such that $N/2^k = 1 \rightarrow k = \log N$

$T(N) = T(N/2^{\log N}) + c*\log N = T(1) + c*\log N$

T(N) = b + c*log N

Hence, the complexity is O(log N)

Complexity of recursive algorithms

```
// Compute Nth  power x
power2(x,N)
{
   if (N==0)
      return 1
   if (N==1)
      return x
   if (N is even)
     return power2( x * x, N/2)
 else
    return power2( x * x, N/2 ) * x
}
```

Space Complexity

Space usage = space used during the execution of function + space used by stack to record recursive calls

= O(1) + number of recursive calls to power2()

= O(1) + O(log N)

= O(log N)

Recurrence Relations

Consider the example we saw earlier

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

$$T(1) = b$$

Such relations are called **recurrence relations** because T(N) is defined recursively.

- We saw how these recurrence relations can be solved.
- Next, we see the solutions for some common recurrence relations

Home work: Solve the recurrence relations shown in upcoming slides

Logarithmic complexity

Recurrence relation:

$$T(N) = T(N/2) + c$$

$$T(1) = b$$

Solution:

$$T(N) = O(\log_2 N)$$

Linear Complexity

Recurrence relation:

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

$$T(1) = b$$

Solution:

$$T(N) = O(N)$$

Linearithmic complexity

Recurrence relation:

$$T(N) = 2*T(N/2) + c*N$$

$$T(1) = b$$

Solution:

$$T(N) = O(N \log_2 N)$$

Quadratic complexity

Recurrence relation:

$T(N) = T(N-1) + c*N$

$T(1) = b$

Solution:

$T(N) = O(N^2)$

Exponential complexity

Recurrence relation:

$T(N) = 2*T(N-1) + c$

$T(0) = b$

Solution:

$T(N) = O(2^N)$

Concluding Remarks (Not Last Slide)

Take home message

- A proof is much stronger than a test
- You should always formally prove the correctness of your algorithm
- Your algorithms must have good space and time complexities

Things to do (this list is not exhaustive)

- Read more about content covered in this lecture
- Solve all the recurrence relations yourself (including the ones we solved in lectures)
- If you do not understand computational complexity, study to develop some background (e.g., watch videos, read other online resources)

Coming Up Next

- O(N Log N) sorting algorithms (e.g., heap sort, merge sort, quick sort)
- Stable/unstable sorting and in-place/out-of-place algorithms

This is the last slide

See you next week ☺