Analyzing Algorithms
Space and Time Complexity

Growth of Functions: Asymptotic Notations

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

Algorithm Specification (Pseudocode Conventions)

- In this course we present most of our algorithms using a pseudocode that resembles C

```
Algorithm sum (A, n)
// finds the sum of n numbers
// stored in an array A
{
    s:=0.0;
    for i:=1 to n do
            s:= s+A[i];
     return s;
}
```

There really is no precise definition of the pseudo-code language

Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm

Analyzing Algorithms

- Criteria for judging algorithms that have a more direct relationship to performance
  - Storage requirement
  - Computing time
- Analyzing an algorithm means predicting the resources that the algo requires
  - Most often it is computational time that we want to measure
- Generally, by analyzing several candidate algorithms for a problem, a most efficient one can be easily identified
- Before we can analyze an algorithm, we must have a model of the implementation technology that will be used, including a model for the resources of that technology and their costs

Two main characteristics for programs

- Time complexity: Execution time (CPU usage)
- Space complexity: The amount of memory required (RAM usage)
- Which measure is more important?
  - Answer often depends on the limitations of the technology available at time of analysis

The Random Access Machine (RAM) Model

- We are assuming a generic one-processor, RAM model of computation as our implementation technology, and our algos will be implemented as a computer program
- In the RAM model, instructions are executed one after another, with no concurrent operations
- The RAM model contains instructions commonly found in real computer
  - Arithmetic (add, subtract, multiply, divide, remainder, floor ceiling etc.)
  - Data movement (load, store, copy)
  - Control (conditional and unconditional branch, subroutine call and return)
- Each such instruction takes a constant amount of time

The RAM Model

- We also assume a limit on the size of each word of data
- Real computers contain other instructions also
  - For e.g., exponentiation (x^y): not a constant time instruction
  - It takes several instructions to compute x^y, if x,y are real nos
- In the RAM model, we do not attempt to model the memory hierarchy

Space Complexity

- Space complexity: The amount of memory, an algo needs to run to completion
  - When memory was expensive, we focus on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory)
  - Space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)

Space Complexity

- The space complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution.
  - This number is computed with respect to the size n of the input data.
  - Core dumps = the most often encountered cause is “memory leaks” – the amount of memory required larger than the memory available on a given system

Space Complexity

- Why is this of concern?
  - We could be running on a multi-user system where programs are allocated a specific amount of space.
  - We may not have sufficient memory on our computer.
  - There may be multiple solutions, each having different space requirements.
  - The space complexity may define an upper bound on the data that the program can handle.

Space Complexity (cont'd)

1. Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:

   - e.g. name of the data collection

   - same size for classifying 2GB or 1MB of texts

2. Variable part: Space needed by variables, whose size is dependent on the size of the problem:

   - e.g. actual text

   - load 2GB of text VS. load 1MB of text

Components of Program Space

- Program space = Instruction space + data space + stack space
- The **instruction space** is dependent on several factors.
  - the compiler that generates the machine code
  - the compiler options that were set at compilation time
  - the target computer

Components of Program Space

- Data space
  - Very much dependent on the computer architecture and compiler

char	1	float	8
short	2	double	8
int	4	long double	16
long	8	pointer	2

Unit: bytes

Components of Program Space

- Data space
  - Choosing a **“smaller” data type** has an effect on the overall space usage of the program.
  - Choosing the **correct type** is especially important when working with arrays.
  - How many bytes of memory are allocated with each of the following declarations?

    ```
    double a[100];
    int matrix[rows][cols];
    ```

Components of Program Space

- Environment Stack Space
  - Every time a function is called, the following data are saved on the stack.
    1. the return address
    2. the values of all local variables and values of formal parameters
    3. the binding of all reference and const reference parameters
  - What is the impact of recursive function calls on the environment stack space?

Space Complexity Summary

- Given what you now know about space complexity, what can you do differently to make your programs more space efficient?
  - Always choose the optimal (smallest necessary) data type
  - Study the compiler.
  - Learn about the effects of different compilation settings.
  - Choose non-recursive algorithms when appropriate.

Time Complexity

- Time taken by a program P = compile time + run time (tp)
- tp(n) for any given n can be obtained experimentally
- Program: typed, compiled, & run on a particular machine
- The execution time is physically clocked
- Difficulties with this experimental approach
  - Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative
  - It is difficult to compare the efficiency of two algorithms unless experiments on their running time have been performed in the same h/w and s/w environments
  - It is necessary to implement and execute an algorithm in order to study its running time experimentally

Time Complexity

- We desire an analytic framework that:
  - Considers all possible inputs
  - Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the h/w and s/w environment
  - Can be performed by studying a high-level description of the algorithm without implementing it or running experiments on it
- In general, the time taken by an algorithm grows with the size of the input
- So, we are interested in determining the dependency of the running time on the size of the input
- Analytic framework aims at associating a function f(n) with each algorithm that characterizes the running time of the algorithm in terms of the input size n

Time Complexity

- So, we need to define the terms “running time” and “size of input”
- Input size:
  - Notion of input size depends on the problem being studied
  - Ex: sorting: the most natural measure is the number of items in the input, array size n
  - Ex: if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph
- Running time of an algorithm on a particular input is the number of primitive operations executed
  - It is convenient to define the notion of step so that it is as machine-independent as possible

Time Complexity

- Consider the view (keeping with the RAM model):
  - A constant amount of time is required to execute each line of our pseudocode
  - One line may take a different amount of time than another line
    - We may assume that each execution of the i^{th} line takes time c_i, where c_i is a constant

Time Complexity

		cost	times	Total operations
1	Algo sum (A, n)	0	-	0
2	{	0	-	0
3	s:=0.0;	c_3	1	c_3
4	for i:=1 to n do	c_4	n+1	$c_4(n+1)$
5	s:= s+A[i];	c_5	n	c_5n
6	return s;	c_6	1	c_6
7	}	0	-	0

$$(c_4+c_5)n+(c_3+c_4+c_6)$$

$$= an+b$$

Observation: Run time grows linearly in n
[if n is doubled, the run time also doubles (approx)]
So Algo sum is a linear time algo

Time Complexity

To add two m x n matrices ‘A’ and ‘B’

	cost	times	Total operations
Algo add (A,B,C,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	c_3	m+1	c_3 (m+1)
for j:=1 to n do	c_4	m(n+1)	c_4m(n+1)
C[i,j] := A[i,j] + B[i , j];	c_5	mn	$c_5$mn
}	0	-	0

$(c_4+c_5)mn+(c_3+c_4)m+c_3$

$= amn+bm+c$

Observations:
Input size given by two numbers
If m>n: better to interchange the two for statements
If this is done total steps becomes amn+bn+c

Time Complexity

- Simplified analysis can be based on:
  - Number of arithmetic operations performed
  - Number of comparisons made
  - Number of times through a critical loop
  - Number of array elements accessed

Example: Polynomial Evaluation

General form of polynomial is

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0$$

where a_n is non-zero for all $n >= 0$

Example: Polynomial Evaluation

- Suppose that exponentiation is carried out using multiplications. Two ways to evaluate the polynomial
- $P(x) = 4x^4 + 7x^3 - 2x^2 + 3x^1 + 6$
- *Brute force method*:
  - p(x) = 4*x*x*x*x + 7*x*x*x - 2*x*x + 3*x + 6
- *Horner's method*:
  - p(x) = (((4*x + 7) * x - 2) * x + 3) * x + 6

Example: Polynomial Evaluation

Analysis for *Brute Force Method*:

$p(x) = a_n * x * x * \ldots * x * x \ +$	n multiplications
$a_{n-1} * x * x * \ldots * x * x \ +$	n-1 multiplications
$a_{n-2} * x * x * \ldots * x * x \ +$	n-2 multiplications
$\ldots +$	…
$a_2 * x * x +$	2 multiplications
$a_1 * x +$	1 multiplication
a_0	

Example: Polynomial Evaluation

Number of multiplications needed in the worst case is

$$T(n) = n + n\text{-}1 + n\text{-}2 + \ldots + 3 + 2 + 1$$

$$= n(n + 1)/2$$

$$= n^2/2 + n/2$$

Example: Polynomial Evaluation

Analysis for *Horner's Method*:

$p(x) = (\ldots (((a_n * x +$	1 multiplication
$a_{n-1}) * x +$	1 multiplication
$a_{n-2}) * x +$	1 multiplication
$\ldots +$	
$a_2) * x +$	1 multiplication
$a_1) * x +$	1 multiplication
a_0	

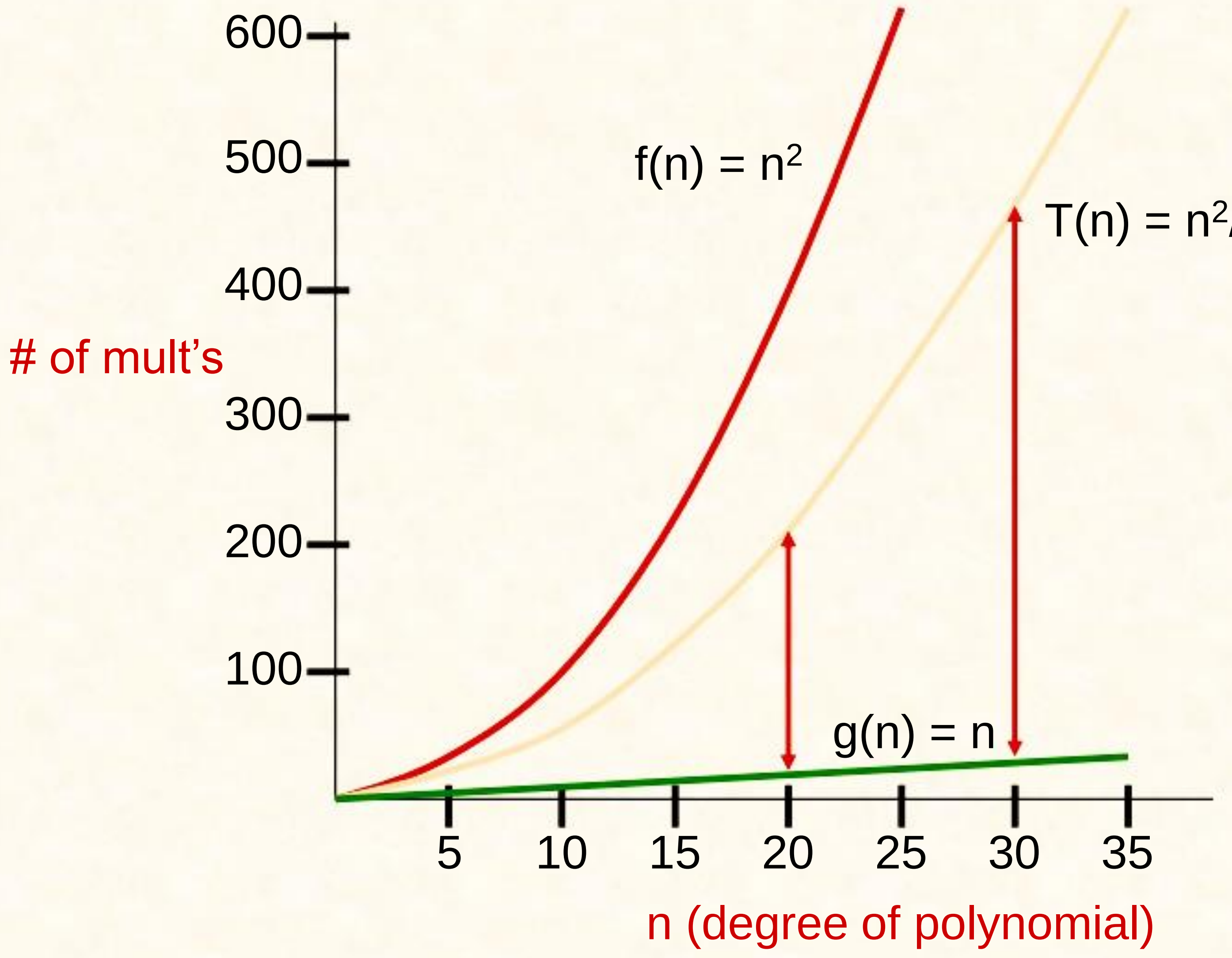
n times

$T(n) = n$, so the number of multiplications is $O(n)$

Example: Polynomial Evaluation

n (Horner)	$n^2/2 + n/2$ (brute force)	n^2
5	15	25
10	55	100
20	210	400
100	5050	10000
1000	500500	1000000

Example: Polynomial Evaluation



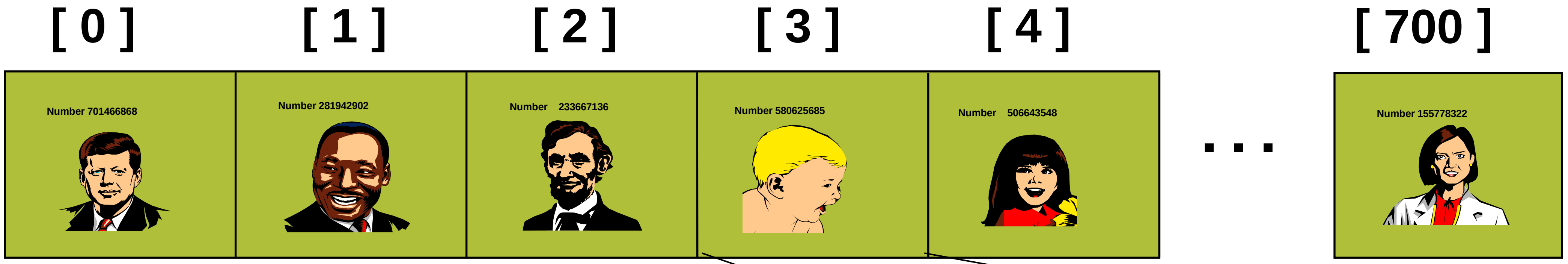
Cases to Consider

- Best Case
  - The least amount of work done for any input set
- Worst Case
  - The most amount of work done for any input set
- Average Case
  - The amount of work done averaged over all of the possible input sets

Problem: Search

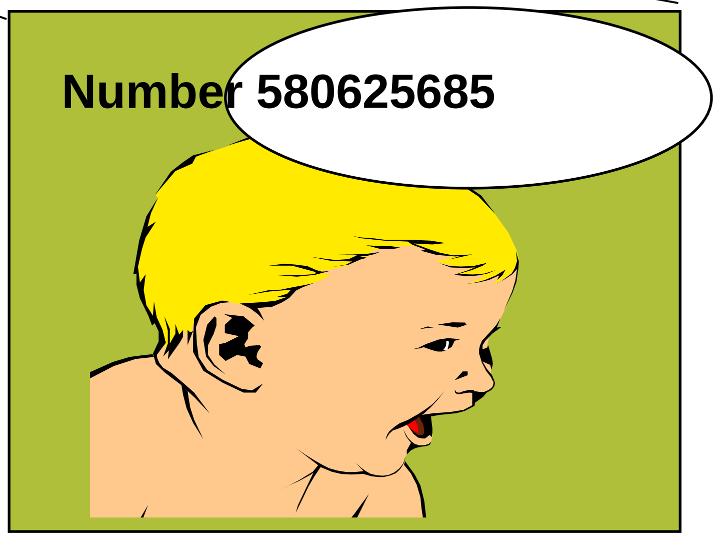
- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

Search



Each record in list has an associated key.
In this example, the keys are ID numbers.

Given a particular key, how can we efficiently retrieve the record from the list?



Serial Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success.

Pseudocode for Serial Search

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns the position of the desired record if found.
// Otherwise, return “not found”
…
for(i = 0; i < n; ++i )
        if(a[i] == desired item)
                return i+1;
```

Serial Search Analysis

- What are the worst and average case running times for serial search?
- Number of operations depends on *n*, the number of entries in the list.

Worst Case Time for Serial Search

- For an array of n elements, the worst case time for serial search requires n array accesses: $O(n)$.
- Consider cases where we must loop over all n records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

Average Case for Serial Search

Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:

- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. *etc.*

The average of all these searches is:

$$(1+2+3+4+5+6+7+8+9+10)/10 = 5.5$$

Average Case Time for Serial Search

Generalize for array size *n.*

Expression for average-case running time:

$(1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2$

Therefore, average case time complexity for serial search is O(n).

Binary Search

- Perhaps we can do better than O(n) in the average case?
- Assume that we are give an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

Binary Search

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

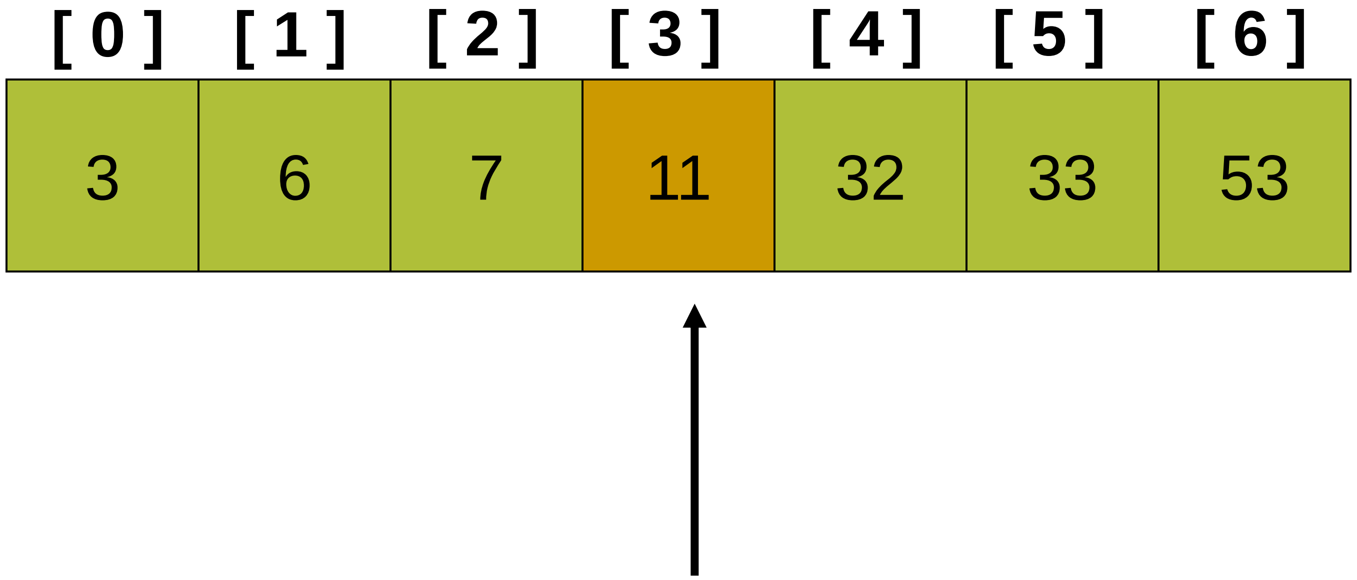
Binary Search

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

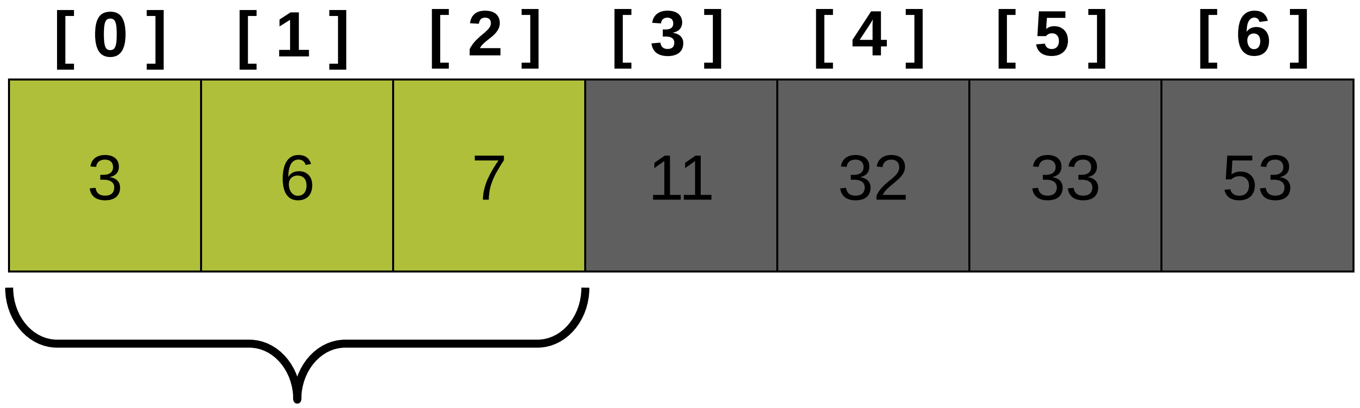
Binary Search

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

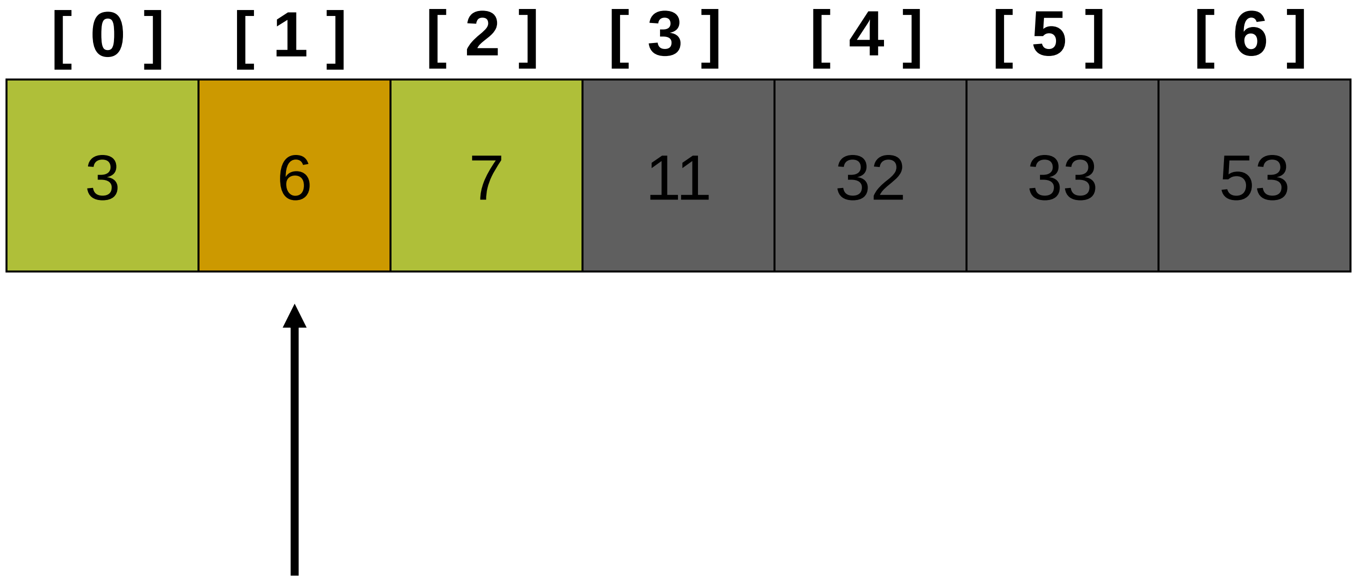
Binary Search



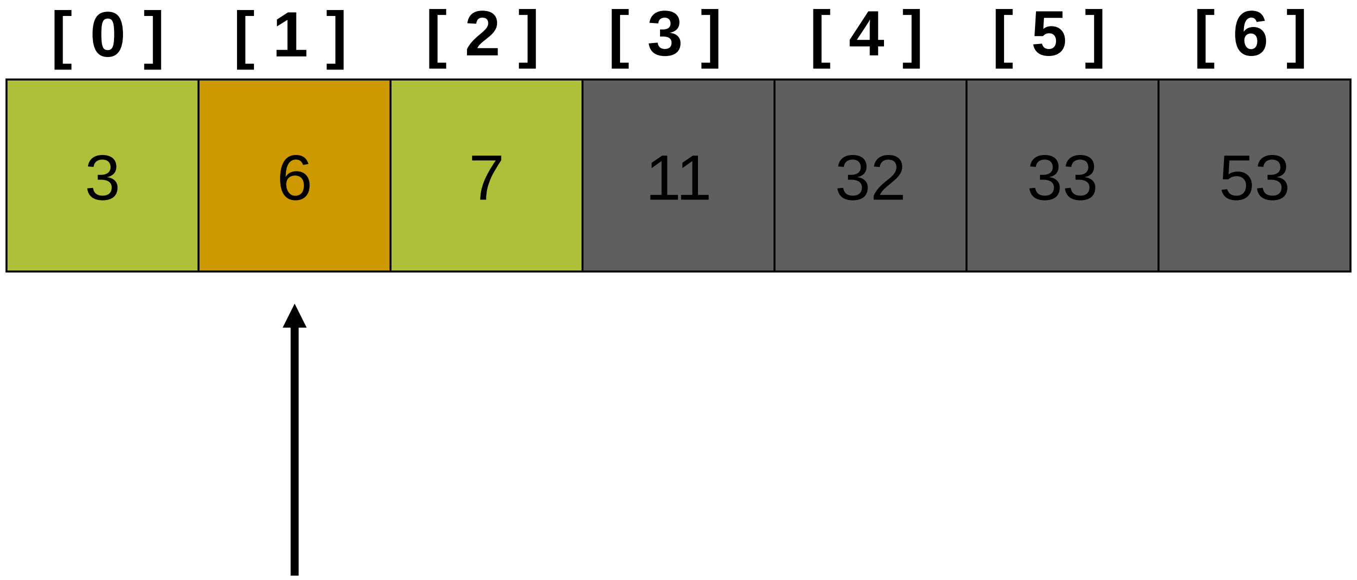
Binary Search



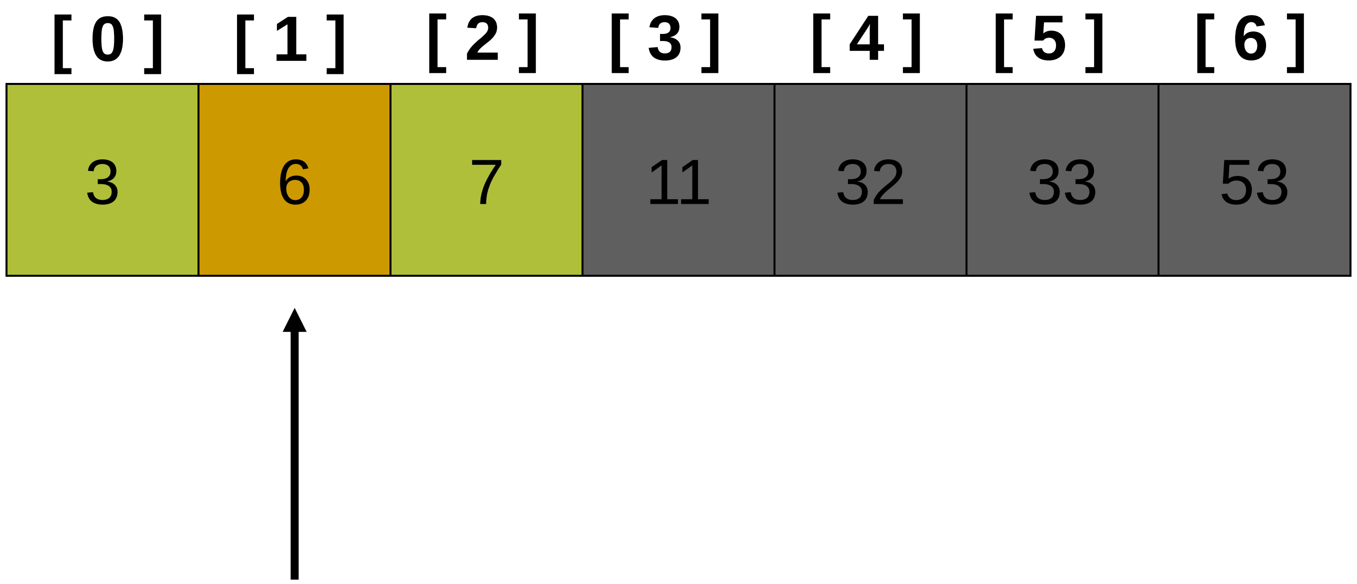
Binary Search



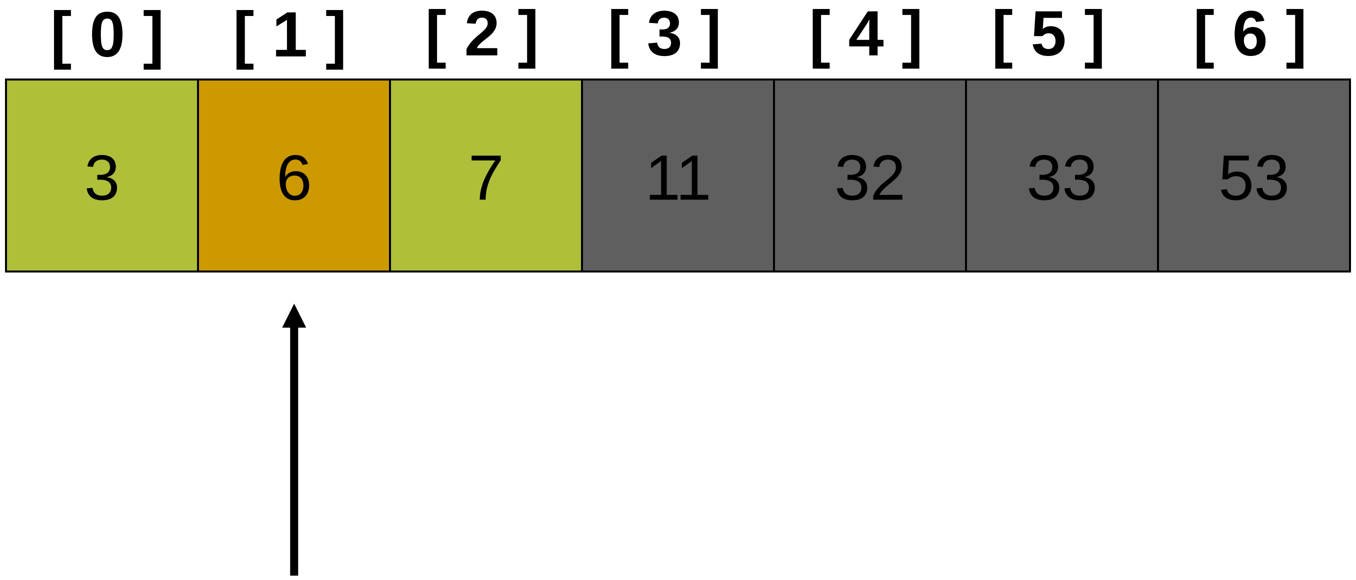
Binary Search



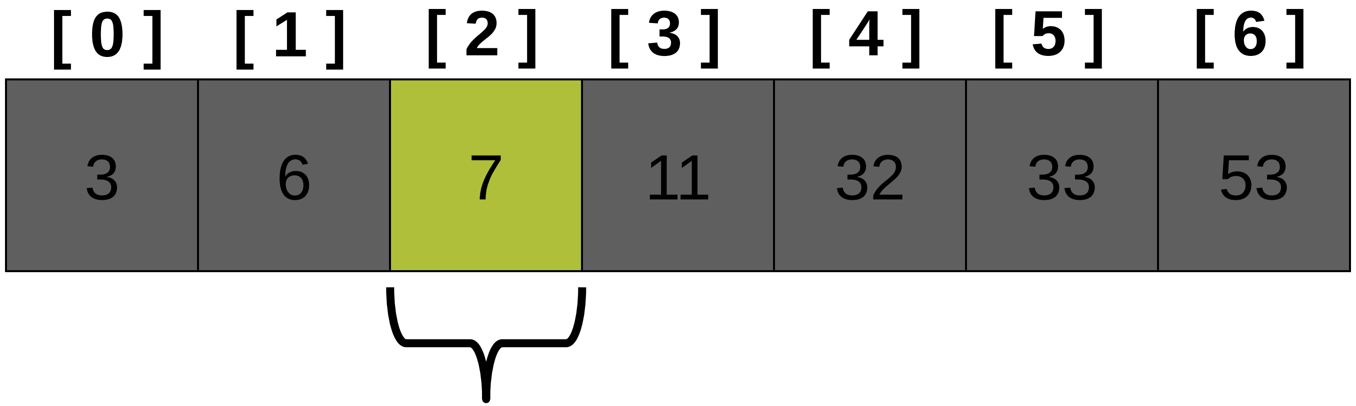
Binary Search



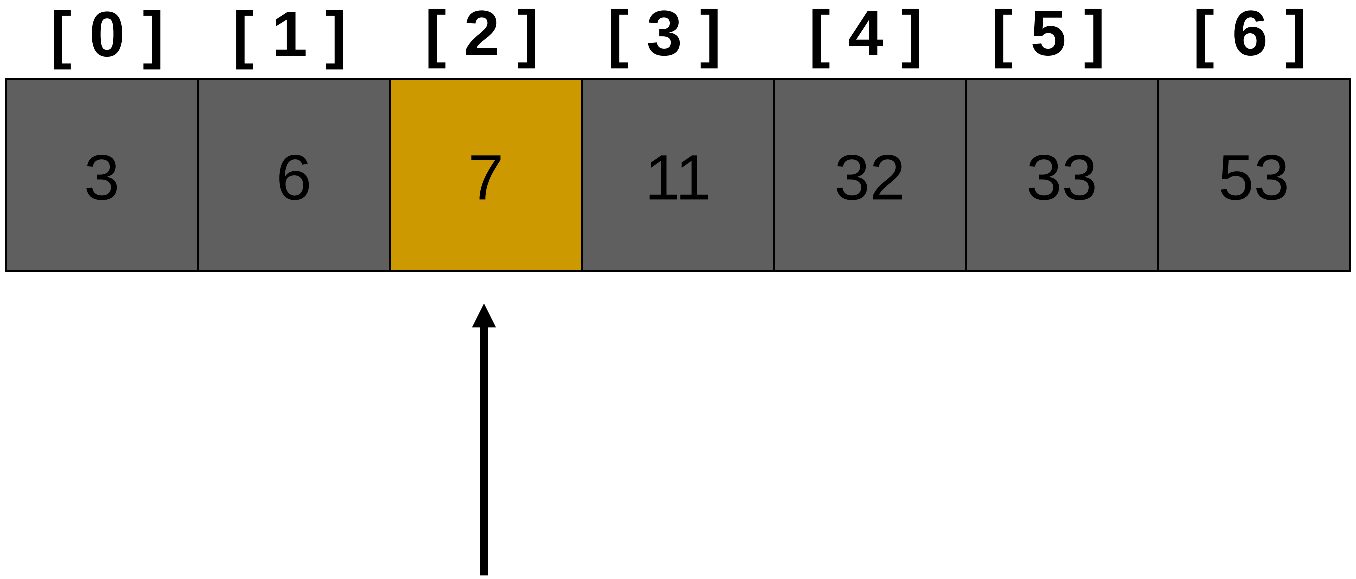
Binary Search



Binary Search



Binary Search



Recursive binary search (cont'd)

- What is the *size factor*?

  The number of elements in (array*[first]* ... *array[last]*)

- What is the *base case(s)*?

  (1) If *first* > *last*, return -1

  (2) If target==*array[mid]*, return mid

- What is the *general case*?

  if target < *array[mid]* search the first half

  if target > *array[mid]*, search the second half

Binary Search (non-recursive)

```
int BinarySearch ( array[ ], target) {
   int first = 0;   int last = array.length-1;
   while ( first <= last ) {
         mid = (first + last) / 2;
         if ( target == array[mid] ) return mid;  // found it
         else if ( target < array[mid] )  // must be in 1st half
               last = mid -1;
         else // must be in 2nd half
               first = mid + 1
    }
    return -1; // only got here if not found above
}
```

Binary Search (recursive)

```
int BinarySearch ( array[ ], first, last, target) {
    if ( first <= last ) {                // base case 1
            mid = (first + last) / 2;
            if ( target == array[mid] ) // found it! // base case 2
                return mid;
            else if ( target < array[mid] )  // must be in 1st half
                    return BinarySearch( array, first, mid-1, target);
            else // must be in 2nd half
                    return BinarySearch(array, mid+1, last, target);
    }
    return -1;
}
```

- No loop! Recursive calls takes its place
- Base cases checked first? (Why? Zero items? One item?)

Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of n?
- Each level in the recursion, we split the array into two halves (divide by two).
- Therefore maximum recursion depth is floor($\log_2 n$) and worst case = $O(\log_2 n)$.
- Average case is also = $O(\log_2 n)$.

Can we do better than $O(\log_2 n)$?

- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$
- Can we do better than this?

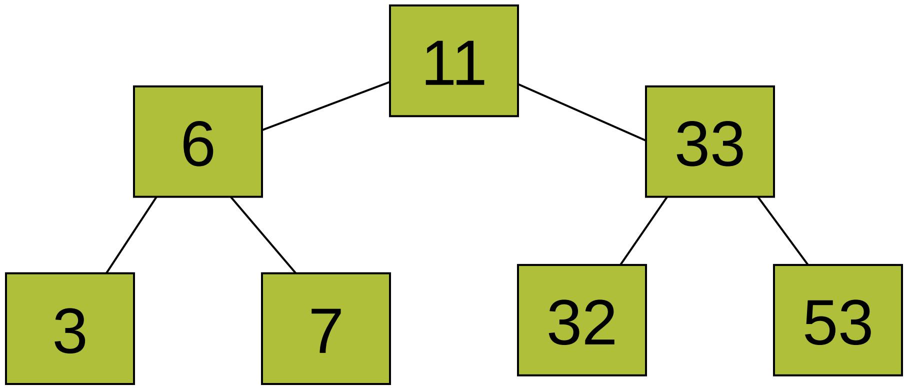
  YES. Use a hash table! (Will be taught later)

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

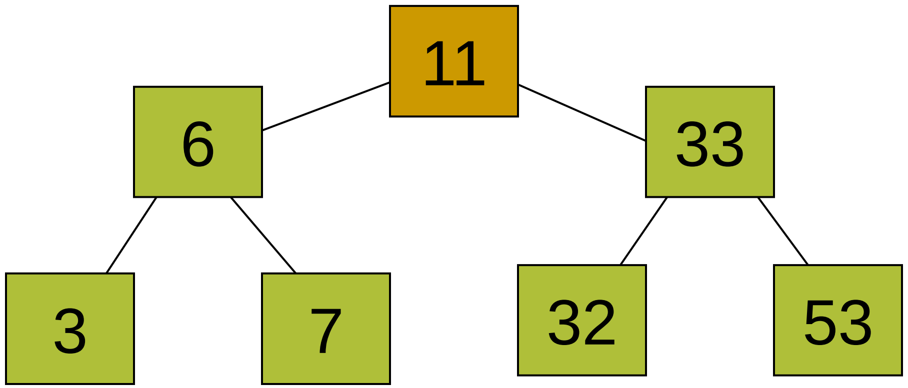


Search for target = 7

Find midpoint:

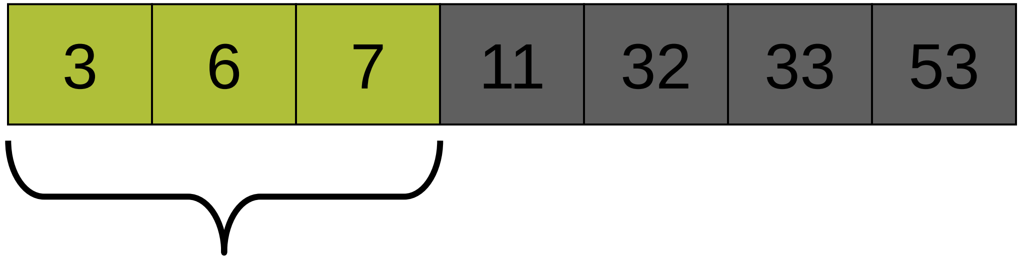
3	6	7	11	32	33	53

Start at root:

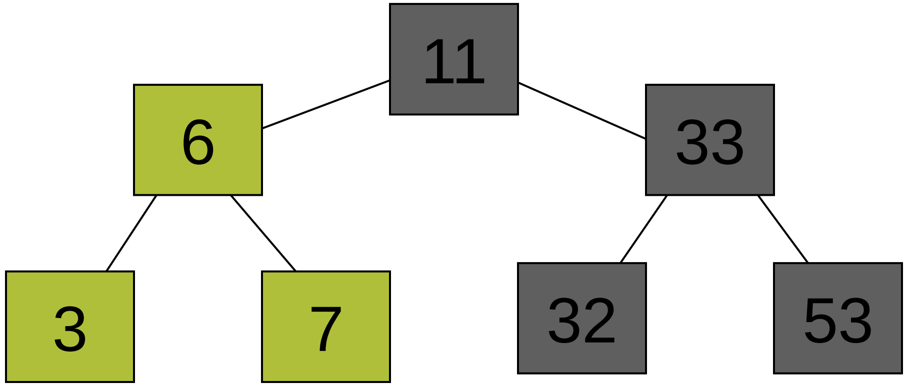


Search for target = 7

Search left subarray:

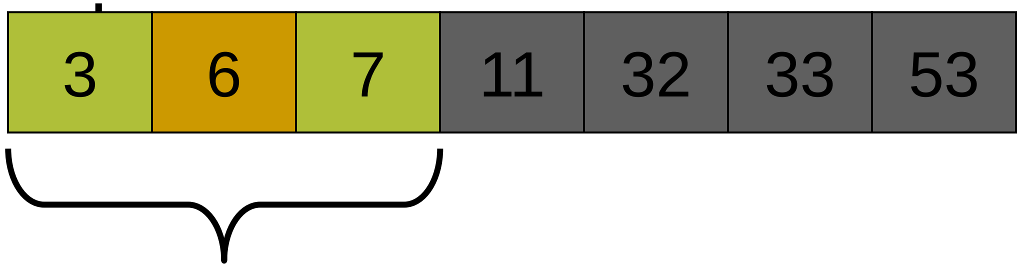


Search left subtree:

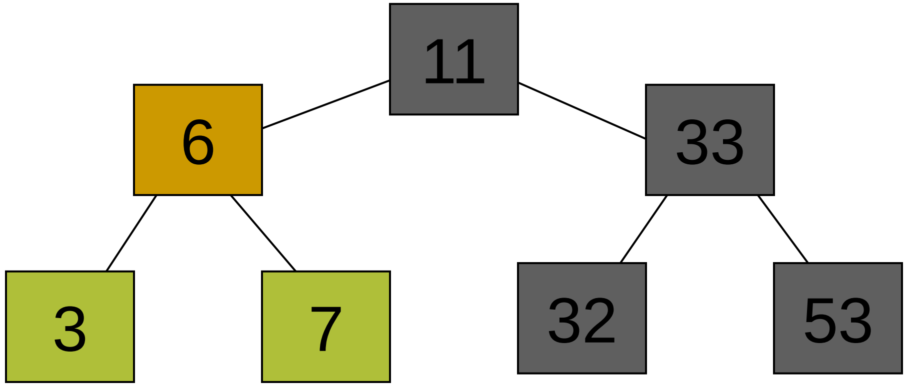


Search for target = 7

Find approximate midpoint of

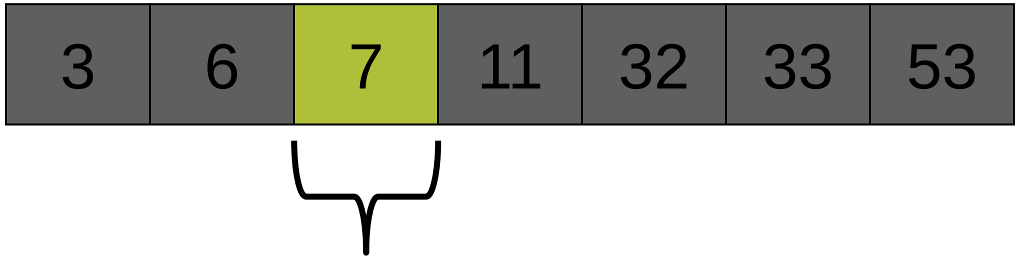


Visit root of subtree:

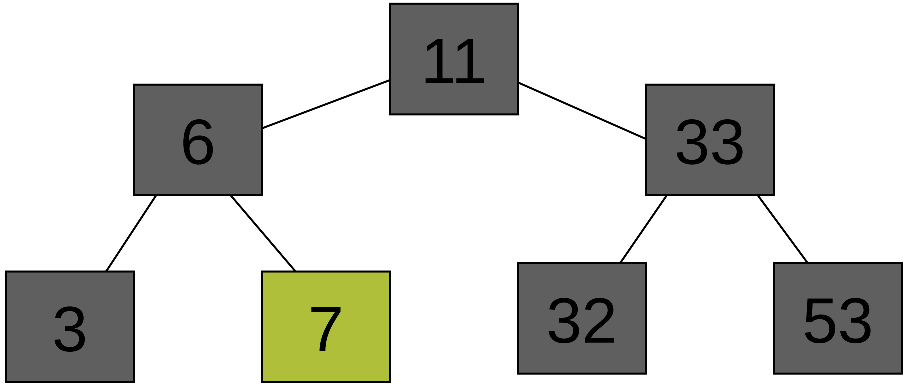


Search for target = 7

Search right subarray:



Search right subtree:



Time Complexity

- Remember our motive behind determining step counts:
  - to be able to compare the time complexities of two algorithms that compute the same function
  - to predict the growth in the runtime as the instance characteristics
- Determining the exact number of instructions is not a worthwhile exercise
- But when the difference between them of two algos is very large (say,3n+2 vs 100n+10); we may safely predict that the algo with complexity 3n+2 will run in less time than the algo with 100n+10 complexity
- But even in this case, it is not necessary to know that the exact step count is 100n+10. Something like, "it's about 80n or 85n or 90n ," is adequate to arrive at the same conclusion

Time Complexity

- Example: algo A: complexity $c_1n^2+c_2n$ and

  algo B: complexity c_3n

  - Algo B will be faster than algo A for sufficiently large values of n
  - For small values of n, either algo could be faster (depending on c_1, c_2, c_3)
    - $c_1=1$, $c_2=2$, $c_3=100$, then $c_1n^2+c_2n \le c_3n$ for **$n \le 98$**
    - $c_1=1$, $c_2=2$, $c_3=1000$, then $c_1n^2+c_2n \le c_3n$ for n **≤ 998**
  - Break-even point

Time Complexity

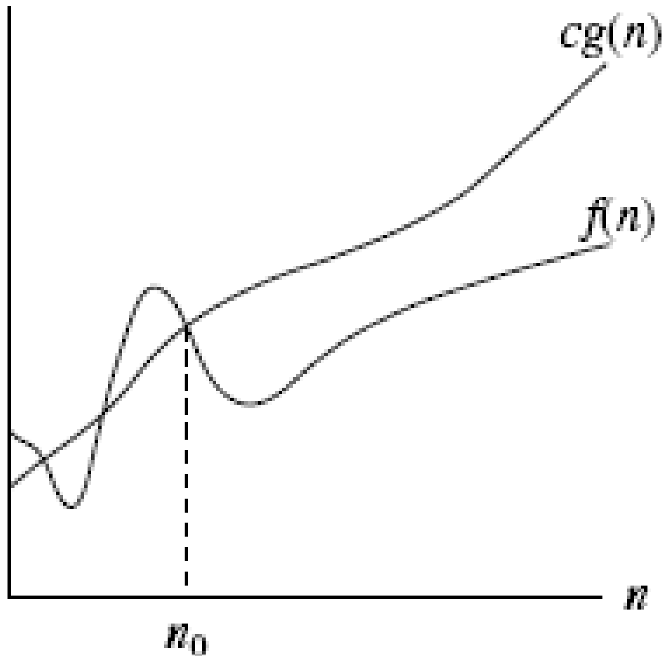
- One more simplifying abstraction: Order of growth
  - We consider only the leading term in the formula, since lower order terms are relatively insignificant for large n
  - We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs
  - Thus we say that time complexity of algo sum (T(n)= 2n+3) is *O*(n) (picking the most significant term: n)
  - We usually consider one algo to be more efficient than other if its worst case running time has a lower order of growth
  - For large enough inputs, a $O(n^2)$ algo runs more quickly in the worst case than a $O(n^3)$ algo

Growth of Functions: Asymptotic Notations

- A terminology has been introduced to enable us to make meaningful statements about the time complexity of an algorithms
- Usually, an algo that is asymptotically more efficient will be the best choice for all but very small inputs
- Several types of asymptotic notations

O-Notation (Big "oh" Notation)

- Asymptotic upper bound
- The function f(n)= O(g(n)) : read as "f of n is big oh of g of n"
- The function f(n)= O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$



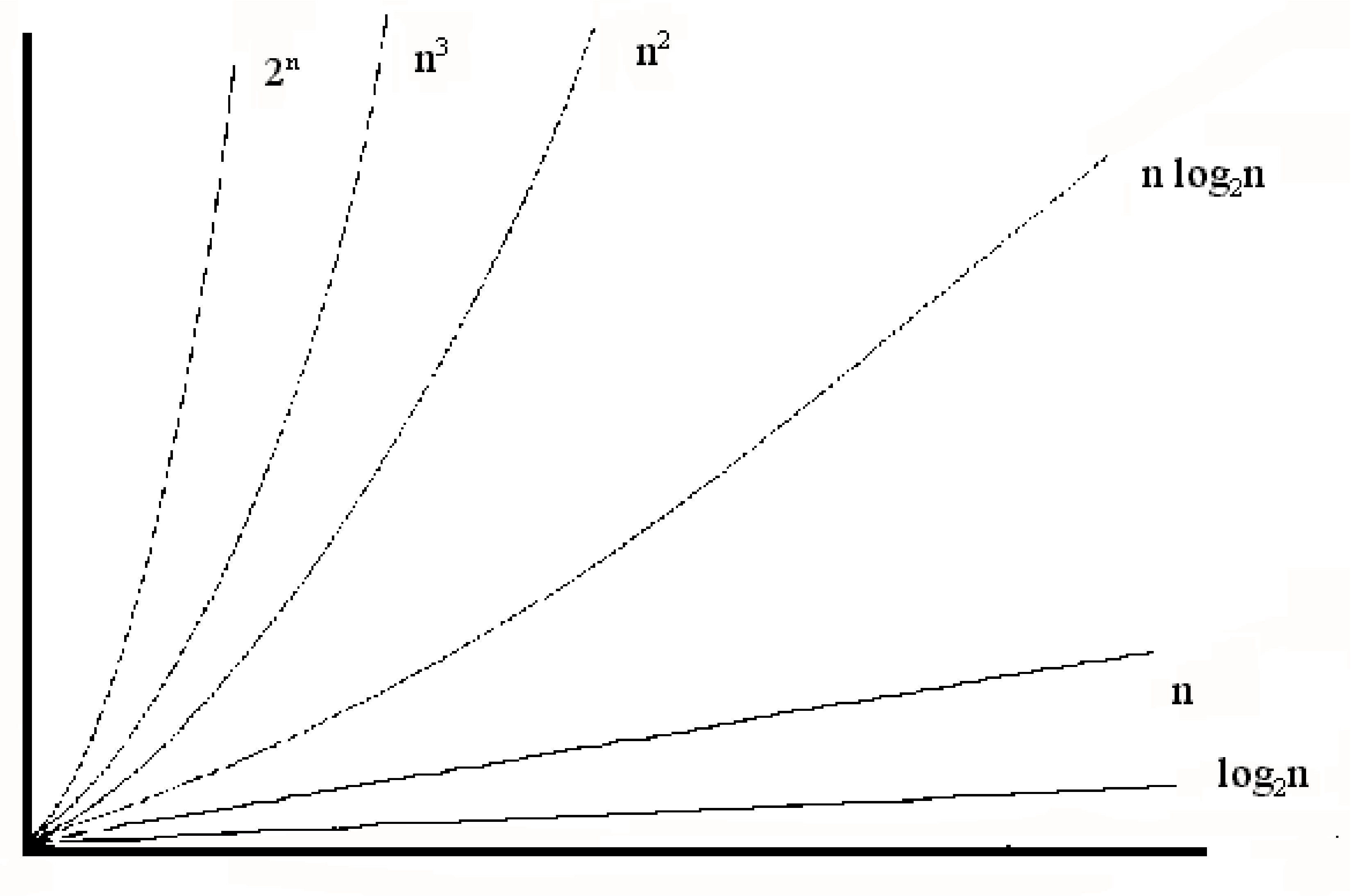
O-Notation

- Example: consider the function: $3n+2 = O(n)$
  - $3n+2 \leq 4n$ for all $n \geq 2$
- $10n^2+2n+4 = O(n^2)$ as $10n^2+2n+4 \leq 11n^2$ for all $n \geq 5$
- $6*2^n+n^2 = O(2^n)$ as $6*2^n+n^2 \leq 7*2^n$ for all $n \geq 4$
- $2n+4 \neq O(1)$
- $10n^2+2n+4 \neq O(n)$
- The statement $f(n)= O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$. It does not say anything about how good this bound is.

Function values

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Common Growth Rates



References:

- Slides and figures have been collected from various Internet sources for preparing the lecture slides of IT2001 course.
- I acknowledge and thank all the authors for the same.
- It is difficult to acknowledge all the sources though.