Algorithm Design & Problem Solving

Week5 Tutorial

Contents

Week 5 – Lab 4

1. Write a program to generate M random integers and put them in an array, then check how random your random number generator is!
2. Generate another sequence of random numbers N and count how many times it occurs in the array – using a sequential search.
3. Run your program for M= 10, 100, 1000 and N= 10, 100, 1000, timing the result using clock()

Week 5 – Lab 4

```
//Initialise array M&N for number of elements S
For i = 1 to S
        M[i]=random()
        N[i]=random()
        R[i]=0 //hold the result of number count
End for
For i= 1 to S               // for each number in M
        for j = 1 to S      // count how many times it occurs in N
                if M[i]==N[j] R[i]++
        end for
End for
// print out the result
For i= 1 to S
        if R[i]>0 print "Number M[i] is in N R[i] times"
End for
```

$O(N^2)$

Week 5 – Lab 4

```
//to do the search recursively, replace the inner loop of the search with
// a recursive function
For i= 1 to S                          // for each number in M
        R[i]=doNCount(N,M[i],S)// count how many times it occurs in N
End for
// count how many times c exists in N[Max]
Function doNCount(Array N, number c, number Max)
        if Max<=0
                return 0
        else if N[Max] == c
                return 1+doNCount(N,c,Max-1)
        else
                return doNCount(N,c,Max-1)
        end if
End function
```

Binary search

- We have seen the linear (sequential) search method.
- Complexity of algorithm is about n/2, i.e. O(n).
- Is there a better algorithm, i.e. more efficient in terms of operations/time.
- The Binary search is such a method.

Binary search

- We consider Binary search for a 1D array of numbers.
- The Binary search requires the data to be in sorted order in the array.

- Imagine looking for a number in the telephone directory if it was not sorted by name.
- We would need to start at the beginning, and do a linear search, very inefficient and slow.

Binary search

- Because **telephone directory is sorted**, we can search it much faster.

- Binary search works in a **similar way** to this.

- We assume the data in the array is sorted, will look at sorting algorithms later.

Binary search

- Element we search for is called the key.
- Array to search is A(n) of size n.

- Let i = 1st index, j = last index=n.
- Compute the middle index k = (i+j)/2
- Check if the key = A(k)?
- If it is, we have found the key, at position =k.
- If key <> A(k) then either the key is in the 1st half of the array or the second half, since the array is sorted.

Binary search

- If key < A(k) we search the sub-array: A(0),A(1),…,A(k-1)
- If key > A(k) we search the sub-array:

  A(k+1), …, A(n – 1)

- Each of these subarrays is about half the size of the original array.

- We apply this same method to the smaller array, since each one is sorted, looking at the middle element first.
- We continue until the element is found, or it isn't in the array.

Binary search

Algorithm. int binary_search(A, n, key)

```
pos = -1                        // assume element not found
i := 0                          //  min index  (initially)
j := n - 1                      //  max index  (initially)
while (i <= j and pos = -1 )
   k = (i + j)/2
   if (key = A(k) ) then       // element found
     pos = k                   // element in k-th position
   else if (key < A(k)) then
      j := k – 1               // last index in 1st half.
   else
       i := k + 1              // 1st index in 2nd half.
  end if
 end while
 return pos
 end binary_search
```

Binary search 1D array

- Search array A for key = 5.
- middle index k = 3. key <> A(3), so
- Search 1st half, i =0, j = k – 1 = 2, k = (0+2)/2=1
- Example A(*):

i			k			j
1	3	5	7	9	11	13

i	k	j
1	3	5

i, j
5

Binary search 1D array

- Let's look at complexity of the Binary search.
- Easiest when $n = 2^k - 1$. Then the two halves each have

  $(n-1)/2 = (2^k - 1 - 1)/2 = (2^k - 2)/2 = 2^{k-1} - 1$ elements.

- Three cases to consider:
  - Best case
  - Worst case
  - Average case
- Best case: 1st element, 1 comparison
- Worst case: The array is halved at each step.
- Eventually the array will have only 1 element.

Binary search 1D array

- After step 1: $n = 2^k - 1$, for some integer k.
- After step 2: $2^{k-1} - 1$
- After step 3: $2^{k-2} - 1$
- After step k: $2^{k-(k-1)} - 1 = 1$
- So after k steps array has only 1 element.
- At most k comparisons are needed.
- $n = 2^k - 1$, $n + 1 = 2^k$
- $\log_2(n+1) = \log_2(2^k) = k$
- $k = \log(n+1)$ comparisons.

Binary search 1D array

- So at most log(n+1) operations are needed.
- For n = 7, log(n+1) = log(8)=3
- For n = 15, log(n+1) = log(16) = 4.
- When n is not = $2^k - 1$, algorithm still works.
- Average case is also log(n+1) operations.
- This is a lot better than linear search.
- log(n+1) << n/2, for large n

Binary search 1D array

- **Example.**
- If $n = 1023 = 2^k - 1 \quad k = 10$
- Linear search: 512 comparisons on average.
- Binary search: 10 comparisons at most.

- Exercises: Find maximum number of comparisons if:
- N = 31, 32, 100, 200, 500, 1000

Example Array: 7 elements

	low			mid			high
i	0	1	2	3	4	5	6
A[i]	10	15	20	25	30	35	40
	10	15	20		30	35	40
	10		20		30		40

BIG-O: Performance Analysis of Algorithms

Constant Time vs Linear Time for an Operation

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes 5000 time units
  - Algorithm B takes 100 * n time units
- Which is better?
  - Algorithm B is better IF our problem size is small, that is, if $n < 50$
  - Algorithm A is better for larger problems, with $n > 50$
- We usually care most about very large problems

Big-O Notation

- To simplify the running time estimation, for a function *f(n),* we ignore the constants and lower order terms.
- When we have a polynomial that describes the time requirements of an algorithm, we simplify it by:
  - Throwing out all but the highest-order term
  - Throwing out all the constants
- E.g. If an algorithm takes $\mathbf{C*n^2 + D*n + E}$ time, we simplify this formula to just $\mathbf{n^2}$
- =>algorithm requires $O(N^2)$ time => this is Big-O notation

Big-O Notation

- Examples:
  - $7*n-2$
  - $3*n^3+20*n^2+5$
  - $3*\log n + 10$
- Compute the complexity time for each algorithm
- The big-Oh notation gives an upper bound on the growth rate of a function
- We can use the big-Oh notation to rank functions according to their growth rate

- Big-O notation is a huge simplification; can we justify it?
  - It only makes sense for large problem sizes
  - For sufficiently large problem sizes, the highest-order term swamps all the rest!
- Consider $F(n) = n^2 + 3n + 5$:

	n=1	n=10	n=100	b=1000
5	5	5	5	5
3n	3	30	300	3000
n^2	1	100	10000	1000000
F(n)	9	135	10305	1003005

2.5 Big-O Notation

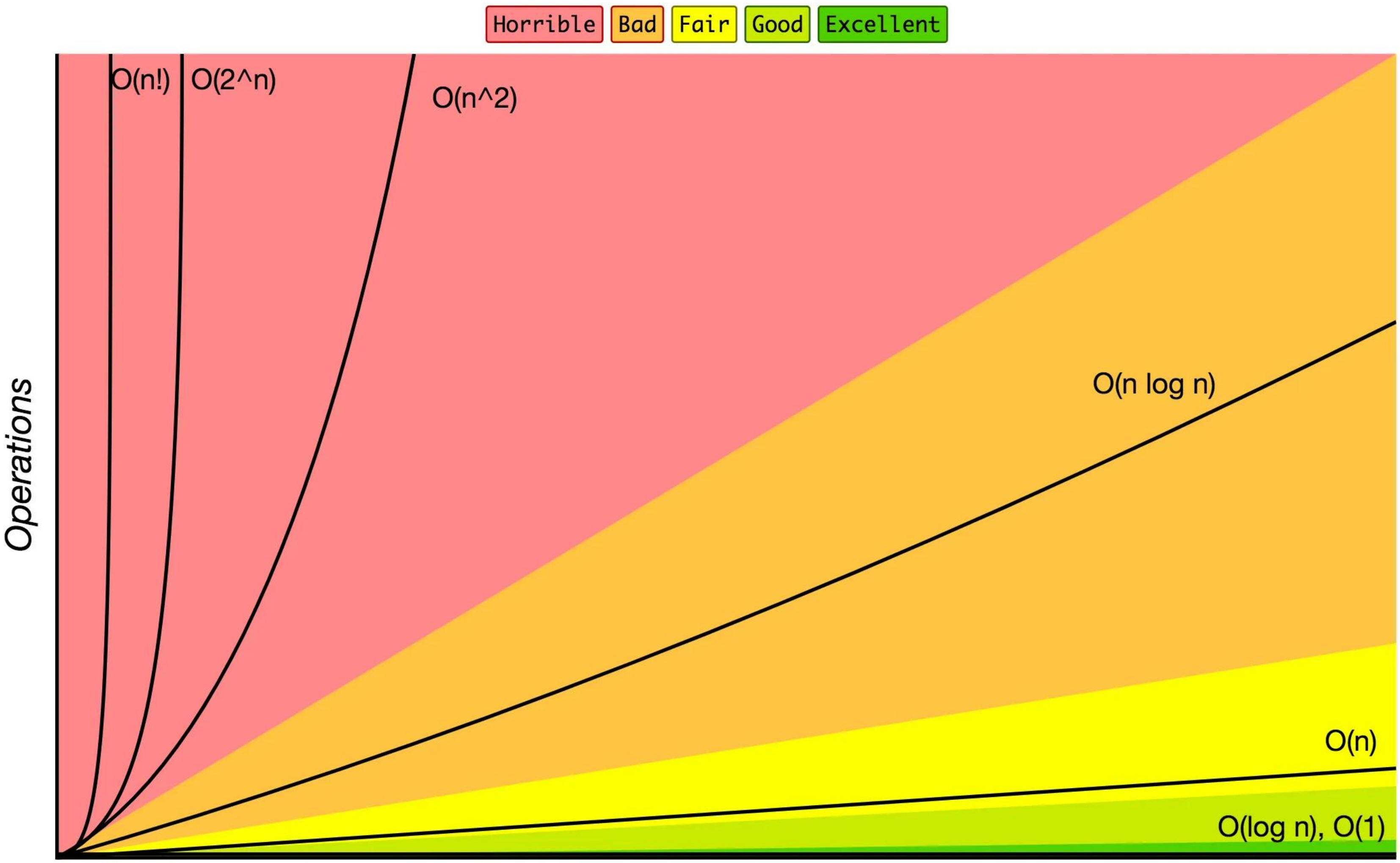
Common Time Complexities

BETTER

$O(1)$	constant time	“order 1 “
$O(\log N)$	log time	
$O(N)$	linear time	“order N”
$O(N \log N)$	log linear time or *linearithmetic*	
$O(N^2)$	quadratic time “order N squared “	
$O(N^3)$	cubic time	
$O(2^N)$	exponential time	

WORSE

BIG-O Cheatsheet (http://bigocheatsheet.com)



Big-O Notation

How to Determine Complexities ?

- How can you determine the running time of a piece of code ?
  - Answer: In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.
- 1. Sequence of statements
  - statement 1;
  - statement 2;
  - ...
  - statement k;

- Total Running Time => adding the times for all statements
  - Total Time = time(statement 1) + time(statement 2) + ... + time(statement k)
  - If each statement is "simple" (only involves basic operations) => the time for each statement is constant => O(1)
  - Total time is also constant => O(1).

- 2. if-then-else statements

```
if (cond) then
      block 1 (sequence of statements)
else
      block 2 (sequence of statements)
end if;
```

either sequence 1 will execute, or sequence 2 will execute

- The worst-case time is the slowest of the two possibilities

MAX(TIME(BLOCK 1), TIME(BLOCK 2))

  - For example, if sequence 1 is O(N) and sequence 2 is O(1)
  - the worst-case time for the whole if-then-else statement would be O(N).

- 3. FOR loops

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

The loop executes N times,
So, the sequence of statements also executes N times

- Total Running Time => N* time(sequence)
  - For example, we assume the statements are O(1)
  - => Total time for the for loop is N * O(1), which

- 4. Nested FOR loops

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

The outer loop executes N times
Every time the outer loop executes => the inner loop executes M times

- The statements in the inner loop execute a total of N * M times
- Total Running Time => N* M *time(sequence)
  - E.g. If we assume the statements are O(1) => the complexity is O(N * M)
- if we change stopping condition of the inner loop from j < M to j < N
- Total complexity for the two loops is O(N2).

5. Statements with method calls

- When a statement involves a method call, the complexity of the statement includes the complexity of the method call.

```
f(k);   // O(1)
g(k);   // O(N)
```

- When a loop is involved, the same rule applies

```
for (j = 0; j < N; j++)
    g(N);
```

- It has complexity (N^2).
- The loop executes N times
- Each method call g(n) is complexity O(N).

Big-O Notation
Properties

- Ignore low-order terms
  - E.g., $O(n^3+4n^2+3n)=O(n^3)$
- Ignore multiplicative constant
  - E.g., $O(5n^3)=O(n^3)$
- Combine growth-rate functions
  - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - E.g., $O(n^2) + O(n*log_2n) = O(n^2 + n*log_2n)$
    - Then, $O(n^2 + n*log_2n) = O(n^2)$

Give the order of growth (as a function of N) of the running times of each of the following code fragments:

```
int sum = 0;
for (int k = N; k > 0; k --)
   for (int i = 0; i < k; i++)
      sum++;
```

```
int sum = 0;
for (int i = 1; i < N; i = i*2)
   for(int j = 0; j < i; j++)
      sum++;
```

```
int sum = 0;
for (int i = 1; i < N; i += 2)
   for (int j = 0; j < N; j++)
      sum++;
```

Quiz ToH Q

```
moveTower (disks, source, dest, spare)
If disk = 0
    Move disk from source to dest
Else
    moveTower (disk-1, source, spare, dest)
    move disk from source to dest
    moveTower (disk-1, spare, dest, source)
```

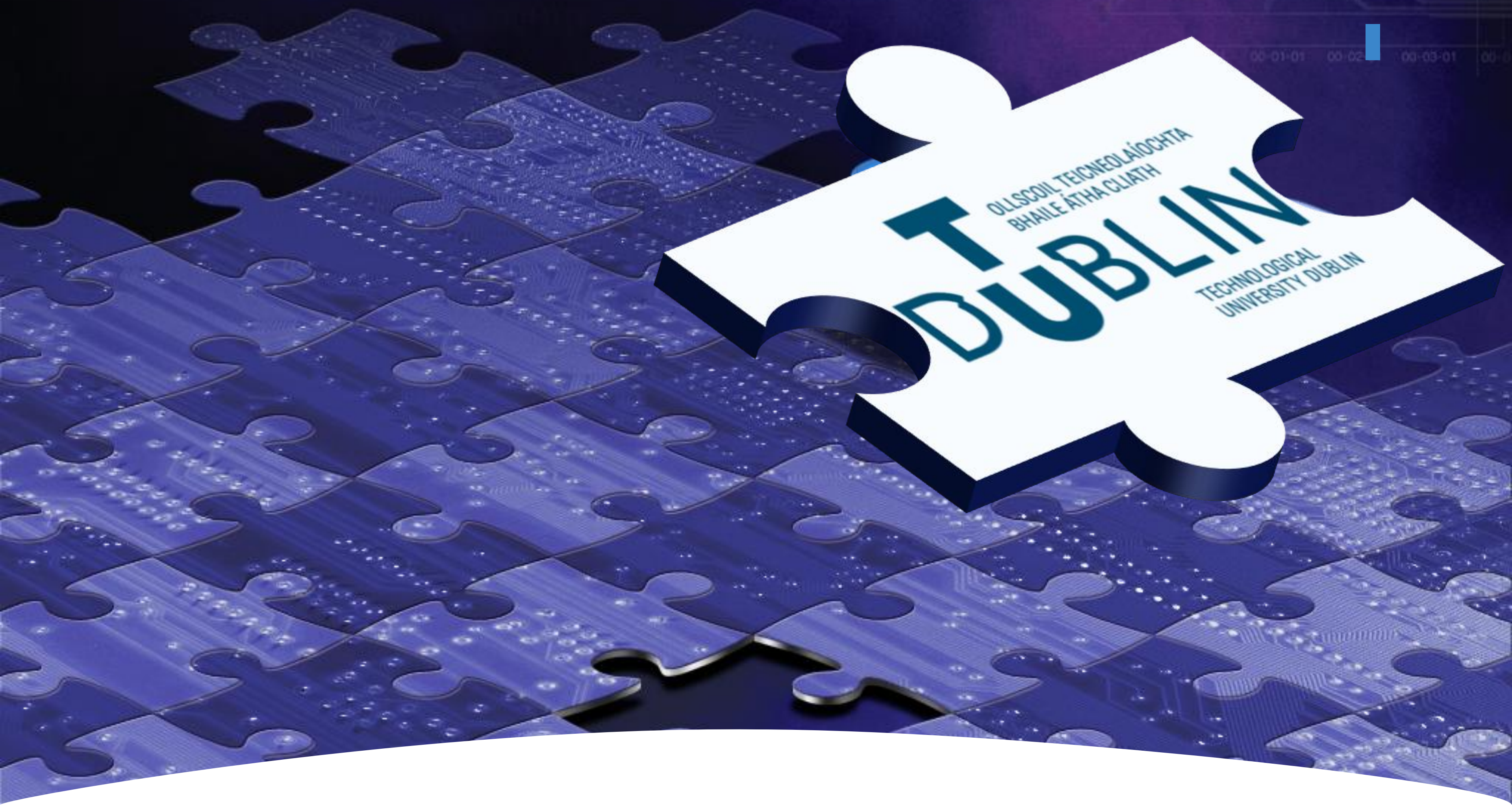
What does the second recursive call in the TOWER OF HANOI algorithm do?

a) Requesting to move a particular disk from the same tower
b) Requesting to move a particular disk from a different tower
c) None of the above

What does the first recursive call do in the TOWER OF HANOI algorithm?

a) Requesting to move a particular disk from the same tower
b) Requesting to move a particular disk from a different tower.
c) Requesting to move a particular disk from a different tower.

Thank You !


OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL
UNIVERSITY DUBLIN