[FIT2004 Sample Notes- Algorithms]

Highlighted yellow means its recommended you know these areas in depth- they are likely to show on the mid-sem and final exam

Week 1:

Lecture and tute participation: 10%
Mid-sem test: 10%
Assignments: 20%
Final exam: 60%

1. Proving correctness of algorithm (loop invariant)

A loop invariant is:
In simple words, a loop invariant is some predicate (condition) that holds for every iteration of the loop (start and end of the loop). For example, let's look at a simple for loop that looks like this:

```
int j = 9;

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

In this example it is true (for every iteration) that i + j == 9. A weaker invariant that is also true is that i >= 0 && i <= 10.

To prove correctness, we know to show that an algorithm:

- Always terminates, and
- It returns the correct result when it terminates

2. How to write a recurrence relation from a piece of code

We want to know how to do this, as it's a way for us to determine the complexity of an algorithm.

A recurrence relation has a: base case + inductive case

Take this code:

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

Base case: T(1) = b, where b is a constant

Our program terminates when we return 1 or x, this is done in constant time, hence b. T(1) is garnered from the case (N== 1), but T(0) = b is also acceptable as the base case.

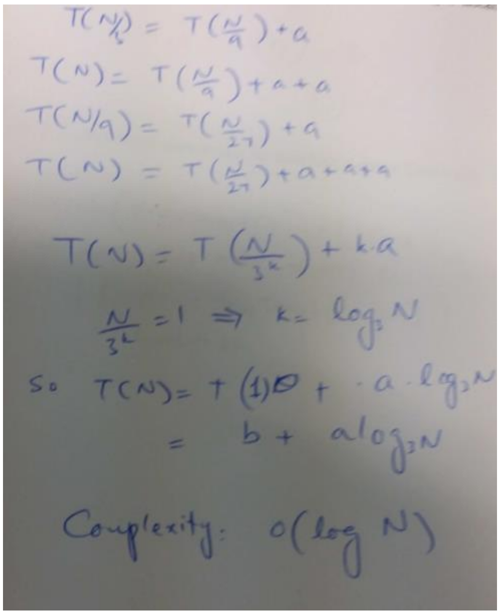
Inductive case: T(N) = T(N-1) + c, where c is a constant

- Look at power(x, N-1), this is T(N-1)
- X * power(x, N-1) takes a constant operation, hence the +c. Almost always expect the +constant when analyzing a piece of coding in the inductive case.

3. Solving a recurrence relation

Take for example this recurrence relation, where b and a are constants. To solve it via **backwards substitution:**

$$T(N) = \begin{cases} T(N/3) + a, & \text{if } N = 3^k \text{ where } k > 0. \\ b & \text{if } N = 1 \end{cases}$$



Also know how to do: Proof by induction. it was not asked in my mid-semester or final exam, but it may appear in yours/a good thing to know.

Week 2:

Comparison based sorting algorithms;

Blue means in these examples that the numbers have been sorted

Selection sort: Find min element in unsorted each time
Example:
Given this array below
2 8 5 3 9 4

2 | 8 5 3 9 4 <- 2 is min element in the array
2 3 | 5 8 9 4 <- swap 3 and 8
2 3 4 | 8 9 5 <- swap 4 and 5
2 3 4 5| 9 8 <- swap 5 and 8
2 3 4 5 8| 9 <- swap 8 and 9

Insertion sort: Place next element in unsorted array to sorted
Example array:
Given this array below

2 8 5 3 9 4
2 | 8 5 3 9 4
2 8 | 5 3 9 4
2 5 8 | 3 9 4
2 3 5 8 | 9 4
2 3 5 8 9 | 4
2 3 4 5 8 9

Summary of comparison-based sorting algorithms

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	O(N)	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	O(N log N)	O(N log N)	O(N log N)	No	Yes
Merge Sort	O(N log N)	O(N log N)	O(N log N)	Yes	No
Quick Sort	O(N log N)	$O(N^2)$ – can be made O(N log N)	O(N log N)	Depends	No

Non comparison-based sorting algorithm:

a) Counting sort: sorts a word alphabetically
- O(n+d) complexity, d is the size of count array for integers

- O(n) complexity for sorting alphabets

Example of what algorithm does (for alphabets): Sort BAAD alphabetically
Create a list of size 26, each letter corresponds to each letter of the alphabet
A List[0] = 2
B List[1] = 1
C List[2] = 0
D List[3] = 1
….

For x in list
    Append character list[x] number of times

Output: AABD

b) Radix sort: O(MN) complexity, N is number of words, M is the length of each word
Sorts multiple words alphabetically.

Example = CAT, ARC, TAC, ART
Example of what the algorithm does:

Sort on third column
TAC
ARC
ART
CAT

Sort on second column
TAC
CAT
ARC
ART

Sort on first column
ARC
ART
CAT
TAC

Final sorted alphabetical list is thus: ART, ARC, CAT, TAC