# UoL logoUniversity of London

# Assessment Coversheet

Complete this coversheet and read the instructions below carefully.

**Candidate Number**:

DP0669

**Degree Title**:

BSc Computer Science

**Course/Module Title**:

Fundamentals of Computer Science

**Course/Module Code:**

CM1025

**Enter the numbers, and sub-sections, of the questions in the order in which you have attempted them:**

**Question 1: a, b, c, d, e, f, g**

**Question 2: a, b, c, d, e, f**

**Date**: 24.09.2020

**Instructions to Candidates**

1. Complete this coversheet and begin typing your answers on the page below, or, submit the coversheet with your handwritten answers (where handwritten answers are permitted or required as part of your online timed assessment).
2. Clearly state the question number, and any sub-sections, at the beginning of each answer and also note them in the space provided above.
3. For typed answers, use a plain font such as Arial or Calibri and font size 11 or larger.
4. Where permission has been given in advance, handwritten answers (including diagrams or mathematical formulae) must be done on light coloured paper using blue or black ink.
5. Reference your diagrams in your typed answers. Label diagrams clearly.

**The Examiners will attach great importance to legibility, accuracy and clarity of expression.**

**Begin your answers on this page**

**PART B**

**Question 1**

1. Construct the truth table for :

The operator procedure order is: Negation (¬) > And (⋀) > Or () > If-Then(→) > If-And-Only-If (↔), then with respects to the brackets our truth table order is:

Let us construct our truth table (Tab.1):

**Table 1 – Truth table for**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **p** | **q** | **r** | **(1) = (p→q)** | **(2) = (p→r)** | **(3) = (1)⋀(2)** | **(4) = (q⋀r)** | **(5) = p→(4)** | **(6) = (3)↔(5)** |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

1. Determine if the formula in part (a) is a tautology or a contradiction and explain your reasoning

Tautology is a formula or assertion that is true in every possible interpretation. Using our truth table, we can confirm that initial statement is tautology, because no matter the values of p, q and r, the statement itself will always be true (For all rows - all possible combinations of values for p,q,r - the values in the final column are “1”).

1. Write the negation of p → q:

The statement p → q is the implication. We can think of implication as of promise – it only become “Not True” or “False” if the premise (p) is “True”, but the conclusion (q) is “False”. Thus, we can conclude that this statement is only false when p is “True” and q is “False”. We construct the truth table to confirm (Tab.2):

**Table 2 – Truth table for**

|  |  |  |
| --- | --- | --- |
| **p** | **q** | **p → q** |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

Then, the negation of this statement must be “True” when the initial statement is “False”, and vice versa. Therefore, we can “reverse” the truth table of (p → q) and get the following (Tab.3):

**Table 3 – Adding negation of the**

|  |  |  |  |
| --- | --- | --- | --- |
| **p** | **q** | **p → q** | **¬ (p → q)** |
| 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |

The only option when this negation is true is presented in the row 2, and we can rephrase it as: “Negation is true, only when p is “True” and q is “False”. The statement p is “True” is can be simplified as – p, and q is “False” as – not q or ¬q. Therefore, we can say that negation of (p → q) is (p AND ¬q) or in the symbols:

1. Each student has a password, which is SIX characters long. Each character is either a digit or a lowercase letter. Each password must contain at least TWO digits. How many possible passwords are there? Show your working:

Let us summarize: 1) There are six slots for the password symbols, 2) There are 26 lowercase letters, and 10 digits, 3) Each password must contain AT LEAST TWO digits. We can rephrase the last condition as 3\*) The only passwords from all possible combinations that doesn’t satisfy our needs the one’s without digits, or with only one digit in them.

So, we can find the number of possible passwords as N = Nall – N0 dig. – N1 dig.

1. If we use all symbols (10 + 26) there are 36 possible variants for one slot, then for six slots (Condition 1) there are Nall = 36\*36\*…= 366.
2. The Password without digit’s has only letter’s => 26 possible variants for one slot, similarly: N0 dig. = 266.
3. The Password with only one digit has 10 possible variants for one slot, and 26 for 5 others. But there are 6 slots => six permutations => N1 dig. = 6 \* 10 \* 265.
4. N = Nall – N0 dig. – N1 dig = 366 – 266 – 6 \* 10 \* 265 = 2’176’782’336 – 308’915’776 – 712’882’560 = 1’154’984’000 (possible passwords)
5. There are 7 red balls, 6 blue balls and 5 green balls in a sack. What is the minimum number of balls one must take out of the sack to guarantee at least 4 balls of the same colour?

To answer this question, we can use the pigeonhole principle. The pigeonhole principle states that: “If N items are put into M containers, with N > M, then at least one container must contain more than one item”. In a more quantified version: “If N=kM+1 items are put into M containers, with N > M, then at least one container must contain at least k+1 items”. Therefore, we can rewrite our condition as: There are 3 containers (M = 3), and we need at least 4 balls (k+1 = 4, k = 3). What is the minimum number of items? From pigeonhole principle: N = kM+1 => N = 3\*3+1 = 10.

We can solve this question by simple reasoning. The worst case is that we will draw balls of consecutive colors, i.e.: “Red” =>” Blue” => “Green” => “Red” =>… Then, when after few before we will draw the last – fourth ball of one of the colors, we already have – 3 \* 3 = 9 balls (3 triplets of R – G - B), the next one – the tenth ball will guarantee that there are at least 4 balls of the same colour.

1. Use mathematical induction to prove n3 − n is divisible by 6, for all n ≥ 0.

Proof by induction consists of 3 basic steps:

1. Base case: For n = 0 we have: 03 – 0 = 0, 0 is divisible by everything, so the base case is true.
2. Then we make inductive hypothesis, that the statement holds for n. Therefore, we assume that n3 – n is divisible by 6 is true.
3. We try to prove using this hypothesis that the statement is also true for n = n+1. Then (n+1)3 – (n+1) is also must be divisible by 6. Let us prove it:

(n+1)3 – (n+1) = n3 + 3n2 + 3n + 1 – n – 1 = n3 – n + 3n2 + 3n = [n3 – n] + [3n(n+1)]

For whole sum to be divisible by 6 we must show that each term is also divisible by 6. From inductive hypothesis we know that [n3 – n] is divisible by 6, so we just need to show that [3n(n+1)] is also multiplier of 6. To prove that we again use induction:

1. Base case: For n = 0 we have: 3\*0 \* (0 + 1) = 0, 0 is divisible by everything, so the base case is true.
2. Then we make inductive hypothesis, that the statement holds for n. Therefore, we assume that [3n(n+1)] is divisible by 6 is true.
3. We try to prove using this hypothesis that the statement is also true for n = n+1. Then [3n(n+1)] is also must be divisible by 6. Let us prove it:

3(n+1) ([n+1]+1) = 3(n+1)(n+2) = 3(n2 + n + 2n + 2) = 3n2 + 9n+6 = [3n2 + 3n] + [6n + 6] = [3n(n+1)] + [6(n+1)]

From inductive hypothesis we know that [3n(n+1)] divisible by 6 is true, and [6(n+1)] is divisible by 6 is also true because it is can be expressed as multiplier of 6. Because both terms are divisible by 6 – the whole sum is divisible by 6, therefore we proved by induction that [3n(n+1)] is divisible by 6. Now, let us return to the first case. Know, because we proved that [3n(n+1)] is also multiplier of 6 we proved that (n+1)3 – (n+1) = [n3 – n] + [3n(n+1)] is divisible by 6, therefore we proved by induction that n3 – n is divisible by 6, for all n >= 0.

1. Use contrapositive to prove if n2 + 6n + 8 is odd then n is odd.

To prove by contrapositive, we use logical equivalence (p→q) ≡ (¬q → ¬p). Therefore, if we prove that negation of the conclusion is also results in negation of premise, we can say that initial implication was also true. In our case [n2 + 6n + 8 is odd] is premise p, and [n is odd] is conclusion q. To prove this implication by contrapositive let us show that negation of q [n is even, or n = 2m | m - integer] results in negation of p [n2 + 6n + 8 is even, or 2k | k - integer].

Let us rewrite the implication using assumption that ¬q is true => n = 2m. Then:

(2m)2 + 6(2m) + 8 = 4m2 + 12m + 8 = 2(2m2 + 6n + 4). If we assume that 2m2 + 6n + 4 = k (some integer), then we see that n2 + 6n + 8 = 2k (where k is integer) is even, therefore we proved that if ¬q is true, then ¬p is also true => (¬q → ¬p) is true, and by contrapositive implication [ If n2 + 6n + 8 is odd then n is odd (p→q) ] is also true.

**Question 2**

1. Heapify the following tree, make every step clear. (Min heap)

Изображение выглядит как часы

Автоматически созданное описание

Figure 1 – The initial binary tree

To heapify the binary tree we start by building min heap: we move from the bottom and look at every parent node. If parent node has a child element that is lesser than itself, than we swap them. This way we move from the bottom to the top. When we built min heap, we switch the top element, with the min from the bottom. And the previous top nod goes to the start of the list.

Let’s look at the example of one loop: We can see from the tree that this is not min heap, so we start arranging. 1st bottom parent nod is 14. It has child element that is smaller than itself, therefore we swap it with lowest – 5. Next bottom parent nod is 24. It is higher than both its children, so we swap again 5 ↔ 24. Next bottom parent – 17 is in min heap form, because its value is smaller than that of its the child elements. Last one is top nod. We swap 20 with 5. But now we see that 3rd row parent nod – 24 is not in min heap form. Swap it with 14. Same way we swap 2nd row parent 20 with 14. Finally, we build min heap. We truncate the top nod and place it’s value at the first position in the list. After that we swap top element with smallest in the lowest row. Then again, we start building min heap. Whole algorithm is presented at the figure 2.

Изображение выглядит как текст

Автоматически созданное описание

Figure 2 – Heap sort (min heap)

1. Using the Master theorem write the time complexity of T(n),

The master theorem is the popular way to analyze the divide-and-conquer algorithms. It’s states, that the time required for such algorithm can be expressed as sum of the times required to split the problems into the smaller ones and then combine solutions of this subproblems together with the time used in each recursive call. T(n) is the total time needed for the algorithm, such that: .

Where: a – number of splits, T(n/b) – time used in each recursion (b is the factor by which main problem is split), and O(f(n)) - is time bound required to divide the problem into small subproblems. Let’s analyze our recursion: a = 1, b = 3, and the upper bound of the f(x) is O(n) = O(nd). The polynomial that describe the complexity of the work to split the main recursion is d = 1. Now, to find the upper time bound of the problem we compare the value of d with critical exponent (c = logba). In our example the critical exponent value is c = log31 = 0. Now we compare the polynomial of the work to split with the work in the recursion call: d = 1 > c = log31 = 0 => Therefore we can say that the work to split and then recombine individual solutions dominates the subproblems and time complexity of the algorithm is T(n)=O(f(x))=n.

1. What is the time complexity of the insertion sort? State the best, worst and average cases.

In insertion search we pick an element and compare it to the element to the left. If it’s smaller than left hand side element we switch them and repeat the process until we found the position of the value, we first picked. The time complexity if the insertion sort is equal to the product between the numbers of elements in the list (n-1) and number of switches (1, 2, …, n-1).The best case for the insertion search if the list is already sorted. Then we pick each element, compare it to the element to the left, but because the list is already sorted we don’t need to switch anything, therefore B(n) = (n-1) \* O(1) = O(n) – because there n elements and we only compare each one once. The worst case is the complete opposite example – the list must be sorted in the reverse order, and for each element we make (n – 1) switches to the left, i.e. 1 switch for the second element, 2 for the third, …, (n-1) for the last, therefore, the time complexity of the worst case W(n) = 1 + 2 + 3 +… + (n-1) = n(n-1)/2 = O(n2). For the average case, each element is about halfway in order => It’s similar to the worst case: A(n) = ½ \* [1 + 2 + 3 + … + (n-1)] = n(n-1)/4 = O(n2).

1. Give one instance of the worst-case and one instance of the best-case input for the insertion sort. Explain your reasoning.

Worst-case input - [6,5,4,3,2,1]. We start from the first – there no elements bigger to the left, next one. 5 < 6 => we compare and switch 2nd element once, to the next. 3rd element 4: 4<6 => we compare and switch it once and then 4<5 – we repeat the process. As we mentioned before – n-th element would be compared and switched maximum of all possible times – (n-1), i.e. 2 time for 3rd, 3 times for 4th, etc. Number of comparisons and switches: 1 + 2 + 3 + 4 + 5 = n(n+1)/2 = 15.

Best-case input – [1,2,3,4,5,6]. We reason the same way, BUT, because the list is already sorted, we don’t need to move anything. We make comparisons to the left, do nothing, and move to the next element. After 5 (n - 1) comparisons we end the insertions sort without any switches.

1. This is the pseudo-code for a recursive algorithm called FACT. Execute it for input n = 6. Present the results of this execution as a table showing the values of x and n at each stage and state the ﬁnal results that are returned.

FACT(n)

1 if n = 1 then

2 return 1

3 else x ← FACT(n − 1)

4 return x.n

Let’s first show first few steps of this code:

FACT(6) => 1) 6 = 1 ? FALSE => 3) X = FACT(5) => 1) 5 = 1 ? FALSE => 3) X = FACT(4) => … FACT(1) => 1) 1 = 1? TRUE => 2) RETURN 1 => X=1 =>…

F(6)=F(5)\*6=(F(4)\*5)\*6)=(((((F(1)\*2)\*3)\*4)\*5)\*6)=6!=720;

The result of the execution of the following code can be presented in table form:

**Table 4 – Iterations of the FACT (6)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Variables** | **1st iterat.** | **2 st iterat.** | **3 st iterat.** | **4 st iterat.** | **5 st iterat.** | **6 st iterat.** |
| n | 6 | 5 | 4 | 3 | 2 | 1 |
| x | F(5)=120 | F(4)=24 | F(3)=6 | F(2)=2 | F(1)=1 | 1 |
| RETURN | 720 | 120 | 24 | 6 | 2 | 1 |

1. Write the asymptotic functions of the following:

The show that function G(x) is the upper bound of the function F(x) we need to find such value c>0, that F<= c\*G for all x>k. So, we need to find such pairs of c and k:

First, we transfer the polynomial by the highest power:

1. f\*(n) = 10[n\*log(n)] + 6n\*log(n) = 16n\*log(n) – asymptote for f(n), when n>=2.

f\*(n) = |16n\*log(n)| >= |10n+6n\*log(n)| = f(x), for n>=2 => C= 16, k = 2, then:

f(n) is asymptotically bounded by O(nlog(n)).

1. g\*(n) = 2[n2] + 3n2 = 5n2 – asymptote for g(n), when n>=1.

g\*(n) = |5n2| >= |2n+3n2| = g(x), for n>=1 => C= 5, k = 1, then:

g(n) is asymptotically bounded by O(n2).

1. h\*(n) = 8\*log(n) + 3\*log(n) = 11\*log(n) – asymptote for h(n), when n>=2.

h\*(n) = |11\*log(n)| >= |8\*log(n) + 3| = h(x), for n>=2 => C= 11, k = 2, then:

h(n) is asymptotically bounded by O(log(n)).