CSE 417

Algorithms and Computational Complexity

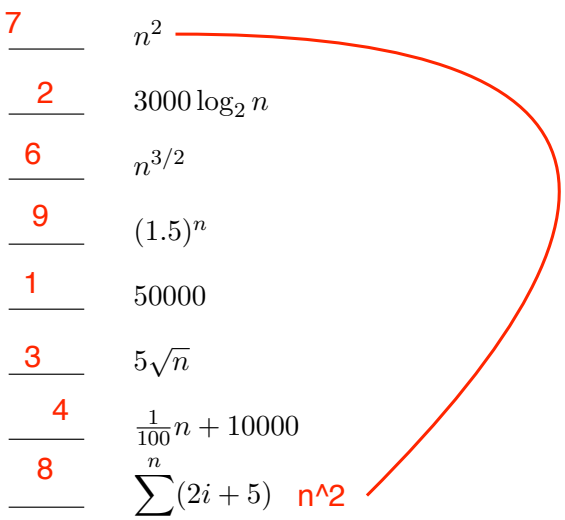
Sample Midterm Exam

W. L. Ruzzo Winter 2019

Take "Sample Midterm" with a big grain of salt. These questions are similar to ones I have used, but are not all from the same year, are not necessarily representative of questions I am likely to use, and it's probably twice as long as it should be..., *but* they do cover appropriate material.

1. Here is a list of 9 functions of n:

7	n^2
2	$3000 \log_2 n$
6	$n^{3/2}$
9	$(1.5)^n$
1	50000
3	$5\sqrt{n}$
4	$\frac{1}{100}n + 10000$
8	$\sum_{i=1}^{n}(2i+5)$ n^2
5	$n \log_2 n$



(a) Number the functions in the list above from 1 to 9, so that if $f(n)$ is numbered less than $g(n)$, then $f(n) = O(g(n))$.

(b) Connect with an arrow any pair $(f(n), g(n))$ on the list that are "equivalent", in the sense that $f(n) = \Theta(g(n))$.

2. Examine the following pseudo-code fragment:

```
// x is an n by n array
for i = 1 to n do {
   for j = i+1 to n do {
      temp    = x[i][j];
      x[i][j] = x[j][i];
      x[j][i] = temp;
   }
}
```

(a) Give the best big-O upper bound you can on the worst case run-time of this algorithm; briefly justify.

(b) Give the best big-Ω lower bound you can on the worst case run-time of this algorithm;briefly justify.

(c) Either give, and briefly justify, a big-Θ bound on its worst case run-time, or explain why this can't be done.

a)

O(n^2)
The inner loop runs n-i times so in total the inner loop runs sum_i=1^n n-i = n^2 - n(n-1)/2 = (n^2 - n)/ 2

(n^2 - n)/2 <= n^2 so O(n^2)

b)

(n^2 - n) / 2 >= (n^2 -n^2/2)/2 = n^2 / 4 = cn^2 so Omega(n^2)

c)

Since O(n^2) = Omega(n^2), then Theta(n^2)

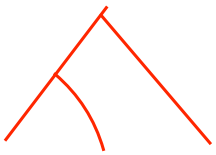
3. Depth first search of an undirected graph divides the edges into two categories. Name them, explain how the depth first search algorithm would distinguish these cases, give one example graph where both kinds of edge arise, another where only one of the two arises, and explain why no other kinds of edges are possible during DFS any undirected graph.

Edges that are traversed as part of DFS (i.e. edges within the DFS tree) and edges going from a child to an ancestor (not necessarily their parent) in the DFS tree.

Whenever a vertex v is expanded during DFS, the edge between v and the vertex that added v to the list of vertices to visit last is part of the tree. All other edges are not.

both

just tree edges



Suppose there exists an edge between a vertex v and a vertex u that is not its ancestor and that edge is not part of the DFS tree. Since u is not the ancestor of v, there exists another vertex w that is the least common ancestor of u and v, and u and v must be in different subtrees under w. In the DFS tree, each subtree is fully explored before recursing back up to the root. Without loss of generality assume v's subtree was expanded first. Well then the edge (v,u) would have been expanded and v would be in the same subtree as w.

Name: Student Number:

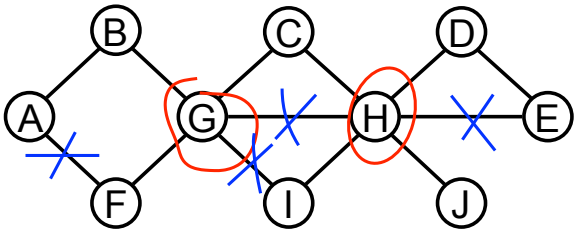
4. Let $G = (V, E)$ be an undirected graph.

(a) Define what it means for a vertex x in G to be an articulation point of G.

(b) Circle the articulation points on the graph below.

(c) Simulate the algorithm presented in lecture to find the articulation points on that graph. Fill in the table below with depth-first numbers and LOW values as calculated by the algorithm. Mark clearly with a heavy 'X' each edge that is *not* an edge of your DFS tree. (As in the homeworks, assume that DFS processes the edges incident to any vertex in *alphabetical order* of their other ends. Start your search at A.)

An articulation point of a graph G is a vertex that when deleted alongside all its adjacent edges, this disconnects G (or increases the number of connected components more generally)



Vertex	DFS Number	LOW
A	1	1
B	2	2
C	4	3
D	6	5
E	7	5
F	10	1
G	3	1
H	5	3
I	8	3
J	9	9

Name: Student Number: 5

5. Recall the "Interval Scheduling" problem: given a set of n intervals, choose a subset of them that has as many intervals as possible such that no two overlap. We considered greedy algorithms for this problem based on several possible "greedy" orderings of the intervals. For each ordering below, state whether the resulting algorithm correctly solves the general problem. If not, give a counterexample.

   (a) **earliest start first**

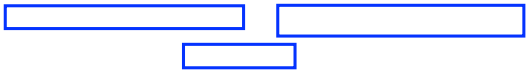
   (b) **earliest finish first** correct

   (c) **shortest duration first**

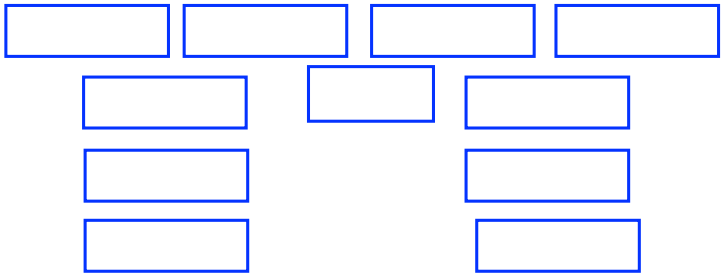
   (d) **fewest conflicts first**

a)

c)



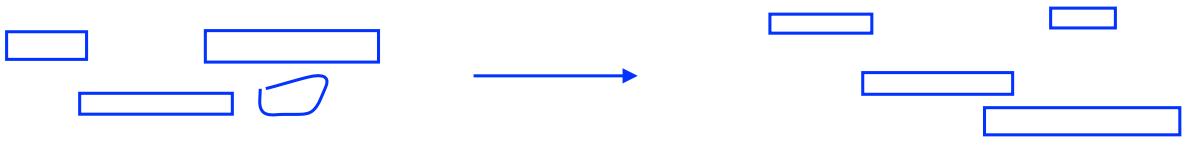
d)



6. Repeat the above for the "Interval Partition Problem," also known as "Interval Coloring."

a) correct

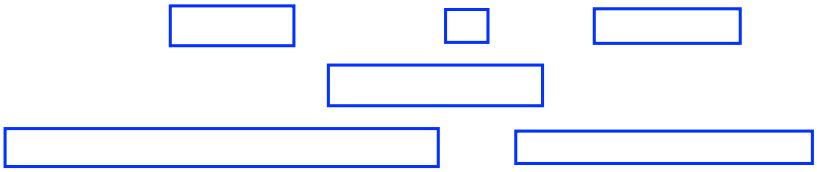
b) earliest finish



c) shortest

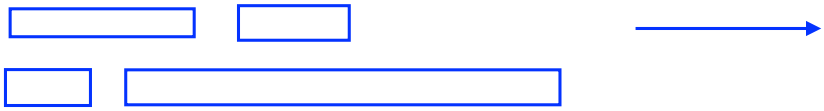
optimal

generated

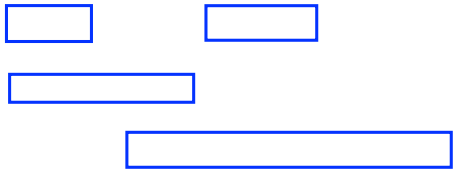


d) least conflicts

optimal



generated



Name: Student Number: 7

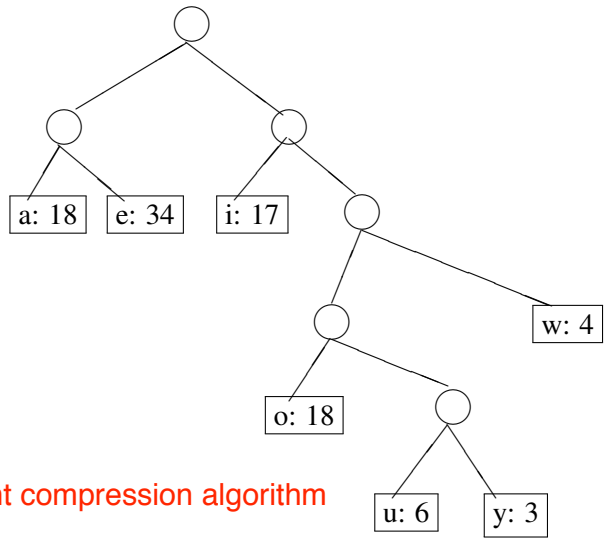
7. Assume you are given a file containing the following numbers of each indicated character.

   (a) *Without* constructing the optimal Huffman tree for this data, prove that the tree shown below is *not* optimal.

   (b) Construct an optimal Huffman tree for this data. Show your work, i.e. how you arrived at this solution.

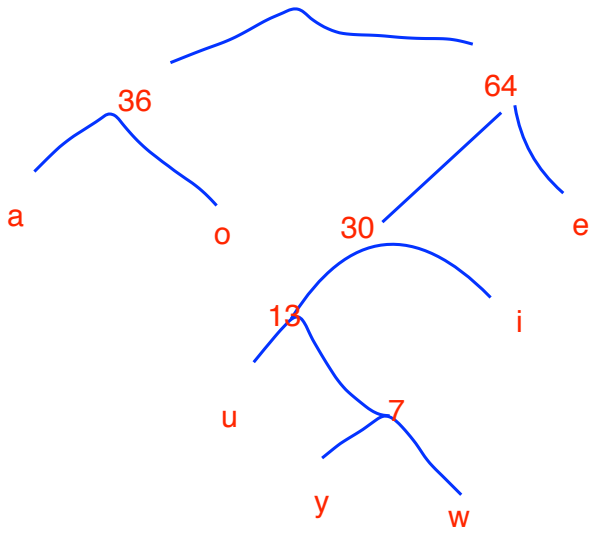
   (c) Give the total number of bits needed to represent the file using this code. For comparison, the obvious 3-bit code would use 300,000 bits.

Letter	Number of Occurrences
a	18,000
e	34,000
i	17,000
o	18,000
u	6,000
y	3,000
w	4,000



a) swapping o and i generates a more efficient compression algorithm

b) smaller value always placed to left; 0 to the left and 1 to the right



a: 00
o : 01
e : 11
i : 101
u : 1000
y : 10010
w: 10011

250,000 bits total

8. Consider the following rather stupid algorithm $M(1, n)$ for computing the maximum of an array $A[1..n]$ of n numbers:

Function $M(i, j)$
  Comment: return the maximum of $A[i..j]$
  if $i = j$
    then return $A[i]$
    else return max($M(i, j-1)$, $M(i+1, j)$)
  end

Give a recurrence relation for $C(n)$, the number of comparisons performed by this algorithm. You do *not* need to solve it.

C(n) =

1 if n == 1
2*C(n-1) if n > 1

9. Let $T[1..n]$ be a sorted array of distinct integers, some of which may be negative. Give an algorithm that can find an index i such that $T[i] = i$, provided such an index exists. Your algorithm should run within time $O(\log n)$.

Since the elements of T are distinct integers in increasing order, if T[i] > i, then it must be true that T[k] > k for all k >= i. Similarly, if T[i] < i then it must be true that T[k] < k for all k <= i. This allows us to quickly reduce the size of our search. Perform a binary like search in this way

```
def find_index(T[1, …, n]):
    i = round(n / 2)

if T[i] == i: return i
else if T[i] > i: return find_index(T[1,. .., i-1])
else return find_index(T[i+1, …, n])
```

Name: *Student Number:* 10

10. Solve the following recurrence. A good big-O estimate is fine; you don't need to give an exact solution. You may assume that n is a power of 3:

$$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ 3T(n/3) + cn, & \text{otherwise.} \end{cases} \quad (1)$$