# COMP4600 — Assignment 1 for 2014

Due date: August 11, 5pm
Late penalty 10% per day

**Note.**
Your marks will depend on the quality of your answers, not only on their correctness.

**Question 1.** [20 marks]

**Important note:** Use the definitions of $O()$, $\Omega()$ and $\Theta()$ in the course notes, not the definition in Cormen et al. (or other textbook).

(a) Simplify the expression $\Theta\big(n^2/\log n + 4n^2(\log n)^3 - 7(\log n)^8\big)$.

(b) Find expressions for the following summations (as a function of $n$) using the $\Theta(\,)$ notation.

$$\sum_{k=1}^{n} (k+4)^6 \log k$$

$$\sum_{k=1}^{n} \frac{k^2}{4^k - 3}$$

$$\sum_{k=1}^{n} (k^2-1)(n-k)^3$$

(c) Prove the following in detail directly from the definitions. If $g(n)$ and $h(n)$ are positive for sufficiently large $n$, and $f(n) = \Omega\big(g(n)\big)$ and $g(n) = \Omega\big(h(n)\big)$, then $f(n) = \Omega\big(h(n)\big)$.

(d) Suppose $g(n) > 0$ for sufficiently large $n$. If $f(n) = \Theta(g(n)) + O(g(n))$, is it true that $f(n) = O(g(n))$? Is it true that $f(n) = \Omega(g(n))$? Prove your claims.

(e) A standard high level description of merge sort is as follows:

```
procedure mergesort(x_1, ..., x_n)
    if n = 1 then
        do nothing
    else
        k := ⌊n/2⌋
        sort x_1, ..., x_k using mergesort
        sort x_{k+1}, ..., x_n using mergesort
        merge the two sorted sublists together
    endif
end
```

The final step is done by a procedure that merges two sorted lists in time $\Theta(m)$, where $m$ is the total number of items in the two lists. We proved in class that mergesort takes time $\Theta(n \log n)$ to sort a list of $n$ items.
Now suppose we accidentally type $\lfloor 2n/3 \rfloor$ instead of $\lfloor n/2 \rfloor$. The correctness is still clear (the main thing to check being that the two sublists are strictly shorter than the whole list). Is the running time still $\Theta(n \log n)$? Prove your answer.

**Question 2.** [10 marks]

In the course notes for backtracking, the general framework is given thus:

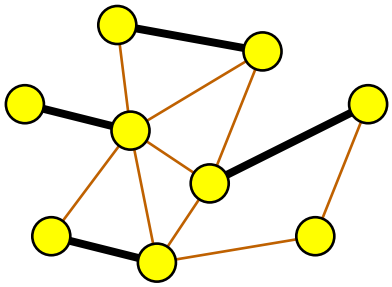
```
procedure search(P : property)
    if test T applies to P then
        output all the solutions having property P
    else
        Say refinement F partitions P into P_1, ..., P_k.
        for i from 1 to k do search(P_i) endfor
    endif
end
```

Another, equivalent, way to write the general framework is to allow the refinement $F$ to also choose which solutions can be handled by the test $T$. For this to work, we need a refinement function $F$ that partitions a property $P$ into $P_0, P_1, \ldots, P_k$ in such a way that property $P_0$ is easy. By "easy" we mean that all the solutions with property $P_0$ can be found immediately, without need for further subdivision. By "partitions" we mean that every solution with property $P$ must have exactly one of the properties $P_0, P_1, \ldots, P_k$.

It is ok for $P_0$ to be empty if there is no easy case of property $P$, but it shouldn't always be empty or else nothing will ever be written. It is also allowed to have $k = 0$, and in fact that is required sometimes if the procedure is to terminate.

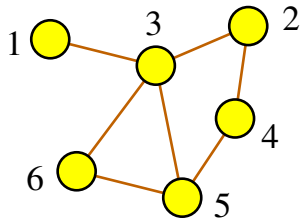
```
procedure altsearch(P : property)
    Say refinement F partitions P into P_0, P_1, ..., P_k.
    output all the solutions having property P_0
    for i from 1 to k do altsearch(P_i) endfor
end
```

A *matching* in a graph is a set of edges such that no two of the edges have a common endpoint. An example of a matching is the set of four bold edges in the following graph.



Using the new framework altsearch() above, describe an algorithm for finding all the matchings of a given graph. You need to decide what properties to use, and define a suitable refinement function $F$.

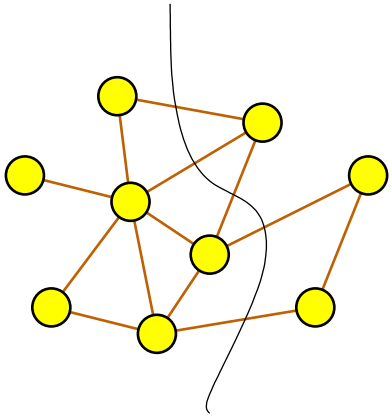
Illustrate the execution of your algorithm using the following graph, by drawing the search tree with labelling that indicates what the properties are and what is output.



**Question 3.** [10 marks]

(a) Convert the alternative backtracking framework altsearch(), as in Question 2, into a branch-and-bound framework for finding a solution that **max**imizes a function $f()$. Include pseudocode that finds an example of a best solution, not just the value of the best solution.

(b) A *cut* in a graph is a division of the vertices into two parts. For example, the thin black line divides the following graph into parts of size 6 and 3.



The *value* of a cut is the number of edges crossing from one side to the other. The value of the cut above is 5. A *maximum cut* is a cut with the largest possible value.

An alternative way of describing a cut is as a colouring of the vertices with two colours. Then the *value* is the number of edges having ends of different colour.

Using the alternative framework you defined in part (a), describe a branch-and-bound procedure for finding a maximum cut in a given graph.

Your answer should include a precise definition of what properties to use, a suitable refinement function $F$, and an upper bound function.

In fact, give two examples of an upper bound function, one very cheap and one more expensive but more powerful. Give an example showing that the second one is more powerful.