Pre-tutorial questions

Do you know the basic concepts of the unit's content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. Basic concepts
   (a) What is an algorithm's worst case running time?
   (b) What does it mean when we say that the algorithm runs in polynomial time?
   (c) What is amortised time? What is expected time, and what's the difference between amortised, expected and worst case time?
   (d) Can you explain the ideas of the $O(\cdot)$, $\Theta(\cdot)$ and $\Omega(\cdot)$ functions? Why do we use these functions?
   (e) What does it mean that these functions are transitive and additive?
2. Data structures
   (a) What are linked lists, queues, stacks and balanced binary trees? What sort of operations are usually supported by these structures?
   (b) What is the height of a balanced binary tree containing n elements?
   (c) What's an AVL tree? How do you insert an element into an AVL tree?
   (d) What is the time complexity of inserting, deleting, and searching for an element in an AVL tree?
   (e) What are pre-order, in-order and post-order traversals of a tree?
   (f) How are priority queues usually implemented?
   (g) What is a heap? How do you insert an element in a heap?
   (h) What's a hash table? What are they used for?
   (i) What's a hash function? Give an example.
3. Graph terminology
   (a) What is a graph $G(V, E)$?
   (b) Graphs are usually represented either as an adjacency matrix or as an adjacency list. Can you explain the two representations?
   (c) What are the advantages/disadvantages between the two different representations?
   (d) What is a simple path in a graph?
   (e) What is a cycle in a graph?
   (f) What is a tree? If a tree has n vertices, how many edges does it have?
   (g) What's the difference between a rooted tree and an unrooted tree?
   (h) What's the difference between a directed graph and an undirected graph?
4. Graph traversals
   (a) What's the difference between BFS and DFS?
   (b) Explain the two search algorithms BFS and DFS.
5. Algorithmic techniques

(a) What is a Greedy algorithm? Example of a Greedy algorithm?
(b) What's a minimum spanning tree of a graph?
(c) How does Prim's algorithm work?
(d) What does Dijkstra's algorithm compute? Can it handle negative edge weights? Why not?
(e) What data structure does Dijkstra's algorithm use to know which vertex to process next?
(f) What are the running times of Prim's and Dijkstra's algorithm?
(g) What is a Divide-and-Conquer algorithm? Example of a Divide-and-Conquer algorithm?

Tutorial

1 Graph algorithms

Problem 1
Give an $O(n)$ time algorithm to detect whether a given undirected graph contains a cycle. If the answer is yes, the algorithm should produce a cycle. (Assume adjacency list representation.)

Problem 2
Trace Prim's algorithm on the graph in Fig. 1 starting at node a. What's the output?

Problem 3
Trace Dijkstra's algorithm on the graph in Fig. 1 starting at node a and ending at g. What's the shortest path?

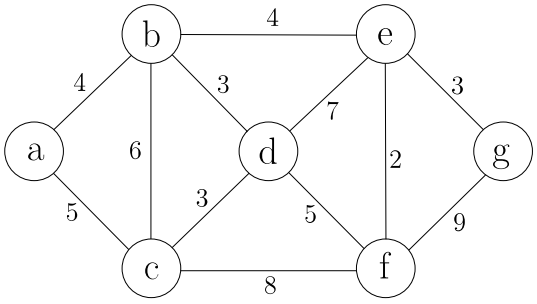

Figure 1: Input graph to Questions 2 and 3.

Problem 4
Consider the following generalization of the shortest path problem where in addition to edge lengths, each vertex has a cost. The objective is to find an s-t path that minimizes the total length of the edges in the path plus the cost of the vertices in the path. Design an efficient algorithm for this problem.

2 Greedy algorithms

Greedy algorithms can be some of the simplest algorithms to implement, but they're often among the hardest algorithms to design and analyze. You can often stumble on the right algorithm but not recognize that you've found it, or might find an algorithm you're sure is correct and be unable to prove its correctness.

The standard way of proving the correctness of a greedy algorithm is by using an *exchange argument*. They work by showing that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without changing the cost of the optimal solution, thereby proving that the greedy solution is optimal. Typically, exchange arguments are set up as follows:

1. **Define your solutions.** You will be comparing your greedy solution X to an optimal solution X_{opt}, so it's best to define these variables explicitly.

2. **Compare solutions.** Next, show that if $X \neq X_{opt}$, then they must differ in some specific way. This could mean that there's a piece of X that's not in X_{opt}, or that two elements of X that are in a different order in X_{opt}, etc. You might want to give those pieces names.

3. **Exchange Pieces.** Show how to transform X_{opt} by exchanging some piece of X_{opt} for some piece of X. You'll typically use the piece you described in the previous step. Then, prove that by doing so, you did not increase the cost of X_{opt} and you therefore have a different optimal solution.

4. **Iterate.** Argue that you have decreased the number of differences between X and X_{opt} by performing the exchange, and that by iterating this process you can turn X_{opt} into X without impacting the quality of the solution. Therefore, X must be optimal. This last step might require a formal argument using an induction proof. However, in most cases this is not needed.

Problem 5

Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then cycle 10 km, then run 3 km. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first contestant swims the 20 laps, gets out, and starts biking. As soon as the first contestant is out of the pool, the second contestant begins swimming the 20 laps; as soon as he/she's out and starts cycling, a third contestant begins swimming ... and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *cycling time* (the expected time it will take him or her to complete the 10 km of cycling), and a projected *running time* (the time it will take him or her to complete the 3 km of running. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts.

What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove the correctness of your algorithm.

Problem 6

Assume that you are given n white and n black dots, lying on a line. The dots appear in any order of black and white. Design a greedy algorithm which connects each black dot with a (different) white dot, so that the total length of wires used to form such connected pairs is minimal. The length of wire used to connect two dots is equal to their distance along the line.

Problem 7

Given a set of n horizontal segments, report the maximum number of segments that can be intersected by a vertical line in $O(n \log n)$ time.

3 Divide-and-Conquer

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of the original problem.
2. Recursively solving these subproblems.
3. Appropriately combining (merging) their answers.

The real work is done in three different places: in the partitioning of problems into subproblems; when the subproblems are so small that they are solved outright; and in the gluing together of partial answers.

The standard way of proving correctness for a divide-and-conquer algorithm is by using induction as follows.

- Base case: Solve trivial instances directly, without recursing.
- Inductive step: Reduce the solution of a given instance to the solution of smaller instances, by recursing. For divide-and-conquer algorithms it usually requires a bit of work to prove that the step of merging two (or more) solutions to smaller problems into the solution for the larger problem.

Problem 8
Consider the following algorithm.

Algorithm 1 REVERSE

```
1: function REVERSE(A)
2:     if |A| = 1 then
3:         return A
4:     else
5:         Let B and C be the first and second half of A
6:         return concatenate REVERSE(C) and REVERSE(B)
7:     end if
8: end function
```

Let $T(n)$ be the running time of the algorithm on a instance of size n. Write down the recurrence relation for $T(n)$ and solve it by unrolling it.

Problem 9
Given an array A holding n objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than $n/2$ positions of A.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

1. Show that if x is a majority element in the array then x is a majority element in the first half of the array or the second half of the array
2. Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.
3. Put these observation together to design a divide an conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$
4. Solve the recurrence by unrolling it.

Problem 10

Suppose we are given numbers a, n, where $n > 0$ is an integer. We wish to calculate the number a^n. What is the quickest way to do this? How many multiplication operations do we have to perform? Of course, we may compute 19^8 by calculating $19 \times 19 = 361$, then calculating $19^3 = 361 \times 19 = 6859$, then $19^4 = 6859 \times 19 = 130321$, and so on, until we get 19^8. This takes seven multiplications in total. Is this the quickest possible? Note that $8 = 2 \times 4$, so we can also write $19^8 = 19^4 \times 19^4$. If we compute 19^4 first, and then square it, we need only one more multiplication. The straightforward method would require four more multiplications: $19^8 = 19^4 \times 19 \times 19 \times 19 \times 19$. Similarly, $19^4 = 19^2 \times 19^2$. So if we calculate $19^2 = 361$ with one multiplication, $19^4 = 361^2 = 130321$ with one more, we get $19^8 = 130321^2 = 16983563041$ with the third multiplication. This cleverer method requires only three multiplications. The method above seems to work when the exponent n is even. What do we do when it is odd? Say, we would like to calculate 19^7. We may write $7 = 6 + 1$, so $19^7 = 19^6 19$, then $19^6 = 19^3 \times 19^3$, and finally $19^3 = 19^2 \times 19$. So $19^3 = 361 \times 19 = 6859$, $19^6 = 6859^2 = 47045881$, and $19^7 = 47045881 \times 19 = 893871739$. The straightforward method of calculation requires 6 multiplications, and we needed only 4 here. We can combine the ideas from the two examples above to get a procedure to calculate the power a n for any pair a, n.

Algorithm 2 Power

```
1: function POWER(n, a)
2:     if  n = 1 then
3:         return a
4:     end if
5:     if n is even then
6:         b =POWER(a,n/2)
7:         return b^2
8:     else
9:         b =POWER(a,(n-1)/2)
10:        return a × b^2
11:    end if
12: end function
```

Prove that the algorithm correctly computes a^n. Can you bound the number of multiplications for each n?