Pre-tutorial questions

Do you know the basic concepts of this week's lecture 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. Algorithmic technique
   (a) What is a Greedy algorithm? Example of a Greedy algorithm?
   (b) What is a Divide-and-Conquer algorithm? Example of a Divide-and-Conquer algorithm?
   (c) What is a Sweepline algorithm? Example of a Sweepline algorithm?
   (d) What is a Dynamic Programming algorithm? Example of a Synamic Programming algorithm?
   (e) What is a Flow Network algorithm? Example of a Flow Network algorithm?

Tutorial

1 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 1
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.

Solution: Let the contestants be numbered $1, \ldots, n$, and let s_i, b_i, r_i denote the swimming, biking, and running times of contestant i. Here is an algorithm to produce a schedule: arrange the contestants in order of decreasing $b_i + r_i$, and send them out in this order. We claim that this order minimizes the completion time.
We prove this by an exchange argument. Let X be the schedule produced by the greedy algorithm and let X_{opt} be an optimal schedule. If X and X_{opt} are identical then we are done. Suppose $X_{opt} \neq X$ then the optimal solution must contain two contestants i and j so that j is sent out directly after i, but $b_i + r_i < b_j + r_j$. We will call such a pair (i, j) an *inversion*. Consider the solution obtained by swapping the orders of i and j. In this swapped schedule, j completes earlier than he/she used to. Also, in the swapped schedule, i gets out of the pool when j previously got out of the pool; but since $b_i + r_i < b_j + r_j$, i finishes sooner in the swapped schedule than j finished in the previous schedule. Hence our swapped schedule does not have a greater completion time, and so it too is optimal.
Continuing in this way, we can eliminate all inversions without increasing the completion time. To be very rigorous here, you would probably proceed by induction to show this, but for most cases (including this case) this is immediate. At the end of this process, we will have a schedule in the order produced by our algorithm, whose completion time is no greater than that of the original optimal order we considered. Thus the order produced by our algorithm must also be optimal.

Problem 2

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.

Solution: Suppose you have a sorted list of the white dot positions and the black dot positions. Then you should match the ith white dot with the ith black dot.
To prove that the algorithm is optimal consider an optimal solution X_{opt} and the solution X produced by the greedy algorithm. Again, if $X = X_{opt}$ then we are done. Otherwise there must exist a matched pair w_i, b_j with $i \neq j$ in X_{opt}. We call such a pair an inversion. To simplify the argument we assume that $i < j$ and that w_i is the leftmost white point that is part of an inversion.
Since w_i is matched to b_j, b_i must be matched to some w_k with $k \geq i$.
We argue that we can remove an inversion in the optimal solution X_{opt} and the cost of the matching will not increase. First consider the cost of matching these two pairs in the optimal solution:

$$C(w_i, b_j) + C(w_k, b_i) = b_j w_i + |w_k b_i|.$$

If we replace these two pairs with (w_i, b_i) and (w_k, b_j) we would get:

$$C(w_i, b_i) + C(w_k, b_j) = |w_i b_i| + |w_k b_j|.$$

There are two cases depending on w_k's relative position between b_i and b_j. When its smaller than both of them the cost stays the same. When its smaller than just b_j one saves $2w_k 2b_i$ by swapping and when its larger than both one saves $2b_j 2b_i$. Therefore we can remove the inversion in X_{opt} and guarantee that we still have an optimal solution. This argument can be iterated until we have no more inversions in the solution and, hence, the greedy must be optimal since it has no inversions.

2 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 3
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 1 Power

```
function POWER($n, a$)
    if  $n = 1$ then
        return $a$
    end if
    if $n$ is even then
        $b$ =POWER(a,n/2)
        return $b^2$
    else
        $b$ =POWER(a,(n-1)/2)
        return $a \times b^2$
    end if
end function
```

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

Solution: The algorithm POWER(a, n) correctly calculates a^n, as long as it correctly calculates the relevant smaller powers of a. We may prove this by induction over n that this happens for any number a.
Proof The base case is $n = 1$, where the algorithm correctly returns a. For the induction hypothesis, assume that the algorithm is correct for all k, with $1 \leq k \leq n - 1$. Consider POWER(a, n). If n is even, the algorithm returns the value b^2, where b =POWER$(a, n/2)$. Since $n > 1$ and even, $n/2$ is an integer, and $1 \leq n/2 \leq n - 1$. By the induction hypothesis, POWER$(a, n/2)$ correctly returns $a^{n/2}$, so the algorithm returns $b^2 = a^n$. If n is odd ($n > 1$) then $(n - 1)/2$ is an integer, and $1 \leq (n - 1)/2 \leq n - 1$. By the induction hypothesis, b =POWER$(a, (n - 1)/2) = a^{(n-1)/2}$, and the algorithm correctly returns $a \times b^2 = a^n$ in this case as well. So the algorithm is correct by induction. □
Consider the number of multiplications secribed by the algorithm, it can be described by the recurrence $T(n) = O(1) + T(n/2)$ which solves to $O(\log n)$. A closer analysis shows that the bound is $2 \log_2 n$.

3 Sweepline

The sweepline technique can be used to solve a variety of different (geometric) problems. The idea is to sweep an "imaginary" line ℓ across the plane while (1) maintaining the status of ℓ and (2) fulfilling an invariant. The status of ℓ only changes at certain discrete event points, thus in-between two incident event points no computation is needed. Typical event points are input points or intersections between segments. When the sweep line encounters an event point the status is updated in such a way that the invariant is guaranteed to hold after the event point has been processed. The status of ℓ is typically an additional data structure that stores information regarding the input that intersects ℓ, and the invariant is usually stating "everything that has already been processed is correct". For example, for the segment intersection reporting algorithm that we went through in the lecture the sweep line is a vertical line that moves from left to right. The status is the input segments intersecting ℓ ordered from top to bottom and the invariant states that "all intersections to the left of ℓ has been reported".

Proving that a sweepline algorithm is correct usually follows immediately from the invariant and the event points. First prove that the status can't change between two consecutive event points and then prove that the invariant holds before and after an event point is processed.

Problem 4
Given a set S of n intervals and a set P of m points on the line, report a point $p \in P$ that is covered by the most number of intervals in S.

Solution: Let U be the endpoints of the intervals in S and the points in P. The points in U will be the event points. Sort the points in U from left to right, and let ℓ be a vertical sweep line that moves from left to right.
During the sweep we will maintain three values:

- s: an integer storing the number of segments intersecting ℓ
- p: a point in P to the left of ℓ that is covered by the most number of intervals in S
- k: an integer storing the number of intervals in S covering p

The variable s is the status of ℓ, while p and k are need for the following invariant: "p stores the point to the left ℓ covered by the most number of intervals from S". From this it's already clear that if we can maintain the invariant during the sweep then the correctness follows immediately.
The algorithm now sweeps ℓ from left to right. Initially $s = 0$, $k = 0$ and p is null. At each event point e it stops and updates p, k and s. We will have three cases depending on the type of point e is:

- $e \in P$: If $s > k$ then set $k = s$ and $p = e$. That is, we found a new point e that is covered by more intervals than any other point to the left of ℓ.
- e is a left endpoint of an interval in $s \in S$: Increment s by one.
- e is a right endpoint of an interval $s \in S$: Decrement s by one.

To prove that the invariant is maintained during the whole sweep we use induction. The base case is when no event points has been encountered. From the initialisation of s, p and k it follows that the invariant and the status is correct at that point. Assume that the invariant and status is correct just before the sweep line encounters an event point e. Next consider what happens when event point e is processed. We have to consider the three cases. If e is a point in p then we just count the number of segments covering e, which is equal to the number of segments intersecting ℓ. If this is larger than k then we update the best found solution found so far (updates p and k). If e is a left or right endpoint then the number of segments intersecting ℓ increases or decreases by one. Thus, after e has been processed the invariant and the status are correct, which proves the correctness.

4 Dynamic programming

Like greedy algorithms, dynamic programming algorithms can be deceptively simple. The algorithms, once written out, are often so straightforward that it's hard to believe they work correctly. Consequently, one of the challenges in writing dynamic programming algorithms is rigorously establishing their correctness. Fortunately, dynamic programming proofs are often relatively straightforward and follow a standard pattern.

Typically, dynamic programming algorithms are based on a recurrence relation involving the optimal solution, so the correctness proof will primarily focus on justifying why that recurrence relation is correct. The general outline of a correctness proof for a dynamic programming algorithm is as following:

1. **Define subproblems.** Dynamic programming algorithms usually involve a recurrence involving some quantity $OPT(...)$ over one or more variables (usually, these variables represent the size of the problem along some dimension). Define what this quantity represents and what the parameters mean. This might take the form "$OPT(k)$ is the maximum number of people that can be covered by the first k cell towers" or "$OPT(u, v, i)$ is the length of the shortest path from u to v of length at most i."

2. **Write a recurrence.** Now that you've defined your subproblems, you will need to write out a recurrence relation that defines $OPT(...)$ in terms of some number of subproblems. Make sure that when you do this you include your base cases.

3. **Prove that the recurrence is correct.** Having written out your recurrence, you will need to prove it is correct. Typically, you would do so by going case-by-case and proving that each case is correct.

4. **Prove the algorithm evaluates the recurrence.** Next, show that your algorithm actually evaluates the recurrence by showing that the table values match the value of OPT and that as you fill in the table, you never refer to a value that hasn't been computed yet. To be fully rigorous, you would probably need to prove this by induction. However, in most cases a few sentences should suffice here.

5. **Prove the algorithm is correct.** Having shown that you've just evaluated the recurrence correctly, your algorithm will probably conclude with something like "return $A[m, n]$". Prove that this table value is the one that you actually want to read.

Problem 5
Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. independent of the values of the weights. Prove the correctness and complexity of your algorithm.

Solution:

1. Let S_i denote an optimal independent set on $\{v_1, \ldots, v_i\}$, and let $OPT[i]$ denote its weight. Define $OPT[0] = 0$ and note that $OPT[1] = w_1$.

2-3. Now, for $i > 1$, either v_i belongs to S_i or it doesn't. In the first case, we know that v_{i-1} cannot belong to S_i, and so $OPT[i] = w_i + OPT[i-2]$. In the second case, $OPT[i] = OPT[i-1]$. Thus we have the recurrence

$$OPT[i] = \max(OPT[i-1], w_i + OPT[i-2]).$$

4. From the recurrence it immediately follows in this simple case that we can compute the values of $OPT[i]$, in increasing order from $i = 1$ to n.

5. $OPT[n]$ is the value we want since it stores the weight of an optimal solution of the elements $v_1, \ldots, v_n$.

Note that we can compute S_n by tracing back through the computations of the *max* operator (similar to what we did for knapsack). Since we spend constant time per iteration, over n iterations, the total running time is $O(n)$.

5 Network Flow

The general idea we've used to solve a problem X with network flows is to "reduce" X to the problem of computing a max flow (or something similar). That is, we modify X into an equivalent problem that can be solved using network flows (for example, bipartite matching). Since we are reducing a problem X to another problem Y the correctness proof requires us to prove that a solution for Y is a solution for X and vice versa. For example, for bipartite matching we proved that if there is a matching of size k in the bipartite graph G then there's a flow of value at most k in the corresponding flow network G', and if there's a flow of value k in G' then there's a matching of size at most k in G.

Problem 6

Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like *Yahoo!* was in the "eyeballs" — the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if the user has told *Yahoo!* that they're a 20-year old computer science student from USyd, the site can throw up a banner ad for apartments in Glebe; on the other hand, if they're a 50-year-old investment banker from Vaucluse the site can display a banner ad pitching Luxury Cars instead.

But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups* $G_1, G_2, \ldots, G_k$. (These groups can overlap; for example G_1 can be equal to all residents of Sydney, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like:

- For a subset $X_i \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now, consider the problem of designing a good *advertising policy* — a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i, are shown an ad provided by advertiser i.)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

Solution: We define a flow network $G = (V, E)$ as follows.

- There is a source s, vertices $v_1, \ldots, v_n$ for each person vertices $w_1, \ldots, w_m$ for each advertiser, and sink t.
- There is an edge of capacity 1 from v_i to each w_j for which person i belongs to a demographic group that advertiser j wants to target.
- There is an edge with a capacity of 1 from s to each v_i; and for each j, there is an edge with lower bound r_j from w_j to t.
- Finally, the source has a demand of $-\sum_j r_j$, and the sink has a demand of $\sum_j r_j$. All other nodes have demand 0.

Now, if there is a valid circulation in this graph, then there is an integer circulation. In such a circulation, one unit of flow on the edge (v_i, w_j) means that we show an ad from advertiser j to person i. With this meaning, each advertiser shows their required number of ads to the appropriate people.
Conversely, if there is a way to satisfy all the advertising contracts, then we can construct a valid circulation as follows. We place a unit of flow on each edge (v_i, w_j) for which i is shown an ad from advertiser j; we put a flow on the edge (w_j, t) equal to the number of ads shown from advertiser j; and we put a unit of flow on each edge (s, v_i) for which person i sees an ad.
Thus, there is a valid circulation in this graph if and only if there is a way to satisfy all the advertising contracts; and the flow values in an integer-valued circulation can be used, as above, to decide which ads to show to which people.