COL 702

Homework III

Due on Sept. 19, 2019

Justify your answers with proper reasonings/proofs.

1. The second minimal spanning tree is one that is distinct from the minimal spanning tree (has to differ by at least one edge) and is an MST if the original tree is ignored (they may even have the same weight). Design an efficient algorithm to determine the second MST.

   The main observation is that we can assume that the second MST will differ from the MST in only one edge.

   Indeed, suppose it differs in two edges: let T denote the MST and T' denote the second MST. Let e and f be edges T which are not there in T'. Now consider adding e to the T'. This will create a cycle C and this cycle must have an edge which e' which is at least as expensive as e and not in T (why?). Then we can remove e' and add e to T' to get a tree of the same or better cost.

   Now we do not know the identity of the edge in which the two trees differ. But we can guess the edge e' which is in $T' \setminus T$. When we add this to T, this will create a cycle. We must drop the edge (other than e') in this cycle which has the highest cost. Running time: $O(mn)$.

2. Let G be a connected graph and T and T' be two different spanning trees of G. We say that T and T' are *neighbours* if T contains exactly one edge that is not in T' (and vice versa). Now from the graph G, we construct a new graph $\mathcal{H}$ as follows: the nodes of $\mathcal{H}$ correspond to the spanning trees of G, and there is an edge between two two nodes of $\mathcal{H}$ if the corresponding spanning trees are neighbours. Is it true that for any connected graph G, $\mathcal{H}$ is connected ? Either give a proof or a counterexample.

   The stated fact is true. Let T and T' be two trees. Suppose they differ in k edges. We will find a tree T'' such that T and T'' are neighbours in $\mathcal{H}$ and T'' and T' will differ in $k-1$ edges only. The proof will then follow by induction on k. Let $e_1, \ldots, e_k$ be the edges in $T' \setminus T$. Add e_1 to T. This will create a cycle C. There must be an edge e in C which is not in T' (because T' does not contain any cycle). Remove e to get T''.

3. Let C be a unit radius circle. An arc of C is given by a pair $[\theta_1, \theta_2]$, where $\theta_1 < \theta_2$ are angles between 0 and 360 degrees. You are given a set of n arcs in the circle and would like to select a subset of arcs of maximum cardinality so that no two of them overlap. Give an efficient algorithm to find an optimal solution.

   Suppose we *knew* one of the arcs which were selected. Then we could remove this arc and the edges in it to get the same problem on a line. We know a greedy algorithm to solve this problem on a line. Thus, the algorithm first cycles over all arcs, and for each such arc, solves the corresponding problem on a line.

4. You are given n jobs and m machines. Each job has a size p_j and needs to be scheduled on one of the machines without preemption. The goal is to minimize the average completion time of the jobs. Design a greedy algorithm and prove that it is optimal.

   The algorithm arranges the jobs in increasing order of the p_j values and considers them in this order. Define the load on a machine as the total sizes of the jobs assigned to it. When consider a job j (in this order), it assigns j to the machine with the minimum load.

   Consider any solution S. First show that given an assignment of jobs to machines, the jobs need to be done in the order of increasing processing time. This follows by a simple exchange argument. For a job j, let n_j denote the number of jobs which are processed after j on the machine in which j is processed (including j). Then it is easy to check that the total completion time of all the jobs can also be written as

$$\sum_j p_j n_j.$$

   Therefore, if there are two jobs j and j' with $p_j < p_{j'}$ but $n_j < n_{j'}$, then we can improve the solution by exchanging the two jobs because

$$p_j n_j + p_{j'} n_{j'} \geq p_j n_{j'} + p_{j'} n_j.$$

   Now starting with any solution, keep on performing such swaps as long as possible. Show that you will end up with the greedy solution.

5. A multi-stack consists of an infinite series of stacks $S_0, S_1, \ldots$ where the ith stack S_i can hold upto 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements of S_i and push them into S_{i+1} (and if S_{i+1} gets full, we need to recurse). Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them into S_i. Again if S_{i+1} becomes empty during this process, we recurse. Assume push and pop operations for each stack takes $O(1)$ time. The pseudo-code is as follows:

```
MultiPush(x):

  i = 0;
  while (S_i is full)
     i = i+1;
  while (i > 0) {
     i = i-1;
     for j=1 to 3^i
        Push(S_{i+1}, Pop(S_i));
  }
  Push(S_0, x);
```

```
MultiPop():
   i = 0;
   while (S_i is empty)
      i = i+1;
   while (i > 0) {
      i = i-1;
      for j=1 to 3^i
          Push(S_i, Pop(S_{i+1});
   }
   return Pop(S_0);
```

- In the worst case, how long does it take to push one more element onto a multistack containing n elements?

  If the first l stacks are full, then time taken is proportional to 3^l, which is $\theta(n)$.

- Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.

  First notice that the number of elements in S_i is a power of 3^{i-1}. So after n operations, the highest occupied stack is S_ℓ, where ℓ is $O(\log n)$. Notice that each element only moves to a higher stack, and so the total amount of movements of the n elements is at most $O(n \log n)$. Hence, the amortized cost is $O(\log n)$.

- Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

  First notice that we the largest level ℓ such that S_ℓ is occupied is at most $O(\log n)$. The reason is that as above, the number of elements in a stack S_i is a multiple of 3^{i-1}. Fix an element x and let us see how many times it is moved from S_i to S_{i+1} and back. Whenever it is moved from S_{i+1} to S_i, all the stacks $S_1, \ldots, S_i$ were empty, and whenever it is moved from S_i to S_{i+1}, all the stacks S_1 to S_i must be a at least one-third full. Suppose elements are moved between these two stacks k times. So the total movement cost is $O(k3^i)$. But notice that $\Omega(k3^i)$ pops and pushes must have happened. So, we can charge the total movement cost to these operations. Since we have to consider all such pairs till $i = O(\log n)$, the desired claim follows.

6. Consider the following process. At all times you have a single positive integer x, which is initially equal to 1. In each step, you can either increment x or double x. Your goal is to produce a target value n. For example, you can produce the integer 10 in four steps as follows:

$$1 \to 2 \to 4 \to 5 \to 10.$$

   Obviously you can produce any integer n using exactly n 1 increments, but for almost all values of n, this is horribly inefficient. Describe and analyze an algorithm to compute the minimum number of steps required to produce any given integer n.

The answer is take the binary representation of n say it is $b_k b_{k-1} ... b_0$, where b_0 is the least significant bit. If n is odd, let n' be the same number as n except for the last bit being 0. We first construct n' recursively and then add 1 to it. If n is even, let n' be the number obtained by removing the last bit b_0. We first construct n' recursively and then double it.

So if the bit representation of n has k 1's and l 0's, then the number of operations is $2k + l$. Now we show that this is optimal by induction on n. If n is odd, then the last operation must be an addition by 1. And so, by induction on $n - 1$, we get the lemma. The more interesting case is when n is even. If the last operation is multiplication by 2, then we are done by induction on $n/2$. So suppose the last operation is addition by 1. Suppose n had k 1's and l 0's. Then it is easy to check that $n - 1$ (in binary) has at least k 1's and $l - 1$ 0's (you need to use the fact that n is even here). By induction hypothesis, any algorithm will require at least $2k + (l - 1)$ operations. Combining this with the last operation shows that the algorithm will take at least $2k + l$ operations, which is what our algorithm is also taking.

7. Suppose you are faced with an infinite number of counters x_i, one for each integer i. Each counter stores an integer mod m, where m is a fixed global constant. All counters are initially zero. The following operation increments a single counter x_i; however, if x_i overflows (that is, wraps around from m to 0), the adjacent counters x_{i1} and x_{i+1} are incremented recursively. Here is the pseudocode:

```
Nudge_m(i)

  x_i = x_i + 1;
  while (x_i >= m) {
      x_i = x_i - m;
      Nudge_m(i+1);
      Nudge_m(i-1);
  }
```

   - Suppose we call $Nudge_3$ n times starting from the initial state when all counters 0. Note that each call can start from any of the counters. Show that the amortized time complexity is $O(1)$.

     The idea is to see what this operation does. Suppose we are calling this function at some counter i. If this counter is not equal to 2, then we just change this counter, and the process ends there. Now suppose the counter is 2, then the process can cascade. Some though will reveal the following: suppose x_i is 2, and let $j \leq i \leq k$ be indices such that $x_j, x_{j+1}, \ldots, x_i, \ldots, x_k$ are all 2, but $x_{j-1}, x_{k+1} \neq 2$. Then this process will replace $x_j, \ldots, x_k$ by 1, and may change x_{j-1} and x_{k+1}. Further, the total number of addition operations is at most $2(k - j) + O(1)$. Therefore, we define the potential function Φ_i after i operations as twice the number of counters which are 2. Now it is easy to check that the time taken plus the change in potential is $O(1)$.

- What is the amortized time complexity in the above question if the global counter m is 2 instead of 3 ?

  In this case, the amortized time complexity after n operations is $O(n)$. The reason is the following: suppose counters $x_i, x_{i+1}, \ldots, x_j$ are all 1. Now if we call increment at any of these counters, the counters $x_{i-1}, x_i, \ldots, x_{j+1}$ will be 1, and the time taken will be at least $j - i$. So if perform such operations n times, the total time taken is $O(n^2)$.