CPSC 320 Sample Final
April, 2016

1. [24 points] Short Answers

   (a) [6 points] Consider the Gale-Shapley stable matching algorithm. Show that at any step of this algorithm the following claim is **true**: *if there is a free man, then there is a woman he has not proposed to yet.*

   Solution: *Proof by contrapositive.* Since each woman keeps the best engagement she has received so far, once a woman gets a proposal she will remain engaged until the end. Hence, if a man m has proposed to every woman, then every woman is engaged at this point. Since # of free men = # of free women, every man (including m) is engaged.

   (b) [4 points] Write down the exact (mathematical) definition of what it means that $f(n) \in O(g(n))$.

   Solution: There exists constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$: $f(n) \leq c \cdot g(n)$.

   (c) [8 points] In each row below, circle the correct statement (either (a) or (b) or (c)) if we know that for every $n \in \mathbb{N}$, $0 < f(n) < g(n)$:

(a) $f(n) \in O(g(n))$	(b) $f(n) \notin O(g(n))$	(c) $f(n)$ may or may not be in $O(g(n))$
(a) $f(n) \in \Omega(g(n))$	(b) $f(n) \notin \Omega(g(n))$	(c) $f(n)$ may or may not be in $\Omega(g(n))$
(a) $f(n) \in o(g(n))$	(b) $f(n) \notin o(g(n))$	(c) $f(n)$ may or may not be in $o(g(n))$
(a) $f(n) \in \omega(g(n))$	(b) $f(n) \notin \omega(g(n))$	(c) $f(n)$ may or may not be in $\omega(g(n))$

   Solution: (a), (c), (c), (b)
   Comments: $f(n) \in O(g(n))$ — we can choose in definition $c = 1$ and $n_0 = 1$. This implies that $f(n) \notin \omega(g(n))$. Whether $f(n) \in \Omega(g(n))$ or $f(n) \in o(g(n))$, cannot be decided based on the fact that $f(n) < g(n)$. For instance, if $f(n) = g(n)/2$, then $f(n) \in \Omega(g(n))$ and $f(n) \notin o(g(n))$, but if $f(n) = n/2$ and $g(n) = n^2$, then $f(n) \notin \Omega(g(n))$ and $f(n) \in o(g(n))$.

   (d) [6 points] What are the two strategies used to show that the greedy solution is an optimal solution? Briefly explain what is the main idea of each strategy.

   Solution:

   - **Staying-Ahead**: show that the greedy solution is as good the optimal solution at every step.
   - **Exchange-Argument**: show that any optimal solution can be transformed to the greedy solution by a sequence of transformations, each preserving the optimality of the solution.

2. [20 points] *Short answers.*

   (a) [10 points] The worst-case running time of an algorithm you wrote satisfies the recurrence relation

   $$T(n) = \begin{cases} aT(n/4) + n^x & \text{if } n \geq 4 \\ \Theta(1) & \text{if } n < 4 \end{cases}$$

   For which values of the constants a and x will the algorithm run in $\Theta(n^2)$ time?
   Hint: there are two cases.

Solution: We can use the Master Theorem, since the recurrence has the right form (as long as $a > 0$). The special function is $s(n) = n^{\log_4 a}$ and $f(n) = n^x$. Hence, depending on comparison of $\log_4 a$ and x, we can end up in each of the three cases:

1. $x < \log_4 a$, we have Case 1. Hence, $T(n) \in \Theta(s(n))$. Since we want $\Theta(n^2)$, we must have $\log_4 a = 2$. Then $a = 16$ and $x < 2$.
2. $x = \log_4 a$, we have Case 2 with $k = 0$. Hence, $T(n) \in \Theta(s(n) \log n)$. There is no choice for x and a, how to get $\Theta(n^2)$ (we can't get rid off the $\log n$ term).
3. $x > \log_4 a$, we have Case 3. Hence, $T(n) \in \Theta(f(n))$. We must have $x = 2$, and then $\log_4 a < 2 = \log_4 16$, so $a < 16$. However, we also need to check the regularity condition: $a(n/4)^2 \leq \delta n^2$. Since $a(n/4)^2 = a/16 \cdot n^2$ and $a < 16$, the regularity condition holds for $\delta = a/16 < 1$.

The answer: $T(n) \in \Theta(n^2)$ if either $a = 16$ and $x < 2$, or $a < 16$ and $x = 2$.

(b) [4 points] In our divide and conquer algorithm to find the closest pair of points in the plane, why was it sufficient, during the merge step, to consider points that were at most 11 positions apart in the list of points in the vertical strip around the dividing line sorted by their y-coordinates?

Solution: Divide the strip into $\delta/2 \times \delta/2$ boxes. Each box can contain at most one point, otherwise two points on the same side of the dividing line are closer than δ apart, which isn't possible by the definition of δ (the smallest distance found between two points on the same side of the dividing line). Hence, if two points 12 or more positions apart on the list, then there are at least two layers of boxes between them, and hence, their distance is at least δ.

(c) [6 points] Assume n is even. Describe an algorithm that find simultaneously the minimum and the maximum among n elements using $3n/2 - 2$ comparisons in the worst case.

Solution: Divide the numbers into $n/2$ pairs. Find the smallest and largest elements for each pair: $(s_1, l_1), \ldots, (s_{n/2}, l_{n/2})$ [$n/2$ comparisons]. Then apply standard algorithm to find the minimum among $s_1, \ldots, s_{n/2}$ ($n/2 - 1$ comparisons) and then the maximum is among $l_1, \ldots, l_{n/2}$ ($n/2 - 1$ comparisons). The total number of comparisons used is exactly $3n/2 - 2$.

3. [20 points] You and a group of your friends want to play tug-of-war (two teams pulling on opposites sides of a rope). Being the only computer scientist in the group, you have been asked to design an algorithm to build two teams A and B that are as equal as possible. You formalize the problem as follows:

- Each person i has a strength s_i.
- You would like the sum of the strengths of the people on team A to be as close to equal as possible to the sum of the strengths of the people on team B. That is, you want to minimize

$$\left| \sum_{i \in A} s_i - \sum_{j \in B} s_j \right|$$

Note that the two teams do not need to have the same number of people.

(a) [12 points] Write a pseudocode for a **greedy** algorithm to build the teams A and B. Your algorithm does not need to always succeed at minimizing the difference in total strength between the two teams, but it should make a good attempt at it.

Solution: Here is one possible algorithm.

```
function BUILDTEAMS(S)
    sort S by decreasing strength
    strengthA ← 0
    strengthB ← 0
    for i ← 0 to length[S] − 1 do
        if strengthA < strengthB then
            add person i to A
            strengthA ← strengthA + strength[i]
        else
            add person i to B
            strengthB ← strengthB + strength[i]
        end if
    end for
    return A, B
end function
```

Comments: The idea here is to make a greedy choice for each interval that will make the teams as equal as possible: add the next person to a weaker team. Also to avoid dealing with a strongest person at the end, which could make the teams unbalanced at the end, the people are sorted by their strength in decreasing order.

(b) [4 points] Analyze the time complexity of your algorithm from part (a). Specify only as much of your implementation (for instance, data structures) as needed for your analysis. Try to obtain as tight bound on the running time as possible, but make sure that your answer is correct (otherwise you will lose all points for this part). Justify your answer!

Solution: The algorithm above runs in $O(n \log n)$ time, because of the sorting step. The remaining part of the algorithm takes time in $O(n)$, because loop iterates n times and each iteration take a constant time.

(c) [4 points] Give an example where your algorithm will not return the optimal solution (the one that minimizes the strength difference between the two teams).

Solution: Suppose we have six people with strengths 10, 9, 6, 5, 4 and 2. The algorithm from part (a) will make two teams with strengths $10 + 5 + 2 = 17$ and $9 + 6 + 4 = 19$. However there is a solution where both total strengths are equal: $10 + 6 + 2 = 9 + 5 + 4 = 18$.

4. [8 points] Let $G = (V, E)$ be an undirected graph with costs $c(e) \geq 0$ for all edges $e \in E$. Assume that the edge costs are distinct. Assume you are given a minimum-cost spanning tree T in G. (You don't need to find or verify it, you can assume it is a minimum-cost spanning tree). Now assume that a new edge is added to E, connecting two nodes $v, w \in V$ with cost c that is distinct from costs of edges already in the graph.

   Describe an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$ time. Please note any assumptions you make about what data structure is used to represent the tree T and the graph G. Justify briefly why your algorithm is correct.

Solution: A simple algorithm is to find a path from v to w in T using BFS or DFS. If we are using a adjacency list, this is linear to the number of edges, since when finding a path we can visit every edge at most twice (once in each direction). If any of the edges on this path has a cost larger than c, then the answer is No, otherwise it's Yes. This follows by a Cycle Property applied to the cycle containing the path and the new edge $\{v, w\}$, which states that the edge with the highest cost is not in any MST of the graph.

5. [12 points] Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some integer d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a **local minimum** if the label x_v is less than the label x_w for all nodes w that are connected to v by an edge.

   You are given such a complete binary tree T, but the labeling is only specified in the following way: for each node v, you can determine the value x_v by **probing** the node v. **Design** a **divide and conquer** recursive algorithm that find a local minimum in a T and show that the number of probes it will make is in $O(\log n)$. Write *pseudocode* to specify your algorithm. Remember to justify that your algorithm will make $O(\log n)$ probes.

Solution: Algorithm:

```
1: function LocalMinimum(T, v)▷ v is either the root or x_v is smaller than the value of the parent of v
2:     if v is a leaf or x_v is less than values of children of v then
3:         return v
4:     else
5:         let u be a child of v such that x_u < x_v
6:         LocalMinimum(T, u)
7:     end if
8: end function
```

The initial call is to LOCALMINIMUM(T, r), where r is the root of T.

Number of probes: Clearly at each recursive call, the algorithm moves down in the tree. Hence, it has to terminate in $\log n$ steps. In each step it probes a constant number of nodes of T. Hence, the total number of probes is $O(\log n)$ as required.

6. [10 points] Consider a marking caching algorithm from class:

```
1:  procedure MARKINGALGORITHM(σ)
2:      start with all cache items unmarked
3:      for a request to item s in σ do
4:          if s is not in the cache then
5:              if all cache items are marked then
6:                  unmark all cache items                    ▷ new phase
7:              end if
8:              evict an unmarked item from the cache
9:          end if
10:         mark s
11:     end for
12: end procedure
```

Let r be the number of phases and let c_j be the number of fresh items in phase j (not requested in the previous phase or not initially in the cache if $j = 1$).

Second, consider the optimal (greedy) algorithm from class (furthest-in-future). Let f_j be the number of evictions/misses the algorithm does in phase j.

(a) [2 points] Argue that $f_1 \geq c_1$.

Solution: Since c_1 items requested in phase 1 are not originally in the cache, they will require c_1 evictions in this phase.

(b) [4 points] Argue that for every $j < r$, $f_j + f_{j+1} \geq c_{j+1}$.

Solution: In phases j and $j+1$, $k + c_{j+1}$ distinct items are requested. k of them might be in the cache at the beginning of phase j. The remaining c_{j+1} of them will require evictions during phases j and $j+1$.

(c) [4 points] Show that the number of evictions/misses of the optimal greedy algorithm is at least $\frac{1}{2}\sum_{j=1}^{r} c_j$.

Solution: Summing all inequalities together, we get on the left hand side: $f_1 + (f_1 + f_2) + (f_2 + f_3) + \cdots + (f_{r-1} + f_r) \leq 2\sum_{j=1}^{r} f_r$. On the right hand side, we get $\sum_{j=1}^{r} c_j$. Hence, the number of evictions of opt. algorithm $= \sum_{j=1}^{r} f_r \geq \frac{1}{2}\sum_{j=1}^{r} c_j$.

7. [14 points] Suppose we have a binary counter. The counter is represented by a singly linked list of bits, where the least-significant (rightmost) bit is at the head of the list. For instance, a counter containing the value 10010 would be represented by the linked list $head \to 0 \to 1 \to 0 \to 0 \to 1 \to nil$. The counter is initialized at 0 (so the linked list is initially $head \to 0 \to nil$). The counter supports the following two operations:

- INCREMENT: adds 1 to the counter.
- DOUBLE: doubles the value of the counter.

Use amortized analysis to show that the worst-case running time of any sequence of n INCREMENT and DOUBLE operations is in $O(n)$.

Solution: Let $\Phi(C_i) =$ the number of ones in the counter. Clearly $\Phi(C_0) = 0$ and $\Phi(C_i) \geq 0$.

- INCREMENT: Assume the counter ends with c ones (preceded by a zero or nil). (If the rightmost bit of the counter is 0, then $c = 0$.) Then the real cost of increment is $c + 1$. The change in the potential function: $\Phi(C_{new}) - \Phi(C_{old}) = -c + 1$, as INCREMENT operation will change c trailing ones to zeros and either change one preceding zero to one or add a one at the end of the linked list. Hence, the amortized cost is $c + 1 - c + 1 = 2$.
- DOUBLE: The real cost is 1, since it's enough to insert a zero at the head of the linked list. The potential function does not change. Hence, the amortized cost is 1.

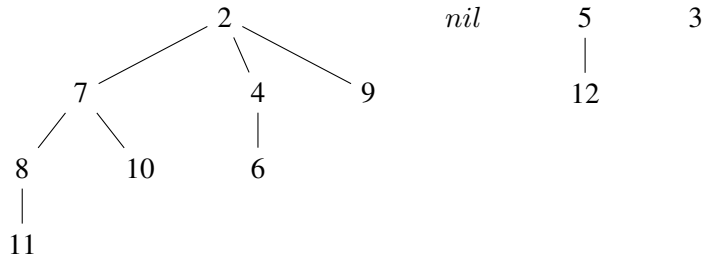
The amortized cost of n operations is at most $2n$, and hence, the real cost of n operations is bounded by $2n \in O(n)$.

8. [16 points] Binomial Heaps.

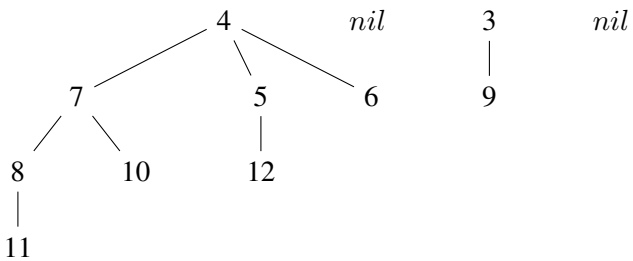
(a) [4 points] Explain why a binomial tree of order k has 2^k nodes.

Solution: A binomial tree of order k can be composed from two binomial trees of order $k - 1$, hence, it has exactly twice more nodes than a binomial tree of order $k - 1$. A binomial tree of order 0 has one node, which is 2^0. The number of nodes keeps doubling with each order increment by one, so a tree of order k has 2^k nodes.

(b) [6 points] Consider the following state of the binomial heap (nil represents the empty element in the ordered list):



Draw the state of the binomial heap after operation EXTRACTMIN(). Circle your final answer.



Solution:

Comments: Alternative solutions are possible: we will have three trees of order 1 (with two nodes each), and the procedure does not clearly specify which two should be merged and which tree will stay in the final result.

(c) [2 points] Assume that a binary heap contains 36 elements. How many binomial trees does it contain and what are their orders.

Solution: Since $36 = 2^5 + 2^2$, the heap contains 2 trees, one with order 5 and other with order 2.

(d) [4 points] Discuss the relationship between inserting an element into a binomial heap and incrementing a binary counter by one. Explain why this implies that building a binomial heap takes $O(n)$ time.

Solution: The orders of the trees of a binomial heap correspond to ones in binary representation of n, where n is the number of elements in the heap. Inserting one element to the heap, will create a binomial tree of order 0, and which is then merged c times with the trees in the heap, where c is the number of trailing ones of the binary representation of n, which exactly corresponds to the work done when incrementing a counter by 1.

Binomial heap is built by performing INSERT n times, which corresponds to incrementing a binary counter (initialized to 0) n times. Since amortized analysis shows that n counter increments take time in $O(n)$, building a heap takes time in $O(n)$ as well.

9. [16 points] **Knapsack Problem**: Given n items, where the i-th item has weights w_i and value v_i, and given value W, the goal is to find a subset S of the items with **the maximum sum of their values** subject to the **restriction** that the total weight of chosen items is **at most** W. Assume that W and $w_1, \ldots, w_n$ are positive integers.

Define the subproblems and specify the **recurrence relation** between the optimal values of subproblems.

Then write a **pseudocode** for a dynamic programming algorithm with complexity $O(nW)$ that finds the value of an optimal solution (it does not need to find an optimal solution). **Do not use** the memoization technique: your algorithm should be iterative, not recursive.

Finally, write a pseudocode for a backtracking algortihm that would reconstruct an optimal solution.

Solution: *Comments:* A natural way how design a recurrence for this problem would be decide whether the last item is added or not. However, if the item is added then in the subproblem we have a smaller limit on the weight than in the original problem. Hence, we will need two parameters i representing the last item to consider and w representing the limit on the weight.

Definition of subproblems (for $i = 0, \ldots, n$ and $w = 0, \ldots, W$): $V[i, w] =$ max. value of selected items among items $1, \ldots, i$ with the total weight at most w.

Recurrence: If the item has weight smaller or equal to the weight limit, we have two choices, as described above. Otherwise, there is only one choice: the item cannot be included. The base case is, when there is 0 items to consider:

$$V[i, w] = \begin{cases} \max\{V[i-1, w], V[i-1, w-w_i] + v_i\} & \text{if } i > 0 \text{ and } w_i \leq w \\ V[i-1, w] & \text{if } i > 0 \text{ and } w_i > w \\ 0 & \text{if } i = 0 \end{cases}$$

Algorithm: To compute $V[i, w]$ we need values of $V[i-1, *]$, so the main loop should iterate i from 0 to n, and inside we can iterate through different values of w between 0 and W in any order.

function KNAPSACK($w_1, \ldots, w_n, v_1, \ldots, v_n, W$)

▷ initialization:

for $w \leftarrow 0$ to W **do**

```
        V[0, w] ← 0
    end for
                                                    ▷ filling up the table:
    for i ← 1 to n do
        for w ← 0 to W do
            V[i, w] ← V[i − 1, w]
            if w_i ≤ w and V[i − 1, w − w_i] + v_i > V[i, w] then
                V[i, w] ← V[i − 1, w − w_i] + v_i
            end if
        end for
    end for
    return V[n, W]
end function
```

Comments: The complexity of this algorithm is $O(nW)$. The algorithm relies on the fact that the weights are integers. If would allow weights to be any positive real numbers, then there would be no way how to iterate through all possible weights (the second parameter). If you are wondering if there is a different efficient algorithm for this problem in this case (either a different DP algorithm or some other algorithm), the answer is No. This problem (with weights being real numbers or even with integer weights that can be exponential in n) is NP-complete, which means that it's very unlikely there is an efficient algorithm for this problem (unless many other hard problems have polynomial-time solutions).

Points total: 140

End of exam.