College of Computer & Information Science
Northeastern University
Fall 2018

CS5800
Algorithms
12 October 2018

Sample Solution to Midterm I

Problem 1. (2 + 2 + 1 + 2 = 7 points) Gale-Shapley algorithm

For the 3 men m_1, m_2, m_3 and the 3 women w_1, w_2, w_3 the following lists of preferences are given:

man	first	second	third
m_1	w_2	w_1	w_3
m_2	w_3	w_2	w_1
m_3	w_1	w_2	w_3

woman	first	second	third
w_1	m_1	m_2	m_3
w_2	m_2	m_1	m_3
w_3	m_1	m_3	m_2

(a) List the stable matching returned by the Gale-Shapley algorithm for the above lists of preferences assuming the **men** propose to the **women**.

Answer:

$$\{(m_1, w_2), (m_2, w_3), (m_3, w_1)\}$$

(b) List the stable matching returned by the Gale-Shapley algorithm for the above lists of preferences assuming the **women** propose to the **men**.

Answer:

$$\{(m_1, w_1), (m_2, w_2), (m_3, w_3)\}$$

(c) Does there exist a man that got a better result in part (a) than in part (b)? If so, indicate which man.

Answer: Yes. m_1.

(d) Prove that if the Gale-Shapley algorithm with the men proposing to the women returns the same matching as the Gale-Shapley algorithm with the women proposing to the men, then there is a unique stable matching for the given instance (i.e., there can be no other stable matching).

Answer: The proof is by contradiction. Let M be the stable matching returned by the Gale-Shapley algorithm (either with the men proposing or the women proposing). If M is not the only stable matching, let M' be a different stable matching. Since M and M' are different, there is a pair (m, w) in M such that m is matched to a different woman $w' \neq w$ in M' and w is matched to a different man $m' \neq m$ in M.

Since M is both man-optimal and woman-optimal, w is the most preferred woman for m *among all matches in stable matchings*. Similarly, m is the most preferred man for w *among all matches in stable matchings*. Since M' is a stable matching, this means w prefers m to m' and m prefers w to w'. But this implies that M' is not stable, leading to a contradiction.

Problem 2. (4 + 4 = 8 points) Asymptotic notation and recurrences

(a) Let $f(n)$ and $g(n)$ be asymptotically positive and monotonically increasing functions. Decide whether the following statement is true, and give a proof or a counterexample.

If $f(n) = O(g(n))$, then $\sqrt[4]{f(n)} = \Omega(\sqrt[4]{g(n)})$.

Answer: False. Take $f(n) = n^2$ and $g(n) = n^4$. Clearly, $n^2 = O(n^4)$. However, $\sqrt[4]{f(n)} = \sqrt{n}$, while $\sqrt[4]{g(n)} = n$; and $\sqrt{n} \neq \Omega(n)$ since $\lim_{n\to\infty} \sqrt{n}/n = 0$.

(b) Let $T(n)$ be defined by:

$$T(n) = \begin{cases} 1, & n = 1; \\ 9T\left(\lfloor \frac{n}{3} \rfloor\right) + n^2, & n > 1. \end{cases}$$

Solve the recurrence to give a tight Θ-bound for $T(n)$. Show your work. You may ignore floors and ceilings in your calculation.

Answer: We use the Master Theorem. We have $a = 9, b = 3$ and $f(n) = n^2$. We obtain $n^{\log_b a} = n^2$. By the second case of the theorem, we obtain $T(n) = \Theta(n^2 \log n)$.

We can also apply the recursion tree method. To solve the recurrence precisely, we can also use the substitution method or induction to prove the same result.

Problem 3. (3 + 4 = 7 points) A New Sorting Algorithm

Consider the following "elegant" sorting algorithm.

```
ElegantSort(A, i, j)
1. if A[i] > A[j]
2.     exchange A[i] ↔ A[j]
3. if i + 1 ≥ j
4.     return
5. k ← ⌊(j − i + 1)/3⌋          // Round down.
6. ElegantSort(A, i, j − k)      // First two-thirds.
7. ElegantSort(A, i + k, j)      // Last two-thirds.
8. ElegantSort(A, i, j − k)      // First two-thirds again.
```

(a) Give a recurrence for the worst-case running-time of ElegantSort$(A, 1, n)$.

Answer:

$$T(n) = 3T(\lceil 2n/3 \rceil) + 1.$$

(b) Solve the recurrence to obtain a tight asymptotic (Θ-notation) bound on the worst-case running time of ElegantSort.

Answer: We use the Master method. Since $b = 3, c = 3/2$, we get $E = \log_{3/2} 3$. We have $f(n) = \Theta(1)$; clearly, there exists an $\varepsilon > 0$ (for example, $\varepsilon = \log_{3/2} 3$) such that $f(n) = O(n^{E-\varepsilon})$. We thus invoke case 1 of the Master theorem to obtain $T(n) = \Theta(n^{\log_{3/2} 3}$.

We can also use the Recursion Tree method, or prove by induction.

(c) 2 Bonus points

Prove that ElegantSort$(A, 1, n)$ correctly sorts the input array $A[1..n]$. Partial credit will be given sparingly.

Answer: Here is a proof by induction, which is not completely formal, but captures the essentials. A more formal proof is given below. For the base case, we consider $n = 1$ and $n = 2$. In the former case, the algorithm does nothing, as expected. In the latter, the swap on line 1 sorts the array.

For the induction hypothesis, suppose ElegantSort correctly sorts any array of length less than n for $n > 2$. Consider input array of length $n > 2$; i.e., the call ElegantSort$(A, 1, n)$. After the initial swap, it makes three recursive calls: the first two-third, the last two-third, and then the first two-third. By the induction hypothesis, each of these calls sort correctly. After the first call (step 5), the first third of the array has elements that are smaller than the second third. As a consequence, none of the elements in the first third of the array after step 5 can be the last third of the array. After the second call (step 6), the elements in the last third are greater than the elements in the middle third. So the elements in the middle third (which are in sorted order) can also not be valid elements in the last third. As a result, elements in the last third are in the correct position. Finally, the third call (step 7) ensures that the first two third of the array is sorted. This makes the whole array sorted.

Formal Proof. To prove that ElegantSort(A, i, j) sorts correctly, we use induction on $\ell = j - i + 1$. (That is, ℓ is the number of elements in $A[i..j]$.) For the induction basis, we consider two cases: $\ell = 1$ or $\ell = 2$, i.e., $i = j$ or $i = j - 1$. For each of these cases, in Steps 1 and 2, $A[i]$ and $A[j]$ are exchanged if $A[i] > A[j]$. Also, in both cases the condition in Step 3 holds. Therefore, ElegantSort(A, i, j) returns after having sorted A.

For the induction hypothesis, we assume that ElegantSort sorts correctly for all arrays of length less than m, where $m > 2$. We now show that ElegantSort sorts correctly for all inputs of length m. Since $m > 2$, the condition in Step 3 is not true; hence ElegantSort does not return in Step 4. Moreover, for the discussion that follows, it does not matter whether the exchange of $A[i]$ and $A[j]$ took place in Step 2. So we now consider Steps 5 through 8. The variable k is set to $\lfloor (j - i + 1)/3 \rfloor$ in Step 5. Since $j - i + 1 \geq 3$, $k \geq 1$; therefore, $j - i - k + 1 = m - k < m$. Therefore, the length of the array $A[i..j - k]$ on which ElegantSort is called in Step 6 is less than m. By the induction hypothesis, it follows that $A[i..j - k]$ is in sorted order after Step 6. Hence, A has the following property: (P1) for all x in $[i..i + k - 1]$ and y in $[i + k..j - k]$, $A[x] \leq A[y]$. In particular, we see that there are at least k elements in $A[i + k..j]$ that are greater than or equal to every element in $A[i..i + k - 1]$. Next, ElegantSort sorts $A[i + k, .., j]$. Again, the length of the array is less than m. So we invoke the induction hypothesis and claim that after Step 7, $A[i + k..j]$ is sorted. Since $A[i + k..j]$ is sorted, A has the following property: (P2) $A[j - k + 1..j]$ is sorted correctly, and (P3) for all x in $[i + k..j - k]$ and y in $[j - k + 1..j]$, $A[x] \leq A[y]$.

Since the elements in indices $[i..i + k - 1]$ do not move, the following property (that follows from (P1)) still holds: (P4) there are at least k elements in $A[i + k..j]$ that are greater than or equal to every element in $A[i..i + k - 1]$. Therefore, combining with property (P3) above, we have: (P5) for all x in $[i..j - k]$ and y in $[j - k + 1..j]$, $A[x] \leq A[y]$. Thus, after step 6, properties (P2) and (P5) hold.

Finally, we apply ElegantSort on $A[i..j - k]$. Again the length of the array, $j - k - i + 1$ is less than m. So applying the induction hypothesis, we obtain that $A[i..j - k]$ is sorted

correctly. Moreover, since the indices $[j - k + 1..j]$ are unaffected, properties (P2) and (P5) still hold. We thus obtain that the entire array $A[i..j]$ is sorted correctly.

Problem 4. (8 points) Finding a local maximum

Given a set of $n \geq 2$ distinct numbers in an array $A[0 \ldots n-1]$, the element $A[i]$ is called a *local maximum* if one of the following holds:

- $i = 0$ and $A[0] > A[1]$, or
- $i = n-1$ and $A[n-1] > A[n-2]$, or
- $0 < i < n-1$ and $A[i-1] < A[i]$ *and* $A[i] > A[i+1]$.

Note that while the maximum element of A is always a local maximum of A, the converse may not be true. For example, given the array $A = [12, 15, 8, 22, 36, 19, 17, 54, 63, 7, 42]$, $A[1] = 15$, $A[4] = 36$, $A[8] = 63$, and $A[10] = 42$ are all local maxima.

Give an algorithm to determine a local maximum of an array A of numbers of length n. If A has multiple local maxima, your algorithm may return any one of them. Derive a tight Θ-bound on your algorithm's worst-case running time, which is defined as the number of elements of A accessed by your algorithm.

Your grade will be determined on the basis of the correctness of your algorithm and its efficiency, given by its worst-case running time. Partial credit may be given for non-optimal algorithms provided they are correct and well explained.

Answer: Here is an algorithm to find a local maximum of A.

findLocalMaximum(A, L, U)
// find a local maximum between $A[L]$ and $A[U]$
if $L = U$: // single element in range
  return A[L]
if $A[L] > A[L+1]$:
  return $A[L]$
if $A[U] > A[U-1]$:
  return A[U]
// $(L - U) \geq 2$
$M = \lfloor (L+U)/2 \rfloor$
if $A[M] > A[M-1]$ and $A[M] > A[M+1]$:
  return $A[M]$
else if $A[M-1] > A[M]$:
  // $A[L] < A[L+1]$ and $A[M] < A[M-1]$
  // There exists a local maximum between $A[L]$ and $A[M-1]$
  findLocalMaximum(A, L, M-1)
else
  // $A[M] < A[M+1]$ and $A[U] < A[U-1]$
  // There exists a local maximum between $A[M+1]$ and $A[U]$
  findLocalMaximum$(A, M+1, U)$

Invoke findLocalMaximum$(A, 0, n-1)$ to solve problem as stated.

Worst-case analysis: The algorithm checks the end points, computes and checks the mid-point in a constant number of accesses of the array elements. It then reduces the problem to finding a local maximum in an array of half the size.

The recurrence relation governing the complexity of the algorithm can be stated as:

$$T(n) = T(n/2) + c,$$

for some constant c. This is similar to binary search and can be shown to be $\Theta(\log n)$.