College of Computer & Information Science
Northeastern University
Spring 2019

CS5800
Algorithms
14 February 2019

Practice Problems for Midterm 1

1. Stable Matching

Decide whether you think each of the following statements is true or false. If it is true, give a brief justification. If it is false, give a counterexample.

(a) *True or False?* In every instance of the stable matching problem, there is a stable matching containing a pair (h, s) such that h is ranked first on the preference list of s and s is ranked first on the preference list of h.

Answer: *False.*
We provide the following counterexample.
Consider a simple example where there are 2 hospitals h_1 and h_2 and two students s_1 and s_2.
h_1 prefers s_1 to s_2; h_2 prefers s_2 to s_1
s_1 prefers h_2 to h_1; s_2 prefers h_1 to h_2
There are 2 stable matchings for this instance of the problem.

$$S_1 = (h_1, s_1, h_2, s_2); S_1 = (h_1, s_2, h_2, s_1)$$

Neither contains a pair (h, s) such that h is ranked first on the preference list of s and s is ranked first on the preference list of h.

(b) *True or False?* If for a given instance of the stable matching problem, the Gale-Shapley algorithm with the hospitals proposing to the students returns the same matching as the Gale-Shapley algorithm with the students proposing to the hospitals, then there is a unique stable matching for this problem instance.

Answer: *True*:
We provide a proof by contradiction. We know that when applying the Gale-Shapley algorithm on sets H and S such that the hospitals propose to the students, the stable matching we get is:

$$M = (h, best(h)) : h \in H$$

Similarly, if the students propose to the hospitals, we get:

$$M = (s, best(s)) : s \in S$$

Assume there is more then 1 possible stable matching. Let M_1 be a stable matching obtained by the G-S algorithm when the hospitals propose to students, and let M_2 be a different stable matching. It follows that for some hospital h_1, there exist students s_1 and s_2, $s_1 \neq s_2$, such that (h_1, s_1) is a pair in M_1, and (h_1, s_2) is a pair in M_2.

We know that h_1 prefers s_1 over s_2. Let h_2 denote the hospital that s_1 is paired with in M_2. The fact that M_2 is stable means that s_1 prefers h_2 over h_1. (Otherwise, h_1 prefers s_1 over its pair s_2, and s_1 prefers h_1 over her pair h_2. That would mean that M_2 is not stable.)

It follows that if the students propose to the hospitals, s_1 will end up with a hospital that she prefers over h_1. In particular, s_1 will not end up with h_1. We see that if there exist more than 2 different stable matchings, then the G-S algorithm returns different matchings depending on whether the men or women propose. It follows that by applying the G-S algorithm twice (once with the men proposing to the women and once with the women proposing to the men), we can determine whether there exists a unique stable matching.

2. Asymptotic Notation and Order of Growth

(a) Sort the following sets of functions in the order of their asymptotic growth:

(i)

$$\log\ n,\ n^{\frac{1}{4}},\ \frac{n}{\log n}$$

Answer;
The correct order is:

$$\log\ n < n^{\frac{1}{4}} < \frac{n}{\log n}$$

We will examine each pair to show the order between consecutive pairs are correct.

$$\lim_{n\to\infty} \frac{log\, n}{n^{\frac{1}{4}}} = \lim_{n\to\infty} \frac{\frac{1}{n}}{1/4\ n^{-3/4}} = \lim_{n\to\infty} \frac{4}{n^{\frac{1}{4}}} = 0$$

Thus $\log\ n$ is $O(n^{\frac{1}{4}})$.

Next we consider $n^{\frac{1}{4}}$ and $\frac{n}{\log n}$.

$$\lim_{n\to\infty} \frac{n^{\frac{1}{4}}}{\frac{n}{\log n}} = \lim_{n\to\infty} \frac{\log\ n}{n^{\frac{3}{4}}} = \lim_{n\to\infty} \frac{\frac{1}{n}}{3/4\ n^{-1/4}} = \lim_{n\to\infty} \frac{4}{3n^{\frac{3}{4}}} = 0$$

This proves that $n^{\frac{1}{4}}$ is $O(\frac{n}{\log n})$. QED.

(ii)

$$2^{\sqrt{\log\ n}},\ n^{\frac{4}{3}},\ 2^n$$

Answer;
The correct order is:

$$2^{\sqrt{\log\ n}} < n^{\frac{4}{3}} < 2^n$$

Once again, we will consider each pair in turn.
$2^{\sqrt{\log\ n}} = O(2^{\log n}) = O(n)$, and $n = O(n^{\frac{4}{3}})$. By transitivity of big-O, we have

$$2^{\sqrt{\log\ n}} = O(n^{\frac{4}{3}}) \tag{1}$$

Next we consider $n^{\frac{4}{3}}$ and 2^n. The former is a polynomial in n and the latter is exponential in n. Therefore,

$$n^{\frac{4}{3}} = O(2^n) \tag{2}$$

From (1) and (2), we have the stated ordering.

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

If $f(n) = \Omega(g(n))$, then $\sqrt{f(n)} = \Omega(\sqrt{g(n)})$

Answer;
$f(n) = \Omega(g(n))$ implies $\exists c > 0$ and $n_0 \geq 0$ such that

$$f(n) \geq c.g(n) \; \forall n \geq n_0$$

As $f(n)$ and $g(n)$ are given to be asymptotically positive and monotonically increasing functions, we can infer that

$$\sqrt{f(n)} \geq \sqrt{c}.\sqrt{g(n)} \; \forall n \geq n_0 \tag{3}$$

Since $\sqrt{c}$ is a positive constant, it follows From (3) above that $\sqrt{f(n)} = \Omega(\sqrt{g(n)})$.

3. Recurrence Relations

Derive an asymptotically tight bound for the following recurrence. Assume that $T(n)$ is $\Theta(1)$ for $n \leq 4$.

$$T(n) = T(n/2) + T(3n/4) + n^2.$$

Answer: Use recursion tree approach. First level would contribute n^2. Second level $13n^2/16$. Third level $(13/16)^2 n^2$. So we have

$$T(n) \leq n^2 + (13/16)n^2 + (13/16)^2 n^2 + \cdots \leq n^2/(1 - 13/16) = O(n^2).$$

T(n) is clearly $\Omega(n^2)$ since it is at least n^2. So $T(n) = \Theta(n^2)$.

4. Finding the ending of an array

You are given an infinite array A in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, using $O(\log n)$ accesses to the array.

Answer:

1. set $i = 1$;
2. while $A[i] < x$: $i = 2 * i$;
3. if $A[i] = x$; return i;
4. Binary search for x in $A[2^{i-1} + 1 \ldots 2^i - 1]$.

The while loop to identify an index i such that $A[i] > x$ is executed at most $\log_2 n + 1$ times. The binary search is also bounded above by $\log_2 n$ comparisons. Thus, the time complexity of the algorithm outlined above is $O(logn)$.

5. Finding the maximum sum subarray

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1 \ldots n]$ of numbers. Formally, the task is to find indices i and j, $1 \leq i \leq j \leq n$ such that the sum $\sum_{k=i}^{j} A[k]$ is as large as possible. Each number in the input array A could be positive, negative, or zero.
For example, for the array of values [-2, 1, -3, 4, -1, 2, 1, -5, 4], the contiguous subarray with the largest sum is [4, -1, 2, 1], with sum 6.

Give an efficient algorithm that takes as input an array $A[1 \ldots n]$ of positive and negative integers, and determines its largest sum subarray, State the worst-case time complexity of your algorithm.

Answer: The solution we outline here uses Divide and Conquer to split the input array of size n into two equal sized arrays of size $n/2$. As with the closest pair problem, the solution to the original problem of size n may be the solution to either subproblem, or one that spans across the two sets of numbers. The combining part then looks at the left half by starting at the midpoint and finding the subsequence that has the largest sum. Similarly, on the right half, the subsequence with the largest sum starting from the element following the midpoint (at its left most index). Combining the two provides the maximum sum subsequence that spans both halves.

The algorithm is outlined below:

```
1 MaxSumSubArray(A, l, r)
2 Comment: Base Case: Only one element
3 if l = r then
4 |   return A[ l ]
5 end
6 Comment: Find the mid point
7 m = ⌊(l + r)/2⌋
8 return max(MaxSumSubArray(A, l, m), MaxSumSubArray(Am + 1, r),
    MaxCrossingSum(Al, m, r))
```

Time Complexity

The MaxCrossingSum function performs a linear scan on the left subset and the right subset to find the maxCrossingSum. So, its time complexity is $\Theta(n)$, where n indicates the size of the array. Therefore, the time complexity of the function MaxSumSubArray is given by the following recurrence relation:

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

. This is a familiar recurrence that we have encountered with MergeSort and has a solution of $T(n) = \Theta(\log\ n)$.

```
1  MaxCrossingSum(Ar, l, m, r)
2  Comment: Include elements on left of mid, m
3  p ← 0
4  lsum ← −10000
5  for i = m; I < l − 1; i = i − 1 do
6  |  p ← p + A[i] if p > lsum then
7  |  |  lsum ← p
8  |  end
9  end
10 Comment: Include elements on right of mid
11 q ← 0
12 rsum ← −1000
13 for i = m + 1; I > r; i = i + 1 do
14 |  q ← p + A[i] if q > rsum then
15 |  |  rsum ← q
16 |  end
17 end
18 Comment: Return sum of elements on left and right of mid
19 return lsum + rsum
```