Problem Set 1 Solutions

This problem set is due **at 11:59pm** on **Thursday, February 12, 2015.**

Exercise 1-1. Asymptotic Growth

Sort all the functions below in increasing order of asymptotic (big-O) growth. If some have the same asymptotic growth, then be sure to indicate that. As usual, $\lg$ means base 2.

1. $5n$
2. $4 \lg n$
3. $4 \lg \lg n$
4. n^4
5. $n^{1/2} \lg^4 n$
6. $(\lg n)^{5 \lg n}$
7. $n^{\lg n}$
8. 5^n
9. 4^{n^4}
10. 4^{4^n}
11. 5^{5^n}
12. 5^{5n}
13. $n^{n^{1/5}}$
14. $n^{n/4}$
15. $(n/4)^{n/4}$

Solution: $4 \lg \lg n < 4 \lg n < n^{1/2} \lg^4 (n) < 5n < n^4$
$< (\lg n)^{5 \lg n} < n^{\lg n} < n^{n^{1/5}} < 5^n < 5^{5n}$
$< (n/4)^{n/4} < n^{n/4} < 4^{n^4} < 4^{4^n} < 5^{5^n}$

Exercise 1-2. Solving Recurrences

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 4T(n/4) + 5n$

Solution: $T(n) = \Theta(n \lg n)$, Case 2 of the Master Theorem.

(b) $T(n) = 4T(n/5) + 5n$

Solution: $T(n) = \Theta(n)$, Case 3 of the Master Theorem.

(c) $T(n) = 5T(n/4) + 4n$

Solution: $T(n) = \Theta(n^{\log_4(5)}) = \Theta(n^{\lg\sqrt{5}})$, Case 1 of the Master Theorem.

(d) $T(n) = 25T(n/5) + n^2$

Solution: $T(n) = \Theta(n^2 \lg(n))$. Case 2 of the Master Theorem.

(e) $T(n) = 4T(n/5) + \lg n$

Solution: $T(n) = \Theta(n^{\log_5 4})$. Case 1 of the Master Theorem.

(f) $T(n) = 4T(n/5) + \lg^5 n\sqrt{n}$

Solution: $T(n) = \Theta(n^{\log_5 4})$. Case 1 of the Master Theorem.

(g) $T(n) = 4T(\sqrt{n}) + \lg^5 n$

Solution: Change variables. Assume that n is a power of 2, let $n = 2^m$. We get $T(2^m) = 4T(2^{m/2}) + m^5$.
If we define $S(m) = T(2^m)$, then we get the recurrence $S(m) = S(m/2) + m^5$. By case 3 of the Master Theorem, $S(m) = \Theta(m^5)$, that is, $T(2^m) = \Theta(m^5)$.
Changing the variable back ($m = \lg n$), we have $T(n) = \Theta(\lg^5(n))$.

(h) $T(n) = 4T(\sqrt{n}) + \lg^2 n$

Solution: Similar to the previous case. Let $n = 2^m$. We get $T(2^m) = 4T(2^{m/2}) + m^2$.
Letting $S(m) = T(2^m)$, we get $S(m) = 4S(m/2) + m^2$. By case 2 of the Master Theorem, $S(m) = \Theta(m^2 \lg m)$, that is, $T(2^m) = \Theta(m^2 \lg m)$.
Changing the variable back ($m = \lg n$), we have $T(n) = \Theta(\lg^2 n \lg\lg n)$.

(i) $T(n) = T(\sqrt{n}) + 5$

Solution: Again similar. Let $n = 2^m$. We get $T(2^m) = T(2^{m/2}) + 5$.
Letting $S(m) = T(2^m)$, we get $S(m) = S(m/2)+5$. This solves to $S(m) = \Theta(\lg m)$, so $T(2^m) = \Theta(\lg m)$. Changing the variable back, we get $T(n) = \Theta(\lg \lg n)$.

(j) $T(n) = T(n/2) + 2T(n/5) + T(n/10) + 4n$

Solution: $\Theta(n \lg n)$, using explicit trees.

Problem 1-1. Restaurant Location [25 points]

Drunken Donuts, a new wine-and-donuts restaurant chain, wants to build restaurants on many street corners with the goal of maximizing their total profit.

The street network is described as an undirected graph $G = (V, E)$, where the potential restaurant sites are the vertices of the graph. Each vertex u has a nonnegative integer value p_u, which describes the potential ***profit*** of site u. Two restaurants cannot be built on adjacent vertices (to avoid self-competition). You are supposed to design an algorithm that outputs the chosen set $U \subseteq V$ of sites that maximizes the total profit $\sum_{u \in U} p_u$.

First, for parts (a)–(c), suppose that the street network G is acyclic, i.e., a tree.

(a) [5 points]
Consider the following "greedy" restaurant-placement algorithm: Choose the highest-profit vertex u_0 in the tree (breaking ties according to some order on vertex names) and put it into U. Remove u_0 from further consideration, along with all of its neighbors in G. Repeat until no further vertices remain.
Give a counterexample to show that this algorithm does not always give a restaurant placement with the maximum profit.

Solution: E.g., the tree could be a line (9, 10, 9).

(b) [9 points]
Give an efficient algorithm to determine a placement with maximum profit.

Solution:
[Algorithm]
We can use dynamic programming to solve the problem in $O(n)$ time. First pick any node u_0 as the root of the tree and sort all the nodes following a depth-first search

(DFS). Store the sorted nodes in array N. Due to the definition of DFS, a parent node appears earlier than all of its children in N.

For each node v, define $A(v)$, the best cost of a placement in the subtree rooted at v if v is included, and $B(v)$, the best cost of a placement in the subtree rooted at v if v is not included. The following recursion equations can be developed for $A()$ and $B()$:

If v is a leaf, then $A(v) = p_v$ and $B(v) = 0$.

If v is not a leaf, then

$$A(v) = p_v + \sum_{u \in v.children} B(u)$$
$$B(v) = \sum_{u \in v.children} \max(A(u), B(u))$$

For each node v in N, in the reverse order, compute $A(v)$ and $B(v)$. Finally the $\max(A(u_0), B(u_0))$ is the maximum profit.

The placement achieving this maximum profit can be derived by recursively comparing $A()$ and $B()$ starting from the root. The root u_0 should be included if $A(u_0) > B(u_0)$ and excluded otherwise. If u_0 is excluded, we go to all of u_0's children and repeat the step; if u_0 is included, we go to all of u_0's grandchildren and repeat the step. This algorithm outputs an optimal placement by making one pass of the tree.

[Correctness]

In the base case where v is a leaf node, the algorithm outputs the optimal placement which is to include the node.

In an optimal placement, a node v is either included, which removes all its children, or not, which adds no constraints. By induction, if all the children of v have correct $A()$ and $B()$ values, then $A(v)$ and $B(v)$ will also be correct and the maximum profit at v is derived. Since the array N is sorted using DFS and processed in the reverse order, child nodes are guaranteed to be processed before their parents.

[Timing Analysis]

Sorting all nodes using DFS takes $O(n)$ time. The time to compute $A(v)$ and $B(v)$, given the values for the children, is proportional to $degree(v)$. So the total time is the sum of all the degrees of all the nodes, which is $O(|E|)$ where $|E|$ is the total number of edges. For a tree structure, $|E| = n - 1$ so the time for finding all $A()$ and $B()$ is $O(n)$. Finally, using the derived $A()$ and $B()$ to find the optimal placement visits each node once and thus is $O(n)$.

Overall, the algorithm has $O(n)$ complexity.

(c) [6 points]

Suppose that, in the absence of good market research, DD decides that all sites are equally good, so the goal is simply to design a restaurant placement with the largest number of locations. Give a simple greedy algorithm for this case, and prove its correctness.

Solution:

[Algorithm]

Similar to Part (b), pick any node u_0 as the root of the tree and sort all the nodes following a depth-first search (DFS) and store the sorted nodes in array N. For each valid node in N, in the reverse order, include it and remove its parent from N.

[Correctness]

Claim: This greedy algorithm yields an optimal placement.

Lemma 1: *For any tree with equal node weights, there is an optimal placement that contains all the leaves.*

If there exists an optimal placement O that excludes some leaf v. Simply add v to O and if necessary, remove its parent. The result is no worse than placement O. Repeat for all leaves until we have an optimal placement with all the leaves included.

The main claim can then be proved using the Lemma 1.

Due to DFS sorting, the first valid node in N, in reverse order, must be a leaf node. According to Lemma 1, it can be included in the optimal placement and its parent excluded. When nodes from N are processed in reverse order, each processed node is the last valid one in the current N and is thus a leaf. And including the leaf is part of an optimal solution for the corresponding subtree. Overall, an optimal placement is derived for the original tree.

[Timing Analysis]

Sorting nodes using DFS takes $O(n)$ time. The greedy algorithm also takes $O(n)$ time since it processes each node once. Overall the algorithm has $O(n)$ complexity.

(d) [5 points]

Now suppose that the graph is arbitrary, not necessarily acyclic. Give the fastest correct algorithm you can for solving the problem.

Solution: A simple algorithm is to try all possible subsets of vertices for U (2^V subsets in total), test whether each has the required independence property (only one node should be included for each edge, $|E|$ edges in total), compute the total profit for each valid solution (which takes $O(V)$), and take the best. This algorithm runs in $O(2^V|E|)$ time. This is the intended solution.

In fact, this problem is exactly Maximum Independent Set problem, which is known to be NP-complete. So unless P = NP, it has no polynomial-time algorithm. (In fact, assuming something stronger called the Exponential Time Hypothesis, there is no $2^{o(V)}$-time algorithm.)

Problem 1-2. Radio Frequency Assignment [25 points]

Prof. Wheeler at the Federal Communications Commission (FCC) has a huge pile of requests from radio stations in the Continental U.S. to transmit on radio frequency 88.1 FM. The FCC is happy to

grant all the requests, provided that no two of the requesting locations are within Euclidean distance 1 of each other (distance 1 might mean, say, 20 miles). However, if any are within distance 1, Prof. Wheeler will get annoyed and reject the entire set of requests.

Suppose that each request for frequency 88.1 FM consists of some identifying information plus (x, y) coordinates of the station location. Assume that no two requests have the same x coordinate, and likewise no two have the same y coordinate. The input includes two sorted lists, L_x of the requests sorted by x coordinate and L_y of the requests sorted by y coordinate.

(a) [3 points]

Suppose that the map is divided into a square grid, where each square has dimensions $\frac{1}{2} \times \frac{1}{2}$. Why must the FCC reject the set of requests if two requests are in, or on the boundary of, the same square?

Solution: Because they are within distance 1. ($\sqrt{2}/2 < 1$.)

(b) [14 points]

Design an efficient algorithm for the FCC to determine whether the pile of requests contains two that are within Euclidean distance 1 of each other; if so, the algorithm should also return an example pair. For full credit, your algorithm should run in $O(n \lg n)$ time, where n is the number of requests.

Hint: Use divide-and-conquer, and use Part (a).

Solution:

[Algorithm]

The intended divide-and-conquer algorihtm is as follows.

Divide: Use list L_x to find a vertical line that divides the requests into two subsets of size approximately $n/2$, and does not pass through any of the requests. The L_x and L_y for both subsets should be computed.

Conquer: If a subset contains less than or equal to a constant C (e.g., $C = 2$) requests, check if any pair of them is within distance 1 and return the pair if it exists. Otherwise if the subset contains more than C requests, divide the subset into smaller subsets.

Merge: We only need to check pairs of requests in a "stripe" S of width 2 centered on the boundary between the two halves. We merge all the requests inside S into a (possibly reduced) list L, and still keep them sorted by y coordinates. For a request $r \in L$, check the distance between r and 7 requests in L following r (which have larger y coordinates). If any pair has distance within 1, return the pair.

[Correctness]

After the divide and conquer phase, if the two subproblems do not find any violation, the only place violations can still happen is in the stripe S, which we check in the merging phase. In the merging phase, it is sufficient to check only 7 requests following r for the following reason.

If we divide stripe S into squares of size 1/2. Each square contains at most one request, otherwise a pair within the same square should have been detected by the subproblems. For a request $r \in L$, a following request within distance 1 must be inside a 2-by-1 square spanning across the boundary. There are only 8 such squares. Thus, excluding r itself, only 7 following requests need to be checked. (Requests with smaller y coordinates have already been checked against r.)

[Timing Analysis]

Dividing the requests takes $O(n)$ time. The merging phase goes through each request in the stripe, and requires constant time for each request. So the merging phase takes $O(n)$ time. We have the following recursion for the time complexity.

$$T(n) \leq 2T(n/2) + cn$$

which solves to $O(n \lg n)$ according to master theorem.

[Other Solutions]

There are other solutions that do not use divide and conquer. Most closely related, one can directly divide L_x into maximal clusters where the points in each cluster have an x extent of at most 1. Then one can compare points between adjacent clusters using a similar algorithm to the above. Less related, if one allows computing the floor of a coordinate, we can assign points to their $\frac{1}{2} \times \frac{1}{2}$ squares, and then use a hash table to check for conflicts within the square or with the eight adjacent squares for requests. But a hash table needs $O(n)$ worse-case time for insertion, though $O(1)$ expected time. This makes the worst-case overall runtime $O(n^2)$. One can instead use a balanced BST, to reduce the worst-case runtime to $O(n \lg n)$.

(c) [8 points]

Describe how to modify your solution for Part (b) to determine whether there are three requests, all within distance 1 of each other. For full credit, your algorithm should run in $O(n \lg n)$ time, where n is the number of requests.

Solution:

The solution is similar to Part (b). For the base case, compute the distance for each 3-tuple in each subset if the number of requests within a subset is less than or equal to C (e.g., $C = 3$).

If none is found, then the remaining possibility is a triangle with two points on one side of the line and one point on the other. Proceed as in Part (b), find the stripe S, divide it into squares.

Now for each request in the reduced list, check its distance from the same constant number of following requests as above, but now looking for two such requests. For each pair, also check the distance between the two requests in the pair.

The extra work is still $O(n)$, so we get the same $O(n \lg n)$ bound.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015