CPT_S 515: Homework #1

Instructor: Zhe Dang

Sheng Guan

Problem 1

1. In Lesson 1, algorithms complexity is measured on input size instead of input values. Please indicate the input size for an algorithm that solves the following problem: Given: a number n and two primes p, q, Question: is it the case that $n = p \cdot q$?

Answer:

The algorithm takes 3 inputs n,p and q.

In order to compute whether $n = p \cdot q$? or not, we need to do one multiplication and one comparison. The time complexity for the above computaton is O(1)

For the storage, we need to store 3 numbers in binary form. The space complexity is O(log(n)+ log(p)+log(q)). The size of input is log(n)+ log(p)+log(q).

Problem 2

2. In Lesson 2, we learned linear-time selection algorithm where the input array of numbers are cut into groups of size 5. Show that, when the group size is 7, the algorithm still runs in linear time.

Answer:

The linear-time selection algorithm with the group of size 7 is shown as below:

Select i-th smallest in an array:

(1)We cut the n numbers into n/7 groups and each group has a size of 7. If the las t group has less than 7 numbers, fill it with really huge numbers to make the size of 7. This step takes O(1).

(2)Then sort each group takes O(n).

(3)After the sort ,we have (n/7) medians.

(4)Select the n/14-th smallest element from the n/7 medians, we call this n/14-th smallest element as Median of Median (MM).

(5)Swap MM with the first element in the original array of n numbers.

(6)After step (5), we update the original array. We do partition on this array.

We will get the index r with the value of MM in the middle. The left side of MM is filled with array values which are less than MM. We call it Low part here. The right side of MM is filled with array values which are greater than MM. We call it High part here. (7) if i==r, return this MM as the i-th smallest; if i<r, recursively run: select i-th smallest in an array of Low part; if i>r, recursively run: select (i-r)-th smallest in an array of High part.

We then analyze the time complexity: The 7 steps except step (4) and step(7) take O(n) time. Step (4) according to the definition is Tw(n/7). step(7) asks us to give an upper bound of the length of Low part and High part. We can derive that there are at least $4 \cdot n/14$ numbers in the original array $\leq$ MM. We will get $\| \text{ Low } \| \geq 4n/14$ and $\| \text{ High } \| \geq 4n/14$. Hence both ‖Low‖ and ‖High‖ will be bounded as 10n/14.

So we can write a formula like:

$$Tw(n) = Tw(5 * n/7) + Tw(n/7) + O(n)$$

Guess from the problem: $Tw(n) = O(n) \leq C * n$, for some C Check:

$$\begin{aligned} Tw(n) &= Tw(5 * n/7) + Tw(n/7) + a * n \\ Tw(n) &\leq C * 5 * n/7 + C * n/7 + a * n \\ &= C * 6 * n/7 + a * n \\ &\leq C * n, foralargeC \end{aligned}$$

In conclusion, when the group size is 7, the algorithm still runs in linear time.

Problem 3

3. In Lesson 2, we learned closest pair algorithm that runs in O(nlogn) time. However, that algorithm can be improved further when additional assumption is made. Here is one. Suppose that there are n^2 bugs sitting on a piece of paper of size n by n. Any two bugs must stay away by at least 1. Each bug is given as a pair of coordinates. Design a linear-time algorithm that finds the closest pair of bugs. (Hint: since the input size is n^2 , the linear time here really means the running time is O(n^2) where the n^2 is the number of bugs.)

Answer:

The algorithm sketch is as below:

1) Divide the space horizontally as O(n) stripes, each stripe is with the constant size that greater than 1, here we can set the size as 2. There are O(n) points in each stripe;

2) In each stripe, for each point, get the closest pair with the point falls in the right-hand-side adjacent stripe , write down the minimum pair m_1. Since for each point, the checking to get the closest pair is constant, this step takes O(n);

3) Continue step 2 and traverse to the n stripes and get the horizontal minimum $m_{horizontal}$, the step (2) and (3) combine together that takes O(n^2)

4) Divide the space vertically as O(n) stripes, each stripe is with the constant size that greater than 1, here we can also set the size as 2. There are O(n) points in each stripe;

5) In each stripe, for each point, get the closest pair with the point falls in the upper-level adjacent stripe, write down the mimimum pair m_2. Since for each point, the checking to get the closest pair is constant, this step takes O(n);

6) Continue step 5 and traverse to the n stripes and get the vertical minimum $m_{vertical}$, the step (4) and (5) combine together that takes O(n^2)

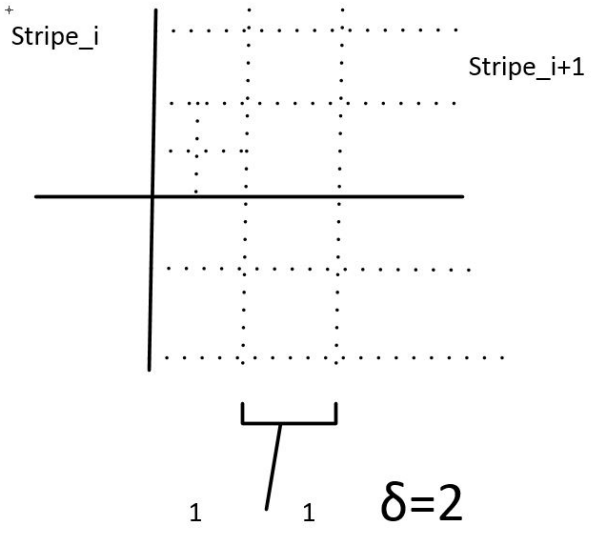
7) Because the pair representation has no direction property, we consider all the minimum distance cases already.

8) Compare and find the global miminum m between 2, $m_{horizontal}$ and $m_{vertical}$, this takes O(1).

In total this algorithm is O(n^2).

In the end we need to prove the most important two points for the correctness of this algorithm. 1)There are O(n) points in each stripe; 2)Each point in the stripe takes constant time checking to find a minimum pair that lies in the adjacent stripe.

Without losing generality, we can let $n = 2^m$, m is a positive integer. Write down the minimum pair. If we divide the whole space equally as stripes with the length 2,we will get 2^{2m-1} stripes. Then naturally we will get the figure 3-1 as below:



First we prove that after the division, in each stripe, we can only allocate O(n) bug points. By dividing the length with 2, we will get 2^{m-1} 2×2 squares, each 2×2 square we can further separate it as $16 \times \frac{1}{2} \times \frac{1}{2}$ squares. For each $\frac{1}{2} \times \frac{1}{2}$square, we know the diagonal is the maximum value with $\frac{\sqrt{2}}{2}$, that is less than 1. So there will be $2^{m-1} \times 16$ points in each stripe, that is O(n).
Then in each stripe with its adjacent stripe, like the divide-conquer solution, for each data point in the stripe, we let the δ equals 2, and find pair that is less than $\delta = 2$ however greater and equal than 1. We only need to consider the nearest 32 points, otherwise the closest pair equals to 2. So we prove the checking here is a small constant number. In the end, we can get the conclusion this adjacency search will take O(n) since we have O(n) points in the stripe.
So the correctness of the above algorithm is proved and it takes O(n^2).

Problem 4

4. Algorithms are to solve problems, or more precisely, to solve problems that have a precise mathematica definition. However, in practice, figuring out what is exactly the problem is not easy at all. (You may search internet) Suppose that you would like to write an algorithm to decide the similarity between two C programs. What would you do?
Answer:
The brute-force solution for this problem is to enumerate all inputs in the input domain and runs each input on two programs to compare the output difference when the input domain is small and feasible to finish the above solution.
However, usually the input domain is large or infinite. Here we would like to construct a computable metrics that can approximate the computation of behavioral similarity. The behavioral similarity here we concern about input/output behaviors of programs.
There is a technique called Dynamic symbolic execution (DSE)[1] that can execute a program both symbolically and concretely to collect constraints from branches and systematically negates part of the collected constraints to generate new program inputs. With the help of DSE, we can use one C program as a reference program and apply DSE on it to generate test inputs which capture the partial behaviors of this C program. Next we will run the other C program under these test inputs and compute the proportion of the agreed inputs over the generated inputs. The proportion of the agreement here serves as an indicator to represent the similarity between two C programs. Because the generated test inputs are based on only the reference program and not sensitive to the other program under analysis, we can further construct a paired program[2] from these two C programs. Given two versions of C class, the paired program will add new branches such that the behavioral differences between the two class versions are exposed and automatically generate a coverage based test generation tool. Then from the coverage based test generation tool, we will get test inputs for covering the added branches to expose behavioral differences. The paired program shares the same input domain with the two programs, and it feeds the same input to both programs and asserts the outputs of the two programs to be the same. When generating test inputs based on the paired program, DSE attempts to generate test inputs passing and failing the assertion, respectively. The inputs passing the assertion indicate that the two programs produce the same output on these inputs, while the inputs failing the assertion indicate that the two programs produce different outputs. Hence, we compute the proportion of test inputs that pass the assertion as the metric to indicate the similarity.
Another way to look at this problem, we can compute the string form and syntax similarity since they are both C programs and probably share the duplicate code. We can go as far as comparing source lines. Source line equality assumes that the cloning process introduced no changes in identifiers, comments, spacing, or other non-semantic changes, and thus limits clone detection to exact matches. When two programs are very similar but white spacing (blanks, tabs, newlines) and comments have difference, which will disable recognition based purely on strings. A simple lexical processor can overcome this particular problem. DUP[3]actually compares strings of lexemes rather than strings of characters to combat this problem. [4]uses

abstract syntax trees(AST) to detect exact and near miss clones over arbitrary program fragments. As a first step in the clone detection process, the source code is parsed and an AST is produced for it. Then the sub-tree clones will be detected. With the help of variable-size sequences of sub-tree clones, the statement and declaration sequences clones will be detected. In the end, more complex near-miss clones will be detected by attempting to generalize combinations of other clones.

Refer to:

[1]Li, Sihan, et al. "Measuring code behavioral similarity for programming and software engineering education." Proceedings of the 38th International Conference on Software Engineering Companion. ACM, 2016.
[2]Taneja, Kunal, and Tao Xie. "DiffGen: Automated regression unit-test generation." Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2008.
[3]Baker, Brenda S. "Finding clones with dup: Analysis of an experiment." IEEE Transactions on Software Engineering 33.9 (2007).
[4]Baxter, Ira D., et al. "Clone detection using abstract syntax trees." Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998.