Advanced Algorithms: Homework 1

Ivani Patel

September 22, 2022

1. In Lesson 1, algorithm's 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 ·q?

   (a) This Question has two possibilities, If an algorithm takes in a single number n and two primes also as single numbers as input, we can use the number of bits in the number. Or we can just take the input number itself as the size.

   In case of n = p.q the input size would be log2n for one number, we have n number and 2 prime numbers p and q So, total input size of algorithm will be (3*log2n)

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.

   (a) If we cut the input array of numbers into 7 groups instead of 5, after partitioning the input array with the median of median, say M, we can obtain a lower bound on the number of elements that are greater than M and a lower bound on the number of elements less than M. For elements greater than M, half of part consisting of 7 elements has at least 4 elements that are greater than M. We really rebate the part containing M and the final part which has size at most 7.

   Therefore, number of elements are grater than M at least $4.(1/2)(n/7) - 2 \geq (2n/7) - 8$

   Similarly, the number of elements that are less than M is also at least $(2n/7) - 8$. Therefore, the algorithm calls itself recursively on a problem of size at most $n - (2n/7) - 8 = (5n/7) + 8$. Thus the recurrence relation for the run-time becomes $T(n) \leq T(n/7) + T((5n/7) + 8) + O(n)$.

   Now, to find solution, $T(n) = c.n$ for constant $c > 0$. we use substitution method to solve this equation.

   Lets assume, $T(n) \leq c.n$ for some $c > 0$.

   $$\begin{aligned} T(n) &\leq T(n/7) + T((5n/7) + 8) + a.n \\ &\leq c.(n/7) + c((5n/7) + 8) + a.n \\ &\leq cn/7 + c((5n/7) + 8)a.n \end{aligned}$$

   The above RHS is c.n if and only if

   $$\begin{aligned} cn/7 + c((5n/7) + 8)a.n &\leq n \\ 8c + a.n &\leq (cn/7) \\ a.n &\leq c((n/7) - 8) \\ c &\geq a.n/((n/7) - 8) = (7an)/(n - 56) \end{aligned}$$

   So we get $n > 56$ and then a constant c exists such that $c > 7a/(1 - (56/n))$. So if we select $n > 2 * 56 = 112$, we have that $(1 - (56/n)) > 1/2$ and $c > 14a$. Because the conditions for selecting c can be satisfied , the Linear-time selection algorithm still runs in linear time.

3. Lesson 2, we learned closest pair algorithm that runs in O(n log n) time. However, that algorithm can be improved further when additional assumption is made. Here is one. Suppose that there are n2bugs 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 n2, the linear time here really means the running time is O(n2) where the n2is the number of bugs.)

   (a) Given that n^2 bugs are sitting on paper of size n by n.And Two bugs must stay away by at least 1. Need to find closest pair of bugs.So, we can calculate the smallest distance in O(nlog n) time using Divide and Conquer strategy.

   Algorithm

   Input: An array of bugs

   Output: The smallest distance between two bugs in the given array.

   Input array is sorted according to x coordinates.

   i. Find the middle point in sorted array, and make middle point M
   ii. Divide the array into two half parts. The first part contains bugs from point P[0] to P[n/2]. And second part contains bugs from point P[n/2+1] to P[n-1].
   iii. Find the smallest distance in both parts. Name the distance dl and dr. Find minimum of dl and dr and name the minimum distance d.
   iv. From these steps we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through P[n/2] and find all the bugs whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such bags.
   v. Sort array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.
   vi. Find the smallest distance in strip[]. Which is O(n).
   vii. Return the minimum of d and distance calculated in above stet.

   Code

```
#include <stdio.h>
#include <float.h>
#include <stdlib.h>
#include <math.h>
int main()
{
bugs P[] = {(9,12),(22,30),(45,10),(35,1),(2,5),(33,27)};
int n=size(P[0]);
printf("The smallest distance is %f", closest(P,n));
return 0;
}
struct bugs
{
int x,y;
};
int compareX(const void*a, const void*b)
```

```
{
bugs *p1=(bugs*)a, *p2=(bugs*)b;
return(p1->x-p2->x);
}
int compareY(const void*a, const void*b)
{
bugs *p1=(bugs*)a, *p2=(bugs*)b;
return(p1->y-p2->y);
}
float dist(bugs p1,bugs p2)
{
return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
float b_Force(Point P[], int n)
{
float min = FLT_MAX;
for (int i = 0; i < n; ++i)
for (int j = i+1; j < n; ++j)
if (dist(P[i], P[j]) < min)
min = dist(P[i], P[j]);
return min;
}
float min(float x, float y)
{
return (x < y)? x : y;
}
float Closest(Point strip[], int size, float d)
{
float min = d;  // Initialize the minimum distance as d
qsort(strip, size, size(bugs), compareY);
for (int i = 0; i < size; ++i)
for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
if (dist(strip[i],strip[j]) < min)
min = dist(strip[i], strip[j]);
return min;
}
float c_Util(Point P[], int n)
{
if (n <= 3)
return b_Force(P, n);
// Find the middle point
int mid = n/2;
Point midPoint = P[mid];
float dl = c_Util(P, mid);
float dr = c_Util(P + mid, n-mid);
// Find the smaller of two distances
float d = min(dl, dr);
```

```
Point strip[n];
int j = 0;
for (int i = 0; i < n; i++)
if (abs(P[i].x − midPoint.x) < d)
strip[j] = P[i], j++;
return min(d, Closest(strip, j, d) );
}
float closest(Point P[], int n)
{
qsort(P, n, size(Point), compareX);
return c_Util(P, n);
}
```

Time Complexity of this algorithm is T(n). Assume we use O(nLogn) sorting algorithm. This algorithm divides all bugs in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finds the closest bugs in strip in O(n) time. So T(n) can be expressed as:

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

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

$T(n) = T(n.Logn.Logn)$

4. Algorithms are to solve problems, or more precisely, to solve problems that have a precise mathematics 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?

An algorithm to decide the similarity between two C Programs

Ivani Patel

September 22, 2022

1 Introduction

To plan an assessment to track down similarity between two C programs, without any hesitation we genuinely need to figure out that: Resemblance can be in various ways, Program's feedback and result or thinking of program or code written in same style. Taking into account as far as possible, assessment changes. In this we depict a technique for looking at two C programs. In here we'll see various cases to find similarities and will see abject strategy for that.

In the event that source code is given, it becomes easy to contrast them and with track down closeness between them. On the off chance that source code isn't given that makes two cases in initial one we think about info and result. Furthermore, in the second one we think about the succession of information and result.

2 Techniques to compare two C programs

2.1 Code is given

In the event that we have code for the program, we can say program P1 and program P2 are white box.We can contrast the program source code straightforwardly with track down the closeness between them.There can be different approaches to differentiating the source code of program: Sentence structure closeness or Reasoning of code is same. We can find these comparable qualities by outline or parse tree or a few different strategies.

2.1.1 Syntax Similarity

A source code has two kinds of information: lexical and essential information. Lexical information connects with factors and the held words like public, if, and for. This language is made from a colossal plan of rarely happening user-defined word and a little plan of much of frequently occurring words. Of course, essential information connects with a plan still not yet decided by reserved words. Among them, essential information is a more huge snippet of data for identifying closeness, since tokens can be successfully different over into various tokens without understanding the subject of a source code.Note that a source program has two sorts of structural information. One is syntactic design that is typically communicated as a parse tree and the other is capability call graph structure.So, We foster an examination calculation that takes advantage of information on the syntax. Since the unit of examination is individual tokens, not an entire line as utilized in text-based comparators, this calculation can bring up the linguistic structure contrasts more precisely than other comparison tools.

The program source code can be normally addressed as a parse tree of which every node indicates variables, reserved words, operators, etc. Involving devices for language acknowledgment and language

syntax we can undoubtedly make an interpretation of source code into parse tree.Since parse tree has syntactic primary data, a measurement for parse tree that reflects whole underlying data is required. The parse tree bit is one of such measurements. It compares parse trees without manually designed structural features. Furthermore, by looking at this we can find comparability between two source code form parse tree strategy.

2.1.2 Semantic Similarity

Presently utilizing graph based structures to catch semantics of Program.In any case, similar to trees, graph comparability experiences a high computational complexity.

As of late, program source codes are composed with object-situated ideas and a few refactoring strategies, so the codes are getting increasingly more modularized at useful level. Since a source code encodes program rationale to tackle an issue, the execution stream at capability level is one of the significant variables to recognize the source code. Thusly, this capability level stream ought to be considered to analyze source codes.

Function call graph of a program depicts how the program executes at function level and how functions are associated with one another. Since the flow of a program is extremely novel according to the endeavor, the likeness between two sources can be resolved using the movements of the programs.Since this flow is tended to as a function call chart, the diagram is the best strategy to see capacity call graphs.

2.2 Code is not given

Code similarity estimation might not exclusively be estimated on source at any point code yet in addition on compiled code. Estimating likeness of compiled code is valuable when the source code is missing or inaccessible. Besides, it can likewise catch dynamic ways of behaving of the projects by executing the compiled code.

Other than the text, token, tree, and chart based approaches, there are a few other elective strategies embraced from different fields of exploration to code comparability estimation like information theory, data recovery, or data mining.These procedures show positive outcomes and open further conceivable outcomes in this examination region.

At the point when code isn't given we can say Program P1 and P2 are secret elements. So we analyze two C programs without code. That can be as far as estimating space involved by memory for program, Contrasting grouping of info and result, looking at working of program, etc.

On the off chance that we take Program P1 and input is i and result is o then i-o grouping is endless, we can track down this arrangements for other program P2. We can find the arrangement for the two projects in which information and result are same. Assuming we give i as an info and we get o as result then we can make succession for that which gives same outcome.

The issue of matching two sequence is a characteristic speculation of that of matching successions of tokens. There exist a few calculations for succession coordinating. Some depend on unique programming plans to track down longest normal aftereffects. Others endeavor to view as a good matching under various models.

Projects can be likewise contrasted with time with run and give yield. The correlation time relies upon how much contrasts between the documents and the spots where the distinctions happen.

Given a program P, and the transformed program P, the definition of obfuscation transformations T is P$\rightarrow$TP1 requiring P and P1 to hold the same observational behaviour. Specifically, legal obfuscation transformation requires: 1) if P fails to terminate or terminates with errors then P may or may not terminate, and 2) P1 must terminate if P terminates.

Above given is model for tracking down Program comparability. In the event that program one rolls out any improvement, another program additionally changes and it influences on its result.So the succession of info and result becomes same. what's more, we get to realize that program is running similarly.

2.3 Conclusion

In this, we have proposed a unique procedure for program relationship. The proposed system registers the comparability between two source codes with the association of two kinds of hidden information isolated from the source codes, and Closeness without given code. That is, the system uses both syntactic information and dynamic information. The syntactic information which gives close by level essential view is associated with the parse tree. The strong information, which is contained in the capacity call graph, gives high and overall level hidden viewpoint. Also, thereafter we dissected program without code.

Along these lines, this shows strategies for tracking down likeness between two C projects with its data.