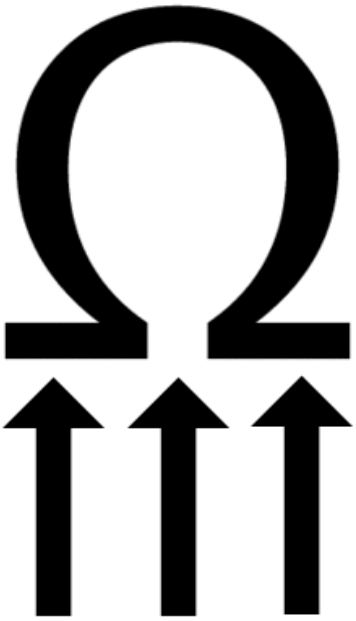
COMPETITIVE PROGRAMMING

Increasing the Lower Bound of Programming Contests



STEVEN HALIM

(PhD*, Instructor)

FELIX HALIM

(PhD candidate)

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

A HANDBOOK FOR ACM ICPC AND IOI CONTESTANTS

2010

Contents

Acknowledgements

.

Steven wants to thank:

- God, Jesus Christ, Holy Spirit, for giving talent and passion in this competitive programming.
- My lovely wife, Grace Suryani, for allowing me to spend our precious time for this project.
- My younger brother and co-author, Felix Halim, for sharing many data structures, algorithms, and programming tricks to improve the writing of this book.
- My father Lin Tjie Fong and mother Tan Hoey Lan for raising us and encouraging us to do well in our study and work.
- School of Computing, National University of Singapore, for employing me and allowing me to teach CS3233 - 'Competitive Programming' module from which this book is born.
- NUS/ex-NUS professors/lecturers who have shaped my competitive programming and coaching skills: Prof Andrew Lim Leong Chye, Dr Tan Sun Teck, Aaron Tan Tuck Choy, Dr Sung Wing Kin, Ken, Dr Alan Cheng Holun, etc.
- Fellow Teaching Assistants of CS3233 and ACM ICPC Trainers @ NUS: Su Zhan, Melvin Zhang Zhiyong, Bramandia Ramadhana, Ngo Minh Duc, etc.
- My CS3233 students in Sem2 AY2008/2009 who inspired me to come up with the lecture notes and CS3233 students in Sem2 AY2009/2010 who help me verify the content of this book plus the Live Archive contribution.

Preface

This is a book that every competitive programmer must read – and master, at least during the middle phase of their programming career: when they want to leap forward from 'just knowing some programming language commands' and 'some algorithms' to become a top programmer.

Typical readers of this book will be: 1). Thousands University students competing in annual ACM International Collegiate Programming Contest (ICPC) [27] regional contests, 2). Hundreds Secondary or High School Students competing in annual International Olympiad in Informatics (IOI) [12], 3). Their coaches who are looking for a comprehensive training materials [9], and 4). Basically anyone who loves problem solving using computer.

Beware that this book is *not* for a novice programmer. When we wrote the book, we set it for readers who have knowledge in basic programming methodology, familiar with at least one programming language (C/C++/Java), and have passed basic data structures and algorithms (or equivalent) typically taught in year one of Computer Science University curriculum.

Due to the diversity of its content, this book is *not* meant to be read once, but several times. There are many exercises and programming problems scattered throughout the body text of this book which can be skipped first if solution is not known at that point of time, but can be revisited in latter time after the reader has accumulated new knowledge to solve it. Solving these exercises help strengthening the concepts taught in this book as they usually contain interesting twists or variants of the topic being discussed, so make sure to attempt them.

Use `http://felix-halim.net/uva/hunting.php`, `www.uvatoolkit.com/problemssolve.php`, and `www.comp.nus.edu.sg/~stevenha/programming/acmoj.html` to help you to deal with UVa [17] problems listed in this book.

We know that one probably cannot win an ACM ICPC regional or get a gold medal in IOI just by mastering the *current version* of this book. While we have included a lot of materials in this book, we are well aware that much more than what this book can offer, are required to achieve that feat. Some pointers are listed throughout this book for those who are hungry for more.

We believe this book is and will be relevant to many University and high school students as ICPC and IOI will be around for many years ahead. New students will require the 'basic' knowledge presented in this book before hunting for more challenges after mastering this book. But before you assume anything, please check this book's contents to see what we mean by 'basic'.

We will be happy if in year 2010 and beyond, the level of competitions in ICPC and IOI increase because many of the contestants have mastered the content of this book. We hope to see many ICPC and IOI coaches around the world, especially in South East Asia, adopt this book knowing that without mastering the topics *in and beyond* this book, their students have no chance of doing well in future ICPCs and IOIs. If such increase in 'required lowerbound knowledge' happens, this book has fulfilled its objective of advancing the level of human knowledge in this era.

To a better future of humankind,
Steven and Felix Halim

PS: To obtain example source codes and PowerPoint slides/other instructional materials (only for coaches), send a personal request email to stevenhalim@gmail.com

Authors' Profiles

Steven Halim, PhD[1], stevenhalim@gmail.com

Steven Halim is currently an instructor in School of Computing, National University of Singapore (SoC, NUS). He teaches several programming courses in NUS, ranging from basic programming methodology, intermediate data structures and algorithms, and up to the 'Competitive Programming' module that uses this book. He is the coach of both NUS ACM ICPC teams and Singapore IOI team. He participated in several ACM ICPC Regional as student (Singapore 2001, Aizu 2003, Shanghai 2004). So far, he and other trainers @ NUS have successfully groomed two-times ACM ICPC world finalist team as well as two silver and two bronze IOI medallists.

Felix Halim, PhD Candidate, felix.halim@gmail.com

Felix Halim is currently a PhD student in the same University: SoC, NUS. In terms of programming contests, Felix has much colorful reputation than his older brother. He was IOI 2002 contestant. His teams (at that time, Bina Nusantara University) took part in ACM ICPC Manila Regional 2003-2004-2005 and obtained rank 10, 6, and 10th respectively. Then, in his final year, his team finally won ACM ICPC Kaohsiung Regional 2006 and thus became ACM ICPC World Finalist @ Tokyo 2007 (Honorable Mention). Today, felix_halim actively joins TopCoder Single Round Matches and his highest rating is a yellow coder.

[1]To be precise, Steven is not yet a PhD at this point of time: April 28, 2010. He is currently waiting for his final PhD thesis defense which is scheduled sometime in May 2010.

Copyright

Chapter 1

Introduction

I want to compete in ACM ICPC World Final!
— **A dedicated student**

In this chapter, we introduce readers to the world of competitive programming. Hopefully you enjoy the ride and continue reading and learning until the very last page of this book, enthusiastically.

1.1 Competitive Programming

'Competitive Programming' in summary, is this: "Given well-known Computer Science (CS) problems, solve them as quickly as possible!".

Let's digest the terms one by one. The term 'well-known CS problems' implies that in competitive programming, we are dealing with solved CS problems and *not* research problems (where the solutions are still unknown). Definitely, some people (at least the problem setter) have solved these problems before. 'Solve them' implies that we must push our CS knowledge to a certain required level so that we can produce working codes that can solve these problems too – in terms of getting the *same* output as the problem setter using the problem setter's secret input data. 'As quickly as possible' is the competitive element which is a very natural human behavior.

Please note that being well versed in competitive programming is *not* the end goal, it is just the means. The true end goal is to produce an all rounded computer scientists/programmers who are much more ready to produce better software or to face the harder CS research problems in the future. The founders of ACM International Collegiate Programming Contest (ICPC) [27] have this vision and we, the authors, agree with it. With this book, we play our little roles in preparing current and future generations to be more competitive in dealing with well-known CS problems frequently posed in recent ICPCs and International Olympiad in Informatics (IOI).

Illustration on solving UVa Problem 10911 (Forming Quiz Teams).

Abridged problem description: Let (x,y) be the coordinate of a student's house on a 2-D plane. There are $2N$ students and we want to **pair** them into N groups. Let d_i be the distance between the houses of 2 students in group i. Form N groups such that $\sum_{i=1}^{N} d_i$ is **minimized**. Constraints: $N \leq 8; 0 \leq x, y \leq 1000$. Think first, try not to flip this page immediately!

Now, ask yourself, which one is you? Note that if you are unclear with the materials or terminologies shown in this chapter, you can re-read it after going through this book once.

- Non-competitive programmer A (a.k.a the blurry one):
  Step 1: Read the problem... confused @-@, never see this kind of problem before.
  Step 2: Try to code something... starting from reading non-trivial input and output.
  Step 3: Realize that all his attempts fail:
  Greedy solution: pair students based on shortest distances gives **Wrong Answer (WA)**.
  Complete search using backtracking gives **Time Limit Exceeded (TLE)**, etc.
  After 5 hours of labor (typical contest time), no **Accepted (AC)** solution is produced.
- Non-competitive programmer B (Give up):
  Step 1: Read the problem...
  Then realize that he has seen this kind of problem before.
  But also remember that he has not learned how to solve this kind of problem...
  He is not aware of a simple solution for this problem: **Dynamic Programming (DP)**...
  Step 2: Skip the problem and read another problem.
- (Still) non-competitive programmer C (Slow):
  Step 1: Read the problem and realize that it is a **'matching on general graph'** problem.
  In general, this problem **must** be solved using **'Edmond's Blossom Shrinking'**.
  But since the input size is small, this problem is solve-able using Dynamic Programming!
  Step 2: Code I/O routine, write recursive top-down DP, test the solution, **debug** >.<...
  Step 3: *Only after 3 hours*, his solution is judged as AC (passed all secret test data).
- Competitive programmer D:
  Same as programmer C, but do all those steps above in less than 30 minutes.
- Very Competitive programmer E:
  Of course, a very competitive programmer (e.g. the red 'target' coders in TopCoder [26]) may solve this 'classical' problem in less than 15 minutes...

1.2 Tips to be Competitive

If you strive to be like competitive programmer D or E in the illustration above: You want to do well to qualify and get a medal in IOI [12]; to qualify in ACM ICPC [27] national, regional, and up to world final; or in many other programming contests, then this book is definitely for you!

In subsequent chapters, you will learn basic to medium data structures and algorithms frequently appearing in recent programming contests, compiled from many sources [19, 6, 20, 2, 4, 14, 21, 16, 23, 1, 13, 5, 22, 15, 46, 24] (see Figure 1.5). But you will not just learn the algorithm, but also how to implement them efficiently and apply them to appropriate contest problem.

Not only that. You will also learn many tiny bits of programming tips from our experience that can be helpful in contest situation. We will start by giving you few general tips below:

Tip 0: Type Code Faster!

No kidding! Although this tip may not mean much as ICPC nor IOI are *not* about typing speed competition, but we have seen recent ICPCs where rank i and rank $i+1$ are just separated by few minutes. When you can solve the same number of problems as your competitor, it is now down to coding skill and ... typing speed.

1.4 Chapter Notes

Figure 1.5: Some Reference Books that Inspired the Authors to Write This Book

This and subsequent chapters are supported by many text books (see Figure 1.5) and Internet resources. Tip 1 is an adaptation from introduction text in USACO training gateway [18]. More details about Tip 2 can be found in many CS books, e.g. [4] chapter 1-5, 17. Reference for Tip 3 are www.cppreference.com, www.sgi.com/tech/stl/ for C++ STL and java.sun.com/javase/6/docs/api for Java API. You can also take a look at Appendix A for some of our examples. For more insights to do better testing (Tip 4), a little detour to software engineering books may be worth trying. There are many other online judges than those mentioned in Tip 5, e.g. SPOJ www.spoj.pl, POJ acm.pku.edu.cn/JudgeOnline, TOJ acm.tju.edu.cn/toj, ZOJ acm.zju.edu.cn/onlinejudge/, etc to name the few.

There are approximately **33 programming exercises** discussed in this chapter.

```
#define REP(i, a, b) \ // all codes involving REP uses this macro
  for (int i = int(a); i <= int(b); i++)
vector<int> pset(1000); // 1000 is just a rough number, adjustable by user
void initSet(int _size) { pset.resize(_size); REP (i, 0, _size - 1) pset[i] = i; }
```

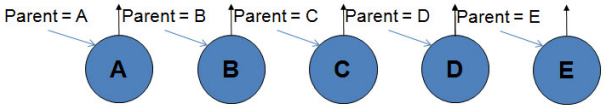


Figure 2.3: Calling initSet() to Create 5 Disjoint Sets

When we want to merge two sets, we call 'unionSet(i, j)' which makes both item 'i' and 'j' to have the same representative item[6] – directly or indirectly (see Path Compression below). This is done by calling 'findSet(j)' – what is the representative of item 'j', and assign that value to 'pset[findSet(i)]' – update the parent of the representative item of item 'i'.

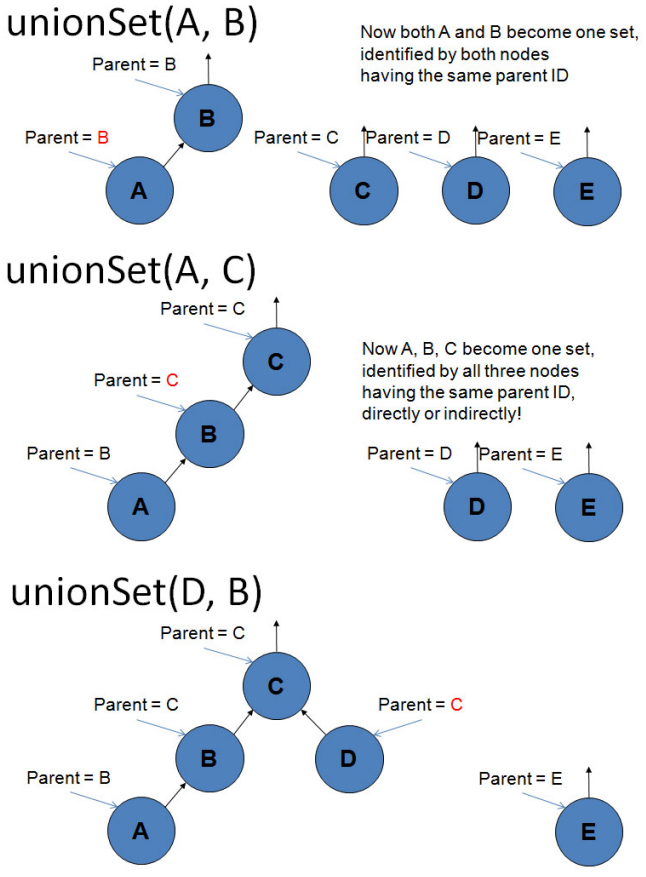


Figure 2.4: Calling unionSet(i, j) to Union Disjoint Sets

In Figure 2.4, we see what is happening when we call unionSet(i, j): every union is simply done by changing the representative item of one item to point to the other's representative item.

```
int findSet(int i) { return (pset[i] == i) ? i : (pset[i] = findSet(pset[i])); }
void unionSet(int i, int j) { pset[findSet(i)] = findSet(j); }
```

[6]There is another heuristic called 'union-by-rank' [4] that can further improve the performance of this data structure. But we omit this enhancing heuristic from this book to simplify this discussion.

In this special box, we want to highlight another problem solving trick called: *Decomposition*!

While there are only 'few' basic algorithms used in contest problems (most of them are covered in this book), the harder problems may require a *combination* of two (or more) algorithms for their solution. For such problem, try to decompose parts of the problem so that you can solve different parts of the problem independently. We illustrate this decomposition technique using a recent top-level programming problems that combines *three* problem solving paradigms that we have just recently learned: Complete Search, Divide & Conquer, and Greedy!

ACM ICPC World Final 2009 - Problem A - A Careful Approach

You are given a scenario of airplane landings. There are $2 \leq n \leq 8$ airplanes in the scenario. Each airplane has a time window during which it can safely land. This time window is specified by two integers a_i, b_i, which give the beginning and end of the closed interval $[a_i, b_i]$ during which the i-th plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \leq a_i \leq b_i \leq 1440$. Then, your task is to:

1. Compute an **order for landing all airplanes** that respects these time windows.
HINT: order = permutation = Complete Search?
2. Furthermore, the airplane landings should be stretched out **as much as possible** so that the minimum achievable time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible!
HINT: Is this similar to 'greedy activity selection' problem [4]?
3. Print the answer split into minutes and seconds, rounded to the closest second.
See Figure 3.7 for illustration: line = time window of a plane; star = its landing schedule.

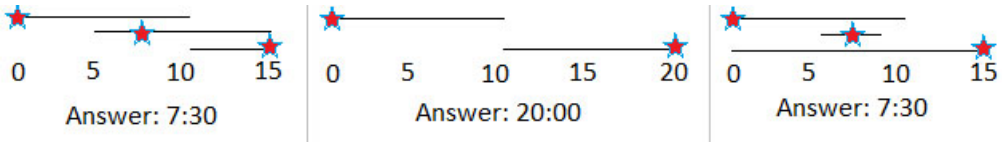


Figure 3.7: Illustration for ACM ICPC WF2009 - A - A Careful Approach

Solution:

Since the number of planes is at most 8, an optimal solution can be found by simply trying all 8! = 40320 possible orders for the planes to land. This is the **Complete Search** portion of the problem which can be easily solved using C++ STL next_permutation.

Now, for each specific landing order, we want to know the largest possible landing window. Suppose we use a certain window length L. We can greedily check whether this L is feasible by forcing the first plane to land as soon as possible and the subsequent planes to land in max(a[that plane], previous landing time + L). This is a **Greedy Algorithm**.

A window length L that is too long/short will overshoot/undershoot b[last plane], so we have to decrease/increase L. We can binary search this L – bisection method – **Divide & Conquer**. As we only want the answer rounded to nearest integer, stopping bisection method when error $\epsilon \leq$ 1e-3 is enough. For more details, please study our AC source code shown in the next page.

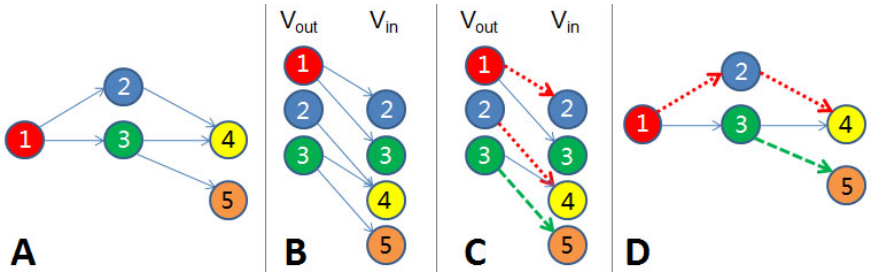


Figure 4.26: Min Path Cover in DAG (from LA 3126 [11])

In general, the min path cover problem in DAG is described as the problem of finding the minimum number of paths to cover *each vertex* in DAG $G = (V, E)$.

This problem has a polynomial solution: Construct a bipartite graph $G' = (V_{out} \bigcup V_{in}, E')$ from G, where $V_{out} = \{v \in V : v \text{ has positive out-degree}\}$, $V_{in} = \{v \in V : v \text{ has positive in-degree}\}$, and $E' = \{(u, v) \in (Vout, Vin) : (u, v) \in E\}$. This G' is a bipartite graph! Finding a matching in bipartite graph G' forces us to select at most one outgoing edge from $v \in V_{out}$ (similarly for V_{in}). DAG G initially has n vertices, which can be covered with n paths of length 0 (the vertex itself). One matching between (a, b) says that we can use one less path as it can cover both vertices in $a \in V_{out}$ and $b \in V_{in}$. Thus if the max cardinality bipartite matching (MCBM) in G' has size m, then we just need $n - m$ paths to cover each vertex in G.

The MCBM in G' that is needed to solve this min path cover in G is discussed below. The solution for bipartite matching is polynomial, thus the solution for min path cover in DAG is also polynomial. Note that path cover in general graph is NP-Complete [41].

Programming Exercises related to DAG:

- Single-Source Shortest Paths in DAG
  1. UVa 10166 - Travel
  2. UVa 10350 - Liftless Eme
- Single-Source Longest Paths in DAG
  1. UVa 103 - Stacking Boxes
  2. UVa 10000 - Longest Paths
  3. UVa 10029 - Edit Step Ladders
  4. UVa 11324 - The Largest Clique (find SCC first then longest path on DAG)
  5. LA 3294 - The Ultimate Bamboo Eater (with 2-D Segment Tree)
  6. Ural 1450 - Russian pipelines
  7. PKU 3160 - Father Christmas flymouse
- Min Path Cover in DAG
  1. LA 2696 - Air Raid
  2. LA 3126 - Taxi Cab Scheme

4.9.3 Bipartite Graph

Bipartite Graph, is a special graph with the following characteristics: the set of vertices V can be partitioned into two disjoint sets V_1 and V_2 and all edges in $(u, v) \in E$ has the property that $u \in V_1$ and $v \in V_2$. The most common application is the (bipartite) matching problem, shown below.

The way BigInteger library works is to simply store the big integer as a (long) string. For example we can store 10^{21} inside a string `num1` = "1,000,000,000,000,000,000,000" without much problem whereas this is already overflow in 64-bit unsigned long long in C/C++. Then, for common mathematical operations, BigInteger library uses a kind of digit-by-digit operations to process the two big integer operands. For example with `num2` = "17", we have `num1 * num2` as:

```
num1         =  1,000,000,000,000,000,000,000
num2         =                              17
               ------------------------------ *
                7,000,000,000,000,000,000,000
               10,000,000,000,000,000,000,00
               ------------------------------ +
num1 * num2 = 17,000,000,000,000,000,000,000
```

Addition and subtraction are two simple operations in BigInteger. Multiplication takes a bit more programming job. Efficient division and raising number to a certain power are more complicated. Anyway, coding these library routines under stressful contest environment can be a buggy affair. Fortunately, Java has BigInteger class that we can use for this purpose (as of 2010, C++ STL currently does not have such library thus it is a good idea to use Java for BigInteger problems).

Java BigInteger (BI) class supports the following basic integer operations: addition – `add(BI)`, subtraction – `subtract(BI)`, multiplication – `multiply(BI)`, division – `divide(BI)`, remainder – `remainder(BI)`, combination of division and remainder – `divideAndRemainder(BI)`, modulo – `mod(BI)` (slightly different with `remainder(BI)`), and power – `pow(int exponent)`. For example, the following simple Java code is the solution for UVa 10925 - Krakovia which simply requires BigInteger addition (sum N large bills) and division (divide the large sum to F friends):

```
import java.io.*;
import java.util.*; // Scanner class is inside this package
import java.math.*; // BigInteger class is inside this package

class Main { /* UVa 10925 - Krakovia */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt(); // N bills, F friends
      if (N == 0 && F == 0) break;
      BigInteger sum = BigInteger.valueOf(0); // use valueOf to initialize
      for (int i = 0; i < N; i++) { // sum the N large bills
        BigInteger V = sc.nextBigInteger(); // for reading next BigInteger!
        sum = sum.add(V); // this is BigInteger addition
      }
      System.out.println("Bill #" + (caseNo++) + " costs " +
        sum + ": each friend should pay " + sum.divide(BigInteger.valueOf(F)));
      System.out.println(); // the line above is BigInteger division
} } }                       // divide the large sum to F friends
```

6.4.2 Applications of Suffix Tree

Assuming that Suffix Tree[4] for a string `S` is already built, we can use it for these applications:

Exact String Matching in $O(|Q| + occ)$

With Suffix Tree, we can find all (exact) occurrences of a query string Q in S in $O(|Q|+occ)$ where $|Q|$ is the length of the query string Q itself and occ is the total number of occurrences of Q in S – no matter how long the string S is. When the Suffix Tree is already built, this approach is *faster* than many exact string matching algorithms (e.g. KMP).

With Suffix Tree, our task is to simply search for the vertex x in the Suffix Tree which represents the query string Q. This can be done with just a root to leaf traversal by following the edge label. Vertex with path-label == Q is the desired vertex x. Then, all the leaves in the subtree rooted at x are the occurrences of Q in S. We can then read the starting indices of such substrings that are stored in the leaves of Suffix Tree.

For example, in the Suffix Tree of `S = 'acacag$'` shown in Figure 6.2, right and `Q = 'aca'`, we can simply traverse from root, go along edge label 'a', then edge label 'ca' to find vertex x with path-label 'aca' (follow the dashed red arrow in Figure 6.2, right). The leaves of this vertex x point to index 1 (substring: `'acacag$'`) and index 3 (substring: `'acag$'`).

Exercise: Now try to find `Q = 'ca'` and `Q = 'cat'`!

Finding Longest Repeated Substring in $O(n)$

With Suffix Tree, we can also find the longest repeated substring in S easily. The deepest internal vertex in the Suffix Tree of S is the answer. This can be done with just $O(n)$ tree traversal.

For example, in the Suffix Tree of `S = 'acacag$'` shown in Figure 6.2, right, the longest repeated substring is '*aca*' as it is the path-label of the deepest internal vertex. Recall that internal vertices represent more than 1 suffices.

Exercise: Find the longest repeated substring in `S = 'cgacattacatta$'`!

Finding Longest Common Substring in $O(n)$

The problem of finding Longest Common **Substring** (not Subsequence)[5] of two **or more** strings can be solved in linear time with Suffix Tree. Consider two strings $S1$ and $S2$, we can build a **generalized Suffix Tree** for $S1$ and $S2$ with two different ending markers, e.g. $S1$ with character '#' and $S2$ with character '$'. Then, we mark each internal vertices with has leaves that represent suffixes of *both* $S1$ and $S2$. We simply report the deepest marked vertex as the answer.

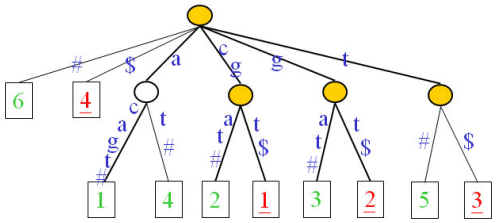


Figure 6.3: Generalized Suffix Tree of S1 = 'acgat#' and S2 = 'cgt$' (Figure from [24])

[4]As Suffix **Tree** is more compact than Suffix **Trie**, we will concentrate on Suffix **Tree**.
[5]In a string 'abcdef', 'bcd' is substring (contiguous) and 'bce' is subsequence (may skip few characters)

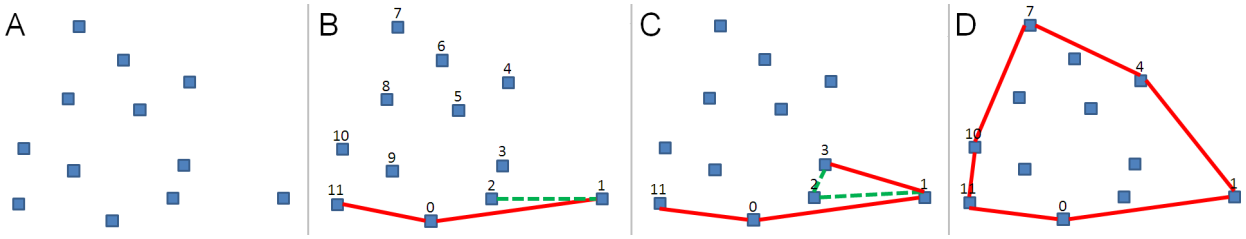


Figure 7.5: Convex Hull $CH(P)$ of Set of Points P

There are several Convex Hull algorithms available. In this section, we choose the simple $O(n \log n)$ Ronald *Graham's Scan* algorithm. This algorithm first sorts all $n-$ points P (Figure 7.5.A) based on its angle w.r.t a point called pivot (the bottommost and rightmost point in P, see point 0 and the counter-clockwise order of the remaining points in Figure 7.5.B). Then, this algorithm maintains a stack S of candidate points. Each point of P is pushed *once* to S and points that are not going to be part of $CH(P)$ will be eventually popped from S (see Figure 7.5.C, the stack previously contains (bottom) 11-0-1-2 (top), but when we try to insert 3, 1-2-3 is a right turn, so we pop 2. Now 0-1-3 is a left turn, so we insert 3 to the stack, which now contains (bottom) 11-0-1-3 (top)). When Graham's Scan terminates, whatever left in S are the points of $CH(P)$ (see Figure 7.5.D, the stack contains (bottom) 11-0-1-4-7-10-11 (top)). The ready implementation of Graham's Scan, omitting parts that have shown earlier like `ccw` function, is shown below:

```
point pivot; // global variable
vector<point> polygon, CH;

int dist2(point a, point b) { // function to compute distance between 2 points
  int dx = a.x - b.x;
  int dy = a.y - b.y;
  return dx * dx + dy * dy;
}

bool angle_cmp(point a, point b) { // important angle-sorting function
  if (area2(pivot, a, b) == 0) // collinear
  return dist2(pivot, a) < dist2(pivot, b); // which one closer

  int d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  int d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  double d = atan2((double)d1y, (double)d1x) - atan2((double)d2y, (double)d2x);
  return (d < 0);
}
```

Bibliography

[1] Ahmed Shamsul Arefin. *Art of Programming Contest (from Steven's Website)*. Gyankosh Prokashoni (Available Online), 2006.

[2] Jon Bentley. *Programming Pearls*. Addison Wesley, 2nd edition, 2000.

[3] Frank Carrano. *Data Abstraction & Problem Solving with C++*. Pearson, 5th edition, 2007.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 2nd edition, 2001.

[5] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw Hill, 2008.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[7] Project Euler. Project Euler.
http://projecteuler.net/.

[8] Michal Forišek. IOI Syllabus.
http://people.ksp.sk/ misof/ioi-syllabus/ioi-syllabus-2009.pdf.

[9] Steven Halim and Felix Halim. Competitive Programming in National University of Singapore. Ediciones Sello Editorial S.L. (Presented at Collaborative Learning Initiative Symposium CLIS @ ACM ICPC World Final 2010, Harbin, China, 2010.

[10] Steven Halim, Roland Hock Chuan Yap, and Felix Halim. Engineering Stochastic Local Search for the Low Autocorrelation Binary Sequence Problem. In *Principles and Practice of Constraint Programming*, pages 640–645, 2008.

[11] Competitive Learning Institute. ACM ICPC Live Archive.
http://acm.uva.es/archive/nuevoportal.

[12] IOI. International Olympiad in Informatics.
http://ioinformatics.org.

[13] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[14] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 1st edition, 2002.

[15] Rujia Liu. *Algorithm Contests for Beginners (In Chinese)*. Tsinghua University Press, 2009.

[16] Rujia Liu and Liang Huang. *The Art of Algorithms and Programming Contests (In Chinese).* Tsinghua University Press, 2003.

[17] University of Valladolid. Online Judge.
http://uva.onlinejudge.org.

[18] USA Computing Olympiad. USACO Training Program Gateway.
http://train.usaco.org/usacogate.

[19] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 2nd edition, 1998.

[20] Kenneth H. Rosen. *Elementary Number Theory and its applications.* Addison Wesley Longman, 4th edition, 2000.

[21] Robert Sedgewick. *Algorithms in C++, Part 1-5.* Addison Wesley, 3rd edition, 2002.

[22] Steven S Skiena. *The Algorithm Design Manual.* Springer, 2008.

[23] Steven S. Skiena and Miguel A. Revilla. *Programming Challenges.* Springer, 2003.

[24] Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction.* CRC Press (Taylor & Francis Group), 1st edition, 2010.

[25] TopCoder. Algorithm Tutorials.
http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static.

[26] TopCoder. Single Round Match (SRM).
http://www.topcoder.com/tc.

[27] Baylor University. ACM International Collegiate Programming Contest.
http://icpc.baylor.edu/icpc.

[28] Adrian Vladu and Cosmin Negruşeri. Suffix arrays - a programming contest approach. 2008.

[29] Wikipedia. Catalan number.
http://en.wikipedia.org/wiki/Catalan_number.

[30] Wikipedia. Combinatorics.
http://en.wikipedia.org/wiki/Combinatorics.

[31] Wikipedia. Cycle-Finding (Detection).
http://en.wikipedia.org/wiki/Cycle_detection.

[32] Wikipedia. Disjoint-set data structure.
http://en.wikipedia.org/wiki/Disjoint-set_data_structure.

[33] Wikipedia. Edmond's matching algorithm.
http://en.wikipedia.org/wiki/Edmonds's_matching_algorithm.

[34] Wikipedia. Eight queens puzzle.
http://en.wikipedia.org/wiki/Eight_queens_puzzle.

[35] Wikipedia. Euler's totient function.
http://en.wikipedia.org/wiki/Euler's_totient_function.

[36] Wikipedia. Gaussian Elimination for Solving System of Linear Equations.
http://en.wikipedia.org/wiki/Gaussian_elimination.

[37] Wikipedia. Great-Circle Distance.
http://en.wikipedia.org/wiki/http://en.wikipedia.org/wiki/Great_circle_distance.

[38] Wikipedia. Longest Path Problem.
http://en.wikipedia.org/wiki/Longest_path_problem.

[39] Wikipedia. Lowest Common Ancestor.
http://en.wikipedia.org/wiki/Lowest_common_ancestor.

[40] Wikipedia. Miller-Rabin Primality Test.
http://en.wikipedia.org/wiki/Miller-Rabin_primality_test.

[41] Wikipedia. Path Cover.
http://en.wikipedia.org/wiki/Path_cover.

[42] Wikipedia. Pick's Theorem.
http://en.wikipedia.org/wiki/Pick's_theorem.

[43] Wikipedia. Sieve of Eratosthenes.

[44] Wikipedia. Tarjan's Strongly Connected Components Algorithm.
http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm.

[45] Wikipedia. The Free Encyclopedia.
http://en.wikipedia.org.

[46] Yonghui Wu. *Programming Book (In Chinese)*. Posts and Telecom Press, 2009.

Index