Algorithms For Interviews

A Problem Solving Approach

Adnan Aziz
Amit Prakash

algorithmsforinterviews.com

Prologue

Let's begin with the picture on the front cover. You may have observed that the portrait of Alan Turing is constructed from a number of pictures ("tiles") of great computer scientists and mathematicians.

Suppose you were asked in an interview to design a program that takes an image and a collection of $s \times s$-sized tiles and produce a mosaic from the tiles that resembles the image. A good way to begin may be to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Here distance in color space can be $L2$-norm over Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform

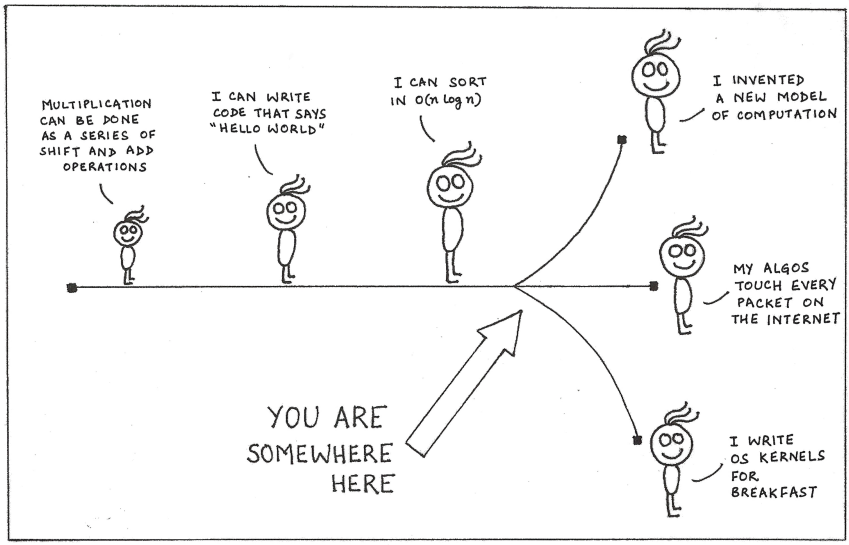

Figure 1. Evolution of a computer scientist

a coarse pixelization (2×2 or 3×3) of each image square and finding the tile that is "closest" to the image square under a distance function defined over all pixel colors (for example, $L2$-norm over RGB values for each pixel). Depending on how you represent the tiles, you end up with the problem of finding the closest point from a set of points in a k-dimensional space.

If there are m tiles and the image is partitioned into n squares, then a brute-force approach would have $O(m \cdot n)$ time complexity. You could improve on this by first indexing the tiles using an appropriate search tree. A more detailed discussion on this approach is presented in Problem 8.1 and its solution.

If in a 45-60 minute interview, you can work through the above ideas, write some pseudocode for your algorithm, and analyze its complexity, you would have had a fairly successful interview. In particular, you would have demonstrated to your interviewer that you possess several key skills:

- The ability to rigorously formulate and abstract real-world problems.
- The skills to solve problems and design algorithms.
- The tools to go from an algorithm to a working program.
- The analytical techniques required to determine the computational complexity of your solution.

Book Overview

Algorithms for Interviews (AFI) aims to help engineers interviewing for software development positions. The primary focus of AFI is algorithm design. The entire book is presented through problems interspersed with discussions. The problems cover key concepts and are well-motivated, challenging, and fun to solve.

We do not emphasize platforms and programming languages since they differ across jobs, and can be acquired fairly easily. Interviews at most large software companies focus more on algorithms, problem solving, and design skills than on specific domain knowledge. Also, platforms and programming languages can change quickly as requirements change but the qualities mentioned above will always be fundamental to any successful software endeavor.

The questions we present should all be solvable within a one hour interview and in many cases, take substantially less time. A question may take more or less time to complete, depending on the amount of coding that is asked for.

Our solutions vary in terms of detail—for some problems we present detailed implementations in Java/C++/Python; for others, we simply sketch solutions. Some use fairly technical machinery, e.g., max-flow, randomized analysis, etc. You will encounter such problems only if you claim specialized knowledge, e.g., graph algorithms, complexity theory, etc.

Interviewing is about more than being able to design algorithms quickly. You also need to know how to present yourself, how to ask for help when you are stuck, how to come across as being excited about the company, and knowing what you can do for them. We discuss the nontechnical aspects of interviewing in Chap-

Problem Solving Techniques

> If there is a problem you can't solve, then there is an easier problem you can solve: find it.
>
> "How To Solve It," G. Pólya, 1945

Developing problem solving skills is like learning to play a musical instrument—a book or a teacher can point you in the right direction, but only your hard work will take you where you want to go. Like a musician, you need to know underlying concepts but theory is no substitute for practice; for this reason, AFI consists primarily of problems.

Great problem solvers have skills that cannot be captured by a set of rules. Still, when faced with a challenging algorithm design problem it is helpful to have a small set of general principles that may be applicable. We enumerate a collection of such principles in Table 1. Often, you may have to use more than one of these techniques.

We will now look at some concrete examples of how these techniques can be applied.

DIVIDE-AND-CONQUER AND GENERALIZATION

A triomino is formed by joining three unit-sized squares in an L-shape. A mutilated chessboard (henceforth 8×8 Mboard) is made up of 64 unit-sized squares arranged in an 8×8 square, minus the top left square. Supposed you are asked to design an algorithm which computes a placement of 21 triominos that covers the 8×8 Mboard. (Since there are 63 squares in the 8×8 Mboard and we have 21 triominos, a valid placement cannot have overlapping triominos or triominos which extend out of the 8×8 Mboard.)

Divide-and-conquer is a good strategy to attack this problem. Instead of the 8×8 Mboard, let's consider an $n \times n$ Mboard. A 2×2 Mboard can be covered with 1 triomino since it is of the same exact shape. You may hypothesize that a triomino placement for an $n \times n$ Mboard with the top left square missing can be used to

Technique	Description
Divide-and-conquer	Can you divide the problem into two or more smaller independent subproblems and solve the original problem using solutions to the subproblems?
Recursion and dynamic programming	If you have access to solutions for smaller instances of a given problem, can you easily construct a solution to the problem?
Case analysis	Can you split the input/execution into a number of cases and solve each case in isolation?
Generalization	Is there a problem that subsumes your problem and is easier to solve?
Data-structures	Is there a data-structure that directly maps to the given problem?
Iterative refinement	Most problems can be solved using a brute-force approach. Can you formalize such a solution and improve upon it?
Small examples	Can you find a solution to small concrete instances of the problem and then build a solution that can be generalized to arbitrary instances?
Reduction	Can you use a problem with a known solution as a subroutine?
Graph modeling	Can you describe your problem using a graph and solve it using an existing algorithm?
Write an equation	Can you express relationships in your problem in the form of equations (or inequalities)?
Auxiliary elements	Can you add some new element to your problem to get closer to a solution?
Variation	Can you solve a slightly different problem and map its solution to your problem?
Parallelism	Can you decompose your problem into subproblems that can be solved independently on different machines?
Caching	Can you store some of your computation and look it up later to save work?
Symmetry	Is there symmetry in the input space or solution space that can be exploited?

Table 1. Common problem solving techniques.

Problem 1.9: Design an efficient algorithm for determining if there exist a pair of indices i, j (not necessarily distinct) such that $A[i] + A[j] = S$.

1.10 ANONYMOUS LETTER

A hash can be viewed as a dictionary. As a result, hashing commonly appears when processing with strings.

Problem 1.10: You are required to write a method that takes two text documents: an anonymous letter L and text from a magazine M. Your method is to return true if L can be written using M and false otherwise. (If a letter appears k times in L, it must appear at least k times in M.)

1.11 PAIRING USERS BY ATTRIBUTES

You are building a social networking site where each user specifies a set of attributes. You would like to pair each user with another unpaired user that specified exactly the same set of attributes.

Specifically, you are given a sequence of users where each user has a unique key, say a 32-bit integer and a set of attributes specified as a set of strings. As soon as you read a user, you should pair it with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should keep him in the unpaired set.

Problem 1.11: How would you implement this matching process efficiently? How would you implement it if we allow an approximate match of attributes as well?

1.12 MISSING ELEMENT

Hashing can be used to find an element which is not present in a given set.

Problem 1.12: Given an array A of integers, find an integer k that is not present in A. Assume that the integers are 32-bit signed integers.

1.13 ROBOT BATTERY CAPACITY

A robot needs to travel along a path that includes several ascents and descents. When it goes up, it uses its battery as a source of energy and when it goes down, it recovers the potential energy back into the battery. The battery recharging process is ideal: on descending, every Joule of gravitational potential energy converts into a Joule of electrical energy that is stored in the battery. The battery has a limited capacity and once it reaches its storage capacity, the energy generated from the robot going down is lost.

Problem 1.13: Given a robot with the energy regeneration ability described above, the mass of the robot m and a sequence of three-dimensional co-ordinates that the

2.10 MERGING SORTED ARRAYS

You are given 500 files, each containing stock quote information for an SP500 company. Each line contains an update of the following form:

```
1232111 131 B 1000 270
2212313 246 S 100  111.01
```

The first number is the update time expressed as the number of milliseconds since the start of the day's trading. Each file individually is sorted by this value. Your task is to create a single file containing all the updates sorted by the update time. The individual files are of the order of 1–100 megabytes; the combined file will be of the order of 5 gigabytes.

Problem 2.10: Design an algorithm that takes the files as described above and writes a single file containing the lines appearing in the individual files sorted by the update time. The algorithm should use very little memory, ideally of the order of a few kilobytes.

2.11 APPROXIMATE SORT

Consider a situation where your data is almost sorted—for example, you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network routes. What would be the most efficient way of restoring the total order?

Problem 2.11: There is a very long stream of integers arriving as an input such that each integer is at most one thousand positions away from its correctly sorted position. Design an algorithm that outputs the integers in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of integers processed.

2.12 RUNNING AVERAGES

Suppose you are given a real-valued time series (e.g., temperature measured by a sensor) with some noise added to it. In order to extract meaningful trends from noisy time series data, it is necessary to perform smoothing. If the noise has a Gaussian distribution and the noise added to successive samples is independent and identically distributed, then the running average does a good job of smoothing. However if the noise can have an arbitrary distribution, then the running median does a better job.

Problem 2.12: Given a sequence of trillion real numbers on a disk, how would you compute the running mean of every thousand entries, i.e., the first point would be the mean of $a[0], \ldots, a[999]$, the second point would be the mean of $a[1], \ldots, a[1000]$, the third point would be the mean of $a[2], \ldots, a[1001]$, etc.? Repeat the calculation for median rather than mean.

3.11 MAXIMIZING EXPRESSIONS

The value of an arithmetic expression depends upon the order in which the operations are performed. For example, depending upon how one parenthesizes the expression $5 - 3 \cdot 4 + 6$, one can obtain any one of the following values:

$$\begin{aligned} -25 &= 5 - \big(3 \cdot (4 + 6)\big) \\ -13 &= 5 - \big((3 \cdot 4) + 6\big) \\ 20 &= (5 - 3) \cdot (4 + 6) \\ -1 &= \big(5 - (3 \cdot 4)\big) + 6 \\ 14 &= \big((5 - 3) \cdot 4\big) + 6 \end{aligned}$$

Given an unparenthesized expression of the form $v_0 \circ_0 v_1 \circ_1 \cdots \circ_{n-2} v_{n-1}$, where $v_0, \ldots, v_{n-1}$ are operands with known real values and $\circ_0, \ldots, \circ_{n-2}$ are specified operations, we want to parenthesize the expression so as to maximize its value.

Problem 3.11: Devise an algorithm to solve this problem in the special case that the operands are all positive and the only operations are $\cdot$ and $+$.

Explain how you would modify your algorithm to deal with the case in which the operands can be positive and negative and $+$ and $-$ are the only operations.

Suggest how you would generalize your approach to include multiplication and division (pretend divide-by-zero never occurs).

Greedy Algorithms

A greedy algorithm is one which makes decisions that are locally optimum and never changes them. This approach does not work generally. For example, consider making change for 48 pence in the old British currency where the coins came in $30, 24, 12, 6, 3, 1$ pence denominations. A greedy algorithm would iteratively choose the largest denomination coin that is less than or equal to the amount of change that remains to be made. If we try this for 48 pence, we get $30, 12, 6$. However the optimum answer would be $24, 24$.

In its most general form, the coin changing problem is NP-hard (*cf*. Chapter 6) but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form $\{1, r, r^2, r^3\}$. *Ad hoc* arguments can be applied to show that it is also optimum for US coins. The general problem can be solved in pseudopolynomial time using DP in a manner similar to Problem 6.1.

3.12 SCHEDULING TUTORS

You are responsible for scheduling tutors for the day at a tutoring company. For each day, you have received a number of requests for tutors. Each request has a specified start time and each lesson is thirty minutes long. You have more tutors than requests. Each tutor can start work at any time. However tutors are con-

symmetric—A may be a contact of B but B may not be a contact of A.) Let's define C to be an extended contact of A if he is either a contact of A or a contact of an extended contact of A.

Problem 4.5: Devise an efficient algorithm which takes a social network and computes for each individual his extended contacts.

4.6 EULER TOUR

Leonhard Euler wrote a paper titled "Seven Bridges of Köenigsberg" in 1736. It is considered to be the first paper in graph theory. The problem was set in the city of Köenigsberg, which was situated on both sides of the Pregel River and included two islands which were connected to each other and the mainland by seven bridges. Euler posed the problem of finding a walk through the city that would cross each bridge exactly once. In the paper, Euler demonstrated that it was impossible to do so.

More generally, an Euler tour of a connected directed graph $G = (V, E)$ is a cycle that includes each edge of G exactly once; it may repeat vertices more than once.

Problem 4.6: Design a linear-time algorithm to find an Euler tour if one exists.

4.7 EPHEMERAL STATE IN A FINITE STATE MACHINE

A finite state machine (FSM) is a set of states S, a set of inputs I, and a transition function $T : S \times I \mapsto S$. If $T(s, i) = u$, we say that s *leads to* u on application of input i. The transition function T can be generalized to sequences of inputs—if $\iota = \langle i_0, i_1, \ldots, i_{n-1} \rangle$, then $T(s, \iota) = s$ if $n = 0$; otherwise, $T(s, \iota) = T\big(T(s, \langle i_0, i_1, \ldots, i_{n-2} \rangle), i_{n-1}\big)$.

The state e is said to be *ephemeral* if there is a sequence of inputs α such that there does not exist an input sequence β for which $T\big(T(e, \alpha), \beta\big) = e$. Informally, e is ephemeral if there is a possibility of the FSM starting at e and getting to a state f from which it cannot return to e.

Problem 4.7: Design an efficient algorithm which takes an FSM and returns the set of ephemeral states.

4.8 TREE DIAMETER

Packets in Ethernet LANs are routed according to the unique path in a tree whose vertices correspond to clients and edges correspond to physical connections between the clients. In this problem, we want to design an algorithm for finding the "worst-case" route, i.e., the two clients that are furthest apart.

Problem 4.8: Let T be a tree, where each edge is labeled with a real-valued distance. Define the diameter of T to be the length of a longest path in T. Design an efficient algorithm to compute the diameter of T.

5.6 LONGEST PALINDROME SUBSEQUENCE

A palindrome is a string which is equal to itself when reversed. For example, the human Y-chromosome contains a gene with the amino acid sequence

$$\langle C, A, C, A, A, T, T, C, C, C, A, T, G, G, G, T, T, G, T, G, G, A, G \rangle,$$

which includes the palindromic subsequences $\langle T, G, G, G, T \rangle$ and $\langle T, G, T \rangle$. Palindromic subsequences in DNA are significant because they influence the ability of the strand to loop back on itself.

Problem 5.6: Devise an efficient algorithm that takes a DNA sequence $D[1, \ldots, n]$ and returns the length of the longest palindromic subsequence.

5.7 PRETTY PRINTING

Consider the problem of arranging a piece of text in a fixed width font (i.e., each character has the same width) in a rectangular space. Breaking words across line boundaries is visually displeasing. If we avoid word breaking, then we may frequently be left with many spaces at the end of lines (since the next word will not fit in the remaining space). However if we are clever about where we break the lines, we can reduce this effect.

Problem 5.7: Given a long piece of text, decompose it into lines such that no word spans across two lines and the total wasted space at the end of each line is minimized.

5.8 EDIT DISTANCES

Spell checkers make suggestions for misspelled words. Given a misspelled string s, a spell checker should return words in the dictionary which are close to s.

One definition of closeness is the number of "edits" it would take to transform the misspelled word into a correct word, where a single edit is the deletion or insertion of a single character.

Problem 5.8: Given two strings A and B, compute the minimum number of edits needed to transform A into B.

5.9 REGULAR EXPRESSION MATCHING

A regular expression is a sequence of characters that defines a set of matching strings. For this problem, we define a simple subset of a full regular expression language:

– Alphabetical and numerical characters match themselves. For example, `aW9` will match that string of 3 letters wherever it appears.

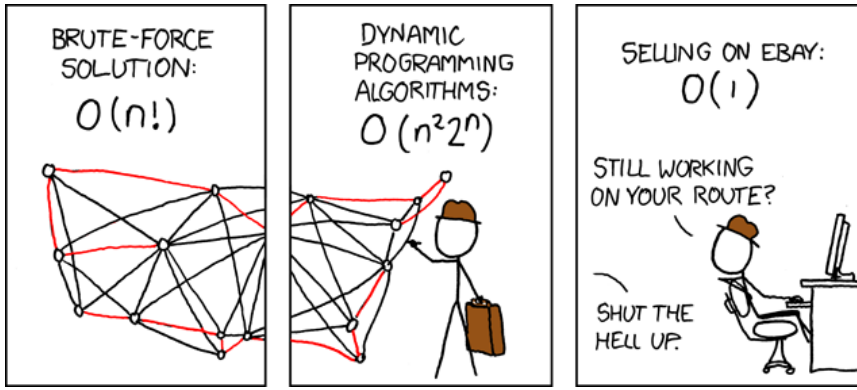

Figure 5. $P = NP$, by XKCD

to it. Often this reduction gives insight into the cause of intractability.

Unless you are a complexity theorist, proving a problem to be intractable is a starting point, not an end point. Remember something is a problem only if it has a solution. There are a number of approaches to solving intractable problems:

- Brute-force solutions which are typically exponential but may be acceptable, if the instances encountered are small.
- Branch-and-bound techniques which prune much of the complexity of a brute-force search.
- Approximation algorithms which return a solution that is provably close to optimum.
- Heuristics based on insight, common case analysis, and careful tuning that may solve the problem reasonably well.
- Parallel algorithms, wherein a large number of computers can work on subparts simultaneously.

6.1 0-1 KNAPSACK

A thief has to choose from n items. Item i can be sold for v_i dollars and weighs w_i pounds (v_i and w_i are integers). The thief wants to take as valuable a load as possible but he can carry at most W pounds in his knapsack.

Problem 6.1: Design an algorithm that will select a subset of items that has maximum value and weighs at most W pounds. (This problem is called the 0-1 knapsack problem because each item must either be taken or left behind—the thief cannot take a fractional amount of an item or take an item more than once.)

The following two problems exhibit structure that can be exploited to come up with fast algorithms that return a solution that is within a constant factor of the optimum (2 in both cases).

may access s for reading or writing while another thread is writing to s. (Two or more readers may access s at the same time.)

One way to achieve this is by protecting s with a mutex that ensures that no thread can access s at the same time as another writer. However this solution is suboptimal because it is possible that a reader $R1$ has locked s and another reader $R2$ wants to access s. There is no need to make $R2$ wait until $R1$ is done reading; instead, $R2$ should start reading right away.

This motivates the first readers-writers problem: protect s with the added constraint that no reader is to be kept waiting if s is currently opened for reading.

Problem 7.5: Implement a synchronization mechanism for the first readers-writers problem.

7.6 Readers-writers with write preference

Suppose we have an object s as in Problem 7.5. In the solution to Problem 7.5, a reader $R1$ may have the lock; if a writer W is waiting for the lock and then a reader $R2$ requests access, $R2$ will be given priority over W. If this happens often enough, W will starve. Instead, suppose we want W to start as soon as possible.

This motivates the second readers-writers problem: protect s with "writer-preference", i.e., no writer, once added to the queue, is to be kept waiting longer than absolutely necessary.

Problem 7.6: Implement a synchronization mechanism for the second readers-writers problem.

7.7 Readers-writers with fairness

The specifications to both Problems 7.5 and 7.6 can lead to starvation—the first may starve writers and the second may starve readers.

The third readers-writers problem adds the constraint that no thread shall be allowed to starve—the operation of obtaining a lock on s always terminates in a bounded amount of time.

Problem 7.7: Implement a synchronization mechanism for the third readers-writers problem. It is acceptable (indeed necessary) that in this solution, both readers and writers have to wait longer than absolutely necessary. (Readers may wait even if s is opened for read and writers may wait even if no one else has a lock on s.)

7.8 Producer-consumer queue

Two threads, the producer P and the consumer Q, share a fixed length array of strings A. The producer generates strings one at a time which it writes into A; the consumer removes strings from A, one at a time.

8.2 SEARCH ENGINE

Modern keyword-based search engines maintain a collection of several billion documents. One of the key computations performed by a search engine is to retrieve all the documents that contain the keywords contained in a given query. This is a nontrivial task because it must be done within few tens of milliseconds.

In this problem, we consider a smaller version of the problem where the collection of documents can fit within the RAM of a single computer.

Problem 8.2: Given a million documents with an average size of 10 kilobytes, design a program that can efficiently return the subset of documents containing a given set of words.

8.3 IP FORWARDING

There are many applications where instead of an exact match of strings, we are looking for a prefix match, i.e., given a set of strings and a search string, we want to find a string from the set that is a prefix of the search string. One application of this is Internet Protocol (IP) route lookup problem. When an IP packet arrives at a router, the router looks up the next hop for the packet by searching the destination IP address of the packet in its routing table. The routing table is specified as a set of prefixes on the IP address and the router is supposed to identify the longest matching prefix. If this task is to be performed only once, it is impossible to do better than testing each prefix. However an Internet core router needs to lookup millions of destination addresses on the set of prefixes every second. Hence it can be advantageous to do some precomputation.

Problem 8.3: You are given a large set of strings S in advance. Given a query string Q, how would you design a system that can identify the longest string $p \in S$ that is a prefix of Q?

8.4 SPELL CHECKER

Designing a good spelling correction system can be challenging. We discussed spelling correction in the context of the edit distance (Problem 5.8). However in that problem, we just considered the problem of computing the edit distance between a pair of strings. A spell checker must find a set of words that are closest to a given word from the entire dictionary. Furthermore, edit distance may not be the right distance function when performing spelling correction—it does not take into account the commonly misspelled words or the proximity of letters on a keyboard.

Problem 8.4: How would you build a spelling correction system?

8.5 STEMMING

When a user submits the query "computation" to a search engine, it is quite possible he might be interested in documents containing the words "computers", "com-

Chapter 9

Discrete Mathematics

> There is required, finally, the ratio between the fluxion of any quantity x you will and the fluxion of its power x^n. Let x flow till it becomes $x + o$ and resolve the power $(x + o)^n$ into the infinite series $x^n + nox^{n-1} + \frac{1}{2}(n^2 - n)o^2x^{n-2} + \frac{1}{6}(n^3 - 3n^2 + 2n)o^3x^{n-3} \cdots$
>
> "On the Quadrature of Curves," I. Newton, 1693

Discrete mathematics comes up in algorithm design in many places such as combinatorial optimization, complexity analysis, and probability estimation. Discrete mathematics is also the source of some of the most fun puzzles and interview questions. The solutions can range from simple application of the pigeon-hole principle to complex inductive reasoning.

Some of the problems in this chapter fall into the category of brain teasers where all you need is one *aha* moment to solve the problem. Such problems have fallen out of fashion because it is hard to judge a candidate's ability based on whether he is able to make a tricky observation in a short period of time. However they are asked enough times that we feel it is important to cover them. Also, these problems are quite a lot of fun to solve.

9.1 Computing the binomial coefficients

The symbol $\binom{n}{k}$ is short form for $\frac{n(n-1)\ldots(n-k+1)}{k(k-1)\ldots3\cdot2\cdot1}$. It is the number of ways to choose a k-element subset from an n-element set.

It is not obvious that the expression defining $\binom{n}{k}$ always yields an integer. Furthermore, direct computation of $\binom{n}{k}$ from this expression quickly results in the

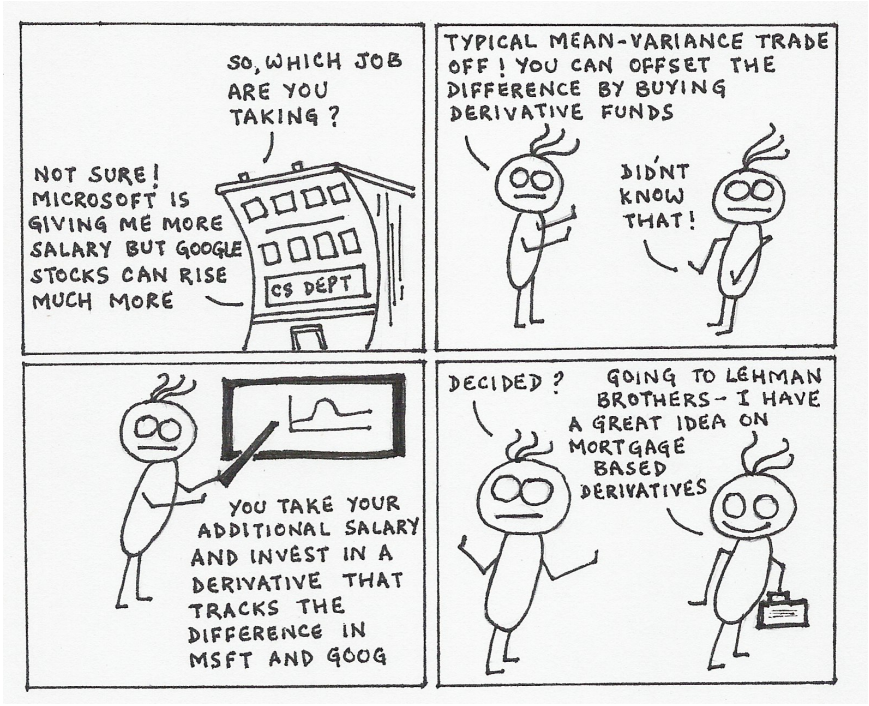

Figure 6. *FINANCIAL ENGINEERING*: an oxymoron widely used *circa* 2008.

10.11 EXPECTED NUMBER OF DICE ROLLS

Bob repeatedly rolls an unbiased 6-sided dice. He stops when he has rolled all the six numbers on the dice. How many rolls will it take, on an average, for Bob to see all the six numbers?

Option pricing

A call option gives the owner the right to buy something—a share, a barrel of oil, an ounce of gold—at a predetermined price at a predetermined time in the future. If the option is not priced fairly, an arbitrageur can either buy or sell the option in conjunction with other transactions and come up with a scheme of making money in a guaranteed fashion. A fair price for an option would be a price such that no arbitrage scheme can be designed around it.

We now consider problems related to determining the fair price for an option for a stock, given the distribution of the stock price for a period of time.

12.3 PRESENTING THE SOLUTION

Once you have a solution, it is important to present it well and do a comprehensive job at it. A lot of these things become simpler if you use a higher level language such as Java. However you should use the language with which you are most familiar. In most scenarios, it is perfectly fine to write a pseudocode as well. Here are some thoughts that could help:

Test for corner cases: For a number of problems, your general idea may work for the majority of the cases but there may be a few obscure inputs where your algorithm (or code) would fail. For example, you could write a binary search code that crashes if the input is an empty array or you may do arithmetic without considering the possibility of integer overflow. It is important to check for these things carefully. One way of doing this is to construct a few test cases and work out the output of your algorithm for them. In many cases, the code to handle some obscure corner cases may be too complicated. In such cases, you should at least mention to the interviewer that you are aware of the problem and you could try to address it if they are interested.

Function signature: Several candidates tend to get this wrong and getting your function signature wrong reflects badly on you. For example, it would be bad if you are writing the code in C language and your function returns an array but you fail to return the size of the array along with the pointer. Another place where function signatures could be important is knowing when to pass parameters by value versus by reference.

Memory management: If you allocate memory in your function, you must ensure that in every execution path, this memory is de-allocated. In general, it is best to avoid memory management operations all together. If you must do this, consider use of scoped pointers.

Syntax: In almost all cases, the interviewers are not evaluating you on the correctness of the syntax of your code. The editors and compilers do a great job at helping you get the syntax right. However you cannot underplay the possibility of an interviewer leaving with the impression that you got most of the syntax wrong since you do not have much experience writing code. Hence once you are done writing your code, you should make a pass over it to avoid any obvious syntax errors before claiming you are done.

12.4 KNOW YOUR INTERVIEWERS

If the organization can share some information about the background of your interviewers, it can help you a great deal. For fresh graduates, it is also important to think from the perspective of the interviewers. Hence we highly recommend reading the next chapter on interviewing from the perspective of an interviewer.

It is also important to note that once you ace the interview, you will have an offer and you would have an important decision to make— is this the organization where you want to work? Interviews are the best time to collect this information.

that have the same hash sequence, which is similar to the original problem. Here k can be varied appropriately to increase or decrease the probability of match for a pair of slightly different attribute sets.

Solution 1.12: The idea here is very similar to hashing. Consider a very simple hash function $F(x) = x \bmod (n+1)$. We can build a bit-vector of length $n+1$ that is initialized to 0 and for every element in A, we set bit $F\big(A[i]\big)$ to 1. Since there are only n elements in the array, there has to be at least one bit in the vector that is not set. That would give us the number that is not there in the array.

An even simpler approach is to find the max (or min) element in the array and return one more (less) than that element. This approach will not work if the extremal elements are the largest (smallest) values in the set that the entries are drawn from.

Solution 1.13: Since the energy is only related to the height of the robot, we can ignore x and y co-ordinates. Let's say that the points where the robot goes in successive order have heights $h_1, \ldots, h_n$. Let's assume that the battery capacity is such that with full battery, the robot can climb up B meters. Then the robot will run out of battery iff there exist integers i and j such that $i < j$ and $h_j - h_i > B$. In other words, in order to go from point i to point j, the robot needs to climb more than B points. So, we would like to pick B such that for any $i < j$, we have $B \geq h_j - h_i$.

If we did not have the constraint that $i < j$, then we could just compute B as $\max(h) - \min(h)$ but this may be an overestimate: consider the case when the robot is just going downwards.

We can compute the minimum B in $O(n)$ time if we keep the running min as we do a sweep. In code:

```
1  double BatteryCapacity(vector<double> h) {
2    if (h.size() < 2) {
3      return 0;
4    }
5    double min = h[0];
6    double result = 0;
7    for (int i = 1; i < h.size(); ++i) {
8      if (h[i] - min > result) {
9        result = h[i] - min;
10     }
11     if (min > h[i]) {
12       min = h[i];
13     }
14   }
15   return result;
16 }
```

Solution 1.14: Let's first consider just the strict majority case. This problem has an elegant solution when you make the following observation: if you take any two distinct elements from the stream and throw them away, the majority element re-

mains the majority of the remaining elements (we assumed there was a majority element to begin with). The reasoning goes as follows: let's say the majority element occurred m times out of n elements in the stream such that $m/n > 1/2$. The two distinct elements that we choose to throw can have at most one of the majority elements. Hence after discarding them, the ratio of the previously majority element could be either $m/(n-2)$ or $(m-1)/(n-2)$. It is easy to verify that if $m/n > 1/2$, then $m/(n-2) > (m-1)/(n-2) > 1/2$.

Now, as we read the stream from beginning to the end, as soon as we encounter more than one distinct element, we can discard one instance of each element and what we are left with in the end must be the majority element.

```
1  string FindMajority(stream* s) {
2    string candidate;
3    int count = 0;
4    string next_word;
5    while (s->GetNext(&next_word)) {
6      if (count == 0) {
7        candidate = next_word;
8        count = 1;
9      } else if (candidate == next_word) {
10       count++;
11     } else {
12       count--;
13     }
14   }
15   return candidate;
16 }
```

The code above assumes there is a majority word in the stream; if no word has a strict majority, it still returns a string but there are no meaningful guarantees on what that string would be.

Solution 1.15: This is essentially a generalization of Problem 1.14. Here instead of discarding two distinct words, we discard k distinct words at any given time and we are guaranteed that all the words that occurred more than $1/k$ times the length of the stream before discarding continue to have more than $1/k$ fraction of copies. For implementing this strategy, we need to keep a hash table of current k candidates. Here is an example code:

```
1  void FindFrequentItems(stream* s, hash_map<string, int>* word_set, int
       k) {
2    word_set->clear();
3    string word;
4    while(s->GetNextWord(&word)) {
5      hash_map<string, int>::iterator i = word_set->find(word);
6      if (i == word_set->end()) {
7        if (word_set->size() == k) {
8          // Hash table is full, decrement all counts
9          // which is equivalent to discarding k distinct
10         // words.
11         for (hash_map<string, int>::iterator j = word_set->begin();
```

check its distance to every other city.

Let's say we have selected the first $i-1$ warehouses $\{c_1, c_2, \ldots, c_{i-1}\}$ and are trying to choose the i-th warehouse. A reasonable choice for c_i is the one that is the farthest from the $i-1$ warehouses already chosen. This can also be computed in $O(n^2)$ time.

Let the maximum distance from any remaining cities to a warehouse be d_m. Then the cost of this assignment is d_m. Let e be a city that has this distance to the warehouses. In addition, the m warehouse cities are all at least d_m apart; otherwise, we would have chosen e and not c_m at the m-th selection.

At least two of these $m+1$ cities have to have the same closest warehouse in an optimum assignment. Let p, q be two such cities and w be the warehouse they are closest to. Since $d(p,q) \leq d(w,p) + d(w,q)$, it follows that one of $d(w,p)$ or $d(w,q)$ is not less than $d_m/2$. Hence the cost of this optimum assignment is at least $d_m/2$, so our greedy heuristic produced an assignment that is within a factor of two of the optimum cost assignment.

Note that the initial selection of a warehouse is immaterial for the argument to work but heuristically, it is better to choose a central city as a starting point.

Solution 6.4: It is natural to try and solve this problem by divide-and-conquer, e.g., determine the minimum number of multiplications for each of x^k and $x^{30/k}$, for different values of k. The problem is that the subproblems are not independent—we cannot just add the minimum number of multiplications for computing x^5 and x^6 since both may use x^3.

Instead we resort to branch-and-bound: we maintain a set of partial solutions which we try to extend to the final solution. The key to efficiency is pruning out partial solutions efficiently.

In our context, a partial solution is a list of exponents that we have already computed. Note that in a minimum solution, we will never have an element repeated in the list. In addition, it suffices to consider partial solutions in which the exponents occur in increasing order since if $k > j$ and x^k occurs before x^j in the chain, then x^k could not be used in the derivation of x^j. Hence we lose nothing by advancing the position of x^k.

Here is code that solves the problem:

```
1  import java.util.LinkedList;
2
3  public class MinExp {
4
5    public static void main( String [] args ) {
6
7      int target = new Integer(args[0]);
8
9      LinkedList<Integer> initEntry = new LinkedList<Integer>();
10     initEntry.add(1);
11
12     LinkedList<LinkedList<Integer>> partials = new LinkedList<
           LinkedList<Integer>>();
```

There are two key components to building such a system: (1.) the front-facing component, by which advertisers can add their advertisements, control when their ads get displayed, how much and how they want to spend their advertising money, and review the performance of their ads and (2.) the ad-serving system which selects which ads to show on the searches.

The front-facing system can be a fairly conventional website design. Users interact with the system using a browser and opening a connection to the website. You will need to build a number of features:

- **User authentication**—a way for users to create accounts and authenticate themselves.
- **User state**—a set of forms to let advertisers specify things like their advertising materials, their advertising budget etc. Also a way to store this information persistently.
- **Performance reports**—a way to generate reports on how and where the advertiser's money is being spent.
- **Human interactions**—even the best of automated systems require occasional human interaction and a way to interfere with the algorithms. This may require an interface for advertisers to be able to contact customer service representatives and an interface for those representatives to interact with the system.

The whole front-end system can be built using, for example, HTML and JavaScript, with a LAMP stack (Linux as the operating system, Apache as the HTTP server, MySQL as the database software, and PHP for the application logic) responding to the user input.

The ad-serving system would probably be a less conventional web service. It needs to choose ads based on their "relevance" to the search, perhaps some knowledge of the user's search history, and how much the advertiser is willing to pay. A number of strategies could be envisioned here for estimating relevance, such as, using information retrieval or machine learning techniques that learn from past user interactions.

The ads can be added to the search results by embedding JavaScript in the results that pulls in the ads from the ad-serving system directly. This helps isolate the latency of serving search results from the latency of serving ad results.

Solution 8.11: The key technical challenge in this problem is to come up with the list of articles—the HTML code for adding these to a sidebar is trivial.

One suggestion might be to add articles that have proven to be popular recently. Another is to have links to recent news articles. A human reader at Jingle could tag articles which he believes to be significant. He could also add tags such as finance, politics, etc. to the articles. These tags could also come from the HTML meta-tags or the page title.

We could also sometimes provide articles at random and see how popular they prove to be; the popular articles can then be shown more frequently.