# NOTE: search “leochen4891” for important info

# Algorithm Analysis

*Problem:* Robot Tour Optimization

*Input:* A set *S* of *n* points in the plane.

*Output:* What is the shortest cycle tour that visits each point in the set *S*?

The quest for an efficient algorithm to solve this problem, called the *traveling*

*salesman problem* (TSP),

There is a fundamental difference between *algorithms*,

which always produce a correct result, and *heuristics*, which may usually do a

good job but without providing any guarantee

*Problem:* Movie Scheduling Problem (Independent Set problem)

*Input:* A set *I* of *n* intervals on the line.

*Output:* What is the largest subset of mutually non-overlapping intervals which can

be selected from *I*?

OptimalScheduling(I)

While (*I \_*= *∅*) do

Accept the job *j* from *I* with the earliest completion date.

Delete *j*, and any interval which intersects *j* from *I*.

The three most common forms of algorithmic

notation are (1) English, (2) pseudocode, or (3) a real programming language

*Take-Home Lesson:* The heart of any algorithm is an *idea*. If your idea is

not clearly revealed when you express an algorithm, then you are using too

low-level a notation to describe it.

An important and honorable technique in algorithm design is to narrow the set of allowable instances until there *is* a correct and efficient algorithm. For example, we can restrict a graph problem from general graphs down to trees, or a geometric problem from two dimensions down to one.

Mathematical induction is usually the right way to verify

the correctness of a recursive or incremental insertion algorithm.

Combinatorial Objects

1. *Permutations*
2. *Subsets*
3. *Trees*
4. *Graphs*
5. *Points*
6. *Polygons*
7. *Strings*

*Take-Home Lesson:* Modeling your application in terms of well-defined structures and algorithms is the most important single step towards a solution.

NP-Complete: set cover problem

To compare the efficiency of algorithms without implementing them, our two most important tools are (1) the RAM model of computation and (2) the asymptotic analysis of worst-case complexity.

n! >> 2^n >> n^3 >> n^2 >> nlogn >> n >> logn >> 1

*• Exponential functions, f*(*n*) = *cn for a given constant c >* 1 – Functions like

2*n* arise when enumerating all subsets of *n* items. As we have seen, exponential

algorithms become useless fast, but not as fast as. . .

*• Factorial functions, f*(*n*) = *n*! – Functions like *n*! arise when generating all

permutations or orderings of *n* items.

*Problem:* Substring Pattern Matching

*Input:* A text string *t* and a pattern string *p*.

*Output:* Does *t* contain the pattern *p* as a substring, and if so where?

time complexity: O(nm)

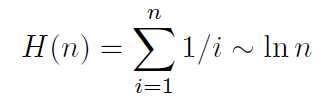
*Problem:* Matrix Multiplication

*Input:* Two matrices, *A* (of dimension *x × y*) and *B* (dimension *y × z*).

*Output:* An *x × z* matrix *C* where *C*[*i*][*j*] is the dot product of the *i*th row of *A*

and the *j*th column of *B*.

Harmonic numbers:



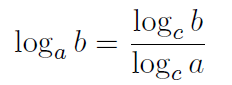
Complexity of Quicksort is the summation

Employing the Harmonic number identity immediately

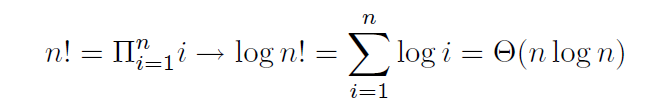
reduces this to Θ(*n* log *n*)

*Take-Home Lesson:* Logarithms arise whenever things are repeatedly halved

or doubled.



n! grows fast unless it is cut by a logarithm



# Data Structures

We will focus on each of the three fundamental abstract data types (containers, dictionaries,

and priority queues) and see how they can be implemented with arrays and lists.

Data structures can be neatly classified as either *contiguous* or *linked*, depending upon whether they are based on arrays or pointers.

Advantages of contiguously-allocated arrays include:

*• Constant-time access given the index*

*• Space efficiency*

*• Memory locality*

dynamic arrays. The cost of doubling its size is amortized. Overall is still O(n)

Balanced Search Trees: red-black tree, splay tree

*Take-Home Lesson:* Building algorithms around data structures such as dictionaries

and priority queues leads to both clean structure and good performance

Hash Table

Collision resolution:

1. *Chaining*
2. *open addressing*

Deletion in an open addressing needs to reinsert all the items in the run following the new hole.

String matching:

A linear *expected-time* algorithm for string matching, called the Rabin-Karp algorithm. It is based on hashing. (p.g. 103) leochen4891

The key idea of hashing is to represent a large object (be it a key, a string, or a substring) using a single number.

# Sorting and Searching

*Take-Home Lesson:* Sorting lies at the heart of many algorithms. Sorting the

data is one of the first things any algorithm designer should try in the quest

for efficiency.

*Problem:* Give an efficient algorithm to determine whether two sets (of size *m* and

*n*, respectively) are disjoint. Analyze the worst-case complexity in terms of *m* and

*n*, considering the case where *m* is substantially smaller than *n*

Stability can be achieved for any sorting algorithm by adding the initial

position as a secondary key

*heapsort*, is actually an implementation of selection sort using priority queue

Heaps are a simple and elegant data structure for efficiently supporting the priority

queue operations insert and extract-min. They work by maintaining a partial order

on the set of elements which is weaker than the sorted order (so it can be efficient

to maintain) yet stronger than random order (so the minimum element can be

quickly identified).

1. *make\_heap, n \* logn , insert to (n+1) place and swim up*
2. *extract minimum, n \* log n, pop 1st item, fill the hole with nth item, and sink down*

Use sink down in the make\_heap process can reduce the heap construction time to near linear, since the leaf nodes don’t need sinking down.

*Problem:* Given an array-based heap on *n* elements and a real number *x*, efficiently

determine whether the *k*th smallest element in the heap is greater than or equal

to *x*. Your algorithm should be *O*(*k*) in the worst-case, independent of the size of

the heap. Hint: you do not have to find the *k*th smallest element; you need only

determine its relationship to *x*.

int heap\_compare(priority\_queue \*q, int i, int count, int x)

{

if ((count <= 0) || (i > q->n) return(count);

if (q->q[i] < x) {

count = heap\_compare(q, pq\_young\_child(i), count-1, x);

count = heap\_compare(q, pq\_young\_child(i)+1, count, x);

}

return(count);

}

*incremental insertion* technique,where we build up a complicated structure on *n* items by first building it on *n−*1 items and then making the necessary changes to add the last item. Incremental insertion proves a particularly useful technique in geometric algorithms

Mergesort: A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements.

Mergesort is a great algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort. Its primary disadvantage

is the need for an auxilliary buffer when sorting arrays. It is easy to merge two

sorted linked lists without using any extra space, by just rearranging the pointers.

However, to merge two sorted arrays (or portions of an array), we need use a third

array to store the result of the merge to avoid stepping on the component arrays

Randomized quicksort runs in Θ(*n* log *n*) time on *any* input, with high probability.

*Problem:* The *nuts and bolts* problem is defined as follows. You are given a collection

of *n* bolts of different widths, and *n* corresponding nuts. You can test whether a

given nut and bolt fit together, from which you learn whether the nut is too large,

too small, or an exact match for the bolt. The differences in size between pairs of

nuts or bolts are too small to see by eye, so you cannot compare the sizes of two

nuts or two bolts directly. You are to match each bolt to each nut.

Give an *O*(*n*2) algorithm to solve the nuts and bolts problem. Then give a

randomized *O*(*n* log *n*) expected time algorithm for the same problem.

**Distribution Sort: Sorting via Bucketing**

**Binary search: Counting occurrences:**

int binary\_search(item\_type s[], item\_type key, int low, int high)

{

int middle; /\* index of middle element \*/

// if (low > high) return (-1); /\* key not found \*/

if (low > high) return **low**; // return low instead of -1 in BS

middle = (low+high)/2;

// if (s[middle] == key) return(middle); // comment this out in BS

if (s[middle] > key)

return( binary\_search(s,key,low,middle-1) );

else

return(binary\_search(s,key,middle+1,high) );

}

The search will proceed to the right half whenever the key is compared to an

identical array element, eventually terminating at the right boundary. Repeating

the search after reversing the direction of the binary comparison will lead us to the

left boundary.

Find square root of n. if n>=1, square root is between 1 and n. Do binary search between l = 1 and r = n, m = (l+r)/2. Improvement can be instead of always testing the midpoint of the interval,these algorithms interpolate to find a test point closer to the actual root.

Recurrence Relations

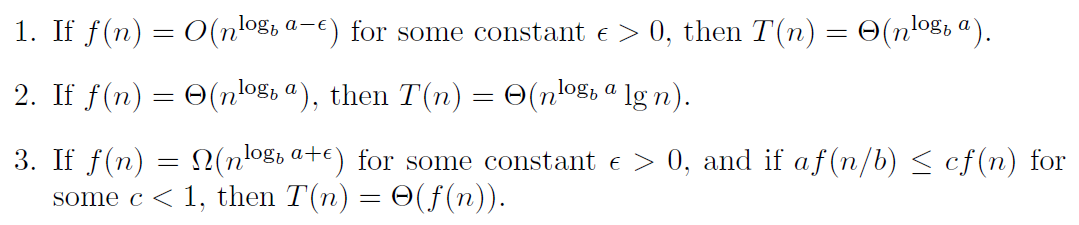
The Fibonacci numbers are described by the recurrence relation *Fn* = *Fn−*1+*Fn−*2

Factorial: an = nan−1, a1 = 1 🡪 an = n!

Divide-and-Conquer Recurrences

*T*(*n*) = *aT* (*n/b*) + *f*(*n*)

*master theorem*



*shellsort:* a substantially more efficient version of insertion sort,

*radix sort*, an efficient algorithm for sorting strings, O(kn), where k is the length of the strings.

# Graph Traversal

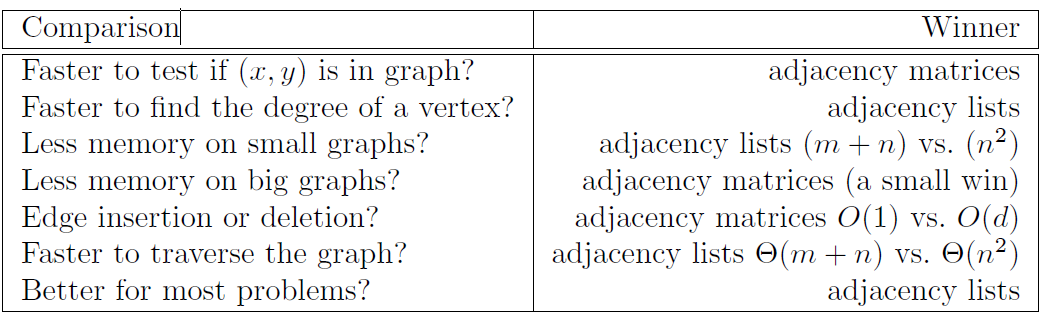
*Take-Home Lesson:* Graphs can be used to model a wide variety of structures

and relationships. Graph-theoretic terminology gives us a language to talk

about them.

Selecting the right graph data structure can have an enormous impact on performance.

Your two basic choices are adjacency matrices and adjacency lists.



*Take-Home Lesson:* Adjacency lists are the right data structure for most

applications of graphs

Graph Traversal:

The key idea behind graph traversal is to mark each vertex when we first visit

it and keep track of what we have not yet completely explored.

Each vertex will exist in one of three states:

*• undiscovered* – the vertex is in its initial, virgin state.

*• discovered* – the vertex has been found, but we have not yet checked out all

its incident edges.

*• processed* – the vertex after we have visited all its incident edges.

We must also maintain a structure containing the vertices that we have discovered

but not yet completely processed

one implementation uses arrays to save status and parent path

bool processed[MAXV+1]; /\* which vertices have been processed \*/

bool discovered[MAXV+1]; /\* which vertices have been found \*/

int parent[MAXV+1]; /\* discovery relation \*/

BFS runs in *O*(*n* + *m*) time on

both directed and undirected graphs. This is optimal, since it is as fast as one can

hope to *read* any *n*-vertex, *m*-edge graph

*Take-Home Lesson:* Breadth-first and depth-first searches provide mechanisms

to visit each edge and vertex of the graph. They prove the basis of most simple,

efficient graph algorithms.

The other important property of a depth-first search is that it partitions the

edges of an undirected graph into exactly two classes: *tree edges* and *back edges*. The

tree edges discover new vertices, and are those encoded in the parent relation. Back

edges are those whose other endpoint is an ancestor of the vertex being expanded,

so they point back into the tree.

*Take-Home Lesson:* DFS organizes vertices by entry/exit times, and edges

into tree and back edges. This organization is what gives DFS its real power.

Our implementation of dfs maintains a notion of traversal *time* for each vertex.

Our time clock ticks each time we enter or exit any vertex. We keep track of the

*entry* and *exit* times for each vertex. (leochen4891)

Back edges are the key to finding a cycle in an undirected graph. If there is no

back edge, all edges are tree edges, and no cycle exists in a tree. But *any* back edge

going from *x* to an ancestor *y* creates a cycle with the tree path from *y* to *x*.

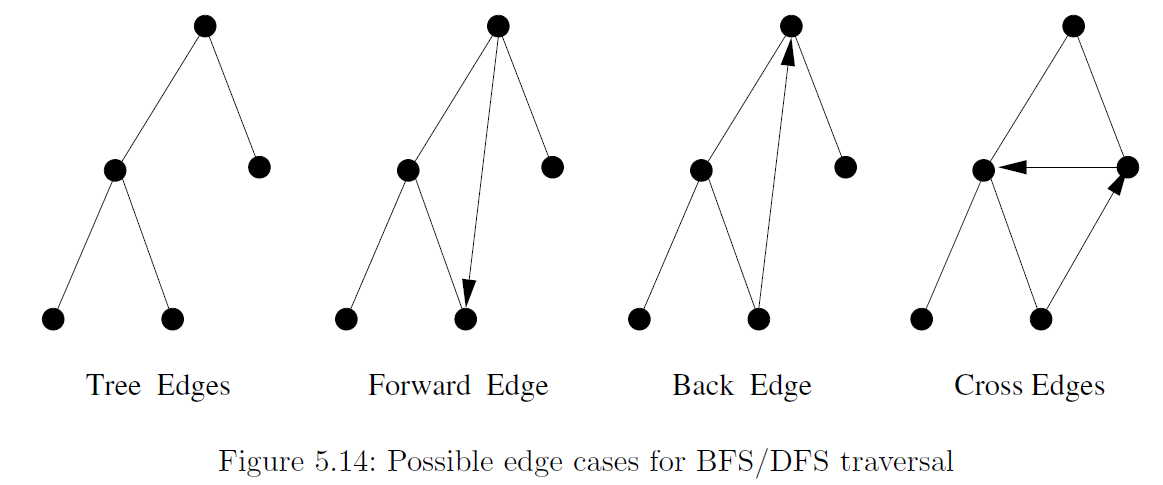
For directed graphs, depth-first search labeling can take on a wider range of

possibilities. Indeed, all four of the edge cases in Figure 5.14 can occur in traversing

directed graphs. Still, this classification proves useful in organizing algorithms on

directed graphs. We typically take a different action on edges from each different

case. The correct labeling of each edge can be readily determined from the state,

discovery time, and parent of each vertex. (leochen4891)  


Topological sorting

Topological sorting is the most important operation on directed acyclic graphs

(DAGs). It orders the vertices on a line such that all directed edges go from left to

right. Each DAG has at least one topological sort. The importance of topological

sorting is that it gives us an ordering to process each vertex before any of its

successors

A directed graph is a DAG if and only if no back edges are encountered. Labeling

the vertices in the reverse order that they are marked *processed* finds a topological

sort of a DAG.

Strongly connected graph. A directed graph is *strongly connected* if there is a directed

path between any two vertices.

It is straightforward to use graph traversal to test whether a graph *G* = (*V,E*)

is strongly connected in linear time. First, do a traversal from some arbitrary vertex

*v*. And reverse the direction of all edges and do it again. The graph is strongly connected iff all vertices in *G* can (1) reach *v* and (2) are reachable from *v*.

# Weighted Graph Algorithms

Greedy algorithms make the decision of what to do next by selecting the best

local option from all available choices without regard to the global structure

Prim’s minimum spanning tree algorithm starts from one vertex and grows the rest

of the tree one edge at a time until all vertices are included. It select the edge of minimum weight between a tree and nontree vertex, add the selected edge and vertex to the tree *Tprim*.

A simple implementation of Prim’s algorithm has time complexity of O(mn), using a priority-queue data structures can reduce it to *O*(*m*+*n* lg *n*), by makingit faster to find the minimum cost edge to expand the tree at each iteration.

Kruskal’s algorithm is an alternate approach to finding minimum spanning trees

that proves more efficient on sparse graphs. In the most simple-minded approach, this can be implemented by breadth-first or depth-first search in a sparse graph with at most *n*

edges and *n* vertices, thus yielding an *O*(*mn*) algorithm.However, a clever data structure called *union-find*, can support such queries in *O*(lg *n*) time.

The Union-Find Data Structure

*• Find(i)* – Find the root of tree containing element *i*, by walking up the parent

pointers until there is nowhere to go. Return the label of the root.

*• Union(i,j)* – Link the root of one of the trees (say containing *i*) to the root

of the tree containing the other (say *j*) so *find*(*i*) now equals *find*(*j*).

We can do both unions and finds in *O*(log *n*), good enough for Kruskal’s algorithm. In fact, union-find can be done even faster,

*Minimum Product Spanning Trees*

Since lg(*a · b*) = lg(*a*) + lg(*b*), the minimum spanning tree on a graph whose

edge weights are replaced with their logarithms gives the minimum product

spanning tree on the original graph

Shortest Paths

Dijkstra’s algorithm is the method of choice for finding shortest paths in an edge and/or vertex-weighted graph. Given a particular start vertex *s*, it finds the shortest path from *s* to every other vertex in the graph, including your desired destination *t*.

Dijkstra works correctly only on graphs without negative-cost edges. The reason

is that midway through the execution we may encounter an edge with weight so

negative that it changes the cheapest way to get from *s* to some other vertex

already in the tree.

Most applications do not feature negative-weight edges, making this discussion

academic. Floyd’s algorithm, discussed below, works correctly unless there are

negative cost cycles, which grossly distort the shortest-path structure.

All-Pairs Shortest Path

We could solve *all-pairs shortest path* by calling Dijkstra’s algorithm from each

of the *n* possible starting vertices. But Floyd’s all-pairs shortest-path algorithm is

a slick way to construct this *n×n* distance matrix from the original weight matrix

of the graph.

Floyd’s algorithm is best employed on an adjacency matrix data structure,

which is no extravagance since we must store all *n*2 pairwise distances anyway

The Floyd-Warshall all-pairs shortest path runs in *O*(*n*3) time, which is asymptotically

no better than *n* calls to Dijkstra’s algorithm. However, the loops are so

tight and the program so short that it runs better in practice. It is notable as one of

the rare graph algorithms that work better on adjacency matrices than adjacency

lists.

Network Flows and Bipartite Matching

The maximum flow from *s* to *t* always equals the weight

of the minimum *s*-*t* cut. Thus, flow algorithms can be used to solve general

edge and vertex connectivity problems in graphs.

# Combinatorial Search and Heuristic Methods

Backtracking is a systematic way to iterate through all the possible configurations

of a search space

Search Pruning

# Dynamic Programming

Fibonacci Numbers： recursion < recursion + caching < DP(iteration using array)

Approximate String Matching

*• Substitution* – Replace a single character from pattern *P* with a different

character in text *T*, such as changing “shot” to “spot.”

*• Insertion* – Insert a single character into pattern *P* to help it match text *T*,

such as changing “ago” to “agog.”

*• Deletion* – Delete a single character from pattern *P* to help it match text *T*,

such as changing “hour” to “our.”

Assigning each operation an equal cost

of 1 defines the *edit distance* between two strings

*Longest Common Subsequence*

*Maximum Monotone Subsequence*

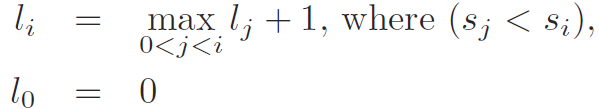
Longest Increasing Sequence

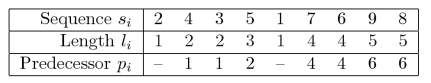
This provides the idea around which to build a recurrence. Define *li* to be the

length of the longest sequence ending with *si*.

The longest increasing sequence containing the *n*th number will be formed by

appending it to the longest increasing sequence to the left of *n* that ends on a

number smaller than *sn*. The following recurrence computes *li*: 



*Take-Home Lesson:* Once you understand dynamic programming, it can be

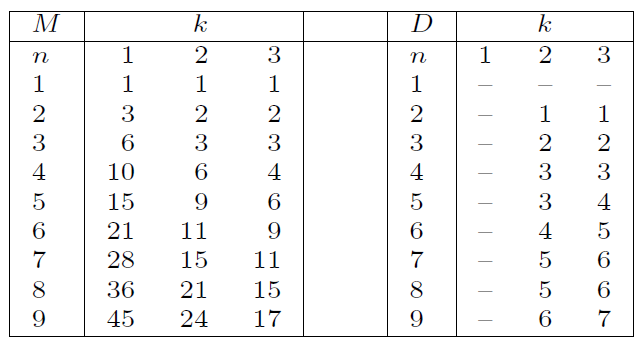
easier to work out such algorithms from scratch than to try to look them up

*Problem:* Integer Partition without Rearrangement

*Input:* An arrangement *S* of nonnegative numbers *{s*1*, . . . , sn}* and an integer *k*.

*Output:* Partition *S* into *k* or fewer ranges, to minimize the maximum sum over all

the ranges, without reordering any of the numbers.



Partitioning {1, 2, 3, 4, 5, 6, 7, 8, 9} into {{1, 2, 3, 4, 5}, {6, 7}, {8, 9}}

Parsing Context-Free Grammars

Minimum Weight Triangulation

*Take-Home Lesson:* For any optimization problem on left-to-right objects,

such as characters in a string, elements of a permutation, points around a

polygon, or leaves in a search tree, dynamic programming likely leads to an

efficient algorithm to find the optimal solution

**Limitations of Dynamic Programming: TSP**

*Problem:* Longest Simple Path

*Input:* A weighted graph *G*, with specified start and end vertices *s* and *t*.

*Output:* What is the most expensive path from *s* to *t* that does not visit any vertex

more than once?

For *unweighted* graphs (where each edge has cost 1), the longest possible simple

path from *s* to *t* is *n −* 1. Finding such *Hamiltonian paths* (if they exist) is an

important graph problem

Dynamic programming algorithms are only as correct as the recurrence relations

they are based on.

Dynamic programming can be applied to any problem that observes the *principle*

*of optimality*. Roughly stated, this means that partial solutions can be optimally

extended with regard to the *state* after the partial solution, instead of the specifics

of the partial solution itself

# Intractable Problems and Approximation Algorithms

The key idea to demonstrating the hardness of a problem is that of a *reduction*,

or translation, between two problems.

*Take-Home Lesson:* Reductions are a way to show that two problems are essentially

identical. A fast algorithm (or the lack of one) for one of the problems

implies a fast algorithm (or the lack of one) for the other

The simplest interesting class of problems have answers restricted to true and

false. These are called *decision problems*. It proves convenient to reduce/translate

answers between decision problems because both only allow true and false as possible

answers.

the traveling salesman decision problem could be defined as:

*Problem:* The Traveling Salesman Decision Problem

*Input:* A weighted graph *G* and integer *k*.

*Output:* Does there exist a TSP tour with cost *≤ k*?

*Take-Home Lesson:* Reductions are a way to show that two problems are essentially

identical. A fast algorithm (or the lack of one) for one of the problems

implies a fast algorithm (or the lack of one) for the other.

*Problem:* Hamiltonian Cycle

*Input:* An unweighted graph *G*.

*Output:* Does there exist a simple tour that visits each vertex of *G* without repetition?

*Problem:* Vertex Cover

*Input:* A graph *G* = (*V,E*) and integer *k ≤ |V |*.

*Output:* Is there a subset *S* of at most *k* vertices such that every *e ∈ E* contains at

least one vertex in *S*?

*Problem:* Independent Set

*Input:* A graph *G* and integer *k ≤ |V |*.

*Output:* Does there exist an independent set of *k* vertices in *G*?

*Problem: General* Movie Scheduling Decision Problem

*Input:* A set *I* of *n sets of* intervals on the line, integer *k*.

*Output:* Can a subset of at least *k* mutually nonoverlapping interval *sets* which can

be selected from *I*?

*Problem:* Maximum Clique

*Input:* A graph *G* = (*V,E*) and integer *k ≤ |V |*.

*Output:* Does the graph contain a clique of *k* vertices; i.e. , is there a subset *S ⊂ V* ,

where *|S| ≤ k*, such that every pair of vertices in *S* defines an edge of *G*?

*Problem:* Satisfiability

*Input:* A set of Boolean variables *V* and a set of clauses *C* over *V* .

*Output:* Does there exist a satisfying truth assignment for *C*—i.e. , a way to set

the variables *v*1*, . . . , vn* true or false so that each clause contains at least one true

literal?

*Problem:* 3-Satisfiability (3-SAT)

*Input:* A collection of clauses *C* where each clause contains exactly 3 literals, over

a set of Boolean variables *V* .

*Output:* Is there a truth assignment to *V* such that each clause is satisfied?

P and NP

We can think of the class *P* as an exclusive club for algorithmic problems that

a problem can only join after demonstrating that there exists a polynomial-time

algorithm to solve it. Shortest path, minimum spanning tree, and the original movie

scheduling problem are all members in good standing of this class *P*.

*The P stands for polynomial-time*.

A less-exclusive club welcomes all the algorithmic problems whose solutions can

be *verified* in polynomial-time. As shown above, this club contains traveling salesman,

satisfiability, and vertex cover, none of which currently have the credentials

to join *P*. We call this less-exclusive club *NP*.

*NP stands for nondeterministic polynomial-time*

We say that a problem is *NP-hard* if, like satisfiability, it is at least as hard

as any problem in NP. We say that a problem is *NP-complete* if it is NP-hard,

and also in NP itself. All the NP-hard problems we have encountered in this book are also NP-complete.

Two-player games such as chess provide examples of problems that are NP-hard, which means they are harder than NP-complete.

Dealing with NP-complete Problems

You won’t find one that quickly solves the problem to optimality in the worst case. You still have three options:

*• Algorithms fast in the average case* – Examples of such algorithms include

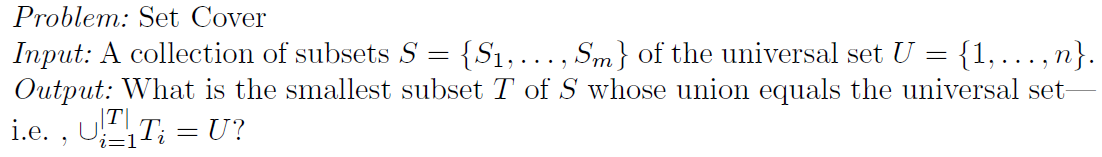
backtracking algorithms with substantial pruning.

*• Heuristics* – Heuristic methods like simulated annealing or greedy approaches

can be used to quickly find a solution with no guarantee that it will be the

best one.

*• Approximation algorithms*



*Take-Home Lesson:* Approximation algorithms guarantee answers that are

always close to the optimal solution. They can provide a practical approach to

dealing with NP-complete problems.

# How to Design Algorithms

A **strategic** question is “Can I model my application as a graph algorithm problem?”

A **tactical** question might be, “Should I use an adjacency list or adjacency matrix data structure to represent my graph?”

If you do not have a global strategy of how you are going to attack your problem, it is pointless to worry about the tactics.

1. Do I really understand the problem?

(a) What exactly does the input consist of?

(b) What exactly are the desired results or output?

(c) Can I construct an input example small enough to solve by hand? What

happens when I try to solve it?

(d) How important is it to my application that I always find the optimal

answer? Can I settle for something close to the optimal answer?

(e) How large is a typical instance of my problem? Will I be working on 10

items? 1,000 items? 1,000,000 items?

(f) How important is speed in my application? Must the problem be solved

within one second? One minute? One hour? One day?

(g) How much time and effort can I invest in implementation? Will I be

limited to simple algorithms that can be coded up in a day, or do I have

the freedom to experiment with a couple of approaches and see which

is best?

(h) Am I trying to solve a numerical problem? A graph algorithm problem?

A geometric problem? A string problem? A set problem? Which

formulation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

(a) Will brute force solve my problem *correctly* by searching through all

subsets or arrangements and picking the best one?

i. If so, why am I sure that this algorithm always gives the correct

answer?

ii. How do I measure the quality of a solution once I construct it?

iii. Does this simple, slow solution run in polynomial or exponential

time? Is my problem small enough that this brute-force solution

will suffice?

iv. Am I certain that my problem is sufficiently well defined to actually

*have* a correct solution?

(b) Can I solve my problem by repeatedly trying some simple rule, like

picking the biggest item first? The smallest item first? A random item

first?

i. If so, on what types of inputs does this heuristic work well? Do these

correspond to the data that might arise in my application?

ii. On what types of inputs does this heuristic work badly? If no such

examples can be found, can I show that it always works well?

iii. How fast does my heuristic come up with an answer? Does it have

a simple implementation?

3. Is my problem in the catalog of algorithmic problems in the back of this

book?

(a) What is known about the problem? Is there an implementation available

that I can use?

(b) Did I look in the right place for my problem? Did I browse through all

pictures? Did I look in the index under all possible keywords?

(c) Are there relevant resources available on the World Wide Web? Did I do

a Google web and Google Scholar search? Did I go to the page associated

with this book, *http://www.cs.sunysb.edu/∼algorith*?

4. Are there special cases of the problem that I know how to solve?

(a) Can I solve the problem efficiently when I ignore some of the input

parameters?

(b) Does the problem become easier to solve when I set some of the input

parameters to trivial values, such as 0 or 1?

(c) Can I simplify the problem to the point where I *can* solve it efficiently?

(d) Why can’t this special-case algorithm be generalized to a wider class of

inputs?

(e) Is my problem a special case of a more general problem in the catalog?

5. Which of the standard algorithm design paradigms are most relevant to my

problem?

(a) Is there a set of items that can be sorted by size or some key? Does this

sorted order make it easier to find the answer?

(b) Is there a way to split the problem in two smaller problems, perhaps

by doing a binary search? How about partitioning the elements into big

and small, or left and right? Does this suggest a divide-and-conquer

algorithm?

(c) Do the input objects or desired solution have a natural left-to-right

order, such as characters in a string, elements of a permutation, or leaves

of a tree? Can I use dynamic programming to exploit this order?

(d) Are there certain operations being done repeatedly, such as searching, or

finding the largest/smallest element? Can I use a data structure to speed

up these queries? What about a dictionary/hash table or a heap/priority

queue?

(e) Can I use random sampling to select which object to pick next? What

about constructing many random configurations and picking the best

one? Can I use some kind of directed randomness like simulated annealing

to zoom in on the best solution?

(f) Can I formulate my problem as a linear program? How about an integer

program?

(g) Does my problem seem something like satisfiability, the traveling salesman

problem, or some other NP-complete problem? Might the problem

be NP-complete and thus not have an efficient algorithm? Is it in the

problem list in the back of Garey and Johnson [GJ79]?

# A Catalog of Algorithmic Problems

## Data Structures:

When choosing a container:

1. What’s the size?
2. Relative frequency of insert, delete and search?
3. The access pattern for keys? Uniform, random, or skewed?
4. Strict response time?

For hash table:

1. How to deal with collision: chaining or open addressing?
2. How big should the table be?
3. What hash function should be?

Note, in substring match, hash codes of successive m-character windows of a string can be computed in constant time instead of O(m), if we choose a proper hash function, such as:



For Binary search tree:

1. If the size is big and insertion of data is not randomly distributed, it must be balanced.
2. red-black tree has guaranteed response time, where splay tree has amortized time cost

For B-trees

1. For data sets so large that they will not fit in main memory (say more than 1,000,000 items) your best bet will be some flavor of a B-tree. Once a data structure has to be stored outside of main memory, the search time grows by several orders of magnitude.
2. The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed

For Skip lists

1. O(logn) search time
2. easy to analyze and implement than balanced trees