# 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)

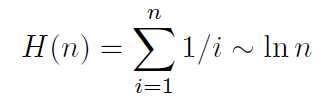
*Problem:* Matrix Multiplication

*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:



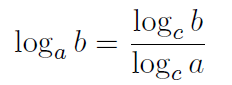
Complexity of Quicksort is the summation

Employing the Harmonic number identity immediately

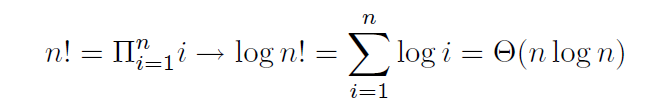
reduces this to Θ(*n* log *n*)

*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

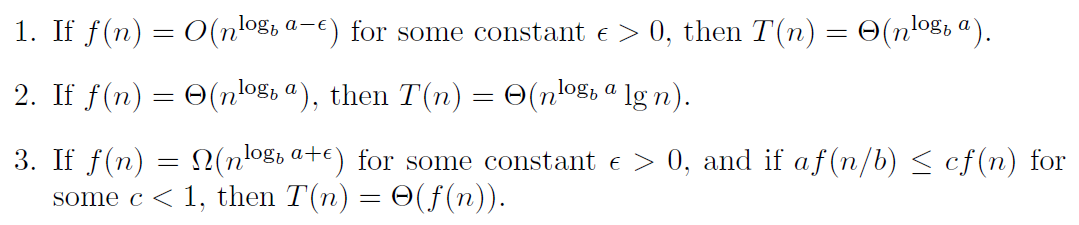
The Fibonacci numbers are described by the recurrence relation *Fn* = *Fn−*1+*Fn−*2

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

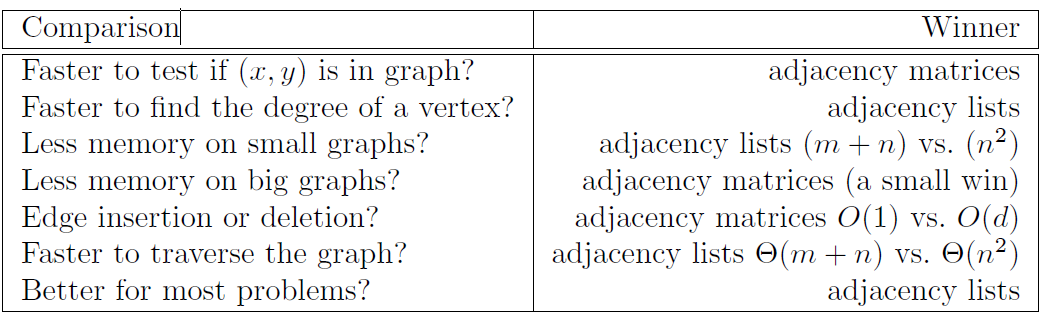
*Take-Home Lesson:* Graphs can be used to model a wide variety of structures

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*.

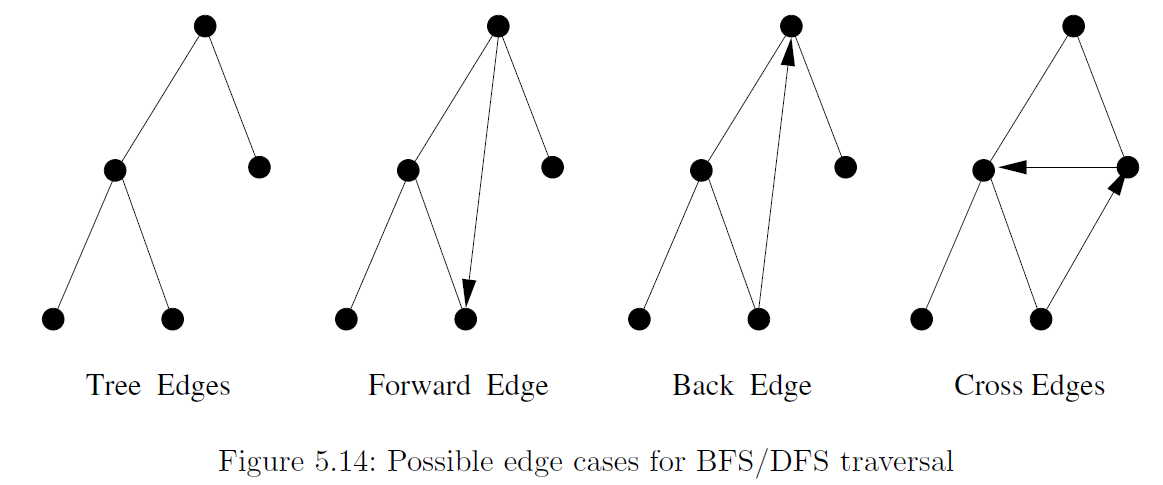
For directed graphs, depth-first search labeling can take on a wider range of

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