Notes of Stanford CS 161: Algorithm
Eric Lu

Topic I: Divide and Conquer

Algorithmic Analysis

To solve a problem, we need to:

1. Design an algorithm
2. Prove its correctness (usually by induction, including base case, inductive case and termination)
3. Prove its complexity (usually runtime, sometimes extra space cost or something else)

Complexity class notations and proof

Big-O:

Let f, g: N -> N, f(n) = O(g(n)) iff there exists n_0 in N, c in R, such that for all n in N, f(n) <= c*g(n) for all n >= n_0.

For algorithmic analysis, we usually care about the un-tight upper bound O(g(n)), for worst-case analysis. There are also Omega(n) for un-tight lower bound (for best-case analysis), omega(n) for tight lower bound, o(n) for tight upper bound, and Theta(n) for both lower and upper bound.

To prove f(n) = O(g(n)), find c and n_0.
To disprove f(n) = O(g(n)), use the definition and prove by contradiction.

Insertion Sort

Algorithm: To sort list A, start from sub-list A[0: 0]. For i in 0 ... n -1, iterate through A[0: i], insert the next element into the previously sorted sub-list.

Proof of correctness:

1. Base case: A[0: 0] (first element only) is vacuously sorted
2. Inductive case: Invariant: at the beginning of iteration i, sub-list A[0: i-1] is sorted. Thus inserting A[i] makes the sub-list A[0: i] sorted.
3. Termination: Since i is arbitrary between 0 and n-1, the algorithm returns correctly sorted A at the end of iteration n.

Proof of runtime:

Double-nested loops, each has at most n iterations, so runtime is in $O(n^2)$.

Divide and Conquer (D&C)

To solve a problem, we take this approach: we first divide the problem into smaller ones, solve each sub-problem, then merge the solutions together. We need to design how to divide, how to conquer, and how to merge. Usually we use recursion.

To prove the runtime of a D&C algorithm, we need to find:

1. How many levels of sub-problem are there
2. How many sub-problem at each level
3. How much time each sub-problem costs

Merge Sort

To sort an array A, we recursively sort each half. The key is how do we merge the sorted two halves: compare the heads, and take the smaller one as the next element.

Proof of correctness:

1. Base case: when a sub-problem consists of only one element, it is vacuously sorted.
2. Inductive case: when merging, the invariant is that each sub-array is sorted. Thus our method of merging correctly returns a sorted array.
3. Termination: since the merging returns a sorted array at arbitrary levels, it returns the correct sorted array at the top level.

Proof of runtime:

At arbitrary level t from the top, we have 2^t sub-problems. Each sub-problem costs time $O(n/2^t)$ to merge. Thus, each level costs O(n) time. There are at most log(n) levels, i.e. t = 0 ... ceil(log(n)). Hence, the runtime is O(nlog(n)).

Integer Multiplication

To multiply two integers each with length n, the brute force algorithm is to multiply digit-to-digit and sum all up. Runtime: $O(n^2)$.

We can decompose the two integers to reduce the total times of multiplication.

Karatsuba's Algorithm:

$$j \times k = (a + b)(c + d) - ac - bd$$

For $j = a \times 10^{n/2} + b, k = c \times 10^{n/2} + d$. The total times of multiplication is reduced.

Proof of correctness: the equation above is correct for all integers j and k.

Proof of runtime: $O(n^{\log[2]3}) = O(n^{1.585})$, see master method below.

Solving recurrences

To solve the runtime of a D&C problem, we usually can easily write the relationship between runtime at a level and its sub-level, i.e.: T(n) = f(T(n/2)) given the partition factor is 2.

We can then apply one of these methods:

1. Draw a recursion tree. List all the runtime of sub-problems by drawing out a tree. Do the summation.
2. Substitution method. Let t be an arbitrary level, find $T(n) = f(T(n/2^t))$ for generic t. Let t be its maximal value, i.e. log(n) and substitute it back into f.
3. Master method. Use the following equations:
   Given $T(n) = aT(n/b) + O(n^d)$:

$$T(n) = \begin{cases} O(n^d \log(n)) \text{ if } a = b^d \\ O(n^d) \text{ if } a < b^d \\ O\left(n^{\log_b(a)}\right) \text{ if } a > b^d \end{cases}$$

The proof of master method is drawing the recursion tree with parameters a, b and d and solve the recursion for the 3 cases algebraically.
However, master method only solves when sub-problems are in the equal size. It cannot solve unequal sizes, e.g. quick sort.

Select-k

To select the kth smallest element in an unsorted list A, given k <= n = |A|, we can choose a pivot (an element in the array), partition around the pivot (make all elements in one side is less than pivot, in the other side is larger), recursively.
If the pivot is random, runtime: $O(n^2)$ for always select the smallest or largest.

Algorithm: Smartly choose pivot by:

1. Split A into m groups
2. Mark the median of each group as candidate pivots
3. The real pivot is the median of candidates, i.e. median of medians

Proof of correctness:
Each time we select a pivot, we know its order in A. If order > k, search on the larger half; if order < k, the smaller half, until order = k. Since each time at least one element can be abandoned (the pivot plus the ignored half), we can always return the correct answer.

Proof of runtime:
Elements in the first half of the groups, with median smaller than or equal to pivot, are guaranteed to be smaller than the pivot. Symmetrically, elements in the second of the groups with median larger than or equal to pivot are guaranteed larger. Hence, we can treat the pivot approximately as the median of A.
By writing out the recurrence and using substitution method, runtime = O(n).

Lower bound of sorting algorithms

For comparison-based sort, the lower bound of runtime is Omega(nlog(n)).
Proof:
For any comparison-based algorithm, it corresponds to a decision tree. For input size n, we can have n! leaves, each leaf corresponding to a permutation. The runtime for a run must be the length of one of all the paths from root to a leaf. The worst-case is at least Omega(length of the longest path).
The shallowest tree is the balanced-tree, where the length of the longest path is log(n!) = nlog(n) using Sterling's approximation. Hence, the lower bound is Omega(nlog(n)).

Linear-time Sorting

Counting sort:

Given the range of elements is known, we count each element by increment the corresponding bucket by 1. Then we can print out the sorted list.

Runtime: O(n+k), given k is the number of distinct elements. Hence, not useful when k is non-polynomial of n.

Bucket sort (generalized counting sort):

Instead of putting one element in a bucket in counting sort, we put a given number of elements in a bucket and sort each bucket using some other sorting algorithm.

Runtime: O(n+k)

Radix sort:

For elements with the same length (or can be concatenate to be the same length, e.g. 5 to 0005) and each digit is comparable, we can iterate through its least significant bit to the most significant bit, using bucket sort on each bit.

Proof of correctness: Invariant: when sorting the ith digit, all less significant digits are sorted. Thus after sorting the ith digit, all digit from the least significant to i are sorted.

Proof of runtime: Based on the runtime of bucket sort, the runtime is O(d(n+k)), given d is the digit.

Stability of sorting

A sorting algorithm is stable iff all the elements with the same content will be in the sequence the same as in the input array, e.g.: sort {5, 5}, if the algorithm is stable, the first 5 is guaranteed to come before the second 5 in the output.

Binary Search Tree (BST) Operations

Search/Select: Return the key if exists, return the leaf that will be the predecessor or the next successor if not exist.

Runtime: O(log(n)) if balanced, O(n) otherwise

Insert: First search, return if exists, add child if not exist.

Runtime: O(log(n)) if balanced, O(n) otherwise

Delete: First search, if exists, there are 3 cases:

1. If the key is a leaf, just delete
2. If the key has one child, move the child up
3. If the key has two children, replace the key with the child that can be the predecessor of the other.

Runtime: O(log(n)) if balanced, O(n) otherwise

Compare BST with ordered linear data structures

	Sorted linked list	Sorted array	BST

Search	O(n) from head	O(log(n)) binary search	O(log(n)) if balanced, O(n) otherwise
Select		O(1) with index	
Insert	O(1) given the pointer is already there	O(n) to copy the elements	
Delete			

Red-Black Tree (RBT)

It is a balanced BST to guarantee O(log(n)) runtime of all four operations. After an insertion/deletion, we fix the tree aiming at maintain the five properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL leaf, the two leaves below the deepest nodes with keys) is black
4. If a node is red, then both children are black
5. All paths from a node to all descendant leaves have the same number of black nodes

Topic II: Randomized Algorithms

Randomized Algorithms

A randomized algorithm contains some random procedure. Note: a procedure is random means it follows discrete uniform probability distribution.
We aim at getting the correct answer with acceptable high probability and an efficient expected runtime.
Generally, randomized algorithms have two categories:

1. Las Vegas algorithms: guaranteed correctness, not runtime
2. Monte Carlo algorithms: guaranteed runtime, not correctness

To prove the runtime of an expected algorithm:

1. Based on the algorithm, find the random variable X and its distribution
2. Convert E[T] as a function of E[X] and solve for E[T]

Quick sort with random pivots

To sort a list A, we randomly choose a pivot, partition around, recursively on each half, until the sub-lists have size 1.

Proof of runtime:
The worst case runtime is $O(n^2)$, given we may pick up the largest or smallest element.
To find the expected runtime, let $X_{a,b} = 1$ if elements a, b are compared, 0 if not.
The runtime is the number of comparisons. Two elements are compared iff one of them is chosen as the pivot before something between them is chosen, e.g: $X_{6,10} = 1$ iff 6 or 10 is chosen as pivot before 7, 8 or 9 is chosen.
Since the choosing is random, it follows discrete uniform distribution:

$$P[X_{a,b} = 1] = \frac{2}{b - a + 1}$$

Thus:

$$E[T] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}]$$

Which is a harmonic series, so E[T] = O(nlog(n))

To improve the algorithm, we can run select-k on the list each time to choose the guaranteed (not expected) median in O(n) time. Hence the guaranteed runtime is O(nlog(n)) for log(n) levels.

Quick select

It is the same as select-k, but instead of choosing the pivot using median of medians, it randomly picks a pivot.

Proof of runtime:
The worst case is $O(n^2)$ if the pivot picked is the largest or smallest.

Define one phase means the select space is shrunken to <= 75%. This definition is because choosing a pivot in the middle 50% guarantees a phase. Thus, at phase k, the sub-list has length <= $n(3/4)^k$. Thus, we need k = ceil($\log_{4/3}n$) phases.
Now we need to find the runtime of a generic phase k. Let X_k be the number of recursive calls in phase k. We have a recursive call iff we fail to get the middle 50%. Hence, X_k follows geometric distribution with probability 50%, so $E[X_k] = 1/(50\%) = 2$. Thus:

$$\begin{aligned} E[T] &= E[\sum X_k \times length\ of\ list\ at\ phase\ k] \\ &= \sum_{k=0}^{\lceil log_{4/3}k \rceil} E[X_k] n(\frac{3}{4})^k \\ &= n \sum_{k=0}^{\lceil log_{4/3}k \rceil} E[X_k](\frac{3}{4})^k \\ &\leq n \sum_{k=0}^{\infty} E[X_k]\left(\frac{3}{4}\right)^k \\ &= n \sum_{k=0}^{\infty} 2(\frac{3}{4})^k \end{aligned}$$

By calculating the upper bound (trick: all convergent series are upper bounded by summation to infinity), E[T] = O(n).

Majority Element
Given a majority element exists (occurs at least ceil(n/2) times) in list A, find that element.

D&C approach:
Recursively call on each sub-list, return the majority elements m1 and m2 from each sub-list. To merge the results, count m1 again on A. If it occurs more than half of times, return m1, otherwise return m2.
Runtime: O(nlog(n))

Randomized approach:
Randomly choose an element, count it on A, return if it is majority.
Worst case runtime: O(∞)
Expected runtime: O(n). Since the total times of picking is a geometric random variable with probability >= 1/2, we expect to pick at most 2 times. The counting costs O(n), so O(n) in total.

Hashing
Given a hash function h: U -> {1...n}, we want O(1) expected runtime, O(n) worst-case runtime for search, insert and delete. We want to lower the probability

of collision as much as possible.

Without randomness: Separate chaining
Each bucket is an unsorted linked list. By pigeon hole principle, at least one bucket has |U|/n items.
Worst case: everything is hashed to one bucket, and need O(|U|) of runtime. Given that usually |U| >> n, it is not preferred.

With randomness: Universal hashing
Randomly choose hash function h from a family H. If H is an exhaustive set (includes all possible h), we can have minimized number of collisions by having O(1) keys hashing to the same bucket as arbitrary key x.
Proof:

$$\mathrm{E}[\text{number of elements in } x\text{'s bucket}]$$

$$= 1 + \sum_{y \neq x} P[h(x) = h(y)]$$

$$= 1 + \sum_{y \neq x} \frac{1}{n}$$

The 1 added is element x itself. If H is exhaustive, the value <= 2, which is O(1).

Universal hash family
The exhaustive hash function set H is huge, for $|H| = n^{|U|}$. We can have smaller hash function sets that satisfies:

$$\mathrm{P}[\mathrm{h}(\mathrm{x}) = \mathrm{h}(\mathrm{y})] \leq \frac{1}{n}$$

Where h is a randomly drawn hash function from the set. Such a hash function set H is called a universal hash family.
There are several typically used universal hash families, such as:

1. Bit representation of exhaustive set: use bits to represent each h, with size |U|log(n)
2. Given U is the integer set {0 ... |U|-1}, we can have:
   $h_{a,\,b}(x) = ax + b \bmod p \bmod n$
   for some prime p >= |U|
3. Given U is a set of bit strings, let A be a log(|U|) by log(n) matrix randomly filled by 0's and 1's:
   $h_A(x) = Ax$

Topic III: Graph Algorithms

Graph Representations

To represent a graph, we can have either adjacency matrix or adjacency list.
Adjacency matrix: rows: sources, columns: destinations, if directed. Each entry has value of the weight. Boolean weights for un-weighted graph.
Adjacency list: a list of vertices, each links to its outgoing vertices with weight recorded

Compare some complexities of the two representations on some basic graph algorithms:

	Matrix	List
Edge membership time	O(1) by checking entry	O(deg(u)) or O(deg(v)) by iterating through all outgoing edges from u or v
Neighbor query time	O(|V|) by iterating a row	O(deg(v)), same as edge membership
Space requirement	$O(\|V\|^2)$	O(|V|+|E|)

Generally, the matrix representation is better for dense graphs, and list better for sparse graphs.

Depth-first Search (DFS)

To traverse a graph (visit all vertices and edges), for an arbitrary vertex, recursively do DFS on all outgoing neighbors. A vertex is visited once it is met, and fully explored after all its outgoing neighbors are fully explored (vacuously fully explored as soon as a vertex with no outgoing neighbor is visited).

Proof of runtime:

$$T = \sum_{v \in V} (1 + N_{out}(v)) = O(|V| + |E|)$$

Find Topological Ordering

It is an application of DFS.
Assume G is a directed acyclic graph (DAG), we need to find one linear order that satisfies the partial order of G. We assign each vertex an enter time when first met in DFS. At every step, current time is increment by 1. The order of enter time is a topological ordering.

Breadth-first Search (BFS)

Similar to DFS, but instead of mark one vertex is searched when it is fully explored, we mark all vertices within the same number of steps as searched.
Proof of runtime:
Same as DFS, T = O(|V|+|E|)

Verify Bipartiteness

It is an application of BFS.

A graph is a bipartite iff it can be 2-colored, i.e. color the vertices in 2 colors such that no edges are among vertices of the same color. We need to find if a graph is a bipartite.

Call BFS on a graph, color vertices reached in odd steps one color, and in even steps another. If doable, return true.

Dijkstra's Algorithm

To solve a single source shortest path problem (SSSP), i.e. given a vertex u in a positively weighted graph G, find the shortest path from u to another arbitrary vertex.

1. Initialize the distance list, with d[u] = 0 and others infinity
2. Calculate the tentative distance of each neighbor v:
$$d[v] = \min\{d[v], d[u]+w(u, v)\}$$
3. Mark u as visited, select the unvisited vertex with minimal distance, repeat until all nodes are visited (or the destination is visited, given the destination).

Proof of correctness:

For all vertex v, a tentative distance d[v] is always greater than or equal to its final distance based on the minimum function. When v is marked as visited, since all weights are positive, d[v] cannot be minimized anymore. Consequently, the algorithm returns the correct shortest distance from u to all other v.

Notice that if G has negatively weighted edges, there might exist a shorter path coming from a visited vertex, and hence Dijkstra will fail.

Proof of runtime:

When selecting the unvisited vertex with minimal d[v], we can implement it using a binary min-heap. The extraction from min-heap takes O(1) of time while insertion takes O(log(|V|)). There are |E| + |V| modification of all d[v]'s during the entire algorithm, so |E| + |V| insertions into the min-heap. Hence the runtime is O((|E|+|V|)log(|V|)) = O(|E|log(|V|)).

By using a Fibonacci heap instead of binary min-heap, the runtime can be improved to O(|E| + |V|log(|V|)).

Strongly connected component (SCC)

A directed graph is strongly connected iff for all pairs of vertices u and v, there exist a path from u to v and a path from v to u.

A graph can be decomposed into strongly connected components, for each sub-graph is an SCC. A single vertex is a vacuous SCC.

Kosaraju's Algorithm

To find all SCC's in a directed graph G, call DFS on G, and record all vertices'

start time (time when visited) and end time (time when fully explored) during DFS. Reverse all edges in G to form the inverse G^T. Choose the unvisited vertex u with the largest end time, call DFS/BFS starting from u and all reachable vertices are in the same SCC as u. Repeat until all vertices are visited.

Proof of correctness:
In the first DFS, for a vertex u and all v with earlier end time than u, there exists paths from u to v in G. Hence in G', if u still has path to v, both paths exist and they are in the same SCC, otherwise they are not in the same SCC.

Proof of runtime:
Each DFS takes time O(|V| + |E|) and reversing edges takes O(|V| + |E|). The total runtime is O(|V| + |E|).

Cut
A cut is a partition of the vertex set V into two non-empty parts. Edges going from one part to another crosses the cut.
A global min cut is a cut with the minimal crossing edges.

Karger's Algorithm
To find a global min cut, randomly pick an edge and contract the two vertices of the edge into one super vertex, affecting all edges connecting to the two vertices. Repeat until two vertices and one edge left, and the min cut is given by this partition, with the final super edge contains all crossing edges.

This algorithm is a Monte Carlo randomized algorithm, with no guarantee in correctness, but a fast runtime.

Proof of probability of correctness:
If a randomly selected and contracted edge is a crossing edge, Karger returns an incorrect answer.
Let |E| = m, given there are k crossing edges in the min cut, and we contract edges $\{e_1, e_2, ..., e_{m-k}\}$:

$$\begin{aligned}
&\mathrm{P}[\text{Karger is correct}] \\
&\quad = \mathrm{P}[e_1 \text{ doesn't cross}] \\
&\quad \times \mathrm{P}[e_2 \text{ doesn't cross} | \text{the previous don't cross}] \times \dots \\
&\quad \times \mathrm{P}[e_{m-k} \text{ doesn't cross} | \text{the previous don't cross}] \\
&\quad \geq \frac{m-k}{m} \times \frac{m-k-1}{m-1} \times \dots \times \frac{1}{k+1} \\
&\quad = \frac{1}{\binom{m}{k}}
\end{aligned}$$

Thus, to find min cut with probability p, repeat Karger $\binom{m}{k}\ln(\frac{1}{1-p})$ times.

Proof of runtime:

For each time running Karger, we cost $O(|V|^2)$ of time based on the algorithm. To guarantee the probability p, we run $\binom{m}{k} \ln\left(\frac{1}{1-p}\right) = O(m^k) = O(|V|^{2k})$times. The total runtime is $O(|V|^{2+2k})$, a polynomial of |V|, which a great improvement from brute force enumeration $O(2^{|V|})$.

Topic IV: Greedy Algorithm

Greedy Algorithm

A greedy algorithm constructs solutions one step at a time, at each step choosing the local optimum. Its advantage is simple to design and often efficient in runtime, but usually difficult to verify correctness or global optimality.

To prove the correctness or global optimality of a greedy algorithm, we have two methods:

1. Greedy stays ahead

Let the greedy algorithm's partial solution at an arbitrary step be A, and let O be an optimal solution at that step. We prove by induction that A is at least as good as O, by using some measurement in the algorithm, e.g. the total time cost after i activities, the distance travelled after i hops by the frog.

2. Greedy swapping argument

If we swap an optimal solution out for the greedy solution, need to prove that the solution is still optimal. We prove by induction that if A and O are the same from step 0 to i, if we append A[i+1] after O[i], the new solution is still optimal. Still, we need some measurement for the proof.

Frog Hopping

A frog can hop at most r steps in a time, starting from 0 and hoping to reach n. It can land on a set of given positions. Given such a path exist, we find the minimal number of hops by choosing the furthest position that can be landed each time.

Proof of correctness:

Legality: The algorithms find a legal path of hopping, since every time the frog always moves forward within the range and lands on a legal position.

Optimality:

1. Greedy stays ahead:

Let H be the series of landed positions be generated from the greedy algorithm, H* be an optimal solution. Need to prove H is at least as good as H*. We use measurement of position p(i, H) and p(i, H*) after i hops, i.e. p(i, H) >= p(i, H*).

Base case: Before step 1, p(0, H) = p(0, H*) = 0, vacuously holds.

Inductive case: Assume p(i, H) >= p(i, H*), at step i+1, the algorithm chooses the furthest feasible position, so p(i+1, H) >= p(i+1, H*).

Termination: Based on the induction, with the same number of steps, H goes at least as far as H*. Since H* is optimal, for any i and H, p(i, H) <= p(i, H*), so p(i, H) = p(i, H*) for all i. Thus H is optimal.

2. Greedy swapping argument:

Assume we append H[i+1] to H*[i] to form H**[i+1], we need to prove H** is optimal.

Base case: Before step 1, p(0, H) = p(0, H*) = 0. We append H[1] to H*[0], that

is, the furthest position can go in one step, which is optimal at step 1.
Inductive case: Assume p(i, H) = p(i, H*), now we append H[i+1] to H*[i], the new position is the furthest position that can go at step i+1 for H*, which is optimal.
Termination: Since i is chosen arbitrarily in inductive step, H is optimal.

Proof of runtime:
We need to hop at most n times, so O(n) in runtime.

Activity Selection
Given a set of activities with start time and end time, need to choose the largest subset with no overlapping. Sort all activities by ascending end time, then choose the activity with the earliest end time, remove all overlapping activities, and repeat until the set is empty.

Proof of correctness:
Legality: The algorithm finds a legal schedule of activities since all activities selected will not be overlapping.

Optimality:
Use greedy stays ahead proof.
We choose the end time of activity i as the measurement. Need to prove that the greedy algorithm produces solution A and optimal solution A*, such that t(i, A) <= t(i, A*).
Base case: t(0, A) = t(0, A*) = 0
Inductive case: Assume t(i, A) <= t(i, A*). The greedy algorithm picks the (i+1)th activity that must start after the ith activity ends, so does the optimal solution. The greedy algorithm choses the activity satisfies this requirement with the smallest end time, so t(i+1, A) <= t(i+1, A*).
Termination: Since i is picked arbitrarily, t(k, A) <= t(k, A*) given at most k activities can be chosen by the greedy algorithm.
Now we need to prove t(k, A) <= t(k, A*) means |A| = |A*|. Since A* is optimal, k = |A| <= |A*|. Assume by contradiction that k < |A*|, i.e. there is a (k+1)th activity in A*. If such an activity exists, it must starts after the kth activity ends, so we can also select it with the greedy algorithm, contradict to the fact that there are only k activities in A. Consequently, |A| = |A*|.

Minimum Spanning Tree (MST)
A minimum spanning tree is a tree (a subset of edges that forms an undirected acyclic graph) that connects all vertices in a graph, with minimal total weight of edges.
We have two greedy algorithms to find an MST, both based on a lemma:
Given a cut on a graph respects a set of edge A, i.e. no edge in A crosses the cut, if A is a subset of an MST, then there is an MST that contains $A \cup \{(u, v)\}$ where

(u, v) is the light edge, i.e. the crossing edge with minimum weight.

Proof:
Assume for contradiction that the MST contains A but not (u, v). By the property of MST, it picks the set of lightest edges unless adding an edge will cause a cycle. Thus, adding (u, v) forms a cycle. Since (u, v) crosses the cut, there must be some other edge (x, y) in the MST and also in the cycle and crossing the cut. Now, we exchange (u, v) for (x, y). The new tree is also MST because: 1, (x, y) is deleted, no cycle exists, 2, (u, v) is light, so the cost can only be equal or smaller.

Prim's Algorithm
To find an MST, randomly pick vertex s and form the cut $\{\{s\}, V - \{s\}\}$.
Greedily add the light edge to the tree.

Proof of correctness:
See the proof of the lemma above.

Proof of runtime:
With the naive method, for each of the |V| vertices, we need to find the light edge by iterating through |E| edges. The total runtime is O(|V||E|).
If the light edge is found using a priority queue implemented by a balanced tree, e.g.: red-black tree, time is reduced to O(|E|log(|V|)).
If the priority queue is implemented by a Fibonacci heap, the runtime is reduced to O(|E|+|V|log(|V|)).

Kruskal's Algorithm
Maintain a forest of trees by greedily adding the cheapest edge.

Proof of correctness:
Although the trees are separated as a forest, it is still a cut, so the proof of the lemma still applies.

Proof of runtime:
The time cost is sorting the edges by weight in the beginning. If we use comparison based sort, runtime is $O(|E|\log(|E|)) = O(|E|\log(|V|^2)) = O(2|E|\log(|V|)) = O(|E|\log(|V|))$.
If the weights can be radix sorted, we have runtime O(|E|).

Topic V: Dynamic Programming

Dynamic Programming

A dynamic programming algorithm breaks up a problem into small sub-problems, with two properties:

1. Optimal substructure: the optimal solution of a problem can be expressed as a function of the optimal solutions of its sub-problems.
2. Overlapping sub-problems: The sub-problems overlaps a lot, so memoization helps.

There are two approaches to solve a dynamic program:

1. Bottom-up: iterate from the smaller problems and solve them, then go to larger problems and solve them using the results of smaller problems.
2. Top-down: recursively solve larger problems by recursively solving smaller problems.

To prove the correctness of a dynamic programming algorithm, we find the relationship between the solution and solutions of sub-problems, and prove this relationship is correct in arbitrary cases.

Bellman-Ford Algorithm (BF)

To solve an SSSP problem, the same goal as Dijkstra, BF always works with negative edge weights. We maintain a list $d^{(k)}$ of length n = |V|, for each k = 0, 1, ..., n-1 corresponding to the results after step k. The relationship is:

$$d^{(k)}[b] = \min\{d^{(k-1)}[b], \min\{d^{(k-1)}[a] + w(a,b)\}\}$$

Proof of correctness:

To find the shortest path to vertex b in step i+1, there are only two cases:

1. The shortest path after step i+1 includes the same vertices as after step i.
2. The shortest path grows from some other vertex a.

The two inputs of the minimum function are respectively the minimal distance of the two cases. Hence the relationship is correct, and so is the algorithm.

Proof of runtime:

We need to go through O(|E|) edges in one step, and there are |V| steps in total. Hence, the runtime is O(|V||E|), slower than Dijkstra's O(|E| + |V|log(|V|)).

BF can detect negative cycle crossing s. If $d^{(|V|)}[s] < 0$, then a negative cycle exists.

Floyd-Warshall Algorithm (FW)

To solve an all-pairs-shortest-path (APSP) problem, we setup an n by n matrix D and update $D^{(k)}$ to store the shortest path from u to v with k = 0 ... |V|-1 steps into $D^{(k)}[u][v]$, with the relationship:

$$D^{(k)}[u][v] = \min\{D^{(k-1)}[u][v], D^{(k-1)}[u][k-1] + D^{(k-1)}[k-1][v]\}$$

Where arbitrary k is the vertex newly included in iteration k.

Proof of correctness:
Similar to BF, the relationship considers two cases:
1. In iteration k, the shortest path from u to v consists of the same set of vertices.
2. In iteration k, the shortest path from u to v includes the newly introduced vertex.

Hence the relationship is correct.

Proof of runtime:
We need to iterate through the |V| by |V| matrix for |V| steps, so the runtime is $O(|V|^3)$.

FW can detect negative cycles, by checking $D^{(|V|)}[v][v]$ for all v. If the value < 0, a negative cycle exists.

Comparison of graph algorithms for shortest paths

	Dijkstra	BF	FW
Problem	SSSP	SSSP	APSP
Runtime	O(|E|+|V|log(|V|)) using Fibonacci heap	O(|V||E|)	$O(\|V\|^3)$
Strengths	-	Works on graphs with negatively weighted edges, and can detect negative cycles.	
Weaknesses	Might not work on graph with negatively weighted edges	-	-

Longest Common Subsequence (LCS)
A subsequence is a sequence formed by deleting some elements from a sequence. A common subsequence is a subsequence for two sequences. To find the LCS of two sequences, say two strings X and Y, let T(i, j) be the length of the LCS for sub-strings X[0: i] and Y[0: j]. The relationship is:

$$T(i,j) = \max\{1 + T(i-1, j-1), \max\{T(i-1, j), T(i, j-1)\}\}$$

Proof of correctness:
There are two cases when calculating T(i, j):
1. X[i] = Y[j]
2. X[i] != Y[j]

The relationship takes the maximum of the two cases, so it is correct.

Proof of runtime:
We need to iterate through the matrix T, which is |X| by |Y|, so the runtime is

O(|X||Y|).

Unbounded Knapsack

Given a set of items, each with a positive integer weight and value, we need to put the maximal value into a knapsack of capacity W.

Initialize a list K sized W, where K[x] is the maximal value that we can take with weight x. We have the relationship:

$$\mathrm{K[x]} = \begin{cases} 0 \text{ if there are no i such that } w_i \leq x \\ \max\{K[x - w_i] + v_i\} \text{otherwise} \end{cases}$$

We can retrieve the answer by looking at K[W].

Proof of correctness:

The relationship considers two cases:

1. We are unable to rearrange the items since the capacity is smaller than any weight of items
2. We are able to rearrange the item as the capacity grows

Hence the relationship is correct and so is the algorithm.

Proof of runtime:

At each iteration in K, we need to find the maximum by going through O(n) weights. There are W iterations in total. Hence the runtime is O(nW).

0/1 Knapsack

For unbounded knapsack, we can take at most unlimited number of a distinct item. Here in 0/1 knapsack, we have only one item for each. Thus we can only decide whether taking the item or not.

Initialize a matrix K sized W by n, where K[x][j] is the maximal value we can take with capacity x and j items. We have relationship:

$$\mathrm{K[x][j]} = \begin{cases} 0 \text{ if x or j} = 0 \\ \max\{K[x][j-1], K[x - w_j][j-1] + v_j\} \text{otherwise} \end{cases}$$

Proof of correctness:

Similar to the proof of unbounded knapsack, the relationship considers both cases.

Proof of runtime:

We iterate through a W by n matrix, each entry costs O(1) of time. Hence the total runtime is O(nW).

Maximal Independent Set in a Tree

In a graph, an independent set is a subset of vertices that no pairs of vertices in this set shares an edge. All vertices are weighted in order to find the maximal independent set.

To find the maximal independent set on an arbitrary graph can be NP-hard, but we can find one in a tree efficiently.
Let A[u] be the weight of maximal independent set in the sub-tree rooted at u.
We have the relationship:

$$A[u] = \max\{w(u) + \sum_{v \in u.grandchildren} A[v], \sum_{v \in u.children} A[v]\}$$

Proof of correctness:
To have an independent set, no vertices in the set share an edge, so there are two cases for arbitrary vertex u:

1. Vertex u is in the set, then all its children must not be included, but grandchildren must all be included.
2. Vertex u is not in the set, then all its children must be included.

Thus the relationship is correct, and so is the algorithm.

Proof of runtime:
We need to traverse through the tree from leaves, which has size |V|, with each iteration O(1) of time. Hence the total runtime is O(|V|).

Topic VI: Other Topics

Intractable problems and NP-completeness

A problem is tractable iff there exists an efficient algorithm, i.e. polynomial time that solves it.

P: the class of all problems that can be solved in polynomial time

NP: the class of all problems that can the answers can be verified in polynomial time

NP-complete: the class of all problems that can be reduced form NP in polynomial time

NP-hard: the class of all problems that are at least as hard as NP

Traveling Salesperson Problem (TSP)

In an undirected weighted graph, we need to find the Hamiltonian cycle (a simple cycle that visits all vertices) with the lowest cost. This problem is known for being NP-hard.

Brute force:

Enumerate all cycles.

Proof of runtime:

A graph has (n-1)!/2 cycles, each costs O(n) time, so the runtime is O(n!), NP-hard.

OPT algorithm:

Given OPT(v, S) is the minimal cost of a path from s to v exactly through vertex set S, v is in S, we have relationship:

$$\mathrm{OPT(v,S)} = \begin{cases} 0 \text{ if } \mathrm{v} = \mathrm{s} \text{ and } \mathrm{S} = \{\mathrm{s}\} \\ \infty \text{ if s is not in S} \\ \min\limits_{u \in S-\{v\}} \{OPT(u, S-\{v\}) + w(u,v)\} \text{ otherwise} \end{cases}$$

Proof of runtime:

S is a subset of V, and we need to iterate through all $2^{|V|} = 2^n$ such subsets. For each S, we need to go through the matrix representation of the graph to retrieve all w(u, v) by a double nested loop. Hence, the runtime is $O(2^n n^2)$, still NP-hard, but improves a lot from brute force.

Parameterized Complexity

Sometimes the complexity depends on parameters other than the input size n, e.g.: the knapsack problem has complexity O(nW), and whether the problem is NP-hard depends on if W is in polynomial or non-polynomial of n.

Hence, we define a problem is fixed-parameter tractable iff there exists an algorithm solves it in O(f(k)p(n)) time, where p(n) is a polynomial of n and f(k) is an arbitrary function of other parameters. That is, if f(k) is controlled as a constant, we have an efficient algorithm to solve the problem.

Vertex Cover

Given an undirected graph with weighted vertices, a vertex cover is a vertex set V' such that all edges are incident to at least one vertex in V'.
Special case: all vertices are weighted 1 (unit weight), called the cardinality vertex cover problem.

Some terminologies needed:
Matching: a set of edges such that no two edges share a common vertex
Maximal matching: a matching, such that if any edge is added, the set is no longer a matching
Maximum matching: the maximal matching that contains the largest possible number of edges

Minimal Vertex Cover for Cardinality Problem

Now we need to find the minimal vertex cover with unit weights.

Greedy approach:
Greedily pick an edge, add its endpoints to the vertex cover, and then remove edges adjacent to the endpoints. Repeat until no vertex is left.

This algorithm has no guarantee of correctness, but is a factor-2 approximation algorithm.
Proof:
Let OPT be the optimal (minimal) cardinality vertex cover in G. Since any vertex cover has to pick at least one endpoint of each matched edge, the size of any maximal matching M provides a lower bound for OPT, i.e.: |M| <= OPT. The algorithm picks a cover that is all endpoints of a maximal matching, which is 2|M|. Thus, |V'| = 2|M| <= 2OPT.

Is there any other lower bounding method to improve this approximation? This is still an open problem.

Minimal Set Cover

Given a universe U of n elements, and a subset collection $S = \{S_1, ..., S_k\}$ of U, each subset associated with a cost. Find the min cost sub-collection that covers all elements of U.

Greedy approach:
Iteratively pick the most uncovered elements from the most "cost-effective" (lowest cost per element) set, remove the covered elements until all elements are covered.

This algorithm has no guarantee in correctness, but is an approximation with guaranteed factor log(n).

Proof:
Index each element in U from e1 to en. When ek is covered, it is from the S_i – C, where S_i is the most cost-effective set and C is the set of currently covered elements.
In any iteration, the remaining sets of the optimal solution can cover the remaining elements at a cost of at most OPT. Thus, since there are |U - C| <= n – k + 1 remaining elements when ek is covered, the average cost of each remaining element <= OPT/(n-k+1). Since ek is from the most cost-effective set, its price must <= OPT/(n-k+1).
Thus, we do the summation of all OPT/(n-k+1) for k from 1 to n, the result <= log(n)*OPT by algebra.

Proof of runtime:
If the cost and subset pairs are implemented using a dictionary, we need to get the most cost-effective set in O(|S|) of time in an iteration, and there are O(|S|) iterations. Hence the runtime is $O(|S|^2)$.
This runtime can be improved if the maximum is retrieved from better data structures, e.g. a max heap.

Approximation schemes
Let P be an NP-hard optimization problem, f_p describe the value to be optimized, e.g.: The total value in the knapsack in the knapsack problem.
Let A(i, epsilon) be an algorithm where i is an instance of P and epsilon > 0 is an error parameter.

A is an approximation scheme if it outputs a solution s such that:
1. $f_p(s)$ <= (1+epsilon)*OPT if P is a minimization problem
2. $f_p(s)$ >= (1-epsilon)*OPT if P is a maximization problem

A is a fully polynomial time approximation scheme (FPTAS) if for each fixed epsilon > 0, its running time is bounded by a polynomial in the size of instance i and 1/epsilon.

Pseudo-polynomial runtime
Pseudo-polynomial runtime means the runtime is polynomial in the numeric value of the input size, but exponential in the length of the bit representation of input, (approximately log(n)).
To be considered weakly efficient, an algorithm must have pseudo-polynomial runtime of input size n.
e.g.: the 0/1 knapsack problem's DP algorithm is pseudo-polynomial, for it has runtime O(nW). Since W is represented as bit string in computation, we actually needs log(W) bits to represent it. However, the computation uses the factor W instead of the actual input size log(W). The runtime is more accurately to be described as $O(n*2^{\text{bits of W}})$, which is exponential in length of the bit string.

Flow

Given a directed weighted graph G, with each edge (u, v) has a positive real number capacity c(u, v). The flow follows:

1. Capacity constraint: 0 <= f(u, v) <= c(u, v)
2. Flow conservation constraint: $\sum_{x \in N_{in}(v)} f(x,v) = \sum_{y \in N_{out}(v)} f(v,y)$

Cost of cut

Based on the same graph, let {S, T} be a cut of G and s is in S, t is in T. The cost of the cut (or can be called as the size of the cut) ||S, T|| = the sum of capacity of crossing edges.

Max-Flow Min-Cut Problem

Given a directed graph G, each edge with a capacity, a source s and a destination t, we need to find the maximum flow from s to t given a path exists. The value of flow |f| is defined to be $\sum_{x \in N_{out}(s)} f(s,x)$. Assume s has no incoming neighbors (source) and t has no outgoing neighbors (sink).

For any flow f and any s-t cut of G, we have |f| <= ||S, T||. In particular, the max flow is the value of the min cut. We can prove this algebraically using the two definitions. Thus, we can find the max flow by finding the min cut.

Ford-Fulkerson Max-Flow Algorithm

Assume G does not have both (u, v) and (v, u) in E. If G has such a pair, we split (u, v) into two edges (u, x) and (x, v) by adding a new node x. Hence the graph is equivalent and the new number of vertices is m + n.

Let f be a flow, try to improve the flow, define residual capacity cf: V×V -> R^+ as:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) \text{ if } (u,v) \in E \\ f(v,u) \text{ if } (v,u) \in E \\ 0 \text{ otherwise} \end{cases}$$

Basically, the residual capacity definition tells us that, if flow f is on edge (u, v) with capacity c, we remove f from c (c = c-f) and we add an edge in the opposite direction with the value of the flow. The new edges form a new graph G_f, which is the residual graph.

The algorithm:

Initialize f with all zeroes flow, and G_f = G. We check using DFS that if t is reachable from s in Gf. If reachable, get the path P from s to t and the min capacity F on P. For each flow f, we update it with new flow f' such that:

$$f'(u,v) = \begin{cases} f(u,v) + F \text{ if } (u,v) \in P \\ f(u,v) - F \text{ if } (v,u) \in P \\ f(u,v) \text{ otherwise} \end{cases}$$

Then update G_f to the corresponding new flow.