14. Computational Complexity

The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest.

Jack Edmonds, 1966

Decision Problems

A decision problem is a question (in some formal system) that has a True or False answer. A decision problem is decidable if there is an algorithm that correctly answers all of its instances. Here are some classic decision problems:

1. Sorted: Is the list \(\langle a_0, a_1, \ldots, a_{n-1}\rangle\) of integers sorted? Sorted can be solved in \(O(n)\) time.

2. Reachability: Given two vertices \(u\) and \(v\) in a graph \(G\), is there a path from \(u\) to \(v\)? Reachability can be solved in \(O(n^2)\) time, where \(n\) is the number of nodes in \(G\).

3. 0—1 Knapsack: Given a knapsack that can hold weight \(C\) and a list of provisions \(\langle p_k : k \in \mathbb{N} \rangle\) each of which has a weight \(w_k\) and value \(v_k\). Is it possible to fill the knapsack with provisions weighing no more than \(C\) and having a total value of \(V\) or greater?

\[
\sum w_k \leq C \\
\sum v_k \geq V
\]

Presburger arithmetic is an example of a class of decidable problems. Presburger arithmetic is the collection of statements \(P\) about the natural numbers \(\mathbb{N}\) that only involve addition, equality, and Boolean operations among sub-expressions. The Presburger axioms are:

1. \(\neg(0 = x + 1)\)

2. \(x + 1 = y + 1 \Rightarrow x = y\)

Propositional logic studies the truth of Boolean expressions (True or False values combined using AND, OR, and NOT, and operations that can be defined from these three basic operations.)

First-order logic introduces quantification of formula that involve variables which determine the truth of a expression.
3. \( x + 0 = x \)

4. \( x + (y + 1) = (x + y) + 1 \)

5. Let \( p(n) \) be a first-order formula in the language of Presburger arithmetic about a natural number \( n \). The induction axiom is:

\[
(p(0) \land (\forall n)(p(n) \Rightarrow p(n + 1))) \Rightarrow (\forall n)(p(n))
\]

If \( P \) is a statement about Presburger arithmetic, then \( P \) is decidable, that is there is an algorithm that decides if \( P \) is True or False. Moreover,

- Presburger arithmetic is **consistent**: If \( P \) is derivable from (Presburger) axioms, then \( \neg P \) cannot be deduced from these axioms.
- Presburger arithmetic is **complete**: For each expression \( P \), only one of \( P \) or \( \neg P \) is True, and the one that is True can be derived from the axioms.
- Presburger arithmetic is **decidable**: There is an algorithm that decides whether proposition \( P \) is True or False.

See (Stansifer, 1984) for additional details on on the history and significance of Presburger’s discoveries.

Likewise, Gödel, and others, proved completeness for logical expressions in the first-order logic (Gödel, 1930).

### Theorem 14: Gödel's Completeness Theorem

Every valid logical expression is provable. Equivalently, every logical expression is either satisfiable or refutable.

On the other hand, Gödel (Gödel, 1992) demonstrated how to construct propositions, from the Peano axioms for general arithmetic, that can not be proven True or False. Gödel realized that natural numbers could be used to name basic symbols, expressions over these symbols, and proofs. Let \( G(s) \) be the Gödel number of symbol \( s \). For instance, if

\[
G(0) = 1 \quad G(+) = 3 \quad G(=) = 5 \quad G(x) = 7
\]

Then the axiom \( x + 0 = x \) has Gödel number

\[
G(x + 0 = x) = 2^73^55^711^7 = 1,131,912,171,637,632
\]

If expression \( Q \) can be derived from \( P \) by some rule of inference, then there is a function \( f \) such that

\[
f(G(P)) = G((Q))
\]

Let \( P(n) \) be a predicate and let \( G = G(P(n)) \) be its Gödel number. Consider \( P(G) \). This expression has a Gödel number, call it \( G' \). And, the development goes on from here, beyond the scope of these notes.
Theorem 15: Gödel's First Incompleteness Theorem

Every consistent formal proof system $F$ about a sufficiently rich arithmetic is incomplete.

Theorem 15 says there are statements about the arithmetic we learned as children that are True but have no proof. Gödel's second theorem says you cannot prove a consistent arithmetic is consistent.

Theorem 16: Gödel's Second Incompleteness Theorem

If $F$ is a consistent formal proof system about a sufficiently rich arithmetic, then there is no proof that $F$ is consistent.

Turing Machines

An algorithm can be thought of as a Turing machine for some decision problem. Informally, a Turing machine uses a transition function $\delta$ to map the current state of the machine and the character read to a next state, a character printed, and a direction to move the read write head.

The next state $k'$ either in $K$, the set of states, or one of three special states: answers y “yes” and n, “no,” or the “halt” state h. The read/write head can move ← “left”, → “right,” or — “stay.”

There are many ways to define a Turing machine. Here is Papadimitriou’s (Papadimitriou, 1994) definition.

Definition 17: Turing Machine

A Turing machine is a 4-tuple $M = (K, \Sigma, \delta, s)$ where:

1. $K$ is a finite set of states
2. $s \in K$ is the initial (start) state
3. $\Sigma$ is an alphabet (a finite set of symbols (characters)). $\Sigma$ contains two special symbols: $\sqcup$ and $\sqcap$, called blank and first, respectively.
4. $\delta$ is a transition function. It maps a (state, character) pair to a triple (next state, character, direction).

$$\delta : (K, \Sigma) \rightarrow (K \cup \{h, y, n\}, \Sigma, \{\leftarrow, \rightarrow, \rightarrow\})$$

Example: Turing machine to add 1

The transition function for a Turing machine can be defined by a state transition table. Consider adding one to a natural
number written in binary, for instance \( n = (101010)_2 = 42 \). Assume after the first symbol \( \triangledown \), each bit is written on a cell of a tape and the read/write head is positioned on the leading, leftmost, most significant bit, 1 in this case. A blank cell, \( \square \), lies after the rightmost, least significant bit.

To add one to the \( n \), the Turing machine

1. Copies the bits from left-to-right until the blank cell is scanned.

2. When a blank is scanned, it backs up (to the left) and turns 1’s into 0’s until the first 0 is found.

3. When the first 0 is found, the machine changes the 0 into a 1 and halts.

This can be described by the state transition table below. It reads: When in state \( q_0 \)

- If 0 or 1 is scanned, stay in state \( q_0 \), leave the bit unchanged, and move the read/write head right.

- If \( \square \) is scanned, move to state \( q_1 \), leave the blank unchanged, and move the head left.

Similar transitions can be read for state \( q_1 \).

\[
\begin{array}{c|ccc}
 & 0 & 1 & \square \\
\hline
q_0 & (q_0, 0, \rightarrow) & (q_0, 1, \rightarrow) & (q_1, \square, \leftarrow) \\
q_1 & (\text{halt}, 1, \leftarrow) & (q_1, 0, \leftarrow) & (\text{halt}, \square, \leftarrow)
\end{array}
\]

The machine can also be described by a state transition diagram.

Consider how this machine operated on \((101010)_2 = 42\). It copies the bits from left-to-right until the blank \( B \) is scanned. It then moves back left and seeing the 0, changes it to 1 and halts.

In a similar manner the string \((101011)_2 = 43\), is changed into \((101100)_2 = 44\).
The Universal Turing Machine

Turing showed (Turing, 1936) and important aspect of his machine: \textit{It is possible to invent a single machine that can simulate any other Turing machine.} That is, there is a universal Turing machine, called \( U \). The input to the universal machine \( U \) is a pair \((M, x)\). The universal machine \( U \) computes \( M(x) \), that is \( U(M, x) = M(x) \).

\[ \langle \text{Universal Machine} \rangle \equiv \]
\[
U(\text{machine } M, \text{ input } x) \{ \\
M(x) \;
\}
\]

The existence of a universal machine leads to undecidable problems, the most famous of which is the Halting Problem.

### Problem 15: The Halting Problem

\textit{Decision Problem:} Given a Turing machine \( M \) and its input \( x \), does \( M \) halt on \( x \)?

There is no algorithm that decides the halting problem. It may be possible to decide is a particular machine \( M \) halts on a particular input \( x \), but there is no algorithm that answers the halting problem for every instance of \( M \) and \( x \).

Define the halting language \( \mathbb{H} \) is the set of all (machine, input) pairs such that \( M \) halts on \( x \).

\[
\mathbb{H} = \{ (M, x) : M(x) \neq \downarrow \}
\]

There is no Turing machine that decides whether or not \((M, x) \in \mathbb{H}\) for all pairs \((M, x)\). The proof is by contradiction.

Consider the thought experiment of executing the pseudo-code below: The program accepts the encoding of a machine \( M \) as input. It runs \( M \) on \( M \), looping forever if \( M(M) \) halts and halts if \( M(M) \) does not halt.

\[ \langle \text{Diagonal Machine} \rangle \equiv \]
\[
\text{diagMac(machine } M) \{ \\
\text{if (} M(\text{M}) \text{) halts then } \{ \text{Loop forever; } \} \\
\text{else } \text{halt; } \\
\}
\]

The diagonalization name comes from running the program on itself.

\[ \langle \text{Diagonalization} \rangle \equiv \]
\[
\text{main diagMac(diagMac)};
\]

\textit{The symbol } \downarrow \text{ stands for “does not halt.”}
Now consider the logic:

- If \( \text{diagMac(diagMac)} \) halts, then \( \text{diagMac(diagMac)} \) loops forever, that is, \( \text{diagMac(diagMac)} \) does not halt.
- On the other hand, if \( \text{diagMac(diagMac)} \) does not halt, then \( \text{diagMac(diagMac)} \) halts.

Therefore, there can be no test (algorithm) that correctly answers:

For all Turing machine \( M \) and for all inputs \( x \), does \( M \) halt on \( x \)?

The traditional proof that the halting problem is undecidable goes something like this:

**Proof: The Halting Problem is Undecidable**

Pretend there is a Turing machine \( M_H \) that decides the halting problem.

\[
\begin{array}{c}
(M, x) \\
\hline
M_H \\
\hline
M(x) \neq \uparrow \\
M(x) = \uparrow
\end{array}
\]

Use \( M_H \) to construct a Turing machine \( D \) that accepts the encoding of a Turing machine \( M \) and runs \( M_H \) on \( (M, M) \). The behavior of \( D \) is this:

1. **D does not halt if \( M \) halts on \( M \).**
   
   If \( (M(M) \neq \uparrow) \), then \( D(M) = \uparrow \).

2. **D halts if \( M \) does not halt on \( M \) (\( M(M) = \uparrow \)).**
   
   If \( (M(M) = \uparrow) \), then \( D(M) \neq \uparrow \)

\[
\begin{array}{c}
M \\
\hline
D \\
\hline
M \text{ does not halt on } M
\end{array}
\]

Consider \( D(D) \)

\[
\begin{array}{c}
D \\
\hline
D \\
\hline
D \text{ does not halt on } D
\end{array}
\]
Determinism versus Non-Determinism

By default, Turing machines are deterministic: Their transition functions $\delta$ are functions. When transitions are relaxed to be relations, the machine is said to be non-deterministic.

**Definition 18: The P and NP Complexity Classes**

The complexity class $P$ is the class of all decision problems where all problem instances can be solved in in polynomial time on a (deterministic) Turing machine. $O(n^k)$, where $n$ is the size of the instance and $k$ is a fixed natural number.

The complexity class $NP$ is the class of all decision problems that solve all instances in polynomial time on a non-deterministic Turing machine.

Intuitively, the class $P$ is the set of all problems that can be solved in polynomial time. Such problems are said to be tractable, even though they may run for a very long time.

The class $NP$ is the set of all problems that, when given an answer (a certificate), the answer can be checked to be correct in polynomial time. Cook in his seminal paper (Cook, 1971) clearly described these ideas and their implications.

**Problem 16: Satisfiability**

*Decision Problem:* Given a Boolean expression $B$ of $n$ literals in conjunctive normal form, does $B$ have a truth assignment?

**Example: SAT Problems**

The expression

$$\phi = (p \lor q) \land \neg p$$
is satisfied by \( p = q = \text{False} \).

On the other hand, the expression

\[
\phi = (p \lor q \lor r) \land (p \lor \neg q) \land (q \lor \neg r) \land (r \lor \neg p) \land (\neg p \lor \neg q \lor \neg r)
\]

is unsatisfiable. Although you can reason about this expression to see it is unsatisfiable. Notice the expression is in conjunctive normal form so a satisfying truth assignment must satisfy all clauses.

The first clause requires at least one of the three variables be True. The next three clauses requires all three values be the same. (If \( p \) is True, then \( r \) must be True, and then \( q \) must be True. On the other hand, if \( p \) is False, then \( q \) must be False, and then \( r \) must be False.)

But, in general, you may need to check all \( 2^n \) truth assignments to confirm an \( n \) variable Boolean expression is never satisfied.

Cook describes the satisfiability (SAT) problem, which is clearly in NP but is not known to be in P. The non-deterministic algorithm guesses a satisfying truth assignment for \( \phi \) and checks that it satisfies each clause in \( \phi \). On the other hand, no polynomial-time deterministic algorithm has ever been discovered for satisfiability. This leads to what is said to be the fundamental problem in theoretical computer science.

**Problem 17: P versus NP**

*Decision Problem: Does \( P = NP \)?*

It is clear that \( P \) is a subset of NP. Whether the two classes of problems are the same remains unknown. I think the consensus is that \( P \neq NP \). “Proofs” that \( P \neq NP \) are proffered every so often, but at this time none has stood and no one knows for certain what the answer is.

Here are some sample NP problems. Reason that they belong to NP by convincing yourself that a answer could be checked in polynomial time.

**Problem 18: Subgraph Isomorphism**

*Given two graphs \( G_0 = (V_0, E_0) \) and \( G_1 = (V, E_1) \). Does \( G_0 \) contain a subgraph \( (V, E) \) such that \( |V| = |V_1|, |E| = |E_1|, \) and is there a one-to-one function \( f : V \mapsto V_1 \) such that \( \{u, v\} \in E \text{ if and only if } \{f(u), f(v)\} \in E_1. \)*

Clearly, if given nodes \( V \) and edges \( E \), their cardinalities can be checked in polynomial time. Likewise, that \( f \) preserves edges and be checked in polynomial time.
**Problem 19: Traveling Salesman**

Given a finite set of cities \( C = \{c_0, \ldots, c_{n-1}\} \), distances \( d(c_i, c_j) \in \mathbb{Z}^+ \) for each pair \((c_i, c_j) \in C\), and a bound \( B \in \mathbb{Z}^+\). Is there a tour of all cities with total length no more than \( B \).

That is, a permutation \( \langle c_{\pi(0)}, c_{\pi(1)}, \ldots, c_{\pi(n-1)} \rangle \) of cities such that

\[
\left[ \sum_{k=0}^{n-1} d(c_{\pi(k)}, c_{\pi(k+1)}) \right] + d(c_{\pi(n)}, c_{\pi(0)}) \leq B
\]

Clearly, given the tour, its cost can be computed in polynomial time.

**Reductions**

A classic problem solving technique is to reduce a new problem to an already solved problem. A reduction is an algorithm that solves problem A by transforming any instance of A to an equivalent instance of previously solved problem B. Such a reduction should be executable in
polynomial time. The notation

$$A \leq_P B$$

means if B can be solved in polynomial time, then A can be solved in polynomial time. This establishes potential ways to design algorithms.

On the other hand, if A cannot be solved in polynomial time, then neither can B, establishing intractability.

Consider reducing matching problem to max-flow.

Example: Matching reduced to Reachability

Given an bipartite graph \((U, V, E)\), where \(|U| = |V| = n\). Construct a network of nodes \(U \cup V \cup \{s, t\}\) where \(s\) is the source and \(t\) is the target (sink), and with edges

\[\{(s, u) : u \in U\} \cup E \cup \{(v, t) : v \in V\}\]

where all capacities equal to 1.

Then the bipartite graph has a matching if and only if the network has a flow of value \(n\).

Consider reducing validity: Is a Boolean expression \(E\) always True. It can be reduced to satisfiability.

Definition 19: Validity of a Boolean Expression

A Boolean expression \(\phi\) is valid if it is True for every assignment of True or False to its variables.

To show that Boolean expression \(\phi\) valid, show that \(\neg \phi\) is not satisfiable. If \(\neg \phi\) has no satisfying truth assignment: \(\neg \phi\) is always False. Therefore, \(\phi\) is always True and valid.

Problem 20: Independent-Set

Given a graph \(G = (V, E)\) and an integer \(k\), is there a subset of vertices \(S \subseteq V\) such that \(|S| \geq k\), and for each edge at most one of its endpoints is in \(S\)?

The graph below shows an independent set of size 6, the black nodes.
**Problem 21: Vertex-Cover**

Given a graph $G = (V,E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge at least one of its endpoints is in $S$?

The graph in problem show a vertex cover of size 4, the white nodes.

**Theorem 17: Reducibility: Vertex-cover and Independent-Set**

There is a polynomial time reduction of vertex-cover to independent-set. A subset of nodes $S$ is an independent set if and only if $V - S$ is a vertex cover.

**Proof: Reducibility: Vertex-cover and Independent-Set**

Let $S$ be an independent set of size $k$. Then $V - S$ is of size $n - k$. If $(u, v)$ is an edge, then either $u \not\in S$ or $v \not\in S$ (or both). Therefore, either $u \in V - S$ or $v \not\in V - S$ (or both). That is, for each edge at least one of its nodes is in $V - S$.

On the other hand, Let $V - S$ be a vertex cover of size $n - k$. Then $S$ is of size $k$. Let $u \in S$ and $v \in S$. It must be $(u, v) \not\in E$ because $V - S$ is a vertex cover. (If $(u, v) \in E$, then at least one of $u$ or $v$ is in a vertex cover.) Therefore, no two nodes in $S$ are joined by an edge, that is, $S$ is an independent set.

**NP-Complete Problems**

The book (Garey and Johnson, 1979) is the classic textbook on NP-completeness. A surprising number of problems have been shown to be NP-complete.
**Definition 20: NP-Complete**

A decision problem \( C \) is NP-complete if:

1. \( C \in \text{NP} \), and
2. Every problem in \( \text{NP} \) is reducible to \( C \) in polynomial time.

Intuitively, NP-complete problems are the hardest in NP. It is not clear that there are any NP-complete problem \( C \). And, showing that every problem in NP reduces to \( C \) seems to be an insurmountable task.

Cook’s theorem addresses the first issue.

**Theorem 18: Cook’s Theorem**

SAT is NP-complete.

The proof is well beyond the scope of this class.

The second issue is addressed by this result.

**Lemma 1: Reduction from NP-complete problems**

Let \( A \) and \( B \) be problems in \( \text{NP} \).
If \( A \) is NP-complete and \( A \leq_P B \), then \( B \) is NP-complete.

**Example: 3SAT is NP-complete**

Let \( \phi \) be a Boolean expression in conjunctive normal form where each clause has at most 3 literals. Does \( \phi \) have a truth assignment?

**Co-NP Problems**

PRIMES and COMPOSITE are examples of complementary problems.

- \( \text{PRIME} = \{ n : n \in \mathbb{N} \text{ and } n \text{ is prime} \} \).
- \( \text{COMPOSITE} = \{ n : n \in \mathbb{N} \text{ and } n \text{ is composite} \} \).

**Theorem 19: \( P \) is closed under complements**

If problem \( X \) is in class \( P \), then its complement \( \overline{X} \) is in \( P \) too.
That is, if \( X \in P \), then \( \overline{X} \in P \), or more simply, \( P = \text{co-P} \).
Proof: P is closed under complements

Let $A$ be a polynomial time deterministic algorithm for decision problem $X$. An algorithm $\bar{A}$ for $\overline{X}$ runs $A$ on an instance $I$ of $X$. If $A$ accepts $I$, then $\bar{A}$ rejects $I$. Conversely, if $A$ rejects $I$, then $\bar{A}$ accepts $I$.

For the class NP, the relationship between NP and co-NP is not as clear.

If $X \in$ NP, then there is a certificate (a True solution) can be checked in polynomial time. The complementary problem $\overline{X}$ requires a polynomial time disqualification. That is, a short proof for no instances.

Definition 21: Co-NP

co-NP = $\{X : \overline{X} \in$ NP$\}$

The COMPOSITE decision problem is: Given a natural number $n > 1$, does it have factors other than 1 and itself? COMPOSITE is in NP. Given the prime factorization, you can quickly check that its product is $n$.

The PRIMES decision problem is: Given a natural number $n > 1$, does it have no factors other than 1 and itself? By definition PRIMES is in co-NP.

The subset sum problem is in NP.

Problem 22: Subset Sum

Let $A$ be a finite set of integers. Does $A$ contain a non-empty subset the sums to 0?

You can check in linear time that values in a non-empty subset sum to 0.

The complementary subset sum problem requires that all non-empty subsets have non-zero sums.

Problem 23: Co-Subset Sum

Let $A$ be a finite set of integers. Does every non-empty subset of $A$ sum to a non-zero value?

Problem 24: Unsatisfiability

Decision Problem: Given a Boolean expression $B$ of $n$ literals in conjunctive normal form, does $B$ have no satisfying truth assignment?

SAT is in NP: Given a truth assignment that satisfies a Boolean expression $B$, it can be checked in polynomial time.
UNSAT is in co-NP by definition, it may not be simple to prove there is no satisfying truth assignment.

**Problem 25: No Hamiltonian Cycle**

*Decision Problem:* Given a graph $G = (V, E)$, is there no simple cycle that contains every node of $V$?

Can give a permutation of nodes to prove there is a Hamiltonian cycle. How can you prove there is no Hamiltonian cycle?

**NP-Hard Problems**

**Definition 22: NP-Hard**

*Decision problem* $H$ is NP-hard if every NP-complete problem $C$ can be reduced to $H$ in polynomial time.

An NP-hard problem is at least as hard as the hardest problems in NP.

The *halting problem* is NP-hard. SAT can be reduced to the halting problem by transforming SAT into a Turing machine that tries all possible truth assignments for an instance $I$ of SAT. When the machine finds a satisfying truth assignment it halts. Otherwise, if there is no satisfying truth assignment, the machine goes into an infinite loop.

**PSPACE**

*Complexity Hierarchy*

Computational complexity is complex. Here is an image from (Papadimitriou, 1994) (created by Sebastian Sardina) that shows the relationship among several complexity classes under common assumptions that have not been fully proven.
14. Computational Complexity

- \( R \)
- \( \text{ELEMENTARY} \)
- \( \text{PSPACE} \)
- \( \text{EXPTIME} \)
- \( \text{EXPTIME} \)
- \( \text{PTIME} \)
- \( \text{LOG Space} \)
- \( \text{LOG Time} \)
- \( \text{NPC} \)
- \( \text{PSPACE} \)
- \( \text{EXPSPACE} \)
- \( \text{2EXPTIME} \)
- \( \text{EXPSPACE} \)
- \( \text{EXPSPACE} \)
- \( \text{ELEMENTARY} \)
Bibliography


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to Algorithms. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


Index

Haskell programming language, 20, 31, 44, 116, 137
Java programming language, 31

Ackermann function, 174
Axioms
  Peano, 192

Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193

Dijkstra’s algorithm, 183
DNA, 146

Empty
  String \( \lambda \), 61

Fibonacci, 44, 46, 142
Floating point
  Machine epsilon, 50
Function
  Ackermann, 174
Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstadter, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabriel, 44
\( \lambda \) calculus, 26, 28
Linear congruence, 47
Listing
  C
    Matrix Chain Multiplication, 158
Listings
  C
    A Minimum Algorithm, 135
    Aho-Corasick, 89
    Boyer-Moore Pattern Matching, 72
    Brute-force Left-to-Right Pattern Matching, 56
    Bubble Sort, 109
    Build a Heap, 126
    Counting Sort, 128
    Direct Address Table Operations, 94
    Edit Distance, 152
    Find Set with Path Compression, 177
    Greatest Common Divisor, 42
    Hash Table with Chaining, 95
    Heap Sort, 127
    Heapify a List, 126
    Inner Product, 155
    Insertion Sort, 112
    Knuth-Morris-Pratt Pattern Matching, 67
    Maximum Subsequence Sum (Cubic), 34
    Maximum Subsequence Sum (Linear), 35
    Merge Sort, 118
    Morris-Pratt Pattern Matching,

Haskell

Partition about a pivot, 120
Quick Sort, 120
Radix Exchange Sort, 132
Radix Sort, 129
Randomized Partition, 136
Randomized Selection, 138
Rational Knapsack, 167
Selection Sort, 114
Shell Sort, 116
Union by rank, 177

Haskell

Ackermann function, 174
Apply a Function Repeatedly, 50
Brute-force Left-to-Right Pattern Matching, 57
Bubble Function, 108
Bubble Sort, 109
Extended Euclidean Algorithm, 48
Fold from the left, 37
Greatest Common Divisor, 44
Inner Product, 155
Insertion Sort, 112
Matrix Multiplication, 155
Maximum Subsequence Sum, 37
Merge Sort, 117
Merging Sorted Lists, 117
Newton’s Square Root Method, 51
Newton’s Square Root Recurrence, 50
Quicksort, 120
Randomized Partition, 137
Relative Error Convergence Test, 50
Sample Code Header, 31
Second of a pair, 36
Selection Sort, 114
Sorted Decision Problem, 106