Computational Complexity

$\mathcal{O}$ Ω Θ

Russell Buehler

January 5, 2015

Chapter 1

Preface

What follows are my personal notes created during my undergraduate algorithms course and later an independent study under Professor Holliday as a graduate student. I am not a complexity theorist; I am a graduate student with some knowledge who is–alas–quite fallible. Accordingly, this text is made available as a convenient reference, set of notes, and summary, but without even the slightest hint of a guarantee that everything contained within is factual and correct. This said, if you find an error, I would much appreciate it if you let me know so that it can be corrected.

Contents

1 Preface **3**

2 A Brief Review of *Algorithms* **1**
- 2.1 Algorithm Analysis . . . 1
  - 2.1.1 O-notation . . . 1
- 2.2 Common Structures . . . 3
  - 2.2.1 Graphs . . . 3
  - 2.2.2 Trees . . . 4
  - 2.2.3 Networks . . . 4
- 2.3 Algorithms . . . 6
  - 2.3.1 Greedy Algorithms . . . 6
  - 2.3.2 Dynamic Programming . . . 10
  - 2.3.3 Divide and Conquer . . . 12
  - 2.3.4 Network Flow . . . 15
- 2.4 Data Structures . . . 17

3 Deterministic Computation **19**
- 3.1 O-Notation . . . 19
- 3.2 Models of Computation . . . 21
  - 3.2.1 Space Bounds . . . 21
  - 3.2.2 On Decision Problems . . . 22
- 3.3 The Deterministic Hierarchies . . . 23
  - 3.3.1 The Deterministic Time Hierarchy Theorem . . . 23
  - 3.3.2 The Deterministic Space Hierarchy Theorem . . . 25
- 3.4 Classifying Problems . . . 26
  - 3.4.1 Boolean Logic . . . 26
  - 3.4.2 Boolean Circuits . . . 27
  - 3.4.3 Graph Theory . . . 28
  - 3.4.4 Sets and Numbers . . . 30
- 3.5 Reduction . . . 31
  - 3.5.1 PTIME-Complete Problems . . . 31

4 Nondeterminism **33**
- 4.1 Nondeterministic Machines . . . 33
- 4.2 Nondeterministic Complexity Classes . . . 35
  - 4.2.1 Nondeterministic Hierarchy Theorems . . . 35
- 4.3 Classifying Problems . . . 38
  - 4.3.1 Boolean Logic . . . 38
  - 4.3.2 Graph Theory . . . 38
  - 4.3.3 Numbers and Sets . . . 39
- 4.4 Unifying the Hierarchies . . . 41
  - 4.4.1 The Reachability Method . . . 41
  - 4.4.2 Nondeterminism and Space . . . 42

4.5 Reduction . . . 44
4.5.1 NPTIME-Complete Problems . . . 44
4.5.2 Strong NPTIME-Completeness and Pseudpolynomial Algorithms . . . 47
4.6 NPTIME and coNPTIME . . . 49
4.6.1 NPTIME as Polynomial Certifiability . . . 49
4.6.2 coNPTIME as Polynomial Disqualifiability . . . 49
4.6.3 NPTIME ∩ coNPTIME . . . 51

5 Function Problems **53**
5.1 Total Function Problems . . . 55

6 Randomization **57**

Chapter 2

A Brief Review of *Algorithms*

§2.1 Algorithm Analysis

2.1.1 O-notation

As usual, $\mathbb{N}$ denotes the set of all natural numbers. By definition, algorithms are step-by-step processes which vary in the number of steps executed, the space used, etc. for a given input based on both the size of that input as well as other algorithm-intrinsic properties of the input. It's obvious that this dependency on the algorithm-relative properties of an input represents a severe roadblock to the general characterization and representation of the relative efficiency (time, space, and otherwise) of algorithms; luckily, it can be abstracted away from. The most common way of doing so is via *worst-case* analysis. There are, however, alternatives: best-case analysis and–more notably–average case analysis. Some thought shows the former to be a rather foolish choice, but the latter is of considerable interest. Unfortunately, the distribution of instances of a particular problem seldom admit of an easy characterization, making average case analysis a pragmatically difficult venture and leaving worst-case analysis as the *de facto* choice.

Accepting this, characterizing the run times, space requirements, etc. of algorithms falls naturally into a functional representation with a parameter, normally n, for the size of the input. In other words, we characterize the time/space/resource requirements of algorithms by functions from $\mathbb{N} \to \mathbb{N}$, e.g. n, $n^2 + 3$, and 3^n. Later, it will be useful to utilize real-valued functions like $\sqrt{n}$ and $\log(n)$; appearances to the contrary, these functions too are to be thought of as functions over $\mathbb{N}$, namely $\max\{\lceil f(n) \rceil, 0\}$ for a real-valued f.

Of course, rather than try to give a function which perfectly represents a given algorithm's usage of a resource–a tall order–our characterization captures usage in a more general sense. For two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, $f(n) = O(g(n))$ (pronounced, "$f(n)$ is big-oh of $g(n)$" or "f is of the order of g") if and only if there are $n_0, c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$. Informally, $f(n) = O(g(n))$ means that f eventually uses the resource no faster than g, modulo a constant. Changing emphasis, this is also notated $g(n) = \Omega(f(n))$. Finally, if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we write $f(n) = \Theta(g(n))$, meaning f and g have the exact same rate of usage.

Given our work thus far, it's easy to show that–for any $p \in \mathbb{N}$, $r > 1$, and resource,

$$O(\log^p(n)) < O(n^p) < O(r^n)$$

> **(Time) Efficient**
> An algorithm is *efficient* if and only if it has a polynomial time bound; that is, its runtime is $O(n^p)$ for some $p \in \mathbb{N}$.

> **(Time) Intractable**
> A problem for which no efficient algorithm exists is *intractable*

Properties

- Transitivity:

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

- Let k be a fixed constant, and let $f_1, f_2, ..., f_k$ and h be functions such that $f_i = O(h)$ for all i. Then, $f_1, f_2, ..., f_k = O(h)$.
- Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$.
- Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.
- For every $b > 1$ and every $x > 0$, we have $log_b(n) = O(n^x)$. Note that the base b is often dropped because $log_a(n) = \frac{log_b(n)}{log_b(a)}$, a constant change.
- For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.
- The cost of storing a number of length n is $log(n)$.

Some particularly tricky problems may require ingenious functions, say $f(n) = n^3$ on even n, but only 1 on odd. It follows that $f(n) \neq O(n^2)$ and $f(n) \neq \Omega(n^2)$.

§2.2 Common Structures

2.2.1 Graphs

Graph
A *graph* is defined as a set of nodes $V = \{1, ..., n\}$ and edges $E = \{\langle u, v\rangle | u, v \text{ nodes } \}$. The cardinality of V and E are traditionally, $n = |V|$ and $m = |E|$.

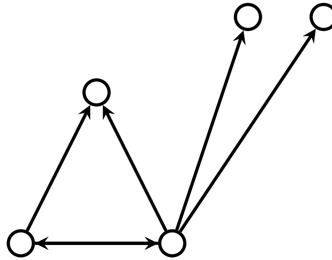


Figure 2.1: A Graph

Graphs are traditionally represented by either an 'adjacency matrix' or an 'adjacency list'. The former is an n x n matrix wherein $M[i, j] = 1$ if $\langle i, j\rangle \in E$ and 0 otherwise. The latter is a list of n elements wherein the ith entry is itself a list of all the neighbors of node i.

	Space	Is $\langle u, v\rangle \in E$?	What are the neighbors of node v?
List	$\Theta(m + n)$	$\Theta(\text{degree}(u) + 1)$	$\Theta(1)$ to return, $\Theta(\text{degree}(u) + 1)$ to print
Matrix	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n)$

Sparse/Dense
A family of graphs is *sparse* if $m = O(n)$ and *dense* if $m = \Omega(n^2)$.

Undirected/Directed
A graph is *directed* if and only if whenever $\langle u, v\rangle \in E$, $\langle v, u\rangle \in E$ as well. A graph is *undirected* just in case it is not directed.

► **Proposition 2.1.**
Let G be an undirected graph on n nodes. Any two of the following statements implies the third.

(i) G is connected

(ii) G does not contain a cycle

(iii) G has $n - 1$ edges

2.2.2 Trees

Tree
A graph G is a *tree* if and only if it is undirected and any two vertices are connected by exactly one *simple path* (a path without any repeated vertices)

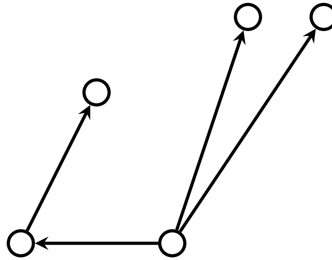


Figure 2.2: A Tree

► **Proposition 2.2.**
Trees don't have cycles

► **Proposition 2.3.**
Every n-node tree has exactly $n - 1$ edges

2.2.3 Networks

Network
A *network* $N = (V, E, s, t, c)$ is a graph (V, E) with two specified notes s (the source) and t (the sink) such that the source s has no incoming edges and the sink t has no outgoing edges. In addition, for each edge $\langle i, j \rangle \in E$, we have a capacity $c(i, j) \in \mathbb{N} - \{0\}$

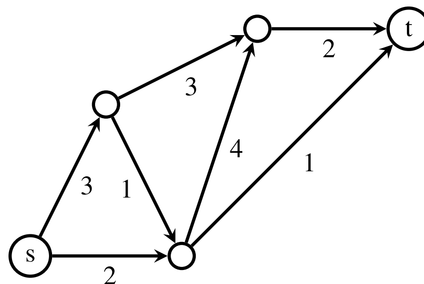


Figure 2.3: A Network

Flow
A *flow* on a network N is a function f assigning an $n \in \mathbb{N}$ to each edge $\langle i, j \rangle \in E$ such that $f(i, j) \leq c(i, j)$ and, for every node other than s and t, the sum of the f-values entering the node is equal to the sum of the f-values leaving it. The *value of a flow* is the sum of the f-values leaving s or–equivalently–entering t.

► **Theorem 2.1** (Max Flow - Min Cut Theorem)**.**
The value of the maximum $s - t$ flow equals the capacity of the minimum $s - t$ cut

---

► **Theorem 2.2** (Cycle-less Maximum Flows)**.**
For all maximum flows on directed graphs, there exists a max flow on the same graph with the same value, but no cycles with positive flow

---

§2.3 Algorithms

Algorithms are divisible into four broad classes:

(1) Greedy Algorithms

(2) Dynamic Programming

(3) Divide and Conquer

(4) Network Flow

What follows is a brief overview of each along with the problems and algorithms traditionally classed under them. The notes in this section are owed to the lectures of Professor David Liben-Nowell.

2.3.1 Greedy Algorithms

Greed is good

– Gordon Gekko, *Wall Street*

A greedy algorithm is an algorithm that uses myopic greed to achieve a global optimum. This strategy, of course, is heavily dependent on the internal structure of the problem. Sometimes myopic greed can always deliver a global optimum; others, it is doomed to mediocrity or worse. There are two main arguments used to show greedy algorithms are in fact optimal: 'Greedy Stays Ahead' and 'Exchange'. The first argues that at no point can optimal be any better than greedy and therefore greedy must be at least as good as the optimal (and therefore is optimal); an example of this style usually accompanies the interval scheduling problem. The 'Exchange' argument, on the other hand, considers the optimal solution (which we assume differs from the greedy) then argues that members of the solution out of place by the greedy criterion at worst maintain the quality of the solution. It follows that by performing a number of these swaps we achieve the greedy solution–which must be at least as good as the optimal (and therefore is optimal); this argument often accompanies the 'minimize lateness' version of the scheduling problem.

Stable Marriage

STABLE MARRIAGE
`Given:` A set M and a set W such that $|M| = |W| = n$ where every member of each has complete preferences over the opposite set
`Return:` A perfect matching of M and W with no instabilities

Matching
A *matching* of M and W is a set of $\langle m, w\rangle$ pairs such that no person is in more than one pair; in a *perfect matching*, no person is left unpaired.

Instability
In a matching, $\langle m, w\rangle$ is an *instability* if and only if there exists $m \in M$, $w \in W$ such that (1) m, w are not matched and (2) m and w each prefer the other to their current match.

Gale-Shapley Proposal Algorithm

```
1 while ∃m ∈ M who hasn't been rejected by all w ∈ W do
2     m proposes to his most preferred w who hasn't yet rejected him;
3     w accepts if single or if she likes him more than her current partner, m';
4 return engaged pairs
```

► **Proposition 2.4** (Gale-Shapley gives a Stable, Perfect Matching)**.**
The Gale-Shapley proposal algorithm terminates is $O(n^2)$ and returns a perfect matching with no instabilities.

Proof Sketch.
Note that each iteration results in a new proposal and only $n * n$ possible; thus, $O(n^2)$. $|W| = |M|$ and women prefer a man over no man–thus, perfect matching. As the algorithm progresses, elements of W can only improve their lot.

Valid Pair
For $m \in M$ and $w \in W$, m and w are a *valid pair* if and only if there exists a stable, perfect matching in which m and w are matched. The *best valid partner for* m/w is the highest ranked member of the opposite set for which m/w has a valid pair.

► **Proposition 2.5** (Gale-Shapley and Best Valid Partners)**.**
The Gale-Shapley Proposal Algorithm returns a perfect matching in which each m is paired with their best valid partner, each w their worst.

Stable Marriage Variations:

1. Polygamy or Stable Assignment (Hospitals and Residents)

   (a) A lazy solution can be achieved by entering the polygamists as multiple individuals; a proof that this solution is optimal can be achieved by showing that a hospital-resident instability implies a m-w instability in Gale-Shapley.

   (b) We can do better by having residents propose to hospitals and having each hospital maintain a min heap of partners, booting the worst when all slots are filled.

   (c) We can even account for number discrepancies by creating dummy hospitals ("Unemployment") or loser residents ("Worse than the worst").

   (d) In terms of runtime, the above can be $O(klog(k)m^2)$ vs. Gale-Shapley's $O(k^2m^2)$, for k slots per hospital and m total hospitals.

Reachability

Reachability
`Given:` A graph $G = (V, E)$ and two nodes $1, n \in V$
`Return:` Is there a path from 1 to n?

There are two standard algorithms used to search a graph: Breadth First Search (BFS) and Depth First Search (DFS). These algorithms represent two variations on the same general strategy. Both move from node to node, but BFS does so in a FIFO manner and DFS a LIFO manner.

Breadth First Search

```
1 Initialize seen[v] := false for all nodes v except s;
2 Initialize seen[s] := true ;
3 Insert s into an empty queue Q ;
4 while Q is not empty do
5     Dequeue u from Q;
6     → Do something with u ← ;
7     for every neighbor v of u with seen[v] = false : do
8         seen[v] := true;
9         Enqueue v into Q;
```

Depth First Search

```
1 Initialize seen[v] := false for all nodes v except s;
2 Initialize seen[s] := true;
3 Push s into an empty stack S ;
4 while S is not empty do
5     Pop u from S ;
6     → Do something with u ←;
7     for every neighbor v of u with seen[v] = false : do
8         seen[v] := true;
9         Push v onto S ;
```

The runtime of the BFS and DFS algorithms are $O(m + n)$ if implemented using an adjacency list or $O(n^2)$ if implemented by an adjacency matrix.

Shortest Path

Shortest Path
Given: A graph $G = \langle V, E\rangle$, weights $w : E \to \mathbb{Z}^{\geq 0}$, and $s \in V$
Return: The length of all shortest paths from s in G

Dijkstra's Algorithm

1 Initialize $D[v] := +\infty$ for all nodes v except s;
2 Initialize $D[s] := 0$;
3 **while** not all $D[u]$'s are finite **do**
4 Let v be the node such that $D[v] = +\infty$ and the quantity $d_v := \min_{u:\langle u,v\rangle \in E} D[u] + w(u, v)$ is minimized;
5 If $d_v = +\infty$, then return D; otherwise, set $D[v] := d_v$;
6 **return** D

Minimum Spanning Trees

Minimum Spanning Trees (MST)
Given: A connected graph $G = \langle V, E\rangle$ and weights $w : E \to \mathbb{Z}^{\geq 0}$
Return: $T = \langle V, E' \subseteq E\rangle$ a tree such that $\sum_{e \in E'} w(e)$ is minimized

Note that in the unweighted case we have a simple solution in the form of the BFS tree; adding weights, however, complicates the problem.

► **Proposition 2.6** (Cycle Rule).
Let C be any cycle in G. Let $e \in C$ be the strictly heaviest edge in C. Then e is not in any MST of G.

► **Proposition 2.7** (Cut Rule)**.**
Let $\langle S, V-S\rangle$ be any cut in G. Let e be the strictly cheapest edge crossing the cut $\langle S, V-S\rangle$. Then e is in every MST of G.

Surprisingly, there exists a large number of greedy algorithms that solve the MST quickly. Among the best known are the 'Reverse-Delete Algorithm' (Delete heaviest edge, unless doing so disconnects the graph), 'Kruskal's Algorithm' (Add cheapest edge unless doing so creates a cycle), and 'Prim's Algorithm' (starting from any node, assign it to S, make the cut $\langle S, V-S\rangle$, add the cheapest edge that crosses the cut and the node it connects to to S. Repeat). Rather surprisingly, all of these algorithms can be implemented in $O(mlog(n))$ using the disjoint sets data structure.

INTERVAL SCHEDULING
Given: A set of intervals $\{1, 2, ..., n\}$ each of which contains a start and end time.
Return: A schedule that has the maximum number of intervals possible without conflicts

Consider the following greedy algorithm:

End-Time Greedy Schedule

```
1 Sort the intervals by end time;
2 while There exists an unexamined interval do
3     Consider the next interval;
4     if it doesn't conflict with any earlier accepted intervals:
5         Accept;
6     else:
7         Don't accept
```

Considering the runtime of this algorithm, we see that the intial sort is $O(nlog(n))$ and trumps the rest of the algorithm ($O(n)$).

2.3.2 Dynamic Programming

A bottom up programming technique that emphasizes solving small subproblems once, and then using these to construct possible solutions to combinations of subproblems until, eventually, the problem itself is resolved. Perhaps the most intuitive way to view this procedure is as solving a reduced version of the problem (try to reduce the complexity along a variable dimension) and then using this reduced version to find the possible solutions to a slightly larger version, and so on. Dynamic programming algorithms almost always use tables to store the results of solving smaller subproblems, and emphasize finite loops that mimic recursive calls.

Dynamic Programming algorithms work well under two conditions: (1) solutions have a recursion friendly substructure and (2) there exists a small number of subproblems that are solved many times.

> WEIGHTED INTERVAL SCHEDULING
> `Given:` n jobs, each with a start time, end time, and value
> `Return:` A set S of consistent jobs such that $\sum_{j \in S} v(j)$ is maximized

> SEQUENCE ALIGNMENT
> `Given:` Two strings x and y
> `Return:` An alignment of x and y of minimum cost

We generate a dynamic programming solution by noting that in an optimal alignment M, at least one of the following is true:

(i). $(m, n) \in M$

(ii). The mth position of X is not matched

(iii). The nth position of Y is not matched

and noting that we may impose costs δ and $\alpha_{i,j}$ for a gap or mismatch of letters i and j respectively. It follows immediately that we have the following recurrence relation for $i, j \geq 1$:

$$OPT(i, j) = \min\{\alpha_{x_i, y_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)\}$$

And thus we achieve the following algorithm:

Alignment (X, Y)

```
1 Array A[0...m, 0...n];
2 Initialize A[i, 0] = iδ for each i;
3 Initialize A[0,j] = jδ for each j;
4 for j = 1, ..., n do
5     for i = 1, ..., m do
6         Use the recurrence above to compute A[i, j];
7 return A[m,n]
```

To evaluate the runtime of the algorithm, we need only note that the array has at most mn entries and we spend a constant amount of time on each–yielding a $O(mn)$ bound. It's also worth noting that by translating this problem into a graph, we can achieve the same runtime bound, but a $O(n + m)$ bound on memory usage (6.7).

> WORD SEGMENTATION
> `Given:` A sequence of n letters from $\{A, B, ..., Z\}$ and oracle access to English
> `Return:` A boolean corresponding to whether or not the sequence can be segmented into a sequence of English words.

To solve this problem, think about working backwards from the end of the sequence and assigning either true or false to each letter indicating whether the sequence from that letter to the end is segmentable or not.

SHORTEST PATHS IN WEIGHTED GRAPHS

`Given:` A directed graph $G = \langle V, E\rangle$, weights $W : E \to \mathbb{Z}$, a source s and sink t in V, and no negative cycles

`Return:` The $s - t$ path of minimum weight

It's helpful to note that this problem is very similar to the one solved by Djikstra's Algorithm; indeed, only the possibility of negative weights differentiates the two. Nonetheless, this difference is enough to break Djikstra's algorithm (as well as attempts to translate back to Djikstra's, for example adding a constant amount to each edge to make them all positive. If all paths had the same length this would work, but since the number of edges in a path varies, we actually change the shortest path during this operation) and so we consider other solutions.

The *Bellman-Ford Algorithm* provides our first solution to the given problem, and is based on the following observation: If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e. does not repeat nodes), and hence has at most $n - 1$ edges. Using $OPT(i, v)$ to denote the minimum cost of a $v - t$ path using at most i edges; thus, by the above, our original problem is to compute $OPT(n - 1, s)$. From our definitions, we may generate the following recursive formula:

$$OPT(i, v) = \min\{OPT(i - 1, v), \min_{W \in V}\{OPT(i - 1, w) + c_{vw}\}\}$$

and may construct the following algorithm:

Bellman-Ford

```
1 Array M[0...n − 1, V];
2 Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V;
3 for i = 1, ..., n − 1 do
4     for v ∈ V in any order do
5         Compute M[i, v] using the recurrence OPT above;
6 return M[n − 1, s]
```

Considering the runtime of BELLMAN-FORD we can note that the table M has n^2 entries and each entry takes at most n time, giving a $O(n^3)$ time; we may, however, improve this bound by reexamining our recurrence and utilizing a different variable for the number of edges in the graph–an analysis which gives a $O(mn)$ bound.

In addition, there also exists a second solution to this problem by slowly building up the number of nodes we may use to get from s to t. This process generates the following recursive formula:

$$OPT_k(u, v) = \text{ the optimum path from } u \text{ to } v \text{ where all intermediate nodes are in } \{1 \ldots k\}$$

$$OPT_{k+1}(u, v) = \min\{OPT_k(u, v), OPT_k(u, k + 1) + Opt_k(k + 1, v)\}$$

and algorithm:

Floyd-Warshall

```
1 Fill in OPT_0;
2 for k = 1, ..., n do
3     for all u, v do
4         Fill in OPT_k;
5 return OPT_n(s, t)
```

2.3.3 Divide and Conquer

'Divide and Conquer' describes a class of algorithms which progress by breaking input into several parts and then recursively combining these to generate overall solutions; in general, divide and conquer algorithms don't provide initial polynomial time solutions to hard problems, but rather provide more efficient implementations of problems already known to be solvable in polynomial time (i.e. where even brute force is polynomial). Some of the more famous divide and conquer algorithms are Binary Search, Quicksort, and Mergesort. Examining these and other examples, we find that divide and conquer algorithms go through three basic stages: Divide, Conquer, and Reconstruct. Interestingly, it is usually this last stage that poses most of the difficulties.

Counting Inversions
  Given: Array A[1...n] of integers
  Return: The number of inversions in A ($\langle i, j \rangle$ such that $i < j$ and $A[i] > A[j]$)

To solve this problem, consider breaking inversion into three cases: those on the right of the midpoint, those on the left, and those that cross. This insight leads to the following algorithm:

CI(A)

```
1 if |A| ≤ 1 :
2     return 0
3 else:
4     Divide A into L = A[1...|A|/2], R = A[|A|/2 + 1...|A|];
5     C_L = CI(L);
6     C_R = CI(R);
7     C_X = "Cross-Half" inversions (check all pairs);
8     return C_L + C_R + C_X
```

Using the Master method, we find that the runtime of this algorithm is $\theta(n^2)$–the same as brute force! Examining the algorithm above shows that it is, in fact, the C_X term which is slowing the algorithm; seeking a cleverer implementation yields:

```
1 C_X = 0;
2 Sort L (with a good sort);
3 for Each x ∈ R do
4     Binary search for x in L;
5     C_X = C_X+ the number of y ∈ L such that y > x;
```

Trying the master method again, we find that it falls into none of the cases; however, by induction we can prove $\theta(nlog^2(n))$.

Order Statistics
  Given: A[1...n], $k \in \{1...n\}$
  Return: The kth largest element of A (where the first largest element is the smallest)

Magic5($A[1...n], k \in \{1...n\}$)

```
1  if  k == n == 1 :
2      return A[1]
3  else:
4      Arbitrarily divide A into n/5 piles of 5 numbers each;
5      Find the median of each pile by brute force;
6      Set x as the median of the medians;
7      Partition A into LESS[1...i] and GTR[1...n-i-1];
8      if  i + 1 == k :
9          return x
10     elif  i + 1 < k :
11         return  MAGIC5(GTR, k-i-1)
12     else:
13         return MAGIC5(LESS, k)
```

Rather amazingly, writing out the recurrence relation and calculating the sums for the recursion tree (note that all the operations at each level are $O(n)$) results in an upperbound of $O(n)$ for the entire algorithm.

Closest Points
Given: $\{\langle x_i, y_i \rangle \mid 1 \leq i \leq n\}$
Return: $argmin_{p,q \in P; p \neq q} \; d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$

ClosestPoints ($P[1...n]$)

```
1  if  n <= 3 :
2      Compute all pairwise distance and return the smallest pair;
3  else:
4      boundary := x-coordinate of the median of P with respect to x-coordinate;
5      Left := the points in P with x-coordinate <= boundary (still sorted by y);
6      Right := the points in P with x-coordinate> boundary (still sorted by y);
7      δ_L, p_L, q_L := CLOSESTPOINTS(Left);
8      δ_R, p_R, q_R := CLOSESTPOINTS(Right);
9      δ := min{δ_L, δ_R};
10     DMZ := the points in P with x-coordinate in boundary ±δ (still sorted by y);
11     for Each  p ∈ DMZ do
12         Compare p to the 9 points before and after it in the DMZ order;
13         Let δ_C, p_C, q_C be the distance between (and the points in) the closest such pair.;
14     return δ_i, p_i, q_i  for the smallest  δ_i  for  i ∈ {L, R, C}
```

A rather involved (geometric) pigeonhole argument shows that this reduction actually operates in $O(nlog(n))$ time vs. the brute force $O(n^2)$.

► **Theorem 2.3** (Master Theorem)**.**
Let $T(n) = aT(n/b) + f(n)$ for $a \geq 1$ and $b > 1$ and for some asymptotically positive function f. Then:

(i) if $f(n) = O(n^{log_b(a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \theta(n^{log_b(a)})$. ("The leaves dominate")

(ii) if $f(n) = \theta(n^{log_b(a)})$, then $T(n) = \theta(f(n)log(n))$. ("All levels are equal")

(iii) if $f(n) = \omega(n^{log_b(a)+\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \theta(f(n))$. ("The root dominates"); this case also requires that for all sufficiently large n and some $c > 1$, we require that $af(n/b) \leq cf(n)$.

NOTE: These three cases are not exhaustive; many recurrences aren't solvable via the Master Theorem; Ex: Counting Inversions' $T(n) = 2T(n/2) + nlog(n)$

2.3.4 Network Flow

Network flow represents a slightly less mainstream algorithmic technique the relies heavily on the graph theoretic notion of network flows. Despite its greater anonyminity, nework flow algorithms provide a powerful means of solving many complex problems.

Given a directed graph $G = \langle V, E \rangle$, a capacity function $C : E \rightarrow \mathbb{Z}^{\geq 0}$ and nodes $s, t \in V$ such that there are no edges into s and no edges out of t.

s-t Flow: A function $f : E \rightarrow \mathbb{R}^{\geq 0}$ such that (i) $\forall e \in E,\ f(e) \leq c(e)$ and (ii) $\forall u \notin \{s, t\},\ \sum_{u:(u,v) \in E} f(u, v) = \sum_{u:(v,u) \in E} f(v, u)$ (flow into u = flow out of u), the conservation constraint.

Flow Value : the flow out of the source ($f_{out}(s)$), note that $f_{out}(s) = f_{in}(t)$

Residual Graph : Given a graph G and flow f, we'll define the residual graph G_f with:

- Same nodes as G
- for every edge e for which $f(e) < c(e)$ include the edge in G_f with residual capacity $c(e) - f(e)$ (forward edges).
- for every edge $\langle u, v \rangle$ in G for which $f(u, v) > 0$, include $\langle v, u \rangle$ in G_f with residual capacity $f(u, v)$ (backward edges).

MAXIMUM FLOW
  Given: A directed graph G, capacity function c, source node s, and sink node t
  Return: $s - t$ flow f such that $value(f)$ is maximized

Ford-Fulkerson

```
1 Initially f(e) = 0 for all e in G;
2 while there is an s − t path in the residual graph G_f do
3     Let P be a simple s − t path in G_f;
4     f′ = AUGMENT(f, P);
5     Update f to be f′;
6     Update the residual graph G_f to be G_f′;
7 return f
```

Augment(f, P)

```
1 Let b be the minimum residual capacity of any edge on P with respect to the given flow, f;
2 for each edge (u, v) ∈ P do
3     if e = (u, v) is a forward edge :
4         increase f(e) in G by b
5     else:
6         (u, v) is a backward edge, and let e = (v, u);
7         decrease f(e) in G by b;
8 return f
```

Considering the runtime of FF shows that the algorithm runs in $O(mf^*)$–an inefficient solution given that the input size is about $m + log(f^*)$.

EDMONDS-KARP Heuristic 1: Always choose the augmenting path with largest residual capacity. This produces a PRIM-like algorithm that runs in $O(m^2 log(m) log(f^*))$.

EDMONDS-KARP Heuristic 2: Pick the shortest augmenting path. This addition to FF produces an algorithm that runs in $O(nm^2)$.

BIPARTITE MATCHING
  Given: $G = \langle L \cup R, E \rangle$
  Return: Maximum-sized matching in G

To solve this problem, simply connect the source to all the nodes on the left with capacity 1, assign capacity 1 to all the edges in the graph, and then connect all the nodes on the right to the sink, also with capacity 1. Moreover, because of the bound on our capacities, we can accomplish this using the FF algorithm in $O(mn)$ time.

EDGE-DISJOINT PATHS
Given: A graph G, nodes s and t
Return: Maximum number of $s - t$ paths that don't share any edges

To solve this problem, simply translate every edge into a backward and forward directed, set the capacity of every edge to 1, and then run EDMONDS-KARP (Use the 'No Cycles' theorem). Moreover, because of the bound on our capacities, we can accomplish this using the FF algorithm in $O(mn)$ time.

§2.4 Data Structures

Array- We store n items in a linear sequence of slots. We have $\Theta(1)$-time access to the ith item, but to move an item around involves shuffling the other elements. There is also a fixed size of the array.

- *Dictionary (sorted array):* $\Theta(1)$ create; $\Theta(log(n))$ lookup (binary search); $\Theta(n)$ insert/delete.
- *Stacks / Queues (circular array):* $\Theta(1)$ pop/push or enqueue/dequeue (unless full)

Doubly Linked Lists- We store n items in a linear sequence of linked nodes. We have $\Theta(i)$-time access to the ith item by walking down the list, but we can splice in/out nodes in constant time.

- *Dictionary (unsorted linked list):* $\Theta(1)$ create; $\Theta(n)$ lookup/delete (linear search); $\Theta(1)$ insert.
- *Stacks / Queues:* $\Theta(1)$ pop/push or enqueue/dequeue.

(Balanced) Binary Search Trees- A binary tree stores n nodes, where each node has up to two children ("left" and "right"). Each node in a binary search tree (BST) stores a key, and maintains the property that, for a node u with key u_k, every key contained in u's left subtree has keys $< u_k$ and every key contained in u's right subtree has keys $> u_k$. A balanced binary search tree (AVL tree, red-black tree, ...) has the additional property that the height of the tree is $O(log(n))$. Inserting/deleting in these trees can be done in time proportional to the height of the tree while maintaining the balance.

- *Dictionary:* $\Theta(1)$ create; $\Theta(log(n))$ lookup/insert/delete/min/max (min/max can be made $O(1)$ by explicit tracking)
- *Priority Queue:* $\Theta(1)$ create; $\Theta(log(n))$ insert/max/extract max

Heaps- A complete binary tree (reading top to bottom left to right, no holes) that maintains the 'heap property': for a node x, the priority of node x is higher than the priority of either of x's children. Thus the highest priority node is the root of the tree. We can extract-max by swapping the root for the last leaf and 'bubbling-down' the new root into its rightful place. Similarly we can insert a new last leaf and 'bubble up' that last leaf to it rightful place. While the balanced binary tree dominates heaps in operations, heaps are more memory compact.

- *Priority Queue:* $\Theta(1)$ create/max, $\Theta(log(n))$ insert/extract-max, $\Theta(n)$ search.

Hash Tables- We store n elements in a m-slot array T. Define a $hash\ function\ h : K \rightarrow \{1, ..., m\}$, where K is the keyspace (set of all possible keys), and h "looks random". Element x will be stored in a linked list in $T[h(x)]$. (This is "hashing with chaining": we resolve "collisions"–that is, two keys that hash to the same slot– by creating a linked list of the elements in that cell.) To look up element y, we perform linear search on the linked list $T[h(y)]$. The benefit of hashing: if we have a good hash function that spreads out the keys close to uniformly over the cells of the table, then each linked list will be short, and we'll get excellent performance. The disadvantage of hashing: if we are asked to sort a particularly bad set of keys for our h, then we'll merely replicate the performance of an unlinked linear list.

- *Dictionary:* $\Theta(1)$ create; AVERAGE-CASE: $\Theta(1)$ lookup/insert/delete, but WORST-CASE: $\Theta(n)$ lookup/insert/delete. Even the average-case running time is $\Theta(n)$ for min/max though.

Union Find- A data structure that maintains the connected components in a graph and supports unioning of these components–as though edges were being added to the graph. Thus data structure is very useful for component-based algorithms like Kruskal's.

- *Record Pointers:* $O(n)$ create with data; $O(log(n))$ Find(u) that returns the name of the component containing node u, and $O(log(n))$ Union(A, B) to merge the A and B components into a single set.

Chapter 3

Deterministic Computation

§3.1 O-Notation

Algorithms are step-by-step processes which, in practice, vary wildly in their resource usage (time, space, etc.) based on the particular problem instance under consideration. Some of this variation is due to the size of the instance (in general, bigger instances use more resources), some to algorithm-specific properties of the instances. While it's easy to account for the former, the latter requires an overwhelming verbosity to accurately capture–a price that is, ultimately, far too high. O-notation, our means of quantifying the resource usage of an algorithm, is thus defined as a worst-case analysis, as always giving bounds which hold for even the worst instances of a given size.

Having left the size of instances as an open parameter, our notation naturally falls into a functional representation taking a variable, normally n, for the size of the input. More generally, the resource usage of an algorithm will be captured by functions from $\mathbb{N} \to \mathbb{N}$, e.g. n, $n^2 + 3$, and 3^n. Later, it will be useful to utilize real-valued functions like $\sqrt{n}$ and $\log(n)$; appearances to the contrary, these functions too are to be thought of as functions over $\mathbb{N}$, namely $\max\{\lceil f(n) \rceil, 0\}$ for a real-valued f. Of course, rather than try to give a function which perfectly represents a given algorithm's usage of a resource–a tall order–our characterization captures resource usage in a more general sense. For two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, $f(n) = O(g(n))$ (pronounced, "$f(n)$ is big-oh of $g(n)$" or "f is of the order of g") if and only if there are $n_0, c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$. Informally, $f(n) = O(g(n))$ means that f eventually uses the resource no faster than g, modulo a constant. Changing emphasis, this is also notated $g(n) = \Omega(f(n))$. Finally, if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we write $f(n) = \Theta(g(n))$, meaning f and g have the exact same rate of usage.

Given our work thus far, it's easy to show that–for any $p \in \mathbb{N}$, $r > 1$, and resource,

$$O(\log^p(n)) < O(n^p) < O(r^n)$$

Complexity theorists and computer scientists often speak of problems as tractable or intractable and algorithms as efficient or inefficient; the precise meaning of these terms depends on the resource under consideration–although, later, we'll see that they are not so disconnected as they first appear. The meanings for space and time are given below:

(Time) Efficient
An algorithm is *efficient* if and only if it has a polynomial time bound; that is, its runtime is $O(n^p)$ for some $p \in \mathbb{N}$.

(Space) Efficient
An algorithm is *efficient* if and only if it has a logarithmic space bound; that is, its runtime is $O(\log^p(n))$ for some $p \in \mathbb{N}$.

(Time) Intractable
A problem for which no efficient algorithm exists is *intractable*

(Space) Intractable
A problem for which no space efficient algorithm exists is *intractable*

Properties

- Transitivity:

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

- Let k be a fixed constant, and let $f_1, f_2, ..., f_k$ and h be functions such that $f_i = O(h)$ for all i. Then, $f_1, f_2, ..., f_k = O(h)$.
- Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$.
- Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.
- For every $b > 1$ and every $x > 0$, we have $log_b(n) = O(n^x)$. Note that the base b is often dropped because $log_a(n) = \frac{log_b(n)}{log_b(a)}$, a constant change.
- For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.
- The cost of storing a number of length n is $log(n)$.

Some particularly tricky problems may require ingenious functions, say $f(n) = n^3$ on even n, but only 1 on odd. It follows that $f(n) \neq O(n^2)$ and $f(n) \neq \Omega(n^2)$.

§3.2 Models of Computation

It's well-accepted that Turing machines are a satisfactory model of computation, but–here–our concern is much more specialized than just computation generally. We wish to characterize the relative speed with which various algorithms use a particular resource; it is by no means obvious that Turing machines provide a stable foundation upon which to build, that choosing a different model of computation may not, for instance, make a Turing intractable problem suddenly tractable.

In truth, however, this seems to not be the case. Taking Turing machines as our model of computation, we define–for instance–the length of a computation in the obvious way via steps taken by the Turing machine, taking the size parameter n in our O, Ω, Θ notation to be the length of the input string. Defining the same notions across different models of computation, none yet has been found for which the difference in bounds is not polynomial, e.g. a $O(f(n))$ algorithm under one model always becomes an $O(f(n)^2)$ algorithm under the other. This has lead to a quantitative extension of Church's much beloved thesis: *any reasonable model of computation and the time performance thereof gives a model that is equivalent to Turing machines within a polynomial.* Put slightly differently, so long as our classes are closed under polynomial changes in runtime, we may divide problems into *complexity classes* based on the runtimes in which they can be solved, independent of the actual model of computation in use.

Of course, it remains to decide what kind of objects problems should be taken to be; naïvely (but naturally!) we may wish to take problems to simply be the set of their 'yes' instances (ignore, for the moment, that not all problems are decision problems). Thus, e.g., Reachability will be the set of all triples $\langle G, 1, n\rangle$ for which node 1 can reach node n in graph G. The immediate downside to this suggestion is, of course, twofold:

(1) Not all problems are triples containing a graph and two of its nodes; in general, problems will take an infinite number of different forms with no common features

(2) Most models of computation do not take graphs, flows, networks, etc. as inputs; the form of the problem is thus misleading since, by necessity, the Turing machine (or whatever model of computation is in use) is not really working with the graph or nodes.

Our actual choice is quite close to the naïve suggestion above; instead of allowing actual abstract objects like graphs, nodes, networks, etc. all problems are sets of strings over some alphabet Σ; that is, for some Σ, any problem P is a subset of Σ^*.[1]

> **Syntactic Complexity Class**
> A set of problems forms a *syntactic complexity class* if and only if it is the set of problems that can be solved by abstract machines of some particular type M using $O(f(n))$ of some specific resource(s) R for some $f \in \Gamma$ where Γ is a polynomially-closed set of functions from $\mathbb{N}$ to $\mathbb{N}$ and n is the size of the input

[2] [3]

3.2.1 Space Bounds

As we saw earlier, quantifying the time used by a particular model of computation is usually a rather simple task; on the surface, space seems just as easy. Consider the case of Turing machines. An obvious and natural suggestion is to take the space required by the machine on a particular input string s to be the maximum number of cells in the tape used throughout the computation. Contra intuition, however, this definition is taken to represent a serious overcharge for many algorithms by complexity theorists.

The problem is that the Turing machine is being charged–regardless of how little or how much space is used in the actual computation–for both the space necessary to store the input string and the space necessary to represent the output, collapsing an distinction between algorithms that use only, for example, $\Theta(log(n))$ space for the rest of the computation and algorithms which use $\Theta(n^4)$. To fix this, we need a means of accurately calculating how much space is needed for the actual computation; one which rules out, for example, algorithms which attempt to achieve artificially low space bounds by simply overwriting and using the cells containing the initial input.

[1] A similar move is made in computability theory proper, but rather than arbitrary alphabets Σ, we restrict to $\mathbb{N}$...why isn't the same done here?

[2] Various authors define complexity classes in a manner which allows for them to fail to be closed under polynomial changes; this seems strange since (1) they will change based on the model of computation and (2) such classes seem to never be considered

[3] Arbitrary Linear Speedup grants justification for O notation...but if we want complexity classes to be stable across models of computation this is already enough to justify its use

The solution, as it will often be, is to shift our model of computation to one which is both similar and provably equivalent, but which makes the resource quantification easy. In this case, we introduce Turing machines with three tapes: the first a read-only tape upon which the input is written to start the computation, the second a permanent, write-only tape which the final output is taken from, and the third a read/write tape which operates as normal. Space usage is then defined as the maximum number of cells used on the read/write tape.

The general point here is twofold; in the particular case of space as a resource, algorithms shouldn't be charged for the space used to store the initial input, nor the space used to write the final output. Instead, merely the space used to turn the former into the latter. More generally, however, space gives a paradigmatic example of how resource quantification problems can be solved; namely by constructing a new similar, equivalent-up-to-a-polynomial model of computation which makes the resource consumption explicit and easily quantified.

3.2.2 On Decision Problems

It's easy to in the following deluge of notation and results, but our topic matter will, shortly, narrow significantly; rather than all problems, the notation and definitions below prioritize the consideration of decision problems (tasks in which everything is either a 'yes' or 'no' answer), problems that cannot be rephrased into a decision variant will be forcibly shunted to the side. This isn't, of course, a particularly happy situation, and we will eventually return to these non-decision problems. Nevertheless, this choice allows for easy definitions as well as a host of foundational results.

§3.3 The Deterministic Hierarchies

Proper Complexity Functions
a function $f : \mathbb{N} \to \mathbb{N}$ is a *proper complexity function* if and only if it is both non-decreasing and there is an algorithm which takes a string of any length n as input and returns a string of length $f(n)$ using time $O(n + f(n))$ and space $O(f(n))$

► **Proposition 3.1.**
Suppose a deterministic (or, later, a non-deterministic) algorithm M decides a set $X \subseteq \mathbb{N}$ within time/space $f(n)$, where f is a proper complexity function. Then, there is an algorithm which decides the language in time/space $O(f(n))$, halting precisely at $f(n)$ time/space.

Fixing a model of computation, we introduce the following notation:

■ **Notation.** For a proper complexity function f, define $\textsc{dtime}(f)/\textsc{dspace}(f)$ as the class of problems decidable by a deterministic model of computation in $O(f)$ time / space. Similarly, we will often use k to represent a union of functions over $\mathbb{N}$, e.g.

Deterministic Time Complexity Classes	Deterministic Space Complexity Classes
$\mathbf{LTIME} = \textsc{dtime}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \textsc{dtime}(\log^m(n))$	$\mathbf{L} = \textsc{dspace}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \textsc{dspace}(\log^m(n))$
$\mathcal{P} = \mathbf{PTIME} = \textsc{dtime}(n^k) = \bigcup_{m\in\mathbb{N}} \textsc{dtime}(n^m)$	$\mathbf{PSPACE} = \textsc{dspace}(n^k) = \bigcup_{m\in\mathbb{N}} \textsc{dspace}(n^m)$
$\mathbf{EXPTIME} = \textsc{dtime}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \textsc{dtime}(2^{n^m})$	$\mathbf{EXPSPACE} = \textsc{dspace}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \textsc{dspace}(2^{n^m})$
$\mathbf{2\text{-}EXPTIME} = \textsc{dtime}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \textsc{dtime}(2^{2^{n^m}})$	$\mathbf{2\text{-}EXPSPACE} = \textsc{dspace}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \textsc{dspace}(2^{2^{n^m}})$

In general, it will be helpful to refer to the complements of various decision problems; that is, the decision problem for which all 'no' instances of the original are 'yes' instances and vice versa.

■ **Notation.** Given a decision problem $\textsc{Problem}$, its complement will be denoted $\overline{\textsc{Problem}}$. Similarly, we will often speak of the complements of a complexity class: Given a complexity class C, its complement will be denoted '$\mathbf{co}C$' and defined by $\{A \subseteq \Sigma^* \;:\; \overline{A} \in C\}$

► **Proposition 3.2** (Deterministic Closure under Complementation)**.**
Any deterministic time or space complexity class is closed under complementation

---

3.3.1 The Deterministic Time Hierarchy Theorem

Define, for any class of abstract computing machines,

$$H_f := \{\langle M, x\rangle \;:\; M_{e,f(x)}(x)\downarrow\}$$

► **Lemma 3.1.**
If $f(n) \geq n$ and the class of computing machines is multi-tape Turing machines, then $H_f \in \textsc{dtime}(f(n)^3)$

Proof Sketch.

► **Lemma 3.2.**
If $f(n) \geq n$ and the class of computing machines is multi-tape Turing machines, then $H_f \notin \textbf{DTIME}(f(\lfloor \frac{n}{2} \rfloor))$

Proof Sketch.
Suppose it is; define

► **Theorem 3.1** (The Time Hierarchy Theorem)**.**
If $f(n) \geq n$ is a proper complexity function, then the class $\textbf{DTIME}(f(n))$ relative to multi-tape Turing machines is strictly contained within $\textbf{DTIME}((f(2n+1))^3)$ relative to multi-tape Turing machines

---

Of course, being relative to multi-tape Turing machines, the results above might at first appear to be of limited interest; recall, however, the models of computation differ by at most a polynomial amount with respect to one another. Thus,

---

► **Corollary** ($\mathcal{P} \subset$ **EXPTIME**)**.**
The complexity class $\mathcal{P}$ is a proper subset of **EXPTIME**

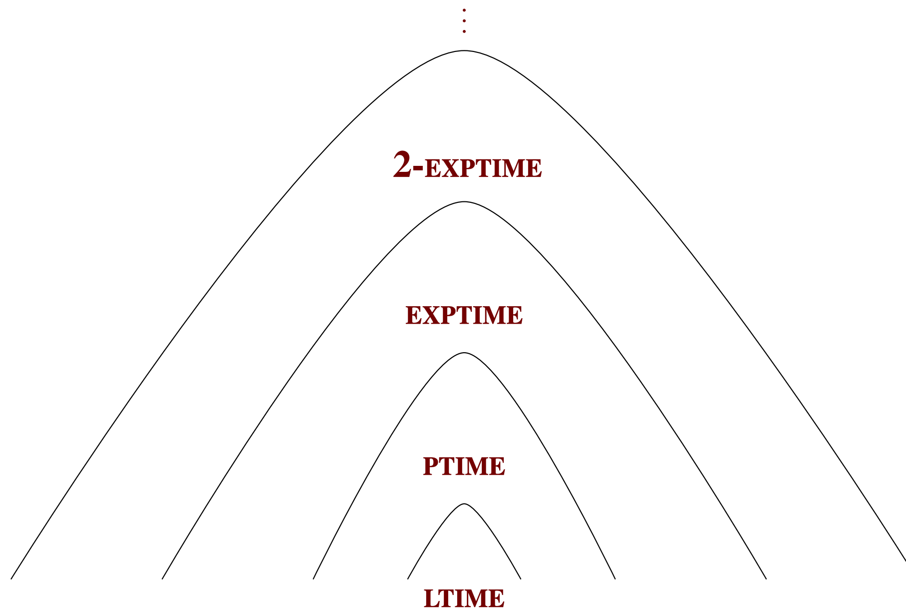


Figure 3.1: Deterministic Time Hierarchy

3.3.2 The Deterministic Space Hierarchy Theorem

Mimicking the lemmas established above for time, we can also show that:

► **Theorem 3.2** (The Space Hierarchy Theorem)**.**
If f is a proper complexity function, then DSPACE$(f(n))$ is a proper subset of DSPACE$(f(n)\log(f(n)))$

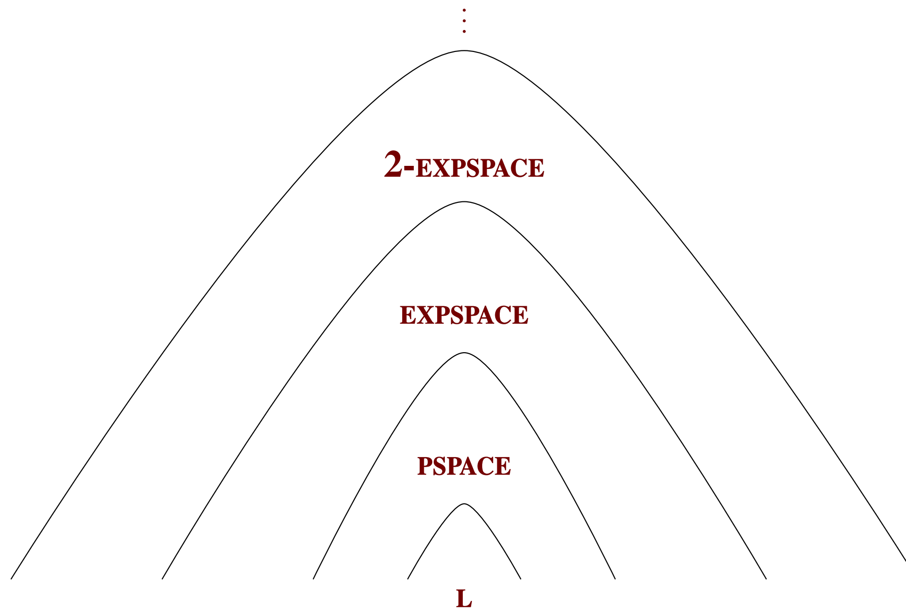


Figure 3.2: Deterministic Space Hierarchy

§3.4 Classifying Problems

Contra much of the presentation up until this point, not all complexity classes are created equal; practical problems aren't evenly distributed across the deterministic hierarchies, and so particular classes are of considerably more interest than others.

3.4.1 Boolean Logic

Formal logic is, unsurprisingly, an ideal area in which to construct problems; we survey a few basic results in this area.

SAT
 Given: a formula of boolean logic in conjunctive normal form
 Return: whether or not the formula is satisfiable

► **Proposition 3.3** (SAT ∈ **EXPTIME**).
SAT is solvable by deterministic models of computation in exponential time; that is, SAT ∈ **EXPTIME**

Proof Sketch.

EXPSAT(φ):

```
1 for each possible assignment do
2     if φ is true under the assignment:
3         return True
4 return False
```

■ **Fact.** Whether or not SAT is in $\mathcal{P}$ is an open problem, although it is heavily suspected that it is not.

While SAT is in **EXPTIME**, a special, but still useful case proves considerably easier.

Horn Clause
A clause containing at most one positive (non-negated) literal

HORNSAT
 Given: a formula of boolean logic in conjunctive normal form all of whose clauses are Horn clauses
 Return: whether or not the formula is satisfiable

► **Proposition 3.4** (HORNSAT ∈ $\mathcal{P}$).
HORNSAT is solvable by deterministic models of computation in polynomial time; that is, HORNSAT ∈ **PTIME**

3.4.2 Boolean Circuits

Alternatively, boolean formulas and the problem of satisfiability can be conceived of as, surprisingly enough, a problem over circuits:

Boolean Function
An n-ary boolean function f is a function $f : \{True, False\}^n \to \{True, False\}$

It's obvious that any given formula can be associated with the boolean function which it defines; similarly, it's easy to recast boolean functions into a circuit representation:

Boolean Circuit
A boolean circuit is a graph $C = (V, E)$ where

(i) the graph has no cycles (and so we may assume edges always go from lower numbered nodes to higher)

(ii) every node i has a sort $s(i)$ such that $s(i) \in \{\wedge, \vee, \neg, \top, \bot\} \cup \{x_1, x_2, \dots\}$

  (a) For any node i, if $s(i) \in \{\top, \bot\} \cup \{x_1, x_2, \dots\}$, the indegree of i is 0

  (b) For any node i, if $s(i) \in \{\neg\}$, the indegree of i is 1

  (c) For any node i, if $s(i) \in \{\wedge, \vee\}$, the indegree of i is 2

(iii) (Optional) there is only one node with outdegree 0

In this context, nodes are called *gates*; nodes with indegree 0 *inputs*. Nodes with an outdegree of 0 are *outputs*.

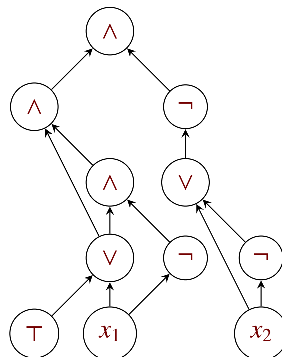


Figure 3.3: A Boolean Circuit

> CIRCUIT SAT
> Given: a Boolean circuit C
> Return: whether or not there is an assignment of truth values to the inputs for which C outputs true

It's easy to see that CIRCUIT SAT is on par with SAT, and so we expect it to lie in **EXPTIME**. Consider, however, a slightly easier variation:

> CIRCUIT VALUE
> Given: a Boolean circuit C with no variable gates
> Return: whether or not C outputs true

► **Proposition 3.5** (Circuit Value∈ **PTIME**, **L**).
Circuit Value lies in both **PTIME** and **L**

Proof Sketch.
For the time bound,

CircuitValue(C):

```
1 for i from 1 to n do
2     Compute the value of node i; if node i is true and has outdegree == 0:
3         return True
4 return False
```

and for the space bound,

CircuitValueRec(C):

```
1 return FindValue(C, n)
```

FindValue(C,gate):

```
1 if indegree(gate) == 0:
2     return value of gate
3 if indegree(gate) == 1:
4     return not(FindValue(C, input gate))
5 if sort(gate) == ∧:
6     return FindValue(C, input gate 1) AND FindValue(C, input gate 2)
7 else:
8     return FindValue(C, input gate 1) OR FindValue(C, input gate 2)
```

3.4.3 Graph Theory

The first-order language of graph theory, $\mathcal{L}_G$, has signature $\langle G \rangle$ consisting in only a single binary relation symbol; this relation is–intuitively–taken to assert that there exists an edge from its first input to its second. As usual, any sentence $\varphi \in \mathcal{L}_G$ represents a property of graphs and can be viewed as giving rise to a decision problem. Rather than name these individually, we take them as instances of a larger decision problem:

φ-Graphs
    Given: a sentence $\varphi \in \mathcal{L}_G$ and a model M of $\mathcal{L}_G$
    Return: whether or not $M \Vdash \varphi$

► **Theorem 3.3** (φ-Graphs $\in$ **PTIME**).
φ-Graphs is solvable in polynomial time; that is, φ-Graphs $\in$ **PTIME**

Proof Sketch.
Induction on formulas

► **Theorem 3.4** (φ-Graphs $\in$ **L**).
φ-Graphs is solvable in $O(log(n))$ space; that is, φ-Graphs $\in$ **L**

Proof Sketch.

The results above are, quite clearly, powerful; classifying a host of decision problems in one sweep. Unfortunately, the assumption that $\varphi \in \mathcal{L}_G$ limits us more than it might, at first, seem to:

► **Theorem 3.5** (Reachability is not Definable in $\mathcal{L}_G$)**.**
There is no $\varphi(x, y) \in \mathcal{L}_G$ such that $\varphi(x, y)$ defines Reachability from x to y.

Reachability is, however, in **PTIME**, so–perhaps–there is a natural extension of first-order logic which correctly captures all and only those problem which are in **PTIME**. The most natural candidate is, of course, second-order logic. It's easy to show that Reachability is, in fact, definable in the second-order equivalent of $\mathcal{L}_G$; unfortunately, it's just as easy to show that Hamilton Path–a problem suspected to not reside in **PTIME**–is also definable.

Hamilton Path
Given: a graph G
Return: a path on G which visits each node exactly once

Luckily, we've already encountered a natural class of formulas which serve to reduce the complexity of the problems expressible:

► **Theorem 3.6.**
For any existential, Horn, second-order sentence $\exists P\varphi$, the corresponding decision problem $\exists P\varphi$-Graphs is in **PTIME**

---

[4]

Proof Sketch.

Moving to undirected graphs, we have a problem familiar to anyone who's taken an algorithms course:

Min Cut
Given: an undirected graph G and a goal K
Return: whether or not there exists a cut (partition of V) of value less than of equal to K

Here, as there, the solution is to leverage one of the polynomial time maximum flow algorithms:

► **Proposition 3.6.**
Min Cut ∈ **PTIME**

Proof Sketch.
Pick a node in G and run a maximum flow algorithm to every other node; the smallest such flow gives the minimum cut in the normal way.

[4] Pg. 116, why do we only need to consider a string of universals?

3.4.4 Sets and Numbers

LINEAR PROGRAMMING
Given: a system of linear inequalities in n variables with integer coefficients
Return: whether or not the system has a solution

► Proposition 3.7.
LINEAR PROGRAMMING $\in$ **PTIME**

---

§3.5 Reduction

(Efficient) Reduction
Given two sets L_1 and L_2, L_1 is reducible to L_2 if and only if there is a function R deterministically computable in $O(\log(n))$ space such that, for any x,

$$x \in L_1 \Leftrightarrow R(x) \in L_2$$

► **Proposition 3.8.**
$\mathcal{P}$, **L**, **PSPACE**, and **EXPTIME** are closed under reduction

---

Proof Sketch.
Suppose not; it's easy to show that the corresponding classes collapse into one

3.5.1 PTIME-Complete Problems

► **Theorem 3.7** (CIRCUIT VALUE is **PTIME**-Complete)**.**
The problem CIRCUIT VALUE is **PTIME**-Complete

Proof Sketch.
By earlier, CIRCUIT VALUE $\in$ **PTIME**. Let A be an arbitrary **PTIME**-decidable set over some alphabet Σ. By definition, then, it is decided by some particular deterministic model of computation M (pick your favorite type). Unfortunately, it's now necessary to delve into the implementation / construction of M and design an **L** algorithm which encodes M into a circuit. For the particular case of multi-string Turing machines, see Papadimitriou pg. 170.

After proving Fagin's theorem (see **NPTIME** complete problems), it's tempting to think that the converse of our result for **PTIME** holds as well; unfortunately, this is not the case. In particular, restricting to horn clauses engenders a fundamental lack of expressibility, the inability to define a successor function and thus count. To achieve a parallel result for **PTIME**, however, adding this to our language proves sufficient to overcome the difficulties:

► **Theorem 3.8.**
The class of all graph-theoretic properties expressible in Horn existential second-order logic with successor is precisely $\mathcal{P}$

Proof Sketch.
Identical to Fagin's theorem

► **Corollary** (HORN SAT is **PTIME**-Complete)**.**
The problem HORN SAT is **PTIME**-Complete

Proof Sketch.
This result follows from proving that **PTIME** is exactly the class of problems...and I don't quite get this

[5]

[5] This doesn't really make sense to me; is it that any **PTIME** problem is equivalent to some **PTIME** problem involving φ-graphs?

Chapter 4

Nondeterminism

§4.1 Nondeterministic Machines

Reasonable models of computation are–surprisingly–not the only models of computation which hold considerable interest to complexity theorists; indeed, many of the most famous problems in complexity theory are simply to give a rigorous proof that abstract machines of such and such a type are, up to a polynomial, more/less/just as efficient as some more traditional model of computation. The most famous of these alternative machines are the nondeterministic variants on the traditional models of computation.

By construction, the traditional models of computation are deterministic; a given input always generates precisely the same behavior, either failing to halt or both halting and returning some specific output. Introducing nondeterminism is simply to break this tight connection between input and output, allowing the machine/function/what-have-you to sometimes behave one way and sometimes behave another for no reason whatever. Such a non-deterministic machine is said to accept an instance of a decision problem if and only if some sequence of non-deterministic choices causes the machine to accept (even if others do not). The complexity of these non-deterministic machines, however, is still defined as an upperbound on any given computation. At face value, this seems to represent a serious undercharge; non-deterministic machines are, just like deterministic machines, charged resources based only on the overall 'depth' of their computational activity. Unlike their deterministic counterparts, however, nondeterministic machines have the capability to branch, to explore different computation paths while only accruing costs based on the longest such path. The general picture, then, is as follows:

Figure 4.1: Deterministic Computation

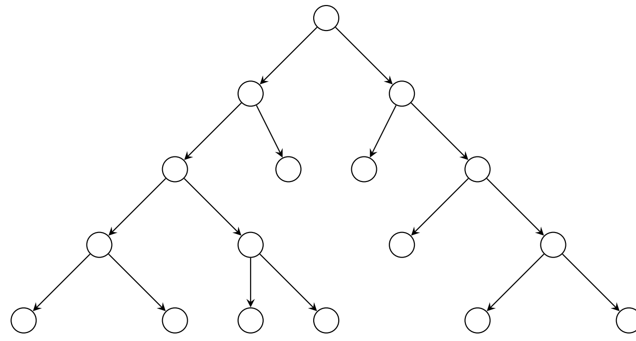


Figure 4.2: Nondeterministic Computation

§4.2 Nondeterministic Complexity Classes

Adding nondeterminism to a standard model of computation produces a new class of abstract machines–allowing, as earlier, the definition of NTIME and NSPACE.

> **NTIME/NSPACE**
> For a proper complexity function f, define $\text{NTIME}(f)/\text{NSPACE}(f)$ as the class of problems solvable by a nondeterministic model of computation in $O(f)$ time / space.

This, as before, gives rise to a sequence of new, nondeterministic complexity classes:

Nondeterministic Time Complexity Classes	Nondeterministic Space Complexity Classes
$\textbf{NLTIME} = \text{NTIME}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \text{NTIME}(\log^m(n))$	$\textbf{NLSPACE} = \text{NSPACE}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \text{NSPACE}(\log^m(n))$
$\textbf{NP} = \textbf{NPTIME} = \text{NTIME}(n^k) = \bigcup_{m\in\mathbb{N}} \text{NTIME}(n^m)$	$\textbf{NPSPACE} = \text{NSPACE}(n^k) = \bigcup_{m\in\mathbb{N}} \text{NSPACE}(n^m)$
$\textbf{NEXPTIME} = \text{NTIME}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \text{NTIME}(2^{n^m})$	$\textbf{NEXPSPACE} = \text{NSPACE}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \text{NSPACE}(2^{n^m})$
$\textbf{2-NEXPTIME} = \text{NTIME}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \text{NTIME}(2^{2^{n^m}})$	$\textbf{2-NEXPSPACE} = \text{NSPACE}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \text{NSPACE}(2^{2^{n^m}})$

4.2.1 Nondeterministic Hierarchy Theorems

► **Theorem 4.1** (Nondeterministic Time Hierarchy Theorem)**.**
For any two proper complexity functions f and g such that $f(n+1)$ is $O(g(n))$, then there is a problem which is in $\text{NTIME}(g(n))$, but not $\text{NTIME}(f(n))$

Proof Sketch.

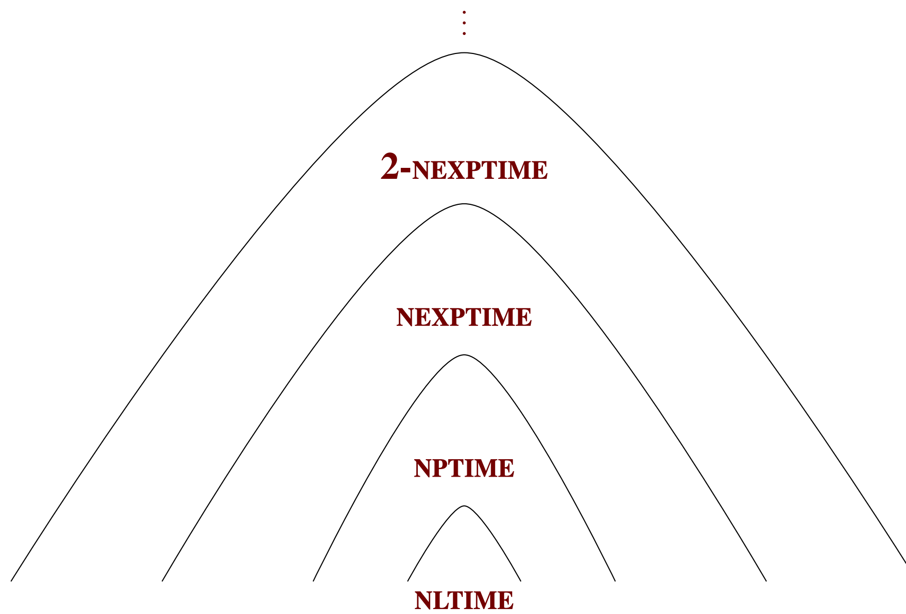


Figure 4.3: Nondeterministic Time Hierarchy

► **Theorem 4.2** (Nondeterministic Space Hierarchy Theorem)**.**
For any proper complexity function f, $\text{NSPACE}(f(n)) \subset \text{NSPACE}(\log(f(n))f(n))$

Proof Sketch.

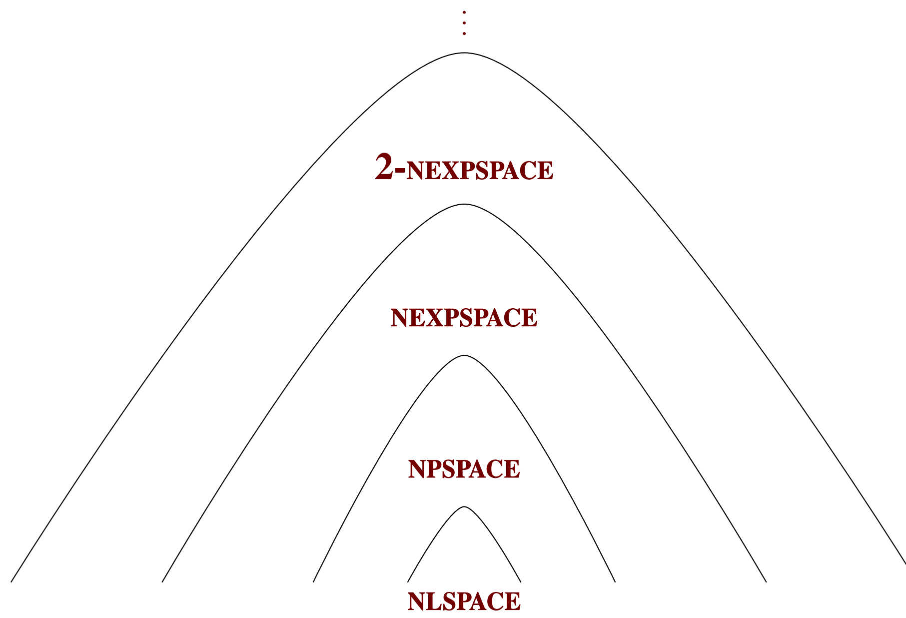


Figure 4.4: Nondeterministic Space Hierarchy

§4.3 Classifying Problems

Continuing our cubbyhole mathematics from the previous chapter, we now have the capability to locate a host of natural computational problems in our new, nondeterministic hierarchies.

4.3.1 Boolean Logic

Turing first to one of our most fundamental problems,

► Proposition 4.1 (SAT ∈ **NPTIME**, **NPSPACE**).
SAT is solvable by nondeterministic models of computation in polynomial time and in polynomial space; that is, SAT ∈ **NPTIME** and SAT ∈ **NPSPACE**

Proof Sketch.
A single algorithm suffices to prove both:

NSAT(φ):

```
1 for each new literal encountered do
2 |  Nondeterministically select T or F;
3 if the assignment satisfies φ:
4 |  return True
5 else:
6 |  return False
```

4.3.2 Graph Theory

Similarly,

► Proposition 4.2.
HAMILTON PATH is in **NPTIME** and **NPSPACE**

Proof Sketch.

NHamiltonPath(G):

```
1 for i from 1 to n do
2 |  Nondeterministically select the next node in the path;
3 if no node appears more than once in the path:
4 |  return True
5 else:
6 |  return False
```

Our previous result connecting second-order logic and **PTIME** can also be extended to establish **NPTIME** as an upperbound on the complexity of decision problems generated from second-order sentences of a particular type:

► Theorem 4.3.
For any existential, second-order sentence $\exists P\varphi$ from the second-order language of graphs, the problem $\exists P\varphi$-GRAPHS is in **NPTIME**

Proof Sketch.
The algorithm guesses non-deterministically an extension for P after which the problem is reduced to a first-order φ-GRAPHS instance

Shifting into the domain of undirected graphs ($G = (V, E)$ where E contains unordered pairs) gives a host of natural computational problems:

INDEPENDENT SET
Given: an undirected graph G and a goal K
Return: whether or not there is a independent set (set of vertices unconnected by edges) of size K

CLIQUE
Given: an undirected graph G and a goal K
Return: whether or not there is a clique (set of vertices with all possible edges between them) of size K

NODE COVER
Given: an undirected graph G and a goal K
Return: whether or not there is a node cover (set of vertices such that any edge of G is connected to at least one) of size K

A bit of thought shows that all three of these problems are inextricably related; any instance of CLIGUE on a graph G is an instance of INDEPENDENT SET on the inverted graph G^{-1}. Similarly, I is a maximal independent set if and only if $V - I$ is a minimal node cover. It's easy, then, to establish the following sequence of results:

► **Proposition 4.3.**
INDEPENDENT SET $\in$ **NPTIME**, **NPSPACE**

► **Corollary.**
CLIQUE $\in$ **NPTIME**, **NPSPACE**

► **Corollary.**
NODE COVER $\in$ **NPTIME**, **NPSPACE**

4.3.3 Numbers and Sets

To continue the trend established thus far, there are a large number of problems residing in **NPTIME** and **NPSPACE** involving sets and numbers; a brief listing is given below:
[1]

TRIPARTITE MATCHING
Given: three sets of size n–A, B, C– and a ternary relation $T \subseteq A \times B \times C$
Return: a set of size n containing triples from T, no two of which have a component in common

[1] Are these supposed to be decision problems?

Set Covering

`Given:` a goal K and a family of sets $F = \{S_1, \ldots, S_n\}$ all of which are subsets of a finite set U

`Return:` a set of K sets whose union is U

Exact Cover by 3-Sets

`Given:` a family of sets $F = \{S_1, \ldots, S_n\}$ all of which contain 3 elements and are subsets of a finite set U of size $3m$ for $m \in \mathbb{N}$

`Return:` a set of n sets whose union is U

Set Packing

`Given:` a goal K and a family of sets $F = \{S_1, \ldots, S_n\}$ all of which are subsets of a finite set U

`Return:` a set of K pairwise disjoint sets

Integer Programming

`Given:` a system of linear inequalities in n variables with integer coefficients

`Return:` whether or not the system has an integer solution

Knapsack

`Given:` a weight limit W and a set of n items each with a positive weight w_i and positive value v_i

`Return:` a set of items which gives the highest value possible while respecting the weight limit

Bin Packing

`Given:` a number of bins B, a capacity for the bins C, and a set of n natural numbers $a_1, \ldots, a_n$

`Return:` whether or not the numbers can be partitioned among the bins in a way that respects the capacity constraint C

§4.4 Unifying the Hierarchies

At present, we have constructed four distinct hierarchies and entirely ignored the possibility of crossing resource / model of computation lines and relating classes in one to those of another–in point of fact, this is one of the primary concerns of complexity theory.

4.4.1 The Reachability Method

One of the primary proof techniques for proving cross-hierarchy relations is to translate a problem in one class into an instance of a problem known to lie in the other; REACHABILITY is particularly well-suited for this, as we show below.

► **Theorem 4.4** (Relations Between Classes)**.**
Suppose that f is a proper complexity function. Then, for any reasonable model of computation, we have $(i) - (iii)$ along with a variation on (iv):

(i) $\textbf{DTIME}(f(n)) \subseteq \textbf{NTIME}(f(n))$

(ii) $\textbf{DSPACE}(f(n)) \subseteq \textbf{NSPACE}(f(n))$

(iii) $\textbf{NTIME}(f(n)) \subseteq \textbf{DSPACE}(f(n))$

(iv) $\textbf{NSPACE}(f(n)) \subseteq \textbf{DTIME}(c^{\log(n)+f(n)})$

Proof Sketch.
Parts (i) and (ii) are trivial; let $A \subseteq \mathbb{N}$ be an element of $\textbf{NTIME}(f(n))$. By definition, there exists a non-deterministic machine M which decides A; we use it to construct a deterministic M' deciding A in $O(f(n))$ space. Our general strategy is to decide A via exhaustive use of M'. Assuming a reasonable implementation of non-determinism, M can not only generate a sequence of non-deterministic choices for M (the length of the sequence must be $O(f(n))$ since M is), it can list out all possible such sequences–deleting each from memory before writing the next. Note next that, given a sequence of choices, the space necessary to run M on those choices is $O(f(n))$ since only $O(f(n))$ characters can be written in $O(f(n))$ time. Putting these together, we may run M on all possible sequences of choices all the while using $O(f(n))$ space–just as required.

For (iv), we demonstrate with a k-string nondeterministic machine M with dedicated input and output tapes. Recall that the configuration of a Turing machine on input x is a snapshot of its computation on x at some point in time; for a Turing machine as described above, any configuration can be represented as a number $\leq n$ and a tuple of strings with length $2k+1$, each entry of which is at most $f(n)$ (the contents of all the tapes). Given a fixed alphabet Σ, there are thus at most $n(2k+1)|\Sigma|^{f(n)}$ possible configurations. Thus, for a suitable choice of c_1 which need only depend on the Turing machine, we have an upperbound of $nc_1^{f(n)} = c_1^{\log(n)+f(n)}$.

Define next the *configuration graph* of M, denoted $G(M, x)$ for input x, by taking the set of nodes to be all possible configurations of M on x and placing an edge between two configurations if and only if M transitions from the former to the latter (a simple procedure). Deciding whether $x \in A$ for some $A \subseteq \mathbb{N}$ is thus equivalent to finding a path from the starting configuration to the accepting one in the defined graph–that is, REACHABILITY. Taking a generous c_2n^2 bound on reachability, we have a bound of $c_2c_1^{2(\log(n)+f(n))}$ which gives the desired bound for $c = c_1^2c_2$.

► **Corollary** (The Complexity Tower).

$$\mathbf{L} \subseteq \mathbf{NLSPACE} \subseteq \mathbf{PTIME} \subseteq \mathbf{NPTIME} \subseteq \mathbf{PSPACE}$$

By the space hierarchy theorem, we know also that $\mathbf{L} \subset \mathbf{PSPACE}$–but where precisely the inclusions are proper is still unknown.

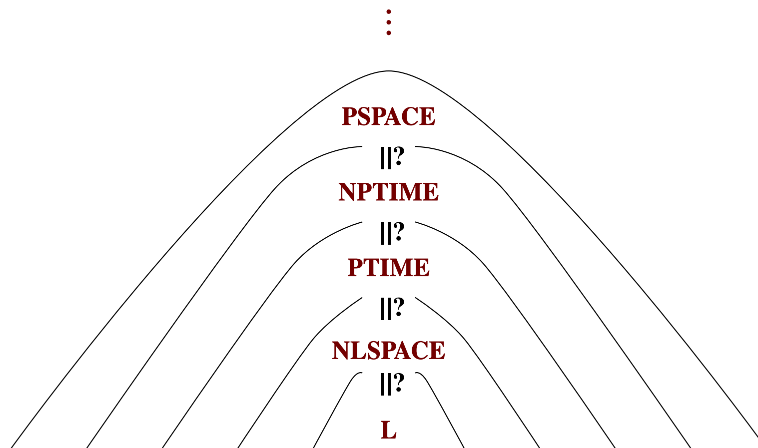


Figure 4.5: The Complexity Tower

4.4.2 Nondeterminism and Space

Thus far, our attempts to combine the hierarchies have been largely symmetric, an observation which encourages the belief the nondeterminism is no more powerful space-wise than time-wise. Interestingly, this belief is likely false; the following sequence of results shows that, pending resolution of $\mathbf{PTIME} = \mathbf{NPTIME}$, adding nondeterminism for space is far weaker than for time. Previously, we established that $\textsc{nspace}(f(n)) \subseteq \textsc{dspace}(k^{\log(n)+f(n)})$–can we do better?

► **Theorem 4.5** (Savitch's Theorem).
REACHABILITY $\in \textsc{dspace}(\log^2(n))$ for multitape Turing machines

Proof.
Let G be a graph with n nodes, x and y nodes of G, and $i \geq 0$.

Path(G,x,y,i)

```
1 if i == 0:
2     if x == y or x and y are adjacent:
3         return True
4 else:
5     for each node z do
6         if Path(G, x, z, i − 1) and Path(G, z, y, i − 1):
7             return True
8 return False
```

A bit of thought shows that $Path(G, x, y, i)$ returns true if and only if there is a path from x to y in G of length at most 2^i–making $i = \log(n)$ sufficient for our purposes. The given algorithm is, as a means of solving REACHABILITY, obviously quite time intensive; it's space requirements, however, are incredibly minimalistic. Note that the only things which must be stored are the inputs for each successive call–$O(\log^2(n))$ at maximum depth. □

► **Corollary.**
NSPACE$(f(n)) \subseteq$ DSPACE$(f^2(n))$ for any proper complexity function $f(n) \geq \log(n)$

Proof Sketch.
Combine the previous results

► **Corollary.**
PSPACE = NPSPACE

That is, nondeterminism adds nothing to the computational power of polynomially bounded space algorithms. Working in another direction,

---

► **Theorem 4.6** (The Immerman-Szelepscényi Theorem)**.**
Given a directed graph G and a node x, the number of nodes reachable from x in G can be computed by a nondeterministic machine within space $\log(n)$

Proof Sketch.

► **Corollary.**
If $f \geq \log(n)$ is a proper complexity function, then NSPACE$((f(n)) =$ coNSPACE$(f(n))$

Proof Sketch.

§4.5 Reduction

The intuition behind reductions are exactly those of computability theory generally; the major difference lies in the difference in constraints on the reduction:

> **Efficient Reduction**
> Given two sets L_1 and L_2, L_1 is reducible to L_2 if and only if there is a function R deterministically computable in $O(\log(n))$ space such that, for any x,
> $$x \in L_1 \Leftrightarrow R(x) \in L_2$$

► **Proposition 4.4** (Closure Under Reduction)**.**
PTIME, **NPTIME**, **coNPTIME**, **L**, **NLSPACE**, **PSPACE**, and **EXPTIME** are closed under reduction

---

Proof Sketch.
Suppose not; note that the relevant classes collapse into one, contra the hierarchy theorems

4.5.1 NPTIME-Complete Problems

Boolean Logic

► **Theorem 4.7** (Cook's Theorem)**.**
SAT is **NPTIME**-complete

Proof Sketch.
By earlier, SAT $\in$ **NPTIME**. It's easy to show that CIRCUIT SAT reduces to SAT. Let A be an arbitrary **NPTIME**-decidable set over some alphabet Σ. By definition, then, it is decided by some particular non-deterministic model of computation M (pick your favorite type). Like CIRCUIT VALUE, our strategy is to design an **L** algorithm R which encodes M into an instance of CIRCUIT SAT. Again, the particular details of the proof vary greatly depending on M; see Papadimitriou pg. 170 for an example using multi-string Turing machines.

The **NPTIME**-complete status of SAT immediately gives rise to a related problem: characterizing where, exactly, the jump from **PTIME** to **NPTIME** occurs (if there is one). This has naturally spawned a number of weakenings and variations on SAT:

> 3-SAT
> `Given:` a Boolean formula φ in CNF with exactly 3 literals in each clause
> `Return:` whether or not φ is satisfiable

> 2-SAT
> `Given:` a Boolean formula φ in CNF with exactly 2 literals in each clause
> `Return:` whether or not φ is satisfiable

Each of the above are special cases of SAT, and thus are members of **NPTIME**, **NPSPACE**. It is not, however, obvious that they don't also reside in a lower class; indeed, it's easy to show that the latter does:

► **Proposition 4.5** (2-SAT $\in$ **PTIME**)**.**
The problem 2-SAT lies in the complexity class **PTIME**

Proof Sketch.
Our strategy is, surprisingly enough, to reduce 2-sat to several instances of reachability:

2SAT(φ):

```
1 for each clause in φ do
2     Add the literals to a set V;
3     Add an edge (α,β) between them if and only if the clause is ¬α ∨ β or β ∨ ¬α;
4 for each unnegated node x in V do
5     if ¬x ∈ V AND Reachable(G,x,¬x):
6         return False
7 return True
```

It now suffices to show that a formula is unsatisfiable if and only if there is at least one path from an x to $\neg x$ in our generated graph, G. This done, we have that 2-SAT $\in$ **NLSPACE**, and thus 2-SAT $\in$ **PTIME**.

By itself, this fact isn't very interesting; however,

► **Proposition 4.6.**
3-SAT is an **NPTIME**-complete problem

Proof Sketch.
A direct reduction of SAT is possible; note that, for example, $p \vee q \vee r \vee s$ can be split into $p \vee q \vee t$ and $r \vee s \vee \neg t$ where t is a new propositional variable appearing nowhere else in the formula.

In other words, the move from two to three literals crosses the (presumed) boundary between **PTIME** and **NPTIME**; exploring the boundary further,

> MAX 2-SAT
> Given: a Boolean formula φ in CNF with 2 literals in each clause and a threshold K
> Return: whether or not there is an assignment which makes at least K clauses in φ true

---

► **Proposition 4.7.**
MAX 2-SAT is **NPTIME**-complete

> NAESAT
> Given: a Boolean formula φ in CNF with exactly 3 literals in each clause
> Return: whether or not there is a truth assignment which doesn't set all three literals in any clause to the same value

---

► **Proposition 4.8.**
NAESAT is **NPTIME**-complete

Graph Theory

Picking up the threads of our earlier work showing second-order existential formulas lie in **NPTIME**,

---

► **Theorem 4.8** (Fagin's Theorem)**.**
The class of all (directed) graph-theoretic properties expressible in existential second-order logic is precisely **NPTIME**

> **Proof Sketch.**
> Since we've already shown that deciding any existential, second-order property lies in **NPTIME**, it only remains to show that any **NPTIME** decidable property can be expressed as an existential second-order formula. The strategy for doing so is simply to show that our second-order logic is expressive enough to describe some class of nondeterministic machines. As usual, this just devolves into the details of particular implementations.

We may also vindicate our assertion that HAMILTON PATH is unlikely to reside in **PTIME**:

► **Proposition 4.9.**
HAMILTON PATH is an **NPTIME**-complete problem

> **Proof Sketch.**
> The obvious nondeterministic algorithm and clever use of 3-SAT will work (Papadimitriou 191)

Turning to undirected graphs, it should come as little surprise that

---

► **Proposition 4.10.**
INDEPENDENT SET is an **NPTIME**-complete problem

> **Proof Sketch.**
> A clever transformation on 3-SAT suffices

► **Corollary.**
CLIQUE is an **NPTIME**-complete problem

► **Corollary.**
NODE COVER is an **NPTIME**-complete problem

Another undirected graph problem, MIN CUT, was encountered classifying problems in the deterministic hierarchies; the corresponding maximization problem shows that minimization and maximization need not be computationally equivalent:

MAX CUT
`Given:` an undirected graph G and a goal K
`Return:` whether or not there exists a cut of value at least K

► **Proposition 4.11.**
MAX CUT is an **NPTIME**-complete problem

Proof Sketch.
The algorithm which nondeterministically choses a set of vertices (and thus a cut) shows MAX CUT ∈ **NPTIME**; hardness can be shown by reducing, for example, NAESAT (Papadimitriou 191)

A simple variation on this problem shows, however, that this need not be the case:

MAX BISECTION
  Given: an undirected graph G and a goal K
  Return: whether or not there exists a cut of value at least K which evenly splits the nodes of G

BISECTION WIDTH
  Given: an undirected graph G and a goal K
  Return: whether or not there exists a cut of value no more than K which evenly splits the nodes of G

► **Proposition 4.12.**
MAX BISECTION is an **NPTIME**-complete problem

Proof Sketch.
A simple trick with MAX CUT suffices

► **Proposition 4.13.**
BISECTION WIDTH is an **NPTIME**-complete problem

Proof Sketch.
Note the connection between MAX BISECTION and the complement of G

Sets and Numbers

Unsurprisingly, each of the problems detailed earlier–TRIPARTITE MATCHING, SET PACKING, SET COVER, EXACT COVER BY 3-SETS, INTEGER PROGRAMMING, KNAPSACK, BIN PACKING–is also **NPTIME**-complete.

4.5.2 Strong NPTIME-Completeness and Pseudpolynomial Algorithms

Strangely enough, not all **NPTIME**-complete problems are created equal; consider, for instance, KNAPSACK. An **NPTIME** algorithm is obvious, but what of a more clever dynamic programming approach? Consider, for instance,

the following:

DynamicKnapsack (W,I,K)

```
1 for w from 1 to W do
2     Define V(w,0) = 0;
3     for i from 0 to |I| - 1 do
4         Set V(i+1, w) = max{V(w,i), v_{i+1} + V(w - w_{i+1}, i)}
5         if V(i+1, w) ≥ K :
6             return True
7 return False
```

[2]

At first glance it might appear that history has been made, that **PTIME** = **NPTIME** has been proven. This is, of course, not quite correct; the complexity of the above is $O(nW)$. In more abstract terms, KNAPSACK is not, as a result of this algorithm, in **PTIME** because the W in the analysis, while contributing $\log(W)$ to n, is not itself polynomially bounded; it may grow exponentially while n grows only polynomially. KNAPSACK is, in a very nontrivial sense, an **NPTIME**-complete problem only because of the W parameter–if it were constrained to polynomial growth, the problem would–provably–be a member of **PTIME**. Problems like KNAPSACK are called, for obvious reasons,*pseudopolynomial*.

> **Strongly NPTIME-Complete**
> A problem is *strongly* **NPTIME**-*complete* if and only if it is **NPTIME**-complete and remains so when we add the constraint that there is a polynomial bound $p(n)$ on the size of the integers appearing in any instance of size n

Given the previous discussion, it should be obvious that KNAPSACK is not strongly **NPTIME**-complete; it is, however, alone in this regard. Every other **NPTIME**-complete problem presented thus far is, in fact, strongly **NPTIME**-complete. Put another way, the computational complexity of the remaining problems doesn't lie in one of two parameters getting out of hand, but rather some difficulty inherent in the problem itself.

[2] He's switched to a budget formulation here

§4.6 NPTIME and coNPTIME

Nondeterminism is fundamentally asymmetric, the criteria for returning a 'yes' and returning a 'no' vastly different. It's unsurprising, then, that the nondeterministic classes in general and **NPTIME** in particular offer our first opportunity to study nontrivial interactions between a complexity class and its complement. Indeed, as we'll soon see, both the asymmetry and prevalence of **NPTIME** problems have a common basis.

4.6.1 NPTIME as Polynomial Certifiability

Thus far, every complexity class encountered has been *syntactic*, firmly based in abstract, formal models of computation rather than some abstract property of a class of machines or problems that one might think to delineate problem complexity by. This latter approach produces *semantic complexity classes*–and, surprisingly enough, our syntactic definition of **NPTIME** corresponds to a very natural one.

For a given alphabet Σ, let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on the strings of the language.

> **Polynomially Decidable**
> A binary relation R is *polynomially decidable* if and only if there is a **PTIME** algorithm deciding it; that is, a **PTIME** algorithm for deciding whether or not $\langle x, y\rangle \in R$ for any tuple $\langle x, y\rangle$

> **Polynomially Balanced**
> A binary relation R is *polynomially balanced* if and only if there is a $k \in \mathbb{N}$ such that for all x and y, if $\langle x, y\rangle \in R$, then $|y| \leq |x|^k$

[3]

► **Proposition 4.14.**
Let $A \subseteq \Sigma^*$; $A \in$ **NP** if and only if there is a polynomially decidable and polynomially balanced relation R such that

$$A = \{x : \exists y[\langle x, y\rangle \in R]\}$$

Proof Sketch.
($\Rightarrow$)
Given an x take y to be an encoded computation (of the relevant **PTIME** algorithm) showing that $x \in A$.

($\Leftarrow$)
Take the algorithm which guesses a y of $\leq |x|^k$, then checks if the guess is correct

Interpretationally, the problems of **NPTIME** all have the property that there is a succinct (read: polynomial) certificate (or witness) for all 'Yes'-instances of the problem. In other words, there is always something (the path required, the satisfying assignment, etc.) that can be supplied which–using only polynomial time–can guarantee the truth of a 'yes' instance. This should not, at first glance, have been at all obvious given the original syntactic definition of **NPTIME**; it does, however, explain why so many natural computational problems lie in **NPTIME**. Anytime a problem is asked to construct some polynomially-complicated abstract object from a finite set possibilities, we have a problem which lies in, at worst, **NPTIME**.

4.6.2 coNPTIME as Polynomial Disqualifiability

As simply the complement to **NPTIME**, the semantic characterization of **NPTIME** carries over readily to **coNPTIME**; just as **NPTIME** represents all problems which have polynomial witnesses to their membership, **coNPTIME** contains

[3] It seems like this k needs to be fixed for the relation; otherwise this is remarkably weak (rewritten to do this)

all problems which have polynomial witness to their non-membership. Perhaps the most basic example of this situation is VALIDITY:

> VALIDITY
> Given: a Boolean formula φ in CNF
> Return: whether or not φ is valid

Note that, in general, while there is no obvious polynomial witness to whether a given φ is valid, there is an obvious polynomial witness which guarantees a given φ is not valid; that is, there is a polynomial *disqualification* to be found in giving a truth assignment which falsifies φ.

► Proposition 4.15.
Let $A \subseteq \Sigma^*$; $A \in$ **coNP** if and only if there is a polynomially decidable and polynomially balanced relation R such that

$$\bar{A} = \{x : \exists y[\langle x, y\rangle \in R]\}$$

It's easy to extend many of our result in **NPTIME** to **coNPTIME**; for example,

---

► Proposition 4.16.
If a set $A \subseteq \Sigma^*$ is **NPTIME**-complete, then its complement $\bar{A} = \Sigma^* - A$ is **coNPTIME**-complete

---

Proof Sketch.
Simply leverage and negate the reduction guaranteed by **NPTIME**-completeness

Indeed, nearly all of our **NPTIME** results either apply directly to **coNPTIME** or do so with an obvious tweak (like switching to universal second-order formulas in Fagin's theorem).

As is the trend, however, the precise relation between **NPTIME** and **coNPTIME** is still unknown; it's conceivable both that **NPTIME** = **coNPTIME** and that the two classes are distinct. This said, the likely situation is the latter, and thus the relationship diagrammed below:

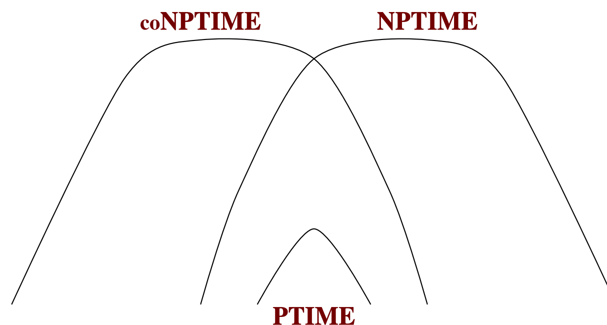


Figure 4.6: **NPTIME** and **coNPTIME** as distinct classes

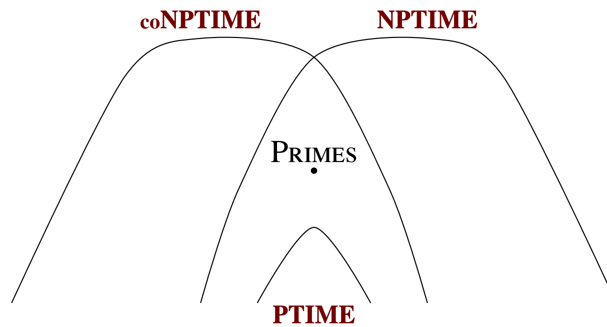
It follows from this picture that both **NPTIME** and **coNPTIME** have their own distinct classes of complete problems and that any overlap occurs below this level.

4.6.3 NPTIME ∩ coNPTIME

Putting our semantic characterizations of **NPTIME** and **coNPTIME** to work, it must be that any problem residing in **NPTIME** ∩ **coNPTIME** has, for every instance, either a polynomial certificate or a polynomial disqualification. Although it is tempting to think so, this doesn't necessarily collapse the problem into $\mathcal{P}$; e.g.,

PRIMES
`Given:` a number N
`Return:` whether or not N is prime

For any N, PRIMES clearly has a polynomial disqualification, namely a number besides 1 and N which divides N. Moreover, some number theory shows that it also has polynomial certificates, albeit of a rather esoteric sort (Papdimitriou 222-227). It follows that PRIMES is an element of **NPTIME** ∩ **coNPTIME**, but no known **PTIME** algorithm exists. That is, the following picture is suspected:



Chapter 5

Function Problems

A basic understanding of the computational difficulty of a great many decision problems in hand, it's time to reconsider one of the basic limitations we've imposed on our work thus far and re-admit non-decision problems into our work.

> **Function Problem**
> A *function problem* is any computational problem which requires a solution more complicated than just 'yes' or 'no'

While our preceding work has been composed entirely of decision problems, it's obvious that–so far as *negative results*–much has been shown about various function problems via their decision problem counterparts. In particular, while our hierarchies technically confine themselves to decision problems, the notion of reduction can be generalized to include function problems:

> **(Efficient) Strong Reduction**
> Given problems P1 and P2 with the same input, P1 is strong reducible to P2 if and only if, given an oracle for $P2$, there is a function R deterministically computable in $O(\log(n))$ space such that for any input x, $R(x)$ gives a solution for P1

[1]

For instance, defining

FSAT
`Given:` a Boolean expression φ in CNF
`Return:` a satisfying valuation for φ if one exists

it follows readily from the preceding work that

► **Proposition 5.1.**
FSAT strong reduces any **NPTIME** problem

More surprisingly, it's also easy to show that

► **Proposition 5.2.**
SAT strong reduces FSAT

Proof Sketch.
Exhaustively search for an assignment by repeatedly replacing more and more atomics by either $\bot$ or $\top$

[1] I'm just making this up, but it's the obvious oracle / Turing reduction counterpart and seems to be what's intended

This connection between **NPTIME** problems and their function counterparts can even, using the semantic definition of **NPTIME**, be formalized:

> **Associated Function Problem**
> Given a decision problem P in **NPTIME**, the *associated function problem for* P, denoted FP, is, for any input x, to give either the verification certificate y for P or return 'no' if no such certificate exists

This, of course, readily extends into a definition of two presumably distinct complexity class:

> **FNPTIME**
> The class of associated function problems for **NPTIME**

> **FPTIME**
> The class of associated function problems for **PTIME**

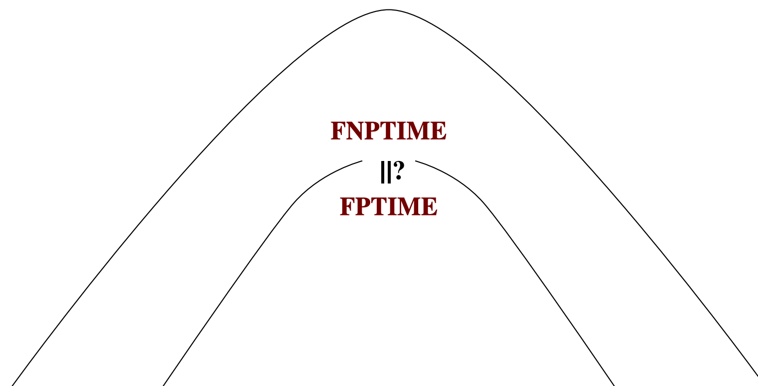


Figure 5.1: Function Problem Classes

We may even define a notion of reduction particular to functions problems:

> **(Efficient) Function Reduction**
> For two function problems A and B, A *reduces to* B if and only if there are string functions R and S, both in **L**, such that for any strings x and z, if x is an instance of A, then $R(x)$ is an instance of B and if z is a correct output of $R(x)$, then $S(z)$ is a correct output of x

The above in hand, it's not hard to show that

► **Proposition 5.3.**
FPTIME and **FNPTIME** are closed under function reduction

► **Proposition 5.4.**
FSAT is **FNPTIME**-complete

---

and thus,

► **Corollary.**
FPTIME = **FNPTIME** if and only if **PTIME** = **NPTIME**

This last result is taken by many to imply that function problems in general aren't any more interesting that decision problems; that, at the macro level, the two have largely identical structure.

§5.1 Total Function Problems

If there is a counterexample to the idea that function problems hold little interest, it's to be found in the total functions problems; that is, those problems which never return 'no'. In terms of decision problem counterparts, all of these correpond to the trivial decision problem which includes everything–nonetheless, they can exhibit complex structure. Among the oldest and most infamous such problems is FACTORING:

FACTORING
Given: $n \in \mathbb{N}$
Return: the prime factorization of n

Although provably total, FACTORING has resisted centuries (knowing or unknowing) of work to produce a **PTIME** algorithm to solve it. FACTORING is thus a problem complicated for a very different reason than many other **FNPTIME** problems (e.g. FSAT)–not because a certificate could fail to exist, but simply because the required certificates are hard to generate.

The semantic class of total function problems like FACTORING–**TFNPTIME**–is widely regarded to occupy a middle ground between **FPTIME** and **FNPTIME**:

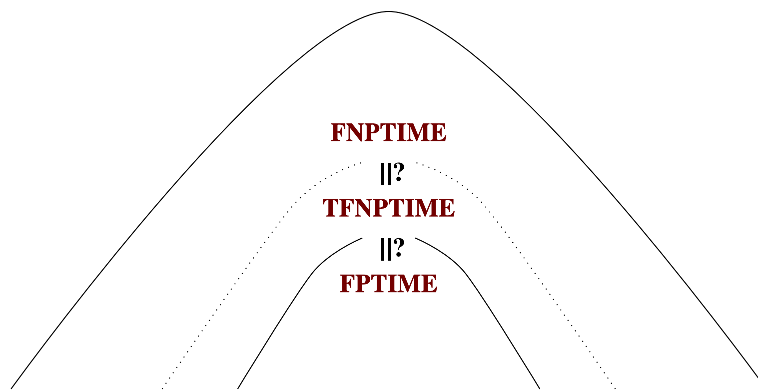


Figure 5.2: Function Problem Classes

Chapter 6

Randomization

The final expansion on deterministic models of computation typically considered is the introduction of the ability to randomize, to 'flip unbiased coins' during computation. As with the introduction of nondeterminism, it's common to tweak the definitions for when an algorithm computes a problem; with randomization, the specific choice

> **Monte Carlo Algorithm**
> A *Monte Carlo algorithm* is a randomized algorithm which never gives false positives and which gives false negatives with probability $\leq p$ for some $p \in [0, 1)$

Definition in hand, the next task is, as usual, to formalize a model of computation, give definitions for time/space/etc, and generate the natural complexity classes. The most obvious choice is to add a randomization command to our model of computation, but–while we will often speak and write this way–there is a more perspicuous representation available. Instead, we adopt as our formalism that of the nondeterministic models from last chapter and simply redefine what it means for such a machine to accept.

> **Monte Carlo Machine**
> A *Monte Carlo machine* M_R is a nondeterministic machine M_D meeting the following criteria:

(i) for a given input size, all computation paths halt after exactly the same number of steps regardless of nondeterministic choices

(ii) all nondeterministic choices are binary

For any input x,

$$M_R(x)\downarrow = 1 \Leftrightarrow \text{ at least half of the computations of } M_D \text{ on } x \text{ give output } 1$$
$$M_R(x)\downarrow = 0 \Leftrightarrow M_D(x)\downarrow = 0$$

The choice to require at least half of the computations to accept in order for the Monte Carlo machine as a whole to accept is, of course, arbitrary; that said, it causes no loss in computational power. Just as with nondeterminism, the definition above again introduces an asymmetry between accepting and declining, one which intuitively matches that inherent in the definition of a Monte Carlo algorithm.

Randomized Time Complexity Classes	Randomized Space Complexity Classes
$\mathbf{RLTIME} = \mathrm{RTIME}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \mathrm{RTIME}(\log^m(n))$	$\mathbf{RL} = \mathbf{RLSPACE} = \mathrm{RSPACE}(\log^k(n)) = \bigcup_{m\in\mathbb{N}} \mathrm{RSPACE}(\log^m(n))$
$\mathbf{RP} = \mathbf{RPTIME} = \mathrm{RTIME}(n^k) = \bigcup_{m\in\mathbb{N}} \mathrm{RTIME}(n^m)$	$\mathbf{RPSPACE} = \mathrm{RSPACE}(n^k) = \bigcup_{m\in\mathbb{N}} \mathrm{RSPACE}(n^m)$
$\mathbf{REXPTIME} = \mathrm{RTIME}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \mathrm{RTIME}(2^{n^m})$	$\mathbf{REXPSPACE} = \mathrm{RSPACE}(2^{n^k}) = \bigcup_{m\in\mathbb{N}} \mathrm{RSPACE}(2^{n^m})$
$\mathbf{2\text{-}REXPTIME} = \mathrm{RTIME}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \mathrm{RTIME}(2^{2^{n^m}})$	$\mathbf{2\text{-}REXPSPACE} = \mathrm{RSPACE}(2^{2^{n^k}}) = \bigcup_{m\in\mathbb{N}} \mathrm{RSPACE}(2^{2^{n^m}})$

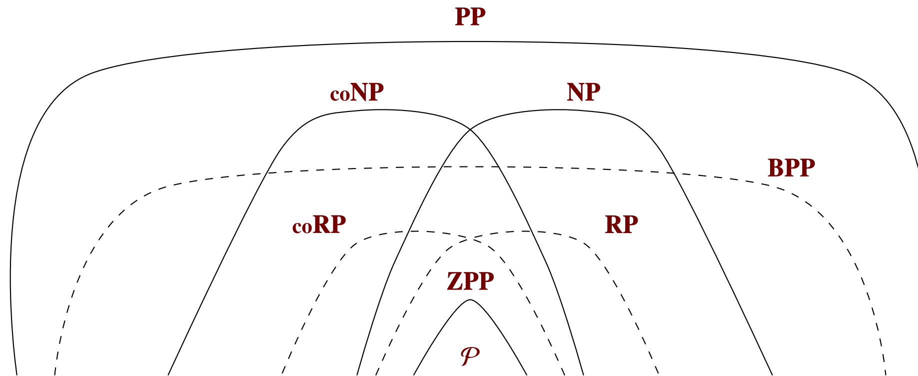


Figure 6.1: Complexity Classes