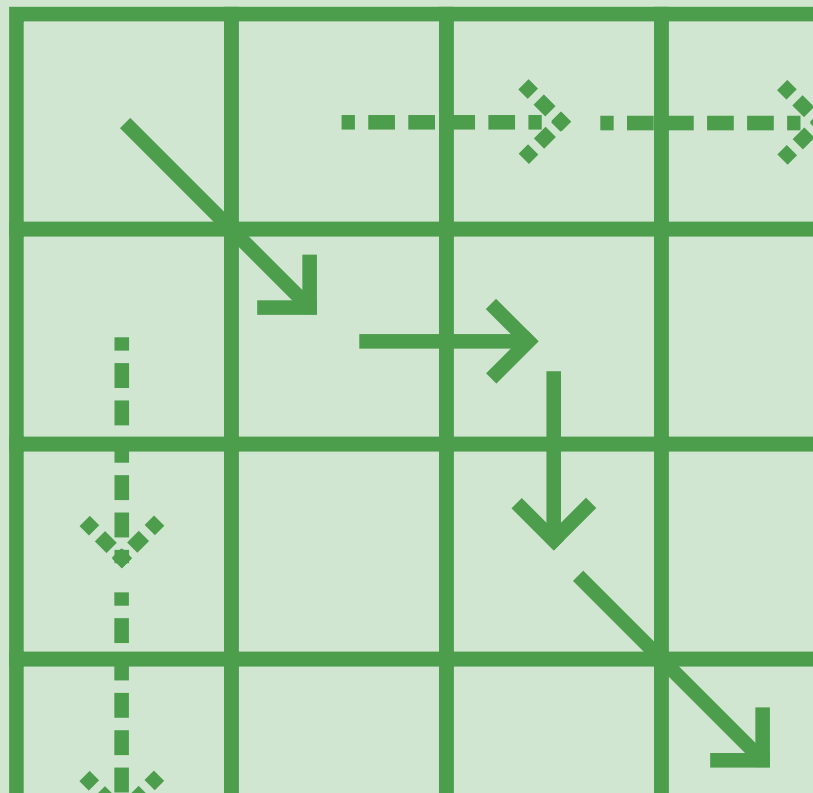


Algorithm Design and Analysis



The cover depicts a dynamic programming matrix used in the edit distance algorithm. Variations of this algorithm such as the Needleman-Wunsch and Smith-Waterman algorithms are frequently used in computational biology to align DNA or protein sequences.

Copyright © 2022 Kevin Gao

TERRA-INCOGNITA.DEV

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

I	Divide and Conquer	
1	Divide and Conquer Primer	11
1.1	Divide and Conquer Paradigm	11
1.2	The Master Theorem	11
1.3	Counting Inversions	12
1.4	Closest Pair	13
1.5	Karatsuba's Integer Multiplication Algorithm	14
2	Strassen's Algorithm	17
2.1	Matrix Multiplication	17
2.2	A Recursive Approach	17
2.3	Strassen's Algorithm	18
3	Fast Fourier Transform	21
3.1	Motivation	21
3.1.1	Coefficient Vector	21
3.1.2	Roots	21
3.1.3	Samples	21

3.2	Operations on Polynomials	22
3.2.1	Coefficient Representation	22
3.2.2	Samples	23
3.2.3	Roots	24
3.2.4	Summary	24
3.3	Fast Fourier Transform	25
3.3.1	Coefficient to Samples (and back)	25
3.3.2	Divide and Conquer	25
3.3.3	Complex Roots of Unity	26
3.3.4	The FFT Algorithm	27
3.4	Inverse Fast Fourier Transform	28
3.5	Fast Polynomial Multiplication	29

II

Greedy Algorithms

4	Interval Scheduling	33
4.1	Interval Scheduling	33
4.2	Proving Optimality	35
4.3	Interval Coloring (Interval Partition)	37
4.4	Greedy Strategy	38
5	Graph Algorithms Using Greedy	41
5.1	Interval Scheduling as Graph Problems	41
5.2	Minimum Spanning Tree	41
5.2.1	Kruskal's Algorithm	41
5.2.2	Prim's Algorithm	41
5.3	Single-Source Shortest Path	41
5.3.1	Dijkstra's Algorithm	41
6	Huffman Encoding	43
7	Generalizing and Formalizing Greedy Algorithms	45
7.1	Priority Model	45
7.2	Global Optimality of Greedy Algorithms	46

III

Dynamic Programming

8	Weighted Interval Selection	51
8.1	Dynamic Programming	51
8.2	Weighted Interval Selection	51

8.3	Computing the Optimal Weight	51
8.3.1	The Dynamic Programming Approach	51
8.3.2	Top-Down DP	53
8.3.3	Bottom-Up DP	53
8.3.4	Comparison of Two Approaches	54
8.4	Computing the Optimal Solution	54
9	Knapsack Problem	55
9.1	Knapsack	55
9.2	DP Algorithm for 0-1 Knapsack	55
9.3	A Different DP Algorithm	56
9.4	FPTAS Approximation for Knapsack	56
10	Graph Algorithms Using DP	59
10.1	Single-Source Shortest Path	59
10.1.1	Bellman-Ford	59
10.2	Maximum Length Path Fails	62
10.3	All-Pairs Shortest Paths	63
10.3.1	Matrix Multiplication	63
10.3.2	Floyd-Warshall	63
10.3.3	Johnson's Algorithm	63
10.4	Traveling Salesman Problem	63
11	Bioinformatics	65
11.1	Dynamic Programming in Computational Biology	65
11.2	Longest Common Subsequence	65
11.3	Edit Distance	65

IV

Network Flow

12	Ford-Fulkerson	69
12.1	Flow Network	69
12.1.1	Definitions	69
12.1.2	Antiparallel Edges and Multiple Sources/Sinks	70
12.2	Ford-Fulkerson Method	70
12.2.1	Idea Behind Ford-Fulkerson	71
12.3	Correctness of Ford-Fulkerson	71
12.3.1	Residue Network and Augmentation	71
12.3.2	Augmenting Paths	73
12.3.3	Max-Flow Min-Cut Theorem	74

13	Edmonds-Karp Algorithm	79
14	Maximum Bipartite Matching	81

V Complexity

VI Linear Programming

15	Linear Programming	87
15.1	Linear Programming Optimization	87
15.2	Standard and Slack Forms	88
15.2.1	Standard Form	88
15.2.2	Conversion From Non-standard Form to Standard Form	89
15.2.3	Slack Form	89
16	Simplex Algorithm for LP	91
16.1	Geometric Intuition	91
16.2	Pivoting	91
16.3	Simplex Algorithm	91
16.4	Duality and Certificate of Optimality	91
17	Linear Programming Problems	93
18	Integer Programming	95

VII Approximation Algorithms

VIII Randomized Algorithms

Appendix

Axioms & Theorems	103
Basic Prerequisite Mathematics	107
Proof Templates	113
Index	123

Bibliography	125
Courses	125
Books	125
Journal Articles	126



Divide and Conquer

1	Divide and Conquer Primer	11
1.1	Divide and Conquer Paradigm	
1.2	The Master Theorem	
1.3	Counting Inversions	
1.4	Closest Pair	
1.5	Karatsuba's Integer Multiplication Algorithm	
2	Strassen's Algorithm	17
2.1	Matrix Multiplication	
2.2	A Recursive Approach	
2.3	Strassen's Algorithm	
3	Fast Fourier Transform	21
3.1	Motivation	
3.2	Operations on Polynomials	
3.3	Fast Fourier Transform	
3.4	Inverse Fast Fourier Transform	
3.5	Fast Polynomial Multiplication	

Chapter 1 Divide and Conquer Primer

1.1 Divide and Conquer Paradigm

Most divide and conquer algorithms follow this paradigm.

- Divide up problem into several subproblems. Note that in some cases, we have to generalize the given problem.
- Recursively solving these subproblems.
- Combining the results from subproblems.

In this chapter and the following chapter, we will examine a few examples of divide-and-conquer algorithms, including but not limited to: maximum subarray, Strassen's matrix multiplication algorithm, quicksort, mergesort, median and selection in sorted array, fast Fourier transform (FFT), inversion counting, etc.

1.2 The Master Theorem

We have used the Master Theorem many times in both the introduction to theory of computation and data structure courses. We will once again present the theorem as it appears in CLRS here. For a careful proof of the theorem, see Section 4.6 of CLRS or Tutorial 7 note for CSC240 (Enriched Introduction to Theory of Computation).

Theorem 1.2.1 — The Master Theorem. Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function. Let $T(n)$ on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $\frac{n}{b}$ is interchangeable with $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$. Then, $T(n)$ has the following asymptotic bounds:

- If $f(n) \in O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
- If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \lg n)$.
- If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) \in \Theta(f(n))$.

An equivalent formulation of the theorem is as follows.

Theorem 1.2.2 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

1.3 Counting Inversions

The objective of array inversion problems is to find the number inversions in an unsorted array compared to a sorted array. That is, given an array A , how many pairs (i, j) are there such that $i < j$ and $A[i] > A[j]$. In other words, it counts the number of element-wise swaps needed in order to sort the given array.

For example, given the array $[8, 4, 2, 1]$, the algorithm should answer 6 since the array has these six inversions: $(8, 4)$, $(4, 2)$, $(8, 2)$, $(8, 1)$, $(4, 1)$, and $(2, 1)$. If we follow these inversions, we will get a sorted array.

An naive algorithm for this problem is naturally to examine every pair of elements in the given array, which will take $O(n^2)$ comparisons. However, due to its resemblance to sorting, it is not hard to come up with a divide-and-conquer algorithm similar to mergesort that solves this problem in $O(n \log n)$ time.

At a high level, the algorithm should follow the aforementioned paradigm:

- Divide: split list into two halves A and B
- Conquer: recursively count inversions in each list
- Combine: count inversions (a, b) with $a \in A$ and $b \in B$
- Return the sum of the tree counts

For the combine step, we assume that the two subarrays A and B are sorted. Then, we can scan A and B from left to right in parallel and compare $A[i]$ and $B[j]$. If $A[i] < B[j]$, then $A[i]$ is not inverted with any element in B . If $A[i] > B[j]$ then $B[j]$ is inverted with every element in left in A .

A pseudocode for the algorithm is shown below. Equivalently, instead of explicitly splitting the array, we can perform this in place by passing around the indices p and q , as shown in Section 2.3.1 of CLRS.

Sort-And-Count(L)

```

1  if  $L.length == 0$ 
2      return  $(0, L)$ 
3   $mid = \lfloor (1 + L.length) / 2 \rfloor$ 
4   $(count-a, A) = \text{Sort-And-Count}(A[1 \dots mid])$ 
5   $(count-b, B) = \text{Sort-And-Count}(A[mid + 1 \dots A.length])$ 
6   $(count-ab, L') = \text{Merge-And-Count}(A, B)$ 
7  return  $(count-a + count-b + count-ab, L')$ 

```

Merge-And-Count(A, B)

```

1   $count = 0$ 
2   $i, j, k = 1$ 
3   $L = []$ 
4  while  $i \leq A.length$  and  $j \leq B.length$ 
5      if  $A[i] \leq B[j]$ 
6           $L[k] = A[i]$ 
7           $i = i + 1$ 
8      else
9           $count = count + 1$ 
10          $L[k] = B[j]$ 
11          $j = j + 1$ 
12      $k = k + 1$ 
13 while  $i \leq A.length$ 
14      $L[k] = A[i]$ 
15      $k = k + 1$ 
16 while  $j \leq B.length$ 
17      $L[k] = B[j]$ 
18      $k = k + 1$ 
19 return  $(count, L)$ 

```

The number of comparisons made by the algorithm is given by this recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise} \end{cases}$$

By Masters Theorem, $T(n) \in O(n \log n)$.

1.4 Closest Pair

Lemma 1.4.1 Let p be a point in the 2δ strip within δ distance horizontally. There are at most 7 points p such that $|y_p - y_q| \leq \delta$.

Proof. p must lie either in the left or right $\delta \times \delta$ square. Within each square, each point has distance at least k from each other. So, we can pack at most 4 such points into one square, and since we have a left square and right square, we have 8 points in total. Other than p , there are at most 7 points. ■

CLOSEST-PAIR($P = p_1, p_2, \dots, p_n$)

```

1  compute a vertical line  $L$  such that half the points
   are on each side                                     //  $O(n \log n)$ 
   // consider sorting based on  $x$ -axis
2   $\delta_1 = \text{CLOSEST-PAIR}(P_L)$ 
3   $\delta_2 = \text{CLOSEST-PAIR}(P_R)$ 
4   $\delta = \min\{\delta_1, \delta_2\}$ 
5  for  $p$  in  $p_1, p_2, \dots, p_n$                          //  $O(n)$ 
6      if Y-DISTANCE( $p, L$ )
7          delete  $p$ 
8  sort remaining points by  $y$ -coordinate                 //  $O(n \log n)$ 
9  for  $p$  in  $p_1, p_2, \dots, p_n$                          //  $O(n)$ 
10     for  $i$  from 1 to 7
11          $p_i = i$ th neighbor of  $p$ 
12         if DISTANCE( $p, p_i$ ) <  $\delta$ 
13              $\delta = \text{DISTANCE}(p, p_i)$ 
14 return  $\delta$ 
```

The number of operations performed by this algorithm is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n \log n) & \text{otherwise} \end{cases}$$

By Masters Theorem, $T(n) \in O(n \log^2 n)$.

Theorem 1.4.2 — Lower Bound for Closest Pair. In a quadratic decision tree model, the closest pair problem requires at least $\Omega(n \log n)$ quadratic tests.

A quadratic decision tree is a comparison tree whose internal nodes are labeled with quadratic comparisons in the form of $x_i < x_j$ or $(x_i - x_k)^2 - (x_j - x_k)^2 < 0$. More generally, each internal node contains a comparison between a polynomial of degree at most 2 and 0 (2nd-order algebraic decision tree).

A hand-wavy proof of this lower bound follows by reduction from the result that the element uniqueness problem has a $\Omega(n \log n)$ lower bound, first shown by Ben-Or in *Lower Bounds For Algebraic Computation Trees*. Element distinctness reduces to 2D closest pair problem.

1.5 Karatsuba's Integer Multiplication Algorithm

To multiply two n -bit integers a and b , we can follow this recursive procedure:

- add two $\frac{1}{2}n$ bit integers
- multiply three $\frac{1}{2}n$ -bit integers recursively
- add, subtract, and shift to obtain the result

$$\begin{aligned}
 a &= 2^{n/2}a_1 + a_0 \\
 b &= 2^{n/2}b_1 + b_0 \\
 ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + a_0 b_1) + a_0 b_0 \\
 &= 2^n \underbrace{a_1 b_1} + 2^{n/2} \underbrace{((a_1 + a_0)(b_1 + b_0) - \underbrace{a_1 b_1} - \underbrace{a_0 b_0})}_{\text{underbrace}} + \underbrace{a_0 b_0}
 \end{aligned}$$

The recursive steps are labeled with underbrace. By Masters Theorem, Karatsuba's algorithm performs

$$T(n) = 3T(\lceil n/2 \rceil) + cn + d \in \Theta(n^{\log_2 3})$$

bit-wise operations.

Chapter 2 Strassen's Algorithm

2.1 Matrix Multiplication

The standard method to multiply two $n \times n$ matrices requires $O(n^3)$ scalar operations (multiplication and addition). The standard algorithm follows from the definition of matrix multiplication that the i, j entry of $C = A \cdot B$ is given by

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.rows$ 
2   $C = n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik}b_{kj}$ 
8  return  $C$ 
```

Clearly, this algorithm runs in $\Theta(n^3)$.

2.2 A Recursive Approach

Without loss of generality, let $n = 2^k$ for some k . Then, we can divide any $n \times n$ matrix into four $n/2 \times n/2$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Since matrices are a non-commutative ring, $n \times n$ matrix multiplication can be realized in $7n/2 \times n/2$ matrix multiplications and 18 matrix additions. Matrix addition only takes $O(n^2)$ scalar operations. Then, we have the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/2) + O(n^2) & \text{otherwise} \end{cases}$$

which implies that $T(n) \in O(n^{\log_2 7})$.

An implementation of this recursive approach is presented in pseudocode below

```

SQUARE-MATRIX-MULTIPLY( $A, B$ )
1   $n = A.rows$ 
2   $C = n \times n$  matrix
3  if  $n == 1$ 
4  else
5      partition  $A, B$ , and  $C$  each into four sub-matrices
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY}(A_{11}, B_{11}) +$ 
           $\text{SQUARE-MATRIX-MULTIPLY}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY}(A_{11}, B_{12}) +$ 
           $\text{SQUARE-MATRIX-MULTIPLY}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY}(A_{21}, B_{11}) +$ 
           $\text{SQUARE-MATRIX-MULTIPLY}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY}(A_{21}, B_{12}) +$ 
           $\text{SQUARE-MATRIX-MULTIPLY}(A_{22}, B_{22})$ 
10 return  $C$ 

```

However, this particular implementation of the recursive approach still has a $O(n^3)$ running time, as demonstrated by solving the recurrence by Masters Theorem.

2.3 Strassen's Algorithm

Strassen's algorithm is a rather peculiar algorithm that multiplies two square matrices using $O(n^{\log_2 7}) = O(n^{2.81})$ scalar operations. It is debatable how practical asymptotically faster matrix multiplication algorithms (including Strassen's algorithm and Coppersmith-Winograd algorithm) are given that their crossover point (the threshold on input size beyond which such algorithms become actually faster) is often quite large for them to be useful in practical applications.

Strassen's algorithm as described in CLRS works as follows:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices. This takes $\Theta(1)$ operations.
2. Create 10 submatrices S_1, S_2, \dots, S_{10} , each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in Step 1. This can be done in $\Theta(n^2)$.
3. Using the submatrices in Step 1 and the 10 matrices in Step 2, recursively compute seven matrix products P_1, P_2, \dots, P_7 . Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. This can also be done in $\Theta(n^2)$ time.

More specifically, in Step 2,

$$S_1 = B_{12} - B_{22}$$

$$S_2 = A_{11} + A_{12}$$

$$S_3 = A_{21} + A_{22}$$

$$S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}$$

$$S_6 = B_{11} + B_{22}$$

$$S_7 = A_{12} - A_{22}$$

$$S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - B_{21}$$

$$S_{10} = B_{11} + B_{12},$$

and in Step 3,

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$

$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$

$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$

$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$$

$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$

$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$$

$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}.$$

In Step 4,

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

Chapter 3 Fast Fourier Transform

3.1 Motivation

A polynomial can be represented in various ways. Each representation has its advantages and downsides.

In general, a polynomial $A(x)$ of degree $n - 1$ can be written as

$$\begin{aligned} A(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \\ &= \sum_{k=0}^{n-1} a_k x^k \end{aligned}$$

3.1.1 Coefficient Vector

For a polynomial $A(x)$ of degree $n - 1$, the coefficient vector contains the coefficient of the polynomial represented as a vector.

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \quad \leftrightarrow \quad \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$$

3.1.2 Roots

Equivalently,

$$A(x) = (x - r_0)(x - r_1) \cdots (x - r_{n-1}) \cdot c$$

where r_0, r_1, \dots, r_{n-1} are the n roots of the polynomial and c is a scale term.

3.1.3 Samples

Given a polynomial, we can take n points (coordinates):

$$(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$$

where $A(x_i) = y_i$ for all $i \in \{0, 1, \dots, n - 1\}$. This representation is also known as the point-value representation.

By the Fundamental Theorem of Algebra, these samples uniquely define a polynomial of degree $n - 1$.

Theorem 3.1.1 — The Fundamental Theorem of Algebra. A univariate polynomial of degree n with complex coefficients has exactly n complex roots.

Corollary 3.1.2 — Uniqueness of Polynomial Interpolation. A degree $n - 1$ univariate polynomial $A(x)$ is uniquely defined by its evaluation at n distinct values of x .

3.2 Operations on Polynomials

There are three primary operations on polynomials: evaluation, addition, and multiplication. The complexity of these operations varies depending on the representation. There is not a single representation that is efficient for all operations, so we need a way to efficiently convert between the representations.

3.2.1 Coefficient Representation

Evaluation

Evaluation is efficient when the polynomial is represented as the coefficient vector using the Horner's rule:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))) \cdots)$$

which can be written in pseudocode as follows

EVALUATE($A = \langle a_0, \dots, a_{n-1} \rangle, x$)

```

1   $y = 0$ 
2  for  $i = n - 1$  to  $0$ 
3       $y = a_i + (x \cdot y)$ 
4  return  $y$ 
```

Addition

Addition is easy in coefficient representation. We simply add each coefficient to get a new coefficient vector.

ADD($A = \langle a_0 \dots a_{n-1} \rangle, B = \langle b_0 \dots b_{n-1} \rangle$)

```

1  for  $j = 0$  to  $n - 1$ 
2       $c_j = a_j + b_j$ 
3  return  $C = \langle c_0, \dots, c_{n-1} \rangle$ 
```

Multiplication

Multiplication is a little bit trickier in coefficient representation. We need to use linear convolution, which takes $O(n^2)$ operations.

$$A(x) \times B(x) = \sum_{j=0}^{2n-2} c_j x^j = \sum_{j=0}^{2n-2} \sum_{k=0}^j a_k b_{j-k} x^j$$

MULTIPLY($A = \langle a_0 \dots a_n \rangle, B = \langle b_0 \dots b_m \rangle$)

```

1  for  $j = 0$  to  $n + m$ 
2       $c_j = 0$ 
3  for  $j = 0$  to  $n$ 
4      for  $k = 0$  to  $n$ 
5           $c_{j+k} = c_{j+k} + a_j \cdot b_k$ 
6   $C = \langle c_0 \dots c_{n+m} \rangle$ 
```

3.2.2 Samples

Evaluation

Evaluation can be done in $O(n^2)$ through interpolation using Lagrange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Addition

Addition is also easy in samples representation.

$$A(x) + B(x) : (x_0, y_0 + z_0), (x_1, y_1 + z_1), \dots, (x_{n-1}, y_{n-1} + z_{n-1})$$

given that $A(x)$ is represented as $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ and $B(x)$ is represented as $(x_0, z_0), \dots, (x_{n-1}, z_{n-1})$.

Multiplication

Multiplication is done in a similar fashion as addition.

$$A(x) \times B(x) : (x_0, y_0 z_0), (x_1, y_1 z_1), \dots, (x_{n-1}, y_{n-1} z_{n-1})$$

given that $A(x)$ is represented as $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ and $B(x)$ is represented as $(x_0, z_0), \dots, (x_{n-1}, z_{n-1})$.

3.2.3 Roots

Evaluation

Takes $O(n)$ operations by substitution and multiplication.

Addition

Impossible in the general case. There is not a general formula that can convert a coefficient vector back to the roots.

Multiplication

Multiplying two polynomials represented by their roots is equivalent to concatenate the list of roots since the resulting polynomial will have the same roots of both polynomials.

3.2.4 Summary

	Coefficient	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

Table 3.1: Comparison between different representations of polynomial

The following chart outlines the conversions required for efficient multiplication of polynomials. The figure is taken from CLRS pp. 904.

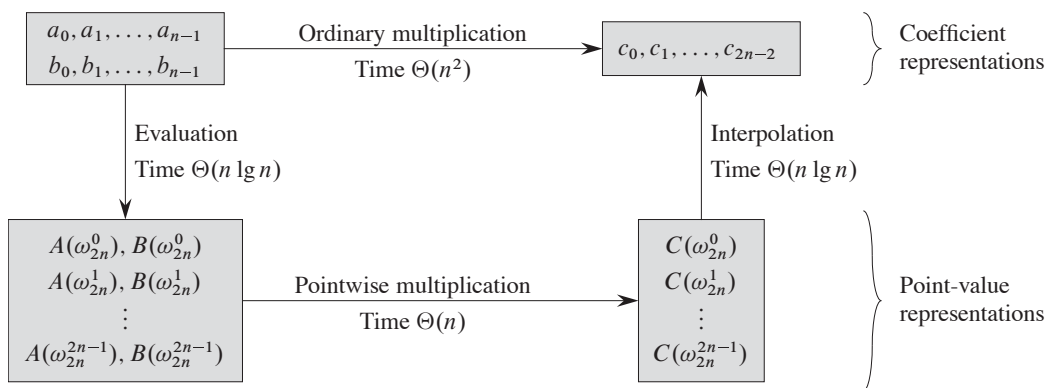


Figure 3.1: Graphical outline of an efficient polynomial multiplication procedure.

3.3 Fast Fourier Transform

3.3.1 Coefficient to Samples (and back)

Conversion from coefficient to samples is similar to evaluation. We fix a set of points x_0, x_1, \dots, x_{n-1} and evaluate the polynomial at these points. This evaluation is equivalent to the following matrix multiplication

$$\mathbf{y} = \mathbf{V}\mathbf{a} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where \mathbf{V} is called the Vandermonde matrix with entries $V_{jk} = x_j^k$ and \mathbf{a} is the coefficient vector of the polynomial $A(x)$.

Evaluating this matrix product takes $\Theta(n^2)$ scalar operations. Similarly, to convert from samples back to polynomial (interpolation), we can compute \mathbf{V}^{-1} using Gaussian elimination with $O(n^3)$ operations, and computing $\mathbf{a} = \mathbf{V}^{-1}\mathbf{y}$ (it is rarely a good idea to invert any matrices; “Don’t invert that matrix!” – John Cook).

In order to obtain efficient algorithms for manipulating polynomials, we need to somehow get rid of the $O(n^2)$ overhead resulted from the interconversion between coefficient and samples.

Note that we simply said “fix $x_0 \dots x_{n-1}$ ” when constructing the Vandermonde matrix without putting any constraints on what values of x to take. By choosing special values for x , we can reduce the conversion overhead to $O(n \log n)$.

3.3.2 Divide and Conquer

We can formulate the evaluation $\mathbf{y} = \mathbf{V}\mathbf{a}$ as a divide and conquer algorithm.

We can come up with an outline of the algorithm by following the divide-and-conquer paradigm

1. Divide: divide the polynomial A into even and odd coefficients (this is equivalent to divide the coefficient vector \mathbf{a} into even and odd entries)

$$A_{\text{even}}(x) = \sum_{k=0}^{\lceil \frac{n}{2} - 1 \rceil} a_{2k} x^k \quad \leftrightarrow \quad \mathbf{a}_{\text{even}} = \langle a_0, a_2, a_4, \dots \rangle$$

$$A_{\text{odd}}(x) = \sum_{k=0}^{\lfloor \frac{n}{2} - 1 \rfloor} a_{2k+1} x^k \quad \leftrightarrow \quad \mathbf{a}_{\text{odd}} = \langle a_1, a_3, a_5, \dots \rangle$$

Note that the degree of the two resulting polynomials is half of the original polynomial.

2. Conquer: recursive conquer $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ for $x \in X^2$ where $X^2 = \{x^2 \mid x \in X\}$.

3. Combine: combine the terms as follows

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$$

for $x \in \mathbf{x}$.

It is obvious that the degree of the polynomial n halves, and hence the recurrence

$$T(n, |X|) = 2T\left(\frac{n}{2}, |X|\right) + O(n + |X|) \in O(n^2).$$

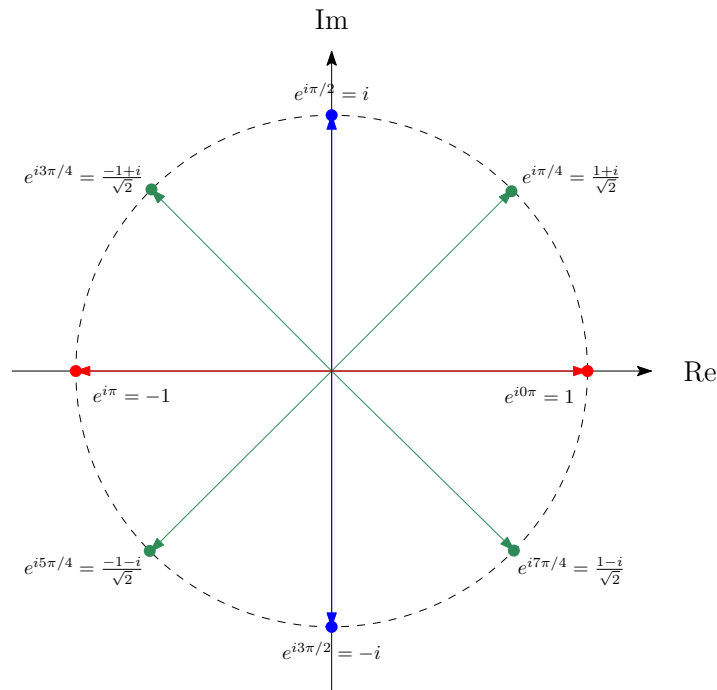
This is no better than the naive approach. The main issue is that we are not halving the size of X . Ideally, we want X to be recursively collapsing.

3.3.3 Complex Roots of Unity

We can construct a collapsing set of x 's via square roots. If we only look at real numbers, taking the square root or squaring a number won't give you fewer or more numbers, but if we broaden our view to complex numbers, we notice that starting from 1, every time we take the square root, the size of the set doubles. The n th root of 1 is called the **complex n th root of unity**. The observation implies that if we start from the n th root of unity and square the elements in the set, the set will collapse every time we square it.

Example: $\{1\} \rightarrow \{1, -1\} \rightarrow \{1, -1, i, -i\} \rightarrow \{1, -1, i, -i, \pm \frac{\sqrt{2}}{2}(1+i), \pm \frac{\sqrt{2}}{2}(-1+i)\}$

The complex roots of unity are spaced equally around the unit circle centered at the origin of the complex plane. These points are of the form $\cos \theta + i \sin \theta = e^{i\theta}$ for $\theta = 0, \frac{2}{n}\pi, \frac{4}{n}\pi, \dots, \frac{2(n-1)}{n}\pi$.



The n th roots of unity where $n = 2^\ell$ for some integer ℓ form a collapsing set since $(e^{i\theta})^2 = e^{i2\theta} = e^{i(2\theta \bmod 2\pi)}$.

Let us formalize this idea and prove that it works.

Lemma 3.3.1 — Halving (Collapsing) Lemma. If $n > 0$ is even, then the squares of the n complex n th roots of unity are the $n/2$ complex $(n/2)$ th roots of unity.

Proof. We know that $\left(e^{\frac{2\pi i}{n}}\right)^{2k} = e^{\frac{4k\pi i}{n}} = \left(e^{\frac{2\pi i}{1/2 \cdot n}}\right)^k$ for any nonnegative integer k .

Furthermore, $\left(e^{\frac{2\pi i}{n}}\right)^{2(k+n/2)} = \left(e^{\frac{2\pi i}{n}}\right)^k$. This implies that for every pair of k and $k + n/2$ share the same square, and that if take the square of every element in the set of n th roots of unity, we get the $n/2$ elements and it follows from algebra that the resulting set contains the $(n/2)$ th roots of unity. ■

3.3.4 The FFT Algorithm

By choosing the x 's in the Vandermonde matrix to be the n th roots of unity, we can make X collapsing along with n . The rest of the algorithm is the same as the divide-and-conquer approach described earlier.

```

FFT-RECURSIVE( $a$ )
1   $n = a.length$ 
2  if  $n == 1$ 
3      return  $a$ 
4   $\omega_n = e^{2\pi i/n}$ 
5   $\omega = 1$ 
6   $a_{even} = (a_0, a_2, \dots, a_{n-2})$ 
7   $a_{odd} = (a_1, a_3, \dots, a_{n-1})$ 
8   $y_{even} = \text{FFT-RECURSIVE}(a_{even})$ 
9   $y_{odd} = \text{FFT-RECURSIVE}(a_{odd})$ 
10 for  $k = 0$  to  $n/2 - 1$ 
11      $y_k = y_{even}[k] + \omega y_{odd}[k]$ 
12      $y_{k+\frac{n}{2}} = y_{even}[k] - \omega y_{odd}[k]$ 
13      $\omega = \omega \omega_n$ 
14 return  $y = (y_0, \dots, y_{n-1})$ 

```

This algorithm takes $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \in O(n \log n)$ operations.

3.4 Inverse Fast Fourier Transform

The inverse discrete fourier transform is an algorithm used to find the coefficients for a polynomial given a set of samples. The transformation is of the form $\mathbf{a}^* \rightarrow \mathbf{V}^{-1} \mathbf{a}^* = \mathbf{a}$. To evaluate this, we need \mathbf{V}^{-1} . In fact, this matrix has a very useful property that allows to find it without actually inverting \mathbf{V} (again, don't invert the matrix).

Lemma 3.4.1 Let \mathbf{V} be the Vandermonde matrix constructed from the set of n th roots of unity. Let $\overline{\mathbf{V}}$ denote the complex conjugate of \mathbf{V} . Then,

$$\mathbf{V}^{-1} = \frac{1}{n} \overline{\mathbf{V}}$$

Proof. We claim that $\mathbf{P} = \mathbf{V} \overline{\mathbf{V}} = n\mathbf{I}$.

Let p_{jk} be the j, k th entry of \mathbf{P} .

$$\begin{aligned} p_{jk} &= (\text{row } j \text{ of } \mathbf{V}) \cdot (\text{column } k \text{ of } \overline{\mathbf{V}}) \\ &= \sum_{m=0}^{n-1} e^{ij2\pi m/n} \overline{e^{ik2\pi m/n}} \\ &= \sum_{m=0}^{n-1} e^{ij2\pi m/n} e^{-ik2\pi m/n} \\ &= \sum_{m=0}^{n-1} e^{i(j-k)2\pi m/n} \end{aligned}$$

Take $j = k$. Then, $p_{jk} = \sum_{m=0}^{n-1} 1 = n$. Otherwise, $p_{jk} = 0$. This means that the diagonal entries are n , and the entries off the diagonal are all 0. Thus, the claim is true.

It follows that $\mathbf{V}^{-1} = 1/n \mathbf{V}$. ■

This lemma implies that for the Inverse Fast Fourier Transform algorithm, we can simply replace $e^{ikr/n}$ with its complex conjugate in the FFT algorithm. The rest of the algorithm is analogous to that for FFT, and it is easy to see that the running of IFFT is also in $O(n \log n)$.

IFFT-RECURSIVE(y)

```

1   $n = y.length$ 
2  if  $n == 1$ 
3      return  $y$ 
4   $\omega_n = e^{-2\pi i/n}$ 
5   $\omega = 1$ 
6   $y_{even} = (y_0, y_2, \dots, y_{n-2})$ 
7   $y_{odd} = (y_1, y_3, \dots, y_{n-1})$ 
8   $a_{even} = \text{IFFT-RECURSIVE}(y_{even})$ 
9   $a_{odd} = \text{IFFT-RECURSIVE}(y_{odd})$ 
10 for  $k = 0$  to  $n/2 - 1$ 
11      $a_k = (a_{even}[k] + \omega a_{odd}[k]) / n$ 
12      $a_{k+\frac{n}{2}} = (a_{even}[k] - \omega a_{odd}[k]) / n$ 
13      $\omega = \omega \omega_n$ 
14 return  $a = (a_0, \dots, a_{n-1})$ 

```

3.5 Fast Polynomial Multiplication

With FFT, we can implement the procedure outlined in Figure 3.1.

To calculate $A(x) \times B(x)$,

1. Compute $A^* = \text{FFT}(A)$ and $B^* = \text{FFT}(B)$. This step is $O(n \log n)$.
2. Compute $C^* = A^* \cdot B^*$ in sample representation. This step is $O(n)$.
3. Compute $C = \text{IFFT}(C^*)$ to get C in coefficient representation. This step is $O(n \log n)$.

Overall, the algorithm has runtime complexity of $O(n \log n)$.



Greedy Algorithms

4	Interval Scheduling	33
4.1	Interval Scheduling	
4.2	Proving Optimality	
4.3	Interval Coloring (Interval Partition)	
4.4	Greedy Strategy	
5	Graph Algorithms Using Greedy	41
5.1	Interval Scheduling as Graph Problems	
5.2	Minimum Spanning Tree	
5.3	Single-Source Shortest Path	
6	Huffman Encoding	43
7	Generalizing and Formalizing Greedy Algorithms	45
7.1	Priority Model	
7.2	Global Optimality of Greedy Algorithms	

Chapter 4 Interval Scheduling

4.1 Interval Scheduling

We begin by considering the interval scheduling problem. The same problem is referred to as the activity selection problem in CLRS.

Consider a set $S = \{a_1, a_2, \dots, a_n\}$ of jobs/activities, each with a start time s_i and finish time f_i . Two jobs are said to be compatible if they don't overlap. More formally, given activities a_i and a_j , they are compatible if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$. That is, if $s_i \geq f_j$ or $s_j \geq f_i$. The goal of the interval scheduling problem is to find a maximum-size subset of mutually compatible jobs.

Let us consider the greedy strategy for solving the interval scheduling problem. Intuitive, the globally optimal solution should leave resources/time open for as many other jobs as possible. This requires us to consider the jobs in some “natural” order:

- Earliest start time: consider jobs in ascending order of s_i
- Earliest finish time: consider jobs in ascending order of f_i
- Shortest interval: consider jobs in ascending order of $f_i - s_i$
- Fewest conflicts: for each job a_i , count the remaining number of conflicting jobs c_i and schedule in ascending order of c_i .

Not all of those strategies work. Here are some counterexamples (Figure 4.1).

Therefore, we choose earliest finish time as our greedy strategy, which we can implement into this recursive algorithm

INTERVAL-SCHEDULING-RECURSIVE(S, k, n)

```
1   $m = k + 1$ 
2  while  $m \leq n$  and  $S[m].s < S[k].f$ 
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{S[m]\} \cup \text{INTERVAL-SCHEDULING-RECURSIVE}(S, m, n)$ 
6  else
7      return  $\emptyset$ 
```

As a precondition, we assume that the array of jobs S is sorted in monotonically increasing order by finish time. $S[i].s$ denotes the starting time of the i th job, and $S[i].f$ denotes the finish time of the i th

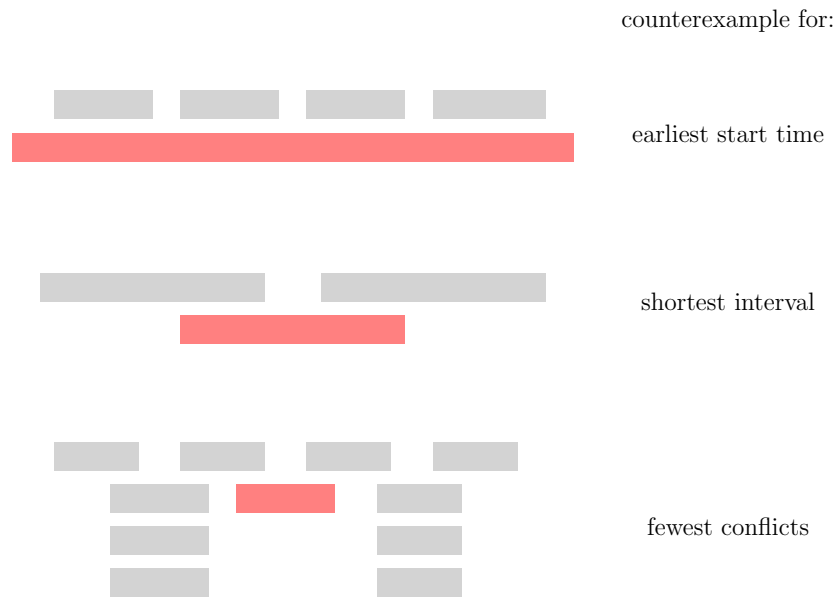


Figure 4.1: Counterexamples for the greedy strategies that do not work for the interval scheduling problem. The interval that satisfies the greedy strategy but leads to incorrect global solution is highlighted in red.

job. k is the job being considered. In each call to the recursive algorithm, we start from $m = k + 1$ and increment m until we find a job with starting time $S[m].s$ strictly lower than the finish time of $S[k]$. This is the job that is selected by our greedy strategy in the current recursive call. After finding this m , we return the union of $\{S[m] = a_m\}$ and the maximum-size subset of S_m returned by the recursive call $\text{INTERVAL-SCHEDULING-RECURSIVE}(S, m, n)$.

The initial call that solves the problem globally is $\text{INTERVAL-SCHEDULING-RECURSIVE}(S, 0, n)$ where n is the number of jobs to be considered.

This algorithm can be easily converted into an iterative algorithm.

$\text{INTERVAL-SCHEDULING}(S)$

```

1   $n = S.length$ 
2   $S' = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $S[m].s \geq S[k].f$ 
6           $S' = S' \cup \{S[m]\}$ 
7           $k = m$ 
8  return  $S'$ 
```

Sorting the array of jobs takes $O(n \log n)$ time, and going over the sorted list and checking each job's compatibility takes $O(n)$ in total.

4.2 Proving Optimality

As we have demonstrated earlier in Figure 4.1, greedy strategy does not always work. It is important for us to ensure (prove) that the locally optimal greedy choice leads to a globally optimal solution. Here, we will present three equally valid proofs for the optimality of the greedy algorithm for the interval scheduling problem.

Theorem 4.2.1 The greedy algorithm using earliest finish time is optimal. More formally, the algorithm returns a maximum-size set of disjoint jobs $\{a_1, \dots, a_m\} \subseteq S$

Proof (by contradiction). Suppose, for contradiction, that the greedy algorithm using earliest finish time is not optimal. Let i_1, i_2, \dots, i_k be the sequence of jobs selected by the algorithm, and let j_1, j_2, \dots, j_m be the correct solution where $m > k$. Let r be the largest possible value such that $i_{r+1} \neq j_{r+1}$. That is, the $(r+1)$ th job is the first job where the two sequences begin to differ.

Both i_{r+1} and j_{r+1} must be compatible with previous choices of i and j , respectively. Let $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r, i_{r+1}, j_{r+2}, \dots, j_m$ be a new sequence of jobs. This is the optimal solution with j_{r+1} replaced with i_{r+1} . By assumption, the greedy solution is not optimal so j_{r+1} is not selected by the greedy algorithm and thus $i_{r+1} \leq j_{r+1}$. So, the new sequence of jobs is still feasible because $i_{r+1} \leq j_{r+1}$. The algorithm is still optimal because all m disjoint jobs are scheduled. But then, i_{r+1} is not different from j_{r+1} , which implies that r is not the maximum value such that $i_{r+1} \neq j_{r+1}$. This is a contradiction. Therefore, the greedy algorithm is optimal. ■

Proof (by induction). Let S_k be the subset of jobs selected by the greedy algorithm after considering the first k jobs in increasing order of finish time. We say that $S_k \subseteq S$ is promising if it can be extended to the optimal solution (S_k is part of the optimal solution). More formally, S_k is promising if there exists $T \subseteq \{a_{k+1}, \dots, a_n\}$ such that $O_k = S_k \cup T$ is optimal.

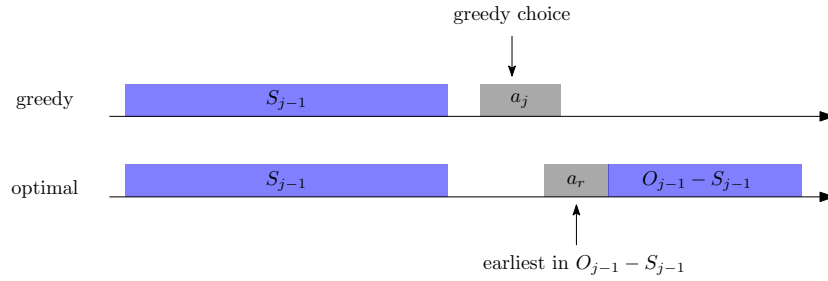
We want to show that for all $k \in \{0, 1, \dots, n\}$, S_k is promising.

Base case: When $k = 0$, $S_0 = \emptyset$. The claim is vacuously true.

Inductive step: Let $j \in \mathbb{N}$ be arbitrary. Suppose that the claim holds for $k = j - 1$ and that S_{j-1} is promising. For S_j , there are two possibilities:

(1) The greedy algorithm did not select job a_j (i.e. $s_{j+1} < f_j$), so $S_j = S_{j-1}$. This implies that a_j is not compatible with some job in S_{j-1} . Since $S_{j-1} \subseteq O_{j-1}$, O_{j-1} does not include a_j . $O_j = O_{j-1}$ does not include a_j , so S_j can be extended to O_j and S_j is promising.

(2) The greedy algorithm selected job a_j (i.e. $s_{j+1} \geq f_j$), so $S_j = S_{j-1} \cup \{a_j\}$. Consider the earliest job a_r in $O_{j-1} - S_{j-1}$. Consider $O_j = O_{j-1} - \{a_r\} \cup \{a_j\}$. O_j is still feasible since jobs in O_{j-1} are disjoint, a_r is the first to finish, and the finish time of a_j is earlier or equal to the finish time of a_r . Hence, S_j can be extended to O_j .



By induction, the claim is true for all $k \in \{0, 1, \dots, n\}$. ■

The previous two proofs use the technique known as the exchange argument. They both rely on the argument that if the greedy solution after j iterations can be extended to an optimal solution, then the greedy solution after $j + 1$ iterations can also be extended to an optimal solution by exchanging some interval with one chosen based on the greedy strategy.

The last proof uses what we call an “**charging argument**”. In this case, the charging argument charges each interval of an optimal solution (or arbitrary solution) to a unique interval in the greedy solution. Charging arguments are also used in approximation algorithms (for example, to prove that an algorithm is a k -approximation).

Proof (by charging argument). Given a set of jobs $S = \{a_1, \dots, a_n\}$, let O be an optimal solution of the interval scheduling problem. Let S' be the solution given by the greedy algorithm using earliest finish time. We want to find a one-to-one function $h : O \rightarrow S'$. For any job $j \in O$, define $h(j)$ as the interval $j' \in S'$ that intersects j and has the earliest finish time amongst intervals in S' intersecting j .

We claim that the h is well-defined so $h(j)$ must exist for all j . This is proved by contradiction. Suppose, for contradiction, that there exists a $j \in O$ where $h(j)$ as we defined does not exist. By definition, this means no interval in S' intersects with j . This implies that j is compatible with every interval in S' , so the greedy strategy would have selected j and j would be part of S' . But then if $j \in S'$, j intersects with itself. This is a contradiction. $h(j)$ **exists** for all $j \in O$. Furthermore, $h(j)$ is **unique** because all intervals in S' are mutually disjoint, and every interval in S' that intersects j have distinct finish time.

We then show that h is **injective**, similarly by contradiction. Assume that there are two intervals $j_1, j_2 \in O$ such that $h(j_1) = h(j_2) = j' \in S'$. Without loss of generality, suppose that the finish time of j_1 is earlier than j_2 . j_1 and j_2 are disjoint because they are both in O , which implies that $f_1 \leq s_2 < f_2$. Since $j' \in S'$, the greedy algorithm must have encountered j' before j_1 and j_2 . Thus, $f' \leq f_1$. But then, this implies that $f' \leq f_1 \leq s_2 < f_2$. That is, j' and j_2 do not overlap. This is a contradiction because if they do not intersect, $h(j_2) \neq j'$. Therefore, h is injective.

Since there exists an one-to-one function $h : O \rightarrow S'$, by the charging argument, the greedy algorithm for the interval scheduling problem is optimal. ■

More generally, we have shown that $|O| \leq |S'|$, which implies the optimality of the algorithm (that the

algorithm returns the maximum-size subset).

4.3 Interval Coloring (Interval Partition)

Let us now consider a modified version of the original interval scheduling problem. Suppose that we are given a set of intervals. We want to color all intervals so that intervals with the same color do not intersect while using the minimum number of colors. This problem is also known as the interval partitioning problem.

Similar to interval scheduling, let's take a look at the few possible choices for our greedy strategy:

- Earliest start time
- Earliest finish time
- Shortest interval
- Fewest conflicts

We can show using counterexamples that the last three heuristics, earliest finish time, shortest interval, and fewest conflicts, do not work. We will prove that the earliest start time greedy choice gives us a globally optimal solution for the interval partitioning problem.

The proof for this is somewhat similar to the charging argument that we used to prove the optimality of interval scheduling. We attempt to bound the size of the solution set in order to show that it is optimal (minimal). In the charging argument, we bound the size by showing that there exists an one-to-one function. In this proof, we will skip that part and instead bound the size directly.

Lemma 4.3.1 Given a set of intervals, let d be the maximum number of intersecting intervals at any time. The number of partitions (colors) given by any algorithms that solve the interval partitioning problem must be at least d .

Proof. By contradiction. ■

Lemma 4.3.2 Let d be defined as in the previous lemma. The greedy algorithm using earliest start time produces a solution with at most d partitions.

Proof. Let d' be the number of partitions produced by the greedy algorithm. Suppose for contradiction that the algorithm used more than d partitions. Consider the first time that the greedy algorithm used $d + 1$ partitions. Suppose this happens when the algorithm is trying to assign a partition to the interval j . This implies that there are d intervals intersecting j . Let s_j be the starting time of j . These d intervals must contain s_j . This is because all the previous d intervals have starting time earlier than s_j . So it must be the case that s_j is after or at the starting time of the other d intervals. But then, this implies that there are $d + 1$ overlapping intervals at s_j , contradicting the fact that there are at most d intersecting intervals at any time. ■

Corollary 4.3.3 The greedy algorithm produces a solution with exactly d partitions.

Proof. Follows immediately from the Lemma 4.3.1 and 4.3.2. ■

Hence the theorem:

Theorem 4.3.4 The greedy algorithm using earliest starting time as greedy choice is optimal.

This greedy algorithm is implemented as follows. Suppose, as a precondition, that S is sorted in increasing order of starting time, and that elements in Q contains objects of INTERVALS that are indexed by finish time.

INTERVAL-PARTITIONING(S)

```

1   $d = 0$ 
   // initialize PQ using finish time as key
2   $Q = \text{PRIORITY-QUEUE}(\text{key} = f)$ 
3  for  $i = 1$  to  $S.\text{length}$ 
4       $k = \text{EXTRACT-MIN}(Q)$ 
5      if  $k == \text{NIL}$ 
6           $d = d + 1$ 
7           $\text{interval} = \text{new INTERVAL}(i.f)$ 
8           $\text{INSERT}(Q, \text{interval})$ 
9      if  $i.s > \text{MIN}(Q)$ 
   // if  $i$  is compatible with  $k$ , allocate  $i$  to  $k$ 
10      $k.f = i.f$ 
11      $\text{INSERT}(Q, k)$ 
12  else
   // otherwise, create new classroom  $d + 1$ 
13      $d = d + 1$ 
14      $\text{INSERT}(Q, k)$ 
15      $\text{interval} = \text{new INTERVAL}(i.f)$ 
16      $\text{INSERT}(Q, \text{interval})$ 
17  return  $d$ 
```

This algorithm runs in $O(n \log n)$ time. This is the case regardless of whether or not we consider sorting as part of the algorithm because the n priority queue operation is going to cost $O(n \log n)$ anyway, assuming the priority is implemented as a min-heap.

4.4 Greedy Strategy

The greedy strategy can be generalized as follows:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing (typically by contradiction or induction) that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

The important property that requires us to prove when implementing a greedy algorithm is the **greedy property**. It tells us that we can assemble a globally optimal solution from locally optimal choices. The exchange argument for the proof (the first two proofs that we have seen for interval scheduling) argues that we can modify the globally optimal solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

Chapter 5 Graph Algorithms Using Greedy

5.1 Interval Scheduling as Graph Problems

5.2 Minimum Spanning Tree

5.2.1 Kruskal's Algorithm

5.2.2 Prim's Algorithm

5.3 Single-Source Shortest Path

5.3.1 Dijkstra's Algorithm

Chapter 6 Huffman Encoding

Chapter 7 Generalizing and Formalizing Greedy Algorithms

7.1 Priority Model

For a given problem, we assume that the input items belong to some set \mathcal{J} . For any execution of the algorithm, the input is a finite subset $\mathcal{I} \subset \mathcal{J}$.

Let $f: \mathcal{J} \rightarrow \mathbb{R}$ be a function. We do not place restriction on the complexity or computability of such function. For an input set $\mathcal{I} = \{I_1, \dots, I_n\}$, the function induces a total ordering \preceq on \mathcal{I} , breaking ties using certain rules (for example, by input ID).

For a fixed order priority algorithm, f and \preceq are set initially before the algorithm observes the input set. For adaptive order, the algorithm computes f and \preceq dynamically. More formally, there is a different function f_i and ordering \preceq_i associated with each iteration i where f_i and \preceq depends on the items $\{I_1, \dots, I_j\}$ considered in prior iterations $j < i$.

Let us first consider fixed order algorithm under this priority model. In each iteration k for $1 \leq k \leq n$, the algorithm observes input element $I_k \in \mathcal{I}$ and based on this input and all previous inputs and decisions, the algorithm makes an irrevocable decision D_k about this input item (this typically involves whether to include or discard the current item).

This model gives us the following template for fixed order priority algorithms

```
1   $\mathcal{J}$  = set of all possible inputs
2   $\preceq$  = a total ordering on  $\mathcal{J}$  (typically induced by a function  $f$ )
3   $\mathcal{I} \subset \mathcal{J}$  = actual input to the algorithm
4   $S = \emptyset$  // items already examined by the algorithm
5   $i = 0$ 
6  while  $\mathcal{I} - S \neq \emptyset$ 
7       $i = i + 1$ 
8       $\mathcal{I} = \mathcal{I} - S$ 
9       $I_i = \min_{\preceq} \{I \in \mathcal{I}\}$  // select min element based on the ordering  $\preceq$ 
10     make an irrevocable decision  $D_i$  concerning  $I_i$ 
11      $S = S \cup \{I_i\}$ 
```

This template can be modified to allow the algorithm to determine the total ordering dynamically based on the elements it has observed so far.

```

1   $\mathcal{J}$  = set of all possible inputs
2   $\mathcal{I} \subset \mathcal{J}$  = actual input to the algorithm
3   $S = \emptyset$  // items already examined by the algorithm
4   $i = 0$ 
5  while  $\mathcal{I} - S \neq \emptyset$ 
6       $i = i + 1$ 
7       $\preceq_i$  = a total ordering on  $\mathcal{J}$  (typically induced by a function  $f_i$ )
8       $\mathcal{I} = \mathcal{I} - S$ 
9       $I_i = \min_{\preceq_i} \{I \in \mathcal{I}\}$  // select min element based on the ordering  $\preceq_i$ 
10     make an irrevocable decision  $D_i$  concerning  $I_i$ 
11      $S = S \cup \{I_i\}$ 
12      $\mathcal{J} = \mathcal{J} - \{I \in \mathcal{I} \mid I \preceq_i I_i\}$ 

```

For greedy algorithm, the algorithm does not have knowledge of the input other than the elements it has observed so far. Sometimes, we allow the algorithm to have some easily computed global information such as the size of the input. The input is said to be chosen by an adversary.

This generalization applies to a broader category of algorithms known as priority-based algorithms. Informally, greedy algorithms always assume that the current iteration could be the last one and the current item being considered might as well be the last item. Hence, an optimal decision is immediate when the algorithm finishes. A more general priority algorithm that are not necessarily greedy does not have such restriction

This formalization was first given by Allan Borodin *et al.* in *(Incremental) Priority Algorithms* (Thirteen Annual ACM-SIAM Symposium on Discrete Algorithms, January 2002) [3].

7.2 Global Optimality of Greedy Algorithms

(From Allan Borodin's notes)

Most greedy algorithms are not optimal. The method we use to show that a greedy algorithm is optimal (when it is) known as the exchange argument often proceeds as follows. At each stage i , we define our partial solution to be promising if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage 0 is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follows. Say that stage $i + 1$ occurs, and that the partial solution after stage i is S_i and that the partial solution after stage $i + 1$ is S_{i+1} , and we know that there is an optimal solution S_{opt} that extends S_i ; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1} . S_{i+1} extends S_i by taking only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1} , and we can just let $S'_{opt} = S_{opt}$ and we are done. The hard part of the induction

step is if S_{opt} does not extend S_{i+1} . In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1} and
- S'_{opt} has value (cost, profit, etc.) at least as good as S_{opt} , so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any solution other than itself. Therefore, if we have proven the above, we know that the solution constructed by the greedy algorithm must be optimal.



Dynamic Programming

8	Weighted Interval Selection	51
8.1	Dynamic Programming	
8.2	Weighted Interval Selection	
8.3	Computing the Optimal Weight	
8.4	Computing the Optimal Solution	
9	Knapsack Problem	55
9.1	Knapsack	
9.2	DP Algorithm for 0-1 Knapsack	
9.3	A Different DP Algorithm	
9.4	FPTAS Approximation for Knapsack	
10	Graph Algorithms Using DP	59
10.1	Single-Source Shortest Path	
10.2	Maximum Length Path Falls	
10.3	All-Pairs Shortest Paths	
10.4	Traveling Salesman Problem	
11	Bioinformatics	65
11.1	Dynamic Programming in Computational Biology	
11.2	Longest Common Subsequence	
11.3	Edit Distance	

Chapter 8 Weighted Interval Selection

8.1 Dynamic Programming

Dynamic programming (DP) is a method for solving complex problem by breaking it down to a collection of simpler subproblems, solving each of those subproblems, and storing their solutions. The next time the same subproblem occurs, instead of recomputing the solution, the algorithm can just use the previously computed solution. Each subproblem is indexed in some way to facilitate the lookup. The technique of storing solutions to subproblems is called **memoization**.

Let's consider the weighted interval selection problem.

8.2 Weighted Interval Selection

The objective of the weighted interval selection problem is to find a non-intersecting set of intervals so that the sum of the weights of intervals in the set is maximized. More formally, given a set of jobs/activities $\{a_1, \dots, a_n\}$ with start time s_i and finish time f_i and weight w_i , we want to find a set S of mutually compatible jobs with highest total weight $\sum_{i \in S} w_i$.

Note that when $w_i = 1$ for all $i = \{1, \dots, n\}$, the weighted interval selection problem is the same as simple interval scheduling.

We first consider greedy algorithms. Possible choices for the greedy choice include: weights, interval length, etc. Unfortunately, all the possible ways of ordering the input items will not only fail to give an optimal solution, but can in fact produce arbitrarily bad solutions for some instances. With a generalized greedy approach, it can be proved that no greedy algorithm can produce a good solution. With certain modifications/extensions to greedy, the algorithm can produce a good approximation (by allowing revocable decisions) and even optimal solution (using local ratio/primal dual algorithms with a reverse delete phase).

8.3 Computing the Optimal Weight

8.3.1 The Dynamic Programming Approach

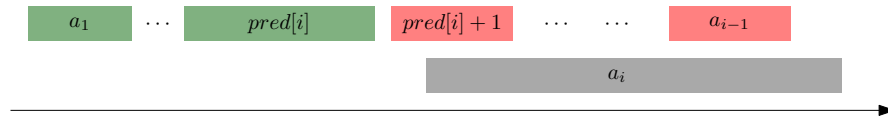
Assume that the jobs (intervals) have been sorted by non-decreasing finish time. Then, in an optimal solution O , either the last interval I_n was selected, or it was not. If not, then we must be using an

optimal solution for the first $n - 1$ intervals. If a_n is in O , then no interval in O can end after the starting time of a_n .

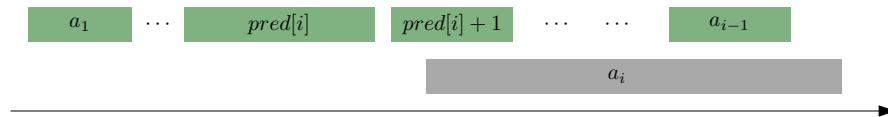
Let us now make this intuition more concrete. Given that the jobs are sorted by non-decreasing finish time, let $pred(i)$ be the largest index $j < i$ such that $f_j < s_i$. This means that jobs a_1, \dots, a_j are compatible with a_i , but a_{j+1}, \dots, a_{i-1} are not compatible.

Let O be an optimal solution. For each job a_i , there are two possibilities:

- Job $a_i \in O$: This implies that $\{pred[i] + 1, \dots, a_{i-1}\}$ are all incompatible with a_i . So, we must select jobs from $\{a_1, \dots, pred[i]\}$.



- Job $a_i \notin O$: This implies that we must select jobs from $\{a_1, \dots, a_{i-1}\}$.



Let define $V : \{a_1, \dots, a_n\} \rightarrow \mathbb{R}$ be defined such that

$$V(i) = \max \text{ total weight of compatible jobs from } \{a_1, \dots, a_i\}$$

More formally, V can be expressed as this recurrence

$$V(i) = \begin{cases} 0 & i = 0 \\ \max\{V(i-1), w_i + V(pred[i])\} & i > 0 \end{cases}$$

Lemma 8.3.1 The two definitions of V are equivalent.

Proof. By induction. ■

Sometimes, the first definition of V in English is represented as an array indexed by i and referred to as a **semantic array**. The second equation of V is called the **Bellman equation**.

This recurrence gives us a clear recursive solution for the weighted interval selection problem.

Consider the time complexity of the naive recursive implementation. $pred[i]$ can be computed using binary search in $O(\log n)$ time, and for all of the n values to be considered, this takes $O(n \log n)$ time. However, the time spent on computing $pred$ turns out to be rather insignificant compared to the running time of the main algorithm.

$$T(n) = T(n-1) + T(pred[n]) \leq T(n-1) + T(n-2) \in \Theta(\varphi^n)$$

where $\varphi \approx 1.618$ is the golden ratio. This is a Fibonacci recurrence.

This is clearly bad. Some solutions are being computed many times unnecessarily. Recall that one of the most important component of dynamic programming is memoization. In many cases, without memoization, dynamic programming simply becomes divide and conquer, or even worse, recurrence that runs in exponential time like this one. Memoization allows us to remember the results that we have already computed and reuse them if needed.

8.3.2 Top-Down DP

In our first dynamic programming implementation, we will store the previously computed values in the array M indexed by the index of each interval.

Assume that prior to calling $\text{COMPUTE-OPT-DP}(n)$, the intervals are sorted in nondecreasing order based on finish time, pred has been precomputed in $O(n \log n)$ time, and M is a global array of size n with every entry set to 0 initially.

$\text{COMPUTE-OPT-DP}(j)$

```

1  if  $M[j] == \text{NIL}$ 
2       $M[j] = \max\{\text{COMPUTE-OPT-DP}(j-1), w_j + \text{COMPUTE-OPT-DP}(\text{pred}[j])\}$ 
3  return  $M[j]$ 
```

For each $i = \{1, \dots, n\}$, there is only one call to $\text{COMPUTE-OPT-DP}(i)$ because previously computed values are stored in the array M . Therefore, there are at most $O(n)$ calls to the recursive procedure. Sorting by finish time and computing pred takes $O(n \log n)$ time, so the overall time complexity of this implementation is $O(n \log n)$. Much better than our original exponential implementation!

8.3.3 Bottom-Up DP

The bottom-up approach differs from the top-down approach in that values of M are precomputed. Instead of using recursive call and checking if $M[i] == \text{NIL}$, we can simply reference to indices of the array where previously computed values are stored. Assume the same preconditions are met prior to calling $\text{COMPUTE-OPT-DP-BOTTOM-UP}$.

$\text{COMPUTE-OPT-DP-BOTTOM-UP}()$

```

1  for  $j = 1$  to  $n$ 
2       $M[j] = \max\{M[j-1], w_j + M[\text{pred}[j]]\}$ 
3  return  $M[n]$ 
```

It can be shown using a simple loop invariant and induction that at the beginning of the i th iteration, $M[j]$ is already computed for all $j < i$ and thus we will not get a null pointer error.

This approach has the same time complexity as the top-down approach.

8.3.4 Comparison of Two Approaches

Top-down may be preferred when not all sub-solutions need to be computed on some inputs. This helps us save some time.

Bottom-up may be preferred when all sub-solutions will always need to be computed for all inputs. The bottom-up approach could be faster as it prevents unnecessary recursive calls which results in unnecessary random memory access. This is because even though recursive dynamic programming does not compute repeated results, there could still be redundant recursive calls, resulting in a large overhead.

8.4 Computing the Optimal Solution

In the previous section, we come up with a dynamic programming algorithm that computes the optimal weight of the solution set for the weighted interval selection problem. Now, we will extend that algorithm so that we get the actual solution set (subset of S that maximizes weight). We only need a really simple modification from the original definition of V . Recall that V is defined as

$$V(i) = \begin{cases} 0 & i = 0 \\ \max\{V(i-1), w_i + V(pred[i])\} & i > 0 \end{cases}$$

Instead of calculating values, we want a subset of the input set.

$$S(i) = \begin{cases} \emptyset & i = 0 \\ S(i-1) & \text{if } V(i) = V(i-1) \\ S(pred[i]) \cup \{a_i\} & \text{otherwise} \end{cases}$$

We can either precompute V and use it directly in the implementation of S . Or alternatively, we can compute V and S simultaneously.

Chapter 9 Knapsack Problem

9.1 Knapsack

In the knapsack problem, we are given a set of n items I_1, \dots, I_n and a size bound B where each item $I_j = (w_j, v_j)$ with w_j being the weight of the item and v_j the value of the item.

A subset of items S is feasible if the sum of the weights of items in S is at most B . The goal of the knapsack problem is to find a feasible set S that maximizes the sum of the values of items in S .

9.2 DP Algorithm for 0-1 Knapsack

We define the semantic array

$$V(i, b) = \text{max profit possible using the first } i \text{ items within weight } b$$

with corresponding Bellman equation

$$V(i, b) = \begin{cases} 0 & \text{if } i = 0 \text{ or } b = 0 \\ \max\{C, D\} & \text{if } w_i \leq b \end{cases}$$

where $C = V(i-1, b)$ and $D = V(i-1, b - w_i) + v_i$.

C corresponds to the decision to not include the i th item in the knapsack, and D correspond to the decision to include the i th item, in which case we deduce w_i from the remaining weight of the knapsack and add v_i to the total profit. Like all DP problems, we need to prove that the semantic array is equivalent to the Bellman equation.

KNAPSACK(S, B)

```
1   $n = S.length$ 
2  for  $b = 0$  to  $B$ 
3       $M[0, b] = 0$ 
4  for  $i = 1$  to  $n$ 
5      for  $b = 0$  to  $B$ 
6          if  $S[i].w > b$ 
7               $M[i, b] = M[i-1, b]$ 
8          else
9               $M[i, b] = \max\{M[i-1, b], S[i].v + M[i-1, b-1]\}$ 
10  $M[n, B]$ 
```

The input to the algorithm: S is the set of all items to be considered, and B is the weight bound of the knapsack. This approach makes an important assumption that the weights are integer.

This DP algorithm solving the 0-1 knapsack problem with n items and weight bound B runs in $\Theta(nB)$ time and uses $\Theta(nB)$ space. As we can see, the running of this algorithm is, unfortunately, not polynomial in the input size. It is **pseudo-polynomial**. It is polynomial in $\log B + \sum_{i=1}^n (\log v_i + \log w_i)$.

If B is small (a polynomial in n), then the algorithm runs in polynomial time. No known exact algorithm runs in polynomial time. The 0-1 knapsack problem is NP-complete.

9.3 A Different DP Algorithm

Consider version of the knapsack problem with a different restriction that the values v_i are integer and small ($V = \sum_{i=1}^n v_i \ll B$). In this case, it makes more sense to derive an algorithm in terms of v instead of B .

The semantic array is defined as follows

$$W(i, v) = \begin{cases} \text{minimum weight required to obtain at least} \\ \text{profit } v \text{ using a subset of } \{I_1, \dots, I_i\} & \text{if possible} \\ \infty & \text{otherwise} \end{cases}$$

The goal is to compute $\max\{v \mid W(n, v) \leq B\}$.

The corresponding Bellman equation is

$$W(i, v) = \begin{cases} \infty & \text{if } i = 0 \text{ and } v > 0 \\ 0 & \text{if } i \leq 0 \text{ or } v \leq 0 \\ \max\{C, D\} & \text{otherwise} \end{cases}$$

where $C = W(i-1, v)$ and $D = W(i-1, v - v_i) + w_i$.

This algorithm is still pseudo-polynomial but the complexity is now $O(nV)$ where $V = v_1 + \dots + v_n$. This is more efficient when $V \ll B$.

9.4 FPTAS Approximation for Knapsack

Because 0-1 knapsack is NP-complete, it is also of theoretical interest to find an efficient approximation algorithm. In the case of the knapsack problem, it is possible to find an algorithm that approximates the result within a specific degree on all possible inputs.

The idea behind the algorithm is as follows: The high order bits/digits of the value v can be used to determine an approximate solution. The fewer high order bits we use, the faster the algorithm runs, but also the worse the approximation.

The goal is to scale the values v using a parameter (rounding factor) ε so that a $(1 + \varepsilon)$ approximation is obtained with time complexity polynomial in n and $1/\varepsilon$.

More precisely, let $\tilde{v} = \lceil v_i/b \rceil b$. This ensures that the constraint that v must be integer for the second algorithm is satisfied. By rounding the values, we get an approximation \tilde{v} of v that is divisible by b . This means we can divide all values of v by ε and get an equivalent problem. Let $\hat{v} = \tilde{v}/b$. This allows us to reduce the size of v so that it becomes smaller than n (or at least bounded by a polynomial in n).

KNAPSACK-APPROX(ε, S)

```

1   $b = (\frac{\varepsilon}{2n}) \cdot \max_i v_i$ 
2  for  $i$  in  $S$ 
3       $\tilde{v} = \lceil v_i/b \rceil b$ 
4       $\hat{v} = \tilde{v}/b$ 
5       $i.v = \hat{v}$ 
6  KNAPSACK-V( $S, \max_i \{v_i\}$ )
```

where KNAPSACK-V is the second DP algorithm discussed above.

KNAPSACK-APPROX is a **FPTAS** (fully polynomial time approximation scheme) algorithm for the 0-1 knapsack problem.

Definition 9.4.1 — FPTAS and PTAS.

An **FPTAS** (Fully Polynomial Time Approximation Scheme) algorithm is one that is polynomial in the encoding of the input and $1/\varepsilon$.

A **PTAS** (Polynomial Time Approximation Scheme) algorithm is one that is polynomial to the encoding of the algorithm but can have any complexity in terms of $1/\varepsilon$ ($1/\varepsilon$ term may not be polynomial).

Chapter 10 Graph Algorithms Using DP

10.1 Single-Source Shortest Path

We have discussed single-source shortest path when we talked about greedy algorithms.

Previously, we have restricted our discussion to graphs without negative weights/cycles. In this chapter, we will talk about algorithms that can be used to general graphs (even one with negative cycles).

Recall that we store the weight of the path from the source s to each node u at $u.d$. Consider a shortest path p from s to a vertex v . If there is a vertex u immediately preceding v on such path, then the path from s to u must also be a shortest path.

One caveat we must consider is that if there is a negative cycle in the graph, one can keep traversing through that negative cycle and yield a path with lower weight. If this happens, the algorithm can get stuck in a negative cycle and d values may never converge. Hence, we need to put one additional constraint: only consider path within certain path length. Let $\delta_k(s, v)$ denote the shortest path from s to v using at most k edges. Similarly, for a vertex v , let $v.d_k$ be the weight of the path from s to v using at most k edges.

In addition, we will define some notations to aid our discussion. If v is immediately reachable from u via one edge, we write $u \rightarrow v$. If v is reachable from u via multiple edges, we write $u \rightsquigarrow v$.

10.1.1 Bellman-Ford

An obvious dynamic programming solution follows from this Bellman equation

$$v.d_k = \begin{cases} 0 & \text{if } k = 0 \text{ and } v = s \\ \infty & \text{if } k = 0 \text{ and } v \neq s \\ \min\{A, B\} & \text{otherwise} \end{cases}$$

where $A = v.d_{k-1}$ and $B = \min\{u.d_{k-1} + w(u, v) \mid (u, v) \in E\}$. To prove the correctness of this equation, we can use induction on the length of the path k .

For a simple shortest path, there are at most $|V| - 1$ edges. There will be $O((|V| - 1)|V|) \in O(|V|^2)$ because we may need to update the d field of all $|V|$ vertices. Each call takes $O(|E|)$ time in order to evaluate $B = \min\{u.d_{k-1} + w(u, v) \mid (u, v) \in E\}$. Therefore, the overall running time of the algorithm is in $O(|V|^2|E|)$.

The dynamic programming algorithm is a somewhat worse version of the Bellman-Ford algorithm, which runs in $O(|V| \cdot |E|)$ time with space complexity $O(|V| + |E|)$.

This rather unfortunate $O(|V|^2|E|)$ running time partly originates from the fact that we are updating the d field of each v for all $v \in |V|$. But if for every v , we are taking $O(|E|)$ time to examine all the edges to update $v.d$ anyway, we might as well just update all the d fields whenever we run a pass through E . This gives us an algorithm with running time $O(|V| \cdot |E|)$. The $|V|$ comes from the $|V| - 1$ iterations to optimize (“relax”) each edge on the simple shortest path of at most $|V| - 1$ edges, and the $|E|$ comes from the pass through E during each of the $|V| - 1$ iterations to update the d fields.

Using the following set of lemmas, we will show that it is safe to do so.

Lemma 10.1.1 — Triangle Inequality. For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof. By contradiction.

Suppose the claim does not hold. In particular, let $\delta(s, v) > \delta(s, u) + w(u, v)$. This implies that the shortest path from s to v has a higher weight than the path $s \rightsquigarrow u \rightarrow v$. But this contradicts the fact that $\delta(s, v)$ is the shortest path. ■

Lemma 10.1.2 — Upper-bound Property. We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ is equal to $\delta(s, v)$, it never changes.

Proof. By induction on the number of relaxations.

Base case: $v.d \geq \delta(s, v)$ after initialization since $v.d = \infty$ for all $v \in V - \{s, v\}$, and for $s.d = 0 = \delta(s, s)$.

Inductive step: Consider the relaxation of an arbitrary edge (u, v) . Assume that $x.d \geq \delta(s, x)$ for all $x \in V$ prior to relaxation. The relaxation of (u, v) can only affect $v.d$. If $v.d$ is updated, then

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{inductive hypothesis} \\ &\geq \delta(s, v) && \text{triangle inequality} \end{aligned}$$

The invariant is maintained after relaxing (u, v) .

Once $v.d = \delta(s, v)$, $v.d$ cannot decrease because $v.d \geq \delta(s, v)$ always holds and relaxation cannot increase $v.d$. ■

Lemma 10.1.3 — No-path Property. If there is no path from s to v , then we always have $v.d = \delta(s, v) = \infty$.

Proof. Follows immediately from the upper-bound property. ■

Lemma 10.1.4 — Convergence Property. If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v) , then $v.d = \delta(s, v)$ at all times afterward.

Proof. By upper bound property, once $u.d = \delta(s, u)$, it no longer changes. In particular, after relaxing (u, v) , we have

$$\begin{aligned} v.d &\leq u.d + w(u, v) && \text{Lemma 24.13, CLRS} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{Lemma 24.1, CLRS} \end{aligned}$$

(Note that Lemma 24.1 was proved in the chapter covering Dijkstra's algorithm in my [notes on data structures](#), CSC265 Notes)

By the upper bound property, $v.d \geq \delta(s, v)$. It follows that $v.d = \delta(s, v)$. ■

Lemma 10.1.5 — Path-relaxation Property. If $p = v_0, v_1, \dots, v_k$ is a shortest path from $s = v_0$ to v_k , and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of other relaxations that occur, including those intermixed with relaxations of edges in p .

Proof. Proof by induction on the i th edge of p that is relaxed.

Base case: $i = 0$. This is before any edge of p have been relaxed. From initialization, $v_0.d = \delta(s, s) = 0$. By upper bound property, $v_0.d$ does not change once it converges to $\delta(s, v_0)$

Inductive step: Assume that $v_{i-1}.d = \delta(s, v_{i-1})$. We relax edge (v_{i-1}, v_i) . By convergence property, after relaxation of this edge, $v_i.d = \delta(s, v_i)$, and by upper bound property, this equality is maintained thereafter. ■

From the path-relaxation property, we can prove the correctness of Bellman-Ford on a graph without negative-weight cycles.

Theorem 10.1.6 — Correctness of Bellman-Ford (without negative cycle). Let BELLMAN-FORD be run on a weighted directed graph $G = (V, E)$ without negative cycles, with source s and weight function $w: E \rightarrow \mathbb{R}$. Then, the algorithm terminates and when it does, $v.d = \delta(s, v)$ for all vertices $v \in V$, from which we can construct a predecessor subgraph G_π that is a shortest path tree.

Proof. We first prove that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If v is not reachable from s , the claim follows from the no-path property. If v is reachable from s , then consider a shortest path $p = v_0, v_1, \dots, v_k$ where $v_0 = s$ and $v_k = v$. Because the shortest path is simple in a graph without negative cycle, so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the outer loop of the algorithm relaxes all $|E|$ edges. Among the edges relaxed in the i th iteration in the $|V| - 1$ total iterations, is (v_{i-1}, v_i) . By path-relaxation property, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$ even if other relaxation steps are intermixed

within the sequence of relaxations of edges $(v_0, v_1), \dots, (v_{k-1}, v_k)$. Hence, the claim holds when v is reachable from s .

By the predecessor-subgraph property (Lemma 24.17, CLRS), G_π is a shortest-path tree.

Termination of the algorithm follows from the fact that the loop-counters of both loops are strictly increasing. ■

Let us finally examine how Bellman-Ford detects negative cycle.

Theorem 10.1.7 — Correctness of Bellman-Ford (negative cycles). If $v.d$ for some vertex v fails to converge after $|V| - 1$ passes, then there exists a negative-weight cycle reachable from s .

Proof. After $|V| - 1$ passes, if we find an edge that can be relaxed, it means that the current path from s to some vertex is not simple and vertices are repeated on that path. Since this cyclic path has less weight than any simple path, the cycle must be a negative-weight cycle. ■

The proofs give a clear picture of how the algorithm should look like.

```

BELLMAN-FORD( $G = (V, E), w, s$ )
1  for  $v \in V - \{s\}$ 
2       $v.d = \infty$ 
3       $v.parent = \text{NIL}$ 
4   $s.d = 0$ 
5  for  $i = 0$  to  $|V| - 1$ 
6      for each edge  $(u, v) \in E$ 
7          if  $v.d > u.d + w(u, v)$ 
8               $v.d = u.d + w(u, v)$ 
9               $v.parent = u$ 
10 for each edge  $(u, v) \in E$ 
11     if  $v.d > u.d + w(u, v)$ 
12         return negative cycle
13 return no negative cycle,  $G_\pi$ 

```

Note that G_π is constructed by following the parent pointer (if exists) of each vertex to s .

As we alluded to earlier, this improved version of the Bellman-Ford algorithm runs in $O(|V| \cdot |E|)$ time.

10.2 Maximum Length Path Fails

Bellman-Ford solves the single-source shortest path problem on a general graph efficiently. It is natural to ask if it is possible to find a **longest simple path** on such general graph. It might be tempting to

simply modify the min dynamic programming function used Bellman-Ford to max. Although this approach works find on a DAG, the algorithm does not guarantee that the resulting path is a simple path when run on a graph with cycles and the algorithm may get stuck in a positive cycle.

This problem is NP-hard, for which no known polynomial-time algorithm exists. A special case of this problem is the Hamiltonian path problem: does a graph $G = (V, E)$ has a simple path of length $|V| - 1$. The Hamiltonian path problem is a variant of the NP-hard **traveling salesman problem** (TSP).

10.3 All-Pairs Shortest Paths

10.3.1 Matrix Multiplication

10.3.2 Floyd-Warshall

10.3.3 Johnson's Algorithm

10.4 Traveling Salesman Problem

Chapter 11 Bioinformatics

11.1 Dynamic Programming in Computational Biology

Dynamic programming is one of the most commonly used algorithm design techniques in bioinformatics and computational biology. In this chapter, we discuss several problems of practical interest in computational biology that can be solved using dynamic programming. Dynamic programming is also the foundation of many more powerful algorithms such as BLAST (basic local alignment search tool). BLAST is probably one of the most used tools among biologists. It allows one to use a query string to search against a database of biological sequences, and return sequences that contain the query or are closely related to the query sequence. The core of any such algorithm is sequence alignment, which is solved using dynamic programming.

Bioinformatics is a subject as much about biology as it is about strings. Strings arise naturally as biological sequences. One of the most common thing to do with strings is comparing them. By comparing two strings and locating the common subsequence, we can find conserved protein motifs, infer evolutionary relationships, and even predicting function of newly discovered genes/proteins. We will look at DP algorithms that solve the so-called pairwise sequence alignment problem.

11.2 Longest Common Subsequence

11.3 Edit Distance

IV

Network Flow

12	Ford-Fulkerson	69
12.1	Flow Network	
12.2	Ford-Fulkerson Method	
12.3	Correctness of Ford-Fulkerson	
13	Edmonds-Karp Algorithm	79
14	Maximum Bipartite Matching	81

Chapter 12 Ford-Fulkerson

12.1 Flow Network

12.1.1 Definitions

We can interpret a graph as a “flow network” and use it to model the flow of materials such as water through pipes, electricity through wires, or goods through supply chain. Each flow network will have a source where everything originates from and a sink (destination) where everything arrives at. We first formally define a **flow network**.

Definition 12.1.1 — Flow Network. A **flow network** $G = (V, E)$ is a directed network in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. Additionally, if $(u, v) \in E$, then $(v, u) \notin E$. If $(u, v) \notin E$, then $c(u, v) = 0$.

In particular, in a flow network, there is a **source** s and **sink** t , and every vertex lies on some path from from s to t . That is, $\forall v \in V$, there exists a path p such that $p = s \rightsquigarrow v \rightsquigarrow t$.

Definition 12.1.2 — Flow. Let $G = (V, E)$ be a flow network with capacity function $c : V \times V \rightarrow \mathbb{R}$, and let s be the source and t be the sink of the network. A **flow** in G is a function $f : V \times V \rightarrow \mathbb{R}$ that satisfies the following properties:

- Capacity constraint: for all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$.
- Flow conservation: for all $u \in V - \{s, t\}$, $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$. That is, the flow into u must equal the flow out from u . When $(u, v) \notin E$, $f(u, v) = 0$.

We define the **value** of a flow f to be

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Sometimes, we abuse notations a little bit and omit the summation by writing $f(s, V)$ where V is the set of vertices. Some also uses the notation $f_{in}(v)$ and $f_{out}(v)$ to denote the flow into and out from v , respectively.

■ **Example 12.1 — Example of a Flow Network.** Below is an example of flow network. On each edge $(u, v) \in E$, we write $f(u, v)/c(u, v)$. The slash is simply a separator and does not mean division.

■

The goal of the maximum flow problem is to find a flow f that maximizes the value of the flow $|f|$. This is also equivalent to finding the maximum flow into the sink t .

The max-flow problem is interesting not only because its practical applications, but also its relation to other problems. Many other problems are closely related to the max-flow problem can be polynomial-time reduced to the max-flow problem.

12.1.2 Antiparallel Edges and Multiple Sources/Sinks

In our definition of a flow network, we requires there to be exactly only one source and one sink, and we prohibit antiparallel edges (i.e. if $(u, v) \in E$, then $(v, u) \notin E$).

However, it is easy to convert a graph with multiple sources and/or sinks and antiparallel edges into a flow network that fit our original definition.

Theorem 12.1.1 Suppose that a graph G contains an edge (u, v) . We can create a new flow network $G' = (V', E')$ where $V' = V \cup \{x\}$ and $E' = E - \{(u, v)\} \cup \{(u, x), (x, v)\}$. G' is obtained by creating a new vertex x and replacing (u, v) by $(u, x), (x, v)$. We set the capacity $c(u, x) = c(x, v) = c(u, v)$. The maximum flow in G' is the same as the maximum flow in G .

Proof. ■

Theorem 12.1.2 Let G be a graph with multiple source vertices and multiple sink vertices. Let G' be constructed as follows. Suppose G has sources $S = \{s_1, \dots, s_i\}$ and sinks $T = \{t_1, \dots, t_j\}$. We add a supersource s and edges (s, s_i) with capacity $c(s, s_i) = \infty$ for all $s_i \in S$. We add a supersink t and edges (t_j, t) for all $t_j \in T$ with capacity $c(t_j, t) = \infty$. More formally, $G' = (V', E')$ such that $V' = V \cup \{s\} \cup \{t\}$ and $E' = E \cup \{(s, s_i) \mid s_i \in S\} \cup \{(t_j, t) \mid t_j \in T\}$. G' has the same maximum flow as G .

Proof. ■

This theorem will become useful later when we look at the maximum bipartite matching problem.

12.2 Ford-Fulkerson Method

Ford-Fulkerson method is a general approach to solve the maximum flow problem. It uses an incremental improve approach by gradually increasing the flow until we reach the maximum. More generally, Ford-Fulkerson can be viewed as a greedy local search algorithm. It is called a method or a scheme because most parts of it can be implemented differently but the general approach remains the same.

The table below summarizes some of the common implementation of the Ford-Fulkerson method. At the end of this chapter, we will discuss the Edmonds-Karp algorithm, and in a later chapter, we will

discuss a different algorithm known as the push-relabel algorithm, which uses a different approach than Ford-Fulkerson.

Algorithm	Implementation	Complexity
Edmonds-Karp (1970)	BFS for finding augmenting path	$O(E ^2 V)$
Dinitz (1970)	BFS + DFS for finding augmenting path	$O(V ^2 E)$
Capacity Scaling (Dinitz, 1972)	Capacity scaling heuristic	$O(E ^2 \log c_{\max})$
Dinitz-Gabow (1973)	Capacity scaling heuristic	$O(V E \log c_{\max})$

Table 12.1: Comparison of multiple algorithms based on the Ford-Fulkerson. c_{\max} in capacity scaling denotes the largest capacity in the network.

12.2.1 Idea Behind Ford-Fulkerson

The idea behind Ford-Fulkerson is quite simple. For each iteration of the algorithm, we find a path from s to t , increase the flow along the path until we hit a bottleneck at one of the edges on the path (the edge with the smallest remaining capacity). This is called augmenting flow, and the path is called augmenting path. When augmenting the flow along the augmenting path, we may also want to decrease the flow along each residue (i.e. reverse) edge so that we can “undo” bad augmentation choices. We will show that when the algorithm terminates, it correctly returns the max-flow.

Note that this is more precisely, only the partial correctness. Surprisingly, depending on the implementation and capacities, the algorithm may not terminate. In particular, the Ford-Fulkerson method may fail to terminate if edge capacities are irrational [13]. In his 1993 paper, Zwick showed a minimal example flow network (with 6 vertices and 8 edges) on which Ford-Fulkerson fails to terminate.

FORD-FULKERSON(G, s, t)

- 1 $f = 0$ for all $(u, v) \in E$
- 2 **while** there exists an augmenting path p in residue network G_f
- 3 augment flow f along p
- 4 **return** f

12.3 Correctness of Ford-Fulkerson

12.3.1 Residue Network and Augmentation

Let us make the idea more concrete by formalizing it and proving the correctness of the Ford-Fulkerson method. We first define what a residue network is.

Given a flow network G , the residue network G_f contains edges with capacities that represent how much more flow we can push through each edge. For each edge (u, v) , we will also add backward edges (v, u) to the residue network that can at most cancel out the flow through the edge (u, v) .

Definition 12.3.1 — Residue Capacity. Let $G = (V, E)$ be a flow network with source s and sink t . Let f be a flow in G , and consider a pair of vertices $u, v \in V$. The **residue capacity** $c_f(u, v)$ is defined as

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

By assumption, since G is a flow network, $(u, v) \in E$ IFF $(v, u) \notin E$, exactly one case in the definition applies.

Definition 12.3.2 — Residue Network. Let $G = (V, E)$ be a flow network and a flow f . The **residue network** of G induced by f , denoted G_f is $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$.

A residue network contains at most twice many edges as the original flow network. If an edge does not exist in G (has zero capacity), it does not exist in G_f either.

Definition 12.3.3 — Augmentation. Let $G = (V, E)$ be a flow network with flow f . G_f is the residue network of G induced by f . Suppose that f' is a flow in G_f . We define the **augmentation** of f by f' , denoted $f \uparrow f'$, to be a function $(f \uparrow f') : V \times V \rightarrow \mathbb{R}$ such that

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that because we allow backward edges in a residue network, we need to subtract $f'(v, u)$ when augmenting f and f' on edge (u, v) . The notion of pushing flow through a reverse edge is called cancellation.

Lemma 12.3.1 Let $G = (V, E)$ be a flow network with source s and sink t , and let f be a flow in G . Let G_f be the residue network of G induced by f , and let f' be a flow in G_f . Then $f \uparrow f'$ is a flow in G and $|f \uparrow f'| = |f| + |f'|$.

Proof. To show that $f \uparrow f'$ is a flow in G , it suffices to show the capacity constraint and flow conservation properties hold.

Capacity constraint: Let $u, v \in V$. Then,

$$(f \uparrow f')(u, v) = f(u, v) + f'(u, v) - f'(v, u) \leq f(u, v) + f'(u, v) \leq f(u, v) + c_f(u, v) = f(u, v) + c(u, v) - f(u, v) = c(u, v).$$

Flow conservation: Fix $u \in V$. Let $In(u) = \{v \in V \mid (u, v) \in E\}$ and $Out(u) = \{v \in V \mid (v, u) \in E\}$. That is, $In(u)$ is the set of vertices with edges to u and $Out(u)$ is the set of vertices with edges from u .

$$\begin{aligned} \sum_{v \in Out(u)} (f \uparrow f')(u, v) - \sum_{v \in In(u)} (f \uparrow f')(v, u) &= \sum_{v \in Out(u)} f(u, v) + \sum_{v \in Out(u)} f'(u, v) - \sum_{v \in Out(u)} f'(v, u) - \\ &\quad \sum_{v \in In(u)} f(v, u) - \sum_{v \in In(u)} f'(v, u) + \sum_{v \in In(u)} f'(u, v) \end{aligned}$$

Regrouping the terms yields

$$\sum_{v \in \text{Out}(u)} f(u, v) - \sum_{v \in \text{In}(u)} f(v, u) + [\sum_{v \in \text{Out}(u)} f'(u, v) + \sum_{v \in \text{In}(u)} f'(u, v)] \\ - [\sum_{v \in \text{Out}(u)} f'(v, u) + \sum_{v \in \text{In}(u)} f'(v, u)],$$

which is equivalent to $\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u)$. We can extend the summation from $\text{In}(u)$ and $\text{Out}(u)$ to V because all the additional terms are 0 by definition of flow and the fact that there $(u, v) \notin E$ for $v \in V - \text{Out}(u)$ and $(v, u) \notin E$ for $v \in V - \text{In}(u)$.

By flow conservation of f and f' individually,

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u) = 0, \\ \text{which implies that } \sum_{v \in \text{Out}(u)} (f \uparrow f')(u, v) = \sum_{v \in \text{In}(u)} (f \uparrow f')(u, v).$$

To show that $|f \uparrow f'| = |f| + |f'|$, set $u = s$ in the previous derivation, and we have

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in \text{Out}(s)} (f \uparrow f')(s, v) - \sum_{v \in \text{In}(s)} (f \uparrow f')(v, s) \\ &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \\ &= |f| + |f'|. \end{aligned}$$

■

12.3.2 Augmenting Paths

Definition 12.3.4 — Augmenting Path. Given a flow network $G = (V, E)$ with source s and sink t and flow f , an **augmenting path** p is a simple path from s to t in the residue graph G_f of G induced by f .

Definition 12.3.5 — Residue Capacity of an Augmenting Path. Let p be an augmenting path for the flow network G . Then, the **residue capacity** of p is defined as the maximum amount by which we can increase the flow on each edge in the p , or more formally,

$$c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ is on } p\}$$

Definition 12.3.6 — Flow of an Augmenting Path. Let $G = (V, E)$ be a flow network, f be a flow in G , and let p be an augmenting path in G_f . Define the flow of the augmenting path $f_p : V \times V \rightarrow \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & (u, v) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

Lemma 12.3.2 The flow f_p on an augmenting path p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Proof. We show that f_p satisfies the capacity constraint and flow conservation.

Capacity constraint: If (u, v) is on p , then $f_p(u, v) = c_f(p) \leq c_f(u, v)$ by definition. If (u, v) is not on p , then $f_p(u, v) = 0 \leq c_f(u, v)$.

Flow conservation: Fix $u \in V - \{s, t\}$. First, consider the case where u is on p . Say, edges (w, u) and (u, v) are on p . Then, since p is a simple path and $u \in V - \{s, t\}$, there is exactly one edge going into and out from u . $\sum_{v \in \text{In}(u)} f_p(v, u) = f_p(w, u) = c_f(p)$ and $\sum_{v \in \text{Out}(u)} f_p(u, v) = f_p(u, v) = c_f(p)$. If (u, v) is not on p , both the flow into u and flow out from u are 0. In both cases, flow conservation is satisfied.

Finally, to show that $|f_p| = c_f(p)$, consider the flow out from s . Since p is a simple path, there is no flow input s , so $|f_p| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) = c_f(p) - 0 = c_f(p)$ ■

Immediately following from Lemma 12.3.2 and 12.3.1, we have the following corollary.

Corollary 12.3.3 Let $G = (V, E)$ be a flow network, let f be a flow in G , and let p be an augmenting path in G_f . Let f_p be a flow on the augmenting path defined previously. Then, $f \uparrow f_p$ is a flow in G with value

$$|f \uparrow f_p| = |f| + |f_p| > |f|$$

Proof. By Lemma 12.3.1, $|f_p| = c_f(p) > 0$. By Lemma 12.3.2, $f \uparrow f_p$ is a flow and has value $|f| + |f_p| > |f|$. ■

As a result of Corollary 12.3.3, we know that augmenting a flow with an augmenting path yields a flow with value strictly greater than the flow prior to augmentation.

12.3.3 Max-Flow Min-Cut Theorem

Our analyses so far shows a promising picture of the correctness of the Ford-Fulkerson method. By iteratively augmenting f with an augmenting path p , we gradually increases the flow in the flow network, and when the procedure halts, we should expect to see that f is equal to the maximum flow. The proof utilizes the max-flow min-cut theorem and shows that if there is no augmenting path, there exists a cut in the flow network that implies that f is the max-flow.

Definition 12.3.7 — Cut and Net Flow Across a Cut. A **cut** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the **net flow** $f(S, T)$ across the cut (S, T) is defined as

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

The **capacity of the cut**, denoted $c(S, T)$ is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

A **minimum cut** is a cut whose capacity is minimum over all cuts of the flow network. The definition of a cut and minimum cut will help us prove the important max-flow min-cut theorem. The proof roughly follows the outline shown in Figure 12.1.

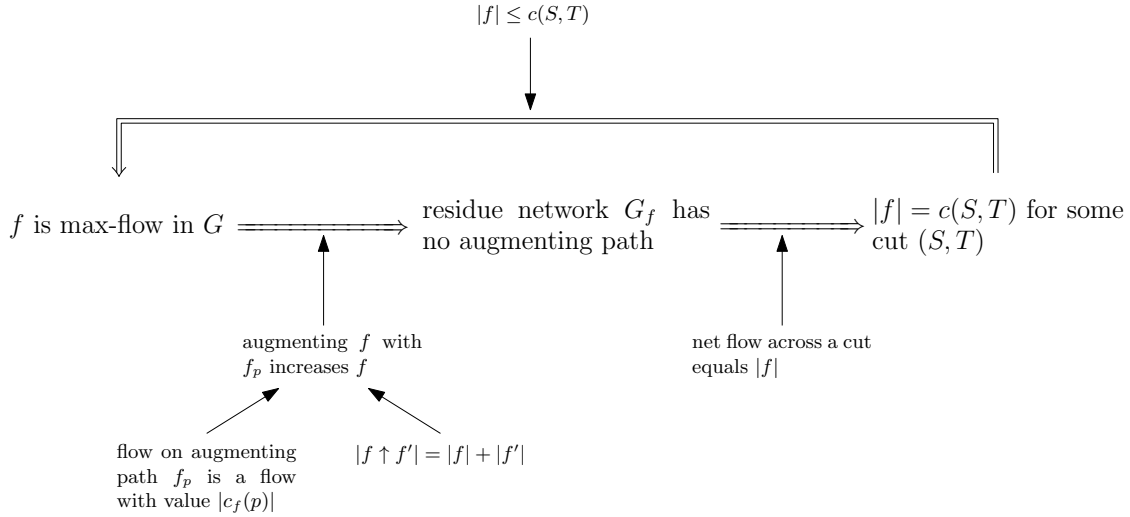


Figure 12.1: Proof outline for the partial correctness of the Ford-Fulkerson method.

Lemma 12.3.4 Let f be a flow in a flow network G with source s and sink t . Let (S, T) be a cut of G . Then, the net flow across the cut (S, T) is $f(S, T) = |f|$.

Proof. Let (S, T) be a cut in G . Consider the following net flows:

- $f(S, T)$: the net flow across the cut (S, T)
- $f(S, V)$: the net flow for edges starting in S
- $f(S, S)$: the net flow within S
- $f(\{s\}, V)$: the net flow from s (i.e. $|f|$)
- $f(S - \{s\}, V)$: the net flow of edges that start from the partition S but not the source s

Fix $u \in V - \{s, t\}$. By flow conservation, we have $\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$. This is true for all $u \in V - \{s, t\}$, so

$$\sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) = 0.$$

This summation can be expanded to

$$\sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) = 0$$

by linearity. Note this expression is equivalent to $f(S - \{s\}, V)$ by definition. So, $f(S - \{s\}, V) = 0$.

Now consider $f(S, S)$. By definition, we can rewrite $f(S, S)$ as

$$\sum_{u \in S} \sum_{v \in S} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(v, u) = 0$$

because for all vertices $x, y \in S$, the term $f(x, y)$ appears exactly once in each summation. Hence, $f(S, S) = 0$.

Since $V = S \cup T$ and $S \cap T = \emptyset$,

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(\{s\}, V) + f(S - \{s\}, V) \\ &= f(\{s\}, V) \\ &= |f| \end{aligned}$$

■

This proof is inspired by the proof presented in the second edition of CLRS. The proof in the third edition employs a more rigorous argument using properties of summation, but I personally find it less intuitive.

A corollary to this lemma shows that cut capacity can be used to bound the value of a flow.

Corollary 12.3.5 The value of any flow f in a flow network G is bounded above by the capacity of any cut of G .

Proof. Let (S, T) be a cut of G and let f be a flow. By Lemma 12.3.4 and capacity constraint,

$$\begin{aligned} |f| = f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \\ &\leq \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u, v) && \text{capacity constraint} \\ &= c(S, T) && \text{definition} \end{aligned}$$

■

Now we finally have everything we need to prove the max-flow min-cut theorem. We will follow the proof outline to prove the equivalences (iff).

Theorem 12.3.6 — Max-Flow Min-Cut Theorem. If f is a flow in a flow network $G = (V, E)$ with source s and sink t , then the following conditions are equivalent

1. f is a maximum flow in G
2. The residue network G_f contains no augmenting paths
3. $|f| = c(S, T)$ for some cut (S, T) of G

Proof.

(1) \Rightarrow (2): Suppose, for contradiction, that f is a maximum flow in G , but G_f has a new augmenting path p . Then, by Corollary 12.3.3, the flow obtained by augmenting f and f_p has a strictly greater value and $f \uparrow f_p$ is a flow in G . This contradicts the assumption that f is a maximum flow.

(2) \Rightarrow (3): Assume (2) holds. Then, there is no path from s to t in the residue network G_f . Let $S = \{v \in V \mid \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$, and let $T = V - S$. Clearly, (S, T) is a cut (this can be trivially proved by showing $s \in S$ and $t \notin S$ because there is no path from s to t). We claim that for each $u \in S$ and $v \in T$, $(u, v) \in E$, $f(u, v) = c(u, v)$. To see why this is true, we use proof by contradiction. Suppose the claim is false. By capacity constraint $f(u, v) \leq c(u, v)$. If $f(u, v) \neq c(u, v)$, $f(u, v) < c(u, v)$ and $(u, v) \in E_f$ because it would have positive residue capacity. But then, this means there would be a path from u to v and $v \in S$, contradicting the assumption that $v \in T$. The other case where $(v, u) \in E$ follows from the same argument with u and v swapped.

(3) \Rightarrow (1): Assume (3) holds. Then, there exists a cut (S, T) such that $|f| = c(S, T)$. By Corollary 12.3.5, $|f| \leq c(S, T)$. This implies that every other flow has value less than f , so f is a maximum flow. ■

This theorem shows that if Ford-Fulkerson terminates, then the resulting flow is the maximum flow. The proof of correctness also suggests the following algorithm that elaborates on the Ford-Fulkerson method outline described earlier.

FORD-FULKERSON($G = (V, E), s, t$)

```

1  for each  $(u, v) \in E$ 
2       $(u, v).f = 0$                                 // initialize flow to 0
3  while there exists a path  $p$  from  $s$  to  $t$  in the residue network  $G_f$ 
4       $c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ is on } p\}$     // residue capacity of path  $p$ 
5      for each  $(u, v)$  on  $p$                             // augment flow
6          if  $(u, v) \in E$ 
7               $(u, v).f = (u, v).f + c_f(p)$ 
8          else  $(v, u).f = (v, u).f - c_f(p)$ 
```

The runtime of Ford-Fulkerson's algorithm/method depends on the way we implement the procedure for finding augmenting path p . In the next chapter, we will look at Edmonds-Karp's algorithm, which implements this operation using BFS.

Chapter 13 Edmonds-Karp Algorithm

Chapter 14 Maximum Bipartite Matching



Linear Programming

- 15 Linear Programming 87**
 - 15.1 Linear Programming Optimization
 - 15.2 Standard and Slack Forms
- 16 Simplex Algorithm for LP 91**
 - 16.1 Geometric Intuition
 - 16.2 Pivoting
 - 16.3 Simplex Algorithm
 - 16.4 Duality and Certificate of Optimality
- 17 Linear Programming Problems 93**
- 18 Integer Programming 95**

Chapter 15 Linear Programming

15.1 Linear Programming Optimization

Linear programming is a commonly used techniques in optimization. In linear programming, we want to optimize (maximize or minimize) a linear function, subject to a set of linear inequalities (called linear constraints).

Definition 15.1.1 — Linear Constraint. Given a set of real numbers a_1, a_2, \dots, a_n and a set of variables x_1, x_2, \dots, x_n , a **linear function** f on those variables is

$$f(x_1, \dots, x_n) = a_1x_1 + a_2x_2 + \dots + a_nx_n = \sum_{j=1}^n a_jx_j$$

If b is a real number and f is a linear function, then $f(x_1, \dots, x_n) = b$ is a linear equality, and $(x_1, \dots, x_n) \leq b$ and $(x_1, \dots, x_n) \geq b$ are linear inequalities. Both linear equalities and linear inequalities are called **linear constraints**.

Geometrically, a linear function forms a line in \mathbb{R}^n . A linear inequality forms a half-space in \mathbb{R}^n . All possible solutions to a linear program lies in a convex region called the **feasible region**. If no such region exists (no solution satisfies all the constraints), we say the optimization problem is **infeasible**. The function we wish to optimize is called the **objective function**, and the value of the objective function evaluated at a particular point is called the **objective value** at that point. The goal of the linear programming problem is to find a feasible solution that maximizes/minimizes the objective value.

Theorem 15.1.1 — Feasible Region for Linear Programming is Convex. Let $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ be two feasible solutions to the linear programming problem, satisfying some particular constraints. Then, for all $\lambda \in [0, 1]$, $\lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2$ is also a solution that satisfies the same constraints.

Proof. Without loss of generality, suppose that the constraints are given as a system of linear equations $\mathbf{Ax} = \mathbf{b}$. Let $\lambda \in [0, 1]$ be arbitrary.

$$\begin{aligned} \mathbf{A}(\lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2) &= \lambda \mathbf{Ax}_1 + \mathbf{Ax}_2 - \lambda \mathbf{Ax}_2 \\ &= \lambda \mathbf{b} + \mathbf{b} - \lambda \mathbf{b} \\ &= \mathbf{b} \end{aligned}$$

Hence, $\lambda \mathbf{x}_1 + (1 - \lambda) \mathbf{x}_2$ is also a solution to the system of linear equation and thus is a feasible solution. ■

Figure 15.1 shows a geometric representation of a linear programming problem with the objective function $x_1 + x_2$.

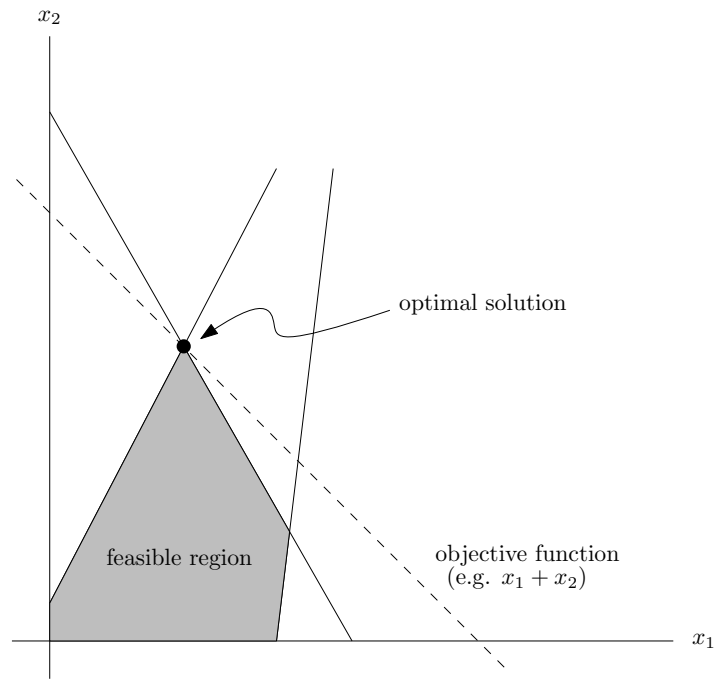


Figure 15.1: A linear programming problem with 2 variables, 4 linear constraints (solid lines), and objective function $x_1 + x_2$ (dashed line).

15.2 Standard and Slack Forms

15.2.1 Standard Form

In standard form, we are given n real numbers c_1, \dots, c_n , m real numbers b_1, \dots, b_m , and mn real numbers a_{ij} for $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$. We wish to

$$\text{Maximize } \sum_{j=1}^n c_j x_j$$

Subject to

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \quad \text{for } i = 1, \dots, m$$

$$x_j \geq 0 \quad \text{for } j = 1, \dots, n$$

or, equivalently, in vectorized form

$$\text{Maximize } \mathbf{c}^\top \mathbf{x}$$

$$\text{Subject to } \mathbf{Ax} \leq \mathbf{b} \text{ and } \mathbf{x} \geq \mathbf{0},$$

where \mathbf{c} is an n -vector, \mathbf{x} is an n -vector, \mathbf{b} is an m -vector, and \mathbf{A} is an $m \times n$ matrix. This can also be written in as a tuple $(\mathbf{A}, \mathbf{b}, \mathbf{c})$ as a shorthand.

This vectorized form is often used in machine learning because vectorized operations can be carried out more quickly using the GPU.

15.2.2 Conversion From Non-standard Form to Standard Form

A linear programming problem might not be in standard form, but it is easy to convert from non-standard form to standard form.

If

- the objective is minimization rather than maximization \Rightarrow flip the sign of coefficients;
- there are variables without nonnegativity constraints \Rightarrow say a variable x_j does not have nonnegativity constraints, replace x_j with $x'_j - x''_j$ and add constraints $x'_j \geq 0$ and $x''_j \geq 0$;
- the constraints are equality constraints rather than $\leq \Rightarrow$ replace the constraint with two non-strict inequality constraints (\leq and \geq);
- the constraints are in the opposite direction (\geq instead of \leq) \Rightarrow multiply both sides by -1

15.2.3 Slack Form

$$\text{Maximize } z = v + \sum_{j=1}^n c_j x_j$$

Subject to

$$s_i = b_i - \sum_{j=1}^n a_{ij} x_j \quad \text{for } i = 1, \dots, m$$

$$x_j, s_i \geq 0 \quad \text{for } j = 1, \dots, n \text{ and } i = 1, \dots, m$$

or, equivalently, in vectorized form

$$\text{Maximize } z = v + \mathbf{c}^\top \mathbf{x}$$

Subject to $\mathbf{s} = \mathbf{b} - \mathbf{A}\mathbf{x}$ and $\mathbf{x}, \mathbf{s} \geq \mathbf{0}$.

We call the variables on the left-hand side of the equality constraints the **basic variables**, and the ones on the right-hand side the **non-basic variables**.

Similarly, a slack form can be concisely defined by a tuple $(N, B, \mathbf{A}, \mathbf{b}, \mathbf{c}, v)$, where N is the set of indices of the non-basic variables, B is the set of indices of the basic variables such that $|N| = n$, $|B| = m$ and $N \cup B = \{1, 2, \dots, n + m\}$.

It is called the slack form because the variable s_i (or in vectorized form \mathbf{s}) measures remaining “slack” or difference between two sides of the original inequality constraints. This is related to the simplex

algorithm where we increase the non-basic variables as much as possible until we hit a bottleneck – having no more slack for that non-basic variable.

Chapter 16 Simplex Algorithm for LP

16.1 Geometric Intuition

16.2 Pivoting

16.3 Simplex Algorithm

16.4 Duality and Certificate of Optimality

Chapter 17 Linear Programming Problems

Chapter 18 Integer Programming

VII Approximation Algorithms



Randomized Algorithms

Commonly Used Axioms & Theorems

Rules of Inference

Axiom 1 — Modus Ponens. $(P \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } Q$

Axiom 2 — Modus Tollens. $(\neg Q \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } \neg P$

Axiom 3 — Hypothetical Syllogism (transitivity).

$((P \text{ IMPLIES } Q) \wedge (Q \text{ IMPLIES } R)) \text{ IMPLIES } (P \text{ IMPLIES } R)$

Axiom 4 — Disjunctive Syllogism. $((P \vee Q) \wedge \neg P) \text{ IMPLIES } Q$

Axiom 5 — Addition. $P \text{ IMPLIES } (P \vee Q)$

Axiom 6 — Simplification. $(P \wedge Q) \text{ IMPLIES } P$

Axiom 7 — Conjunction. $((P) \wedge (Q)) \text{ IMPLIES } (P \wedge Q)$

Axiom 8 — Resolution. $((P \vee Q) \wedge (\neg P \vee R)) \text{ IMPLIES } (Q \vee R)$

Laws of Logic

Axiom 9 — Implication Law. $(P \text{ IMPLIES } Q) \equiv (\neg P \vee Q)$

Axiom 10 — Distributive Law.

$$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$$

$$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$$

$$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \vee (P \wedge Q)) \equiv P$$

$$(P \wedge (P \vee Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \wedge B \equiv B \wedge A$

Axiom 14 — Associativity of AND. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$

Axiom 15 — Identity of AND. $\mathbf{T} \wedge A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \wedge A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \wedge A \equiv A$

Axiom 18 — Contradiction for AND. $A \wedge \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \vee \neg A \equiv \mathbf{T}$

Induction

Axiom 21 — Well Ordering Principle. Every nonempty set of nonnegative integers has a smallest element. i.e., For any $A \subset \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that $\forall a' \in A. a \leq a'$.

Recurrences

Theorem 22 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

Linear Recurrences:

A linear recurrence is an equation

$$f(n) = \underbrace{a_1 f(n-1) + a_2 f(n-2) + \cdots + a_d f(n-d)}_{\text{homogeneous part}} + \underbrace{g(n)}_{\text{inhomogeneous part}}$$

along with some boundary conditions.

The procedure for solving linear recurrences are as follows:

1. Find the roots of the characteristic equation. Linear recurrences usually have exponential solutions (such as x^n). Such solution is called the **homogeneous solution**.

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{k-1} x + a_k$$

2. Write down the homogeneous solution. Each root generates one term and the homogeneous solution is their sum. A non-repeated root r generates the term cr^n , where c is a constant to be determined later. A root with r with multiplicity k generates the terms

$$d_1 r^n \quad d_2 n r^n \quad d_3 n^2 r^n \quad \cdots \quad d_k n^{k-1} r^n$$

where d_1, \dots, d_k are constants to be determined later.

3. Find a **particular solution** for the full recurrence including the inhomogeneous part, but without considering the boundary conditions.
If $g(n)$ is a constant or a polynomial, try a polynomial of the same degree, then of one higher degree, then two higher. If $g(n)$ is exponential in the form $g(n) = k^n$, then try $f(n) = ck^n$, then $f(n) = (bn + c)k^n$, then $f(n) = (an^2 + bn + c)k^n$, etc.
4. Write the **general solution**, which is the sum of homogeneous solution and particular solution.
5. Substitute the boundary condition into the general solution. Each boundary condition gives a linear equation. Solve such system of linear equations for the values of the constants to make the solution consistent with the boundary conditions.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \dots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \dots\}$: the negative integers
- \mathbb{Q} the rational numbers, \mathbb{Q}^+ the positive rationals, and \mathbb{Q}^- the negative rationals.
- \mathbb{R} the real numbers, \mathbb{R}^+ the positive reals, and \mathbb{R}^- the negative reals.

Notation

For any sets A and B , we will use the following standard notation.

- $x \in A$: “ x is an element of A ” or “ A contains x ”
- $A \subseteq B$: “ A is a subset of B ” or “ A is included in B ”
- $A = B$: “ A equals B ” (Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.)
- $A \subsetneq B$: “ A is a proper subset of B ”
(Note that $A \subsetneq B$ if and only if $A \subseteq B$ and $A \neq B$.)
- $A \cup B$: “ A union B ”
- $A \cap B$: “ A intersection B ”
- $A - B$: “ A minus B ” (*set* difference)
- $|A|$: “cardinality of A ” (the number of elements of A)
- \emptyset or $\{\}$: “the empty set”
- $\mathcal{P}(A)$ or 2^A : “powerset of A ” (the set of all subsets of A)
If $A = \{a, 34, \triangle\}$, then $\mathcal{P}(A) = \{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\}$.
 $S \in \mathcal{P}(A)$ means the same as $S \subseteq A$.
- $\{x \in A \mid P(x)\}$: “the set of elements x in A for which $P(x)$ is true”
For example, $\{x \in \mathbb{Z} \mid \cos(\pi x) > 0\}$ represents the set of integers x for which $\cos(\pi x)$ is greater than zero, *i.e.*, it is equal to $\{\dots, -4, -2, 0, 2, 4, \dots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: “the cross product or Cartesian product of A and B ”
 $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
 If $A = \{1, 2, 3\}$ and $B = \{5, 6\}$, then $A \times B = \{(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6)\}$.
- A^n : “the cross product of n copies of A ”
 This is set of all sequences of $n \geq 1$ elements, each of which is in A .
- B^A or $A \rightarrow B$: “the set of all functions from A to B .”
- $f : A \rightarrow B$ or $f \in B^A$: “ f is a function from A to B ”
 f associates one element $f(x) \in B$ to every element $x \in A$.

NUMBER THEORY

For any two natural numbers a and b , we say that a *divides* b if there exists a natural number c such that $b = ac$. In such a case, we say that a is a *divisor* of b (*e.g.*, 3 is a divisor of 12 but 3 is not a divisor of 16). Note that any natural number is a divisor of 0 and 1 is a divisor of any natural number. A number a is *even* if 2 divides a and is *odd* if 2 does not divide a .

A natural number p is *prime* if it has exactly two positive divisors (*e.g.*, 2 is prime since its positive divisors are 1 and 2 but 1 is **not** prime since it only has one positive divisor: 1). There are an infinite number of prime numbers and any integer greater than one can be expressed in a unique way as a finite product of prime numbers (*e.g.*, $8 = 2^3$, $77 = 7 \times 11$, $3 = 3$).

Inequalities

For any integers m and n , $m < n$ if and only if $m + 1 \leq n$ and $m > n$ if and only if $m \geq n + 1$. For any real numbers w , x , y , and z , the following properties always hold (they also hold when $<$ and \leq are exchanged throughout with $>$ and \geq , respectively).

- if $x < y$ and $w \leq z$, then $x + w < y + z$
- if $x < y$, then
$$\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$$
- if $x \leq y$ and $y < z$ (or if $x < y$ and $y \leq z$), then $x < z$

Functions

Here are some common number-theoretic functions together with their definitions and properties of them. (Unless noted otherwise, in this section, x and y represent arbitrary real numbers and k , m , and n represent arbitrary positive integers.)

- $\min\{x, y\}$: “minimum of x and y ” (the smallest of x or y)
 Properties: $\min\{x, y\} \leq x$
 $\min\{x, y\} \leq y$

- $\max\{x, y\}$: “maximum of x and y ” (the largest of x or y)
 Properties: $x \leq \max\{x, y\}$
 $y \leq \max\{x, y\}$
- $\lfloor x \rfloor$: “floor of x ” (the greatest integer less than or equal to x , *e.g.*, $\lfloor 5.67 \rfloor = 5$, $\lfloor -2.01 \rfloor = -3$)
 Properties: $x - 1 < \lfloor x \rfloor \leq x$
 $\lfloor -x \rfloor = -\lceil x \rceil$
 $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
 $\lfloor \lfloor k/m \rfloor / n \rfloor = \lfloor k/mn \rfloor$
 $(k - m + 1)/m \leq \lfloor k/m \rfloor$
- $\lceil x \rceil$: “ceiling of x ” (the least integer greater than or equal to x , *e.g.*, $\lceil 5.67 \rceil = 6$, $\lceil -2.01 \rceil = -2$)
 Properties: $x \leq \lceil x \rceil < x + 1$
 $\lceil -x \rceil = -\lfloor x \rfloor$
 $\lceil x + k \rceil = \lceil x \rceil + k$
 $\lceil \lceil k/m \rceil / n \rceil = \lceil k/mn \rceil$
 $\lceil k/m \rceil \leq (k + m - 1)/m$
 Additional property of $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.
- $|x|$: “absolute value of x ” ($|x| = x$ if $x \geq 0$; $-x$ if $x < 0$, *e.g.*, $|5.67| = 5.67$, $|-2.01| = 2.01$)
 BEWARE! The same notation is used to represent the cardinality $|A|$ of a set A and the absolute value $|x|$ of a number x so be sure you are aware of the context in which it is used.
- $m \operatorname{div} n$: “the quotient of m divided by n ” (integer division of m by n , *e.g.*, $5 \operatorname{div} 6 = 0$, $27 \operatorname{div} 4 = 6$, $-27 \operatorname{div} 4 = -6$)
 Properties: If $m, n > 0$, then $m \operatorname{div} n = \lfloor m/n \rfloor$
 $(-m) \operatorname{div} n = -(m \operatorname{div} n) = m \operatorname{div} (-n)$
- $m \operatorname{rem} n$: “the remainder of m divided by n ” (*e.g.*, $5 \operatorname{rem} 6 = 5$, $27 \operatorname{rem} 4 = 3$, $-27 \operatorname{rem} 4 = -3$)
 Properties: $m = (m \operatorname{div} n) \cdot n + m \operatorname{rem} n$
 $(-m) \operatorname{rem} n = -(m \operatorname{rem} n) = m \operatorname{rem} (-n)$
- $m \bmod n$: “ m modulo n ” (*e.g.*, $5 \bmod 6 = 5$, $27 \bmod 4 = 3$, $-27 \bmod 4 = 1$)
 Properties: $0 \leq m \bmod n < n$
 n divides $m - (m \bmod n)$.
- $\gcd(m, n)$: “greatest common divisor of m and n ” (the largest positive integer that divides both m and n)
 For example, $\gcd(3, 4) = 1$, $\gcd(12, 20) = 4$, $\gcd(3, 6) = 3$
- $\operatorname{lcm}(m, n)$: “least common multiple of m and n ” (the smallest positive integer that m and n both divide)
 For example, $\operatorname{lcm}(3, 4) = 12$, $\operatorname{lcm}(12, 20) = 60$, $\operatorname{lcm}(3, 6) = 6$
 Properties: $\gcd(m, n) \cdot \operatorname{lcm}(m, n) = m \cdot n$.

CALCULUS

Limits and Sums

An infinite sequence of real numbers $\{a_n\} = a_1, a_2, \dots, a_n, \dots$ *converges* to a limit $L \in \mathbb{R}$ if, for every $\varepsilon > 0$, there exists $n_0 \geq 0$ such that $|a_n - L| < \varepsilon$ for every $n \geq n_0$. In this case, we write $\lim_{n \rightarrow \infty} a_n = L$. Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim_{n \rightarrow \infty} a_n = L_1$ and $\lim_{n \rightarrow \infty} b_n = L_2$, then

$$\lim_{n \rightarrow \infty} (a_n + b_n) = L_1 + L_2 \quad \text{and} \quad \lim_{n \rightarrow \infty} (a_n \cdot b_n) = L_1 \cdot L_2.$$

In particular, if c is any real number, then

$$\lim_{n \rightarrow \infty} (c \cdot a_n) = c \cdot L_1.$$

The sum $a_1 + a_2 + \dots + a_n$ and product $a_1 \cdot a_2 \cdot \dots \cdot a_n$ of the finite sequence a_1, a_2, \dots, a_n are denoted by

$$\sum_{i=1}^n a_i \quad \text{and} \quad \prod_{i=1}^n a_i.$$

If the elements of the sequence are all different and $S = \{a_1, a_2, \dots, a_n\}$ is the set of elements in the sequence, these can also be denoted by

$$\sum_{a \in S} a \quad \text{and} \quad \prod_{a \in S} a.$$

Examples:

- For any $a \in \mathbb{R}$ such that $-1 < a < 1$, $\lim_{n \rightarrow \infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim_{n \rightarrow \infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim_{n \rightarrow \infty} (1/n)^a = 0$.
- $\lim_{n \rightarrow \infty} (1 + 1/n)^n = e = 2.71828182845904523536 \dots$

- For any $a, b \in \mathbb{R}$, the *arithmetic* sum is given by:

$$\sum_{i=0}^n (a + ib) = (a) + (a + b) + (a + 2b) + \dots + (a + nb) = \frac{1}{2}(n+1)(2a + nb).$$

- For any $a, b \in \mathbb{R}^+$, the *geometric* sum is given by:

$$\sum_{i=0}^n (ab^i) = a + ab + ab^2 + \dots + ab^n = \frac{a(1 - b^{n+1})}{1 - b}.$$

EXPONENTS AND LOGARITHMS

Definition: For any $a, b, c \in \mathbb{R}^+$, $a = \log_b c$ if and only if $b^a = c$.

Notation: For any $x \in \mathbb{R}^+$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

For any $a, b, c \in \mathbb{R}^+$ and any $n \in \mathbb{Z}^+$, the following properties always hold.

- $\sqrt[n]{b} = b^{1/n}$
- $b^a b^c = b^{a+c}$
- $(b^a)^c = b^{ac}$
- $b^a / b^c = b^{a-c}$
- $b^0 = 1$
- $a^b c^b = (ac)^b$
- $b^{\log_b a} = a = \log_b b^a$
- $a^{\log_b c} = c^{\log_b a}$
- $\log_b(ac) = \log_b a + \log_b c$
- $\log_b(a^c) = c \cdot \log_b a$
- $\log_b(a/c) = \log_b a - \log_b c$
- $\log_b 1 = 0$
- $\log_b a = \log_c a / \log_c b$

BINARY NOTATION

A *binary number* is a sequence of bits $a_k \cdots a_1 a_0$ where each bit a_i is equal to 0 or 1. Every binary number represents a natural number in the following way:

$$(a_k \cdots a_1 a_0)_2 = \sum_{i=0}^k a_i 2^i = a_k 2^k + \cdots + a_1 2 + a_0.$$

For example, $(1001)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 1 = 9$, $(01110)_2 = 8 + 4 + 2 = 14$.

Properties:

- If $a = (a_k \cdots a_1 a_0)_2$, then $2a = (a_k \cdots a_1 a_0 0)_2$, *e.g.*, $9 = (1001)_2$ so $18 = (10010)_2$.
- If $a = (a_k \cdots a_1 a_0)_2$, then $\lfloor a/2 \rfloor = (a_k \cdots a_1)_2$, *e.g.*, $9 = (1001)_2$ so $4 = (100)_2$.
- The smallest number of bits required to represent the positive integer n in binary is called the *length* of n and is equal to $\lceil \log_2(n+1) \rceil$.

Make sure you know how to add and multiply two binary numbers. For example, $(1111)_2 + (101)_2 = (10100)_2$ and $(1111)_2 \times (101)_2 = (1001011)_2$.

Proof Templates

Proof Outlines

LINE NUMBERS: Only lines that are referred to have labels (for example, L1) in this document. For a formal proof, all lines are numbered. Line numbers appear at the beginning of a line. You can indent line numbers together with the lines they are numbering or all line numbers can be unindented, provided you are consistent.

INDENTATION: Indent when you make an assumption or define a variable. Unindent when this assumption or variable is no longer being used.

1. **Implication:** Direct proof of $A \text{ IMPLIES } B$.

L1. Assume A .
:
:
L2. B
 $A \text{ IMPLIES } B$; direct proof: L1, L2

2. **Implication:** Indirect proof of $A \text{ IMPLIES } B$.

L1. Assume $\text{NOT}(B)$.
:
:
L2. $\text{NOT}(A)$
 $A \text{ IMPLIES } B$; indirect proof: L1, L2

3. **Equivalence:** Proof of $A \text{ IFF } B$.

L1. Assume A .
:
:
L2. B
L3. $A \text{ IMPLIES } B$; direct proof: L1, L2
L4. Assume B .
:
:
L5. A
L6. $B \text{ IMPLIES } A$; direct proof: L4, L5
 $A \text{ IFF } B$; equivalence: L3, L6

4. **Proof by contradiction** of A .

L1. To obtain a contradiction, assume $\text{NOT}(A)$.
:
:
L2. B
:
:
L3. $\text{NOT}(B)$
L4. This is a contradiction: L2, L3
Therefore A ; proof by contradiction: L1, L4

5. **Modus Ponens.**

⋮
L1. A
⋮
L2. $A \text{ IMPLIES } B$
 B ; modus ponens: L1, L2

6. **Conjunction:** Proof of $A \text{ AND } B$:

⋮
L1. A
⋮
L2. B
 $A \text{ AND } B$; proof of conjunction; L1, 2

7. **Use of Conjunction:**

⋮
L1. $A \text{ AND } B$
 A ; use of conjunction: L1
 B ; use of conjunction: L1

8. **Implication with Conjunction:** Proof of $(A_1 \text{ AND } A_2) \text{ IMPLIES } B$.

L1. Assume $A_1 \text{ AND } A_2$.
 A_1 ; use of conjunction, L1
 A_2 ; use of conjunction, L1
⋮
L2. B
 $(A_1 \text{ AND } A_2) \text{ IMPLIES } B$; direct proof, L1, L2

9. **Implication with Conjunction:** Proof of $A \text{ IMPLIES } (B_1 \text{ AND } B_2)$.

L1. Assume A .
⋮
L2. B_1
⋮
L3. B_2
L4. $B_1 \text{ AND } B_2$; proof of conjunction: L2, L3
 $A \text{ IMPLIES } (B_1 \text{ AND } B_2)$; direct proof: L1, L4

10. **Disjunction:** Proof of $A \text{ OR } B$ and $B \text{ OR } A$.

⋮
L1. A
 $A \text{ OR } B$; proof of disjunction: L1
 $B \text{ OR } A$; proof of disjunction: L1

11. **Proof by cases.**

L1. C OR $NOT(C)$ tautology
L2. Case 1: Assume C .
 \vdots
 L3. A
L4. C IMPLIES A ; direct proof: L2, L3
L5. Case 2: Assume $NOT(C)$.
 \vdots
 L6. A
L7. $NOT(C)$ IMPLIES A ; direct proof: L5, L6
 A proof by cases: L1, L4, L7

12. **Proof by cases** of A OR B .

L1. C OR $NOT(C)$ tautology
L2. Case 1: Assume C .
 \vdots
 L3. A
 L4. A OR B ; proof of disjunction, L3
L5. C IMPLIES $(A$ OR $B)$; direct proof, L2, L4
L6. Case 2: Assume $NOT(C)$.
 \vdots
 L7. B
 L8. A OR B ; proof of disjunction, L7
L9. $NOT(C)$ IMPLIES $(A$ OR $B)$; direct proof: L6, L8
 A OR B ; proof by cases: L1, L5, L9

13. **Implication with Disjunction:** Proof by cases of $(A_1$ OR $A_2)$ IMPLIES B .

L1. Case 1: Assume A_1 .
 \vdots
 L2. B
L3. A_1 IMPLIES B ; direct proof: L1, L2
L4. Case 2: Assume A_2 .
 \vdots
 L5. B
L6. A_2 IMPLIES B ; direct proof: L4, L5
 $(A_1$ OR $A_2)$ IMPLIES B ; proof by cases: L3, L6

14. **Implication with Disjunction:** Proof by cases of $A \text{ IMPLIES } (B_1 \text{ OR } B_2)$.

- L1. Assume A .
- L2. $C \text{ OR } \text{NOT}(C)$ tautology
- L3. Case 1: Assume C .
- \vdots
- L4. B_1
- L5. $B_1 \text{ OR } B_2$; disjunction: L4
- L6. $C \text{ IMPLIES } (B_1 \text{ OR } B_2)$; direct proof: L3, L5
- L7. Case 2: Assume $\text{NOT}(C)$.
- \vdots
- L8. B_2
- L9. $B_1 \text{ OR } B_2$; disjunction: L8
- L10. $\text{NOT}(C) \text{ IMPLIES } (B_1 \text{ OR } B_2)$; direct proof: L7, L9
- L11. $B_1 \text{ OR } B_2$; proof by cases: L2, L6, L10
- $A \text{ IMPLIES } (B_1 \text{ OR } B_2)$; direct proof. L1, L11

15. **Substitution of a Variable in a Tautology:**

Suppose P is a propositional variable, Q is a formula, and R' is obtained from R by replacing *every* occurrence of P by (Q) .

- L1. R tautology
- R' ; substitution of all P by Q : L1

16. **Substitution of a Formula by a Logically Equivalent Formula:**

Suppose S is a subformula of R and R' is obtained from R by replacing *some* occurrence of S by S' .

- L1. R
- L2. $S \text{ IFF } S'$
- L3. R' ; substitution of an occurrence of S by S' : L1, L2

17. **Specialization:**

- L1. $c \in D$
- L2. $\forall x \in D. P(x)$
- $P(c)$; specialization: L1, L2

18. **Generalization:** Proof of $\forall x \in D. P(x)$.

- L1. Let x be an arbitrary element of D .
- \vdots
- L2. $P(x)$
- Since x is an arbitrary element of D ,
- $\forall x \in D. P(x)$; generalization: L1, L2

19. **Universal Quantification with Implication:** Proof of $\forall x \in D.(P(x) \text{ IMPLIES } Q(x))$.

L1. Let x be an arbitrary element of D .

L2. Assume $P(x)$

\vdots

L3. $Q(x)$

L4. $P(x) \text{ IMPLIES } Q(x)$; direct proof: L2, L3

Since x is an arbitrary element of D ,

$\forall x \in D.(P(x) \text{ IMPLIES } Q(x))$; generalization: L1, L4

20. **Implication with Universal Quantification:** Proof of $(\forall x \in D.P(x)) \text{ IMPLIES } A$.

L1. Assume $\forall x \in D.P(x)$.

\vdots

L2. $a \in D$

$P(a)$; specialization: L1, L2

\vdots

L3. A

Therefore $(\forall x \in D.P(x)) \text{ IMPLIES } A$; direct proof: L1, L3

21. **Implication with Universal Quantification:** Proof of $A \text{ IMPLIES } (\forall x \in D.P(x))$.

L1. Assume A .

L2. Let x be an arbitrary element of D .

\vdots

L3. $P(x)$

Since x is an arbitrary element of D ,

L4. $\forall x \in D.P(x)$; generalization, L2, L3

$A \text{ IMPLIES } (\forall x \in D.P(x))$; direct proof: L1, L4

22. **Instantiation:**

L1. $\exists x \in D.P(x)$

Let $c \in D$ be such that $P(c)$; instantiation: L1

\vdots

23. **Construction:** Proof of $\exists x \in D.P(x)$.

L1. Let $a = \dots$

\vdots

L2. $a \in D$

\vdots

L3. $P(a)$

$\exists x \in D.P(x)$; construction: L1, L2, L3

24. **Existential Quantification with Implication:** Proof of $\exists x \in D.(P(x) \text{ IMPLIES } Q(x))$.

L1. Let $a = \dots$
 \vdots
L2. $a \in D$
 L3. Suppose $P(a)$.
 \vdots
 L4. $Q(a)$
L5. $P(a) \text{ IMPLIES } Q(a)$; direct proof: L3, L4
 $\exists x \in D.(P(x) \text{ IMPLIES } Q(x))$; construction: L1, L2, L5

25. **Implication with Existential Quantification:** Proof of $(\exists x \in D.P(x)) \text{ IMPLIES } A$.

L1. Assume $\exists x \in D.P(x)$.
 Let $a \in D$ be such that $P(a)$; instantiation: L1
 \vdots
 L2. A
 $(\exists x \in D.P(x)) \text{ IMPLIES } A$; direct proof: L1, L2

26. **Implication with Existential Quantification:** Proof of $A \text{ IMPLIES } (\exists x \in D.P(x))$.

L1. Assume A .
 L2. Let $a = \dots$
 \vdots
 L3. $a \in D$
 \vdots
 L4. $P(a)$
L5. $\exists x \in D.P(x)$; construction: L2, L3, L4
 $A \text{ IMPLIES } (\exists x \in D.P(x))$; direct proof: L1, L5

27. **Subset:** Proof of $A \subseteq B$.

L1. Let $x \in A$ be arbitrary.
 \vdots
L2. $x \in B$
 The following line is optional:
L3. $x \in A \text{ IMPLIES } x \in B$; direct proof: L1, L2
 $A \subseteq B$; definition of subset: L3 (or L1, L2, if the optional line is missing)

28. **Weak Induction:** Proof of $\forall n \in N. P(n)$

Base Case:

\vdots

L1. $P(0)$

L2. Let $n \in N$ be arbitrary.

L3. Assume $P(n)$.

\vdots

L4. $P(n+1)$

The following two lines are optional:

L5. $P(n)$ IMPLIES $(P(n+1))$; direct proof of implication: L3, L4

L6. $\forall n \in N. (P(n) \text{ IMPLIES } P(n+1))$; generalization L2, L5

$\forall n \in N. P(n)$ induction; L1, L6 (or L1, L2, L3, L4, if the optional lines are missing)

29. **Strong Induction:** Proof of $\forall n \in N. P(n)$

L1. Let $n \in N$ be arbitrary.

L2. Assume $\forall j \in N. (j < n \text{ IMPLIES } P(j))$

\vdots

L3. $P(n)$

The following two lines are optional:

L4. $\forall j \in N. (j < n \text{ IMPLIES } P(j)) \text{ IMPLIES } P(n)$; direct proof of implication: L2, L3

L5. $\forall n \in N. [\forall j \in N. (j < n \text{ IMPLIES } P(j)) \text{ IMPLIES } P(n)]$; generalization: L1, L4

$\forall n \in N. P(n)$; strong induction: L5 (or L1, L2, L3, if the optional lines are missing)

30. **Structural Induction:** Proof of $\forall e \in S. P(e)$, where S is a recursively defined set

Base case(s):

L1. For each base case e in the definition of S

L2. $P(e)$.

Constructor case(s):

L3. For each constructor case e of the definition of S ,

L4. assume $P(e')$ for all components e' of e .

\vdots

L5. $P(e)$

$\forall e \in S. P(e)$; structural induction: L1, L2, L3, L4, L5

31. **Well Ordering Principle:** Proof of $\forall e \in S.P(e)$, where S is a well ordered set,
i.e. every nonempty subset of S has a smallest element.

L1. To obtain a contradiction, suppose that $\forall e \in S.P(e)$ is false.

L2. Let $C = \{e \in S \mid P(e) \text{ is false}\}$ be the set of counterexamples to P .

L3. $C \neq \emptyset$; definition: L1, L2

L4. Let e be the smallest element of C ; well ordering principle: L2, L3

Let $e' = \dots$

\vdots

L5. $e' \in C$

\vdots

L6. $e' < e$.

L7. This is a contradiction: L4, L5, L6

$\forall e \in S.P(e)$; proof by contradiction: L1, L7

Index

Index

- augmentation (max-flow), 72
- augmenting path, 73
- Bellman equation, 52
- Bellman-Ford algorithm, 59
- charging argument, 36
- complex roots of unity, 26
- convergence property, 61
- cut, 74
- dynamic programming, 51
- fast Fourier transform (FFT), 27
- flow, 69
- flow network, 69
- FPTAS, 57
- greedy property, 39
- inverse fast Fourier transform (IFFT), 28
- Karatsuba's algorithm, 14
- knapsack problem, 55
- linear constraint, 87
- linear equality, 87
- linear function, 87
- linear inequality, 87
- longest simple path, 62
- Master Theorem, 11
- max-flow min-cut theorem, 76
- memoization, 51
- minimum cut, 74
- no-path property, 60
- path relaxation, 61
- path-relaxation property, 61
- pred, 53
- pseudopolynomial, 56
- PTAS, 57
- residue capacity, 72
- residue network, 72
- semantic array, 52
- sink, 69
- source, 69
- Strassen's algorithm, 18
- traveling salesman problem, 62
- triangle inequality (shortest path), 60
- upper-bound property, 60

Bibliography

Courses

- [1] Allan Borodin. *CSC364*. University of Toronto. 2004.
- [2] Allan Borodin. *CSC373S1 Spring 2011*. University of Toronto. 2011.
- [4] Allan Borodin and Sara Rahmati. *CSC373S1 Spring 2020*. University of Toronto. 2020.
- [6] Erik Demaine and Srin Devadas. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2011.
- [7] Erik Demaine, Srin Devadas, and Nancy Lynch. *6.046J Design and Analysis of Algorithms*. Massachusetts Institute of Technology. 2015.
- [8] Faith Ellen. *CSC240S1 Winter 2021*. University of Toronto. 2021.
- [9] Faith Ellen. *CSC265F1 Fall 2021*. University of Toronto. 2021.
- [11] Mauricio Karchmer, Anand Natarajan, and Nir Shavit. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2021.

Books

- [5] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [10] Jeff Erickson. *Algorithms*. 1st. 2019. ISBN: 978-1792644832.
- [12] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 1st. Pearson, 2005. ISBN: 978-0321295354.

Journal Articles

- [3] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. “(Incremental) Priority Algorithms”. In: *Algorithmica* 37.4 (2003), pages 295–326. DOI: 10.1007/s00453-003-1036-3 (cited on page 46).
- [13] Uri Zwick. “The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate”. In: *Theoretical Computer Science* 148.1 (1995), pages 165–170 (cited on page 71).