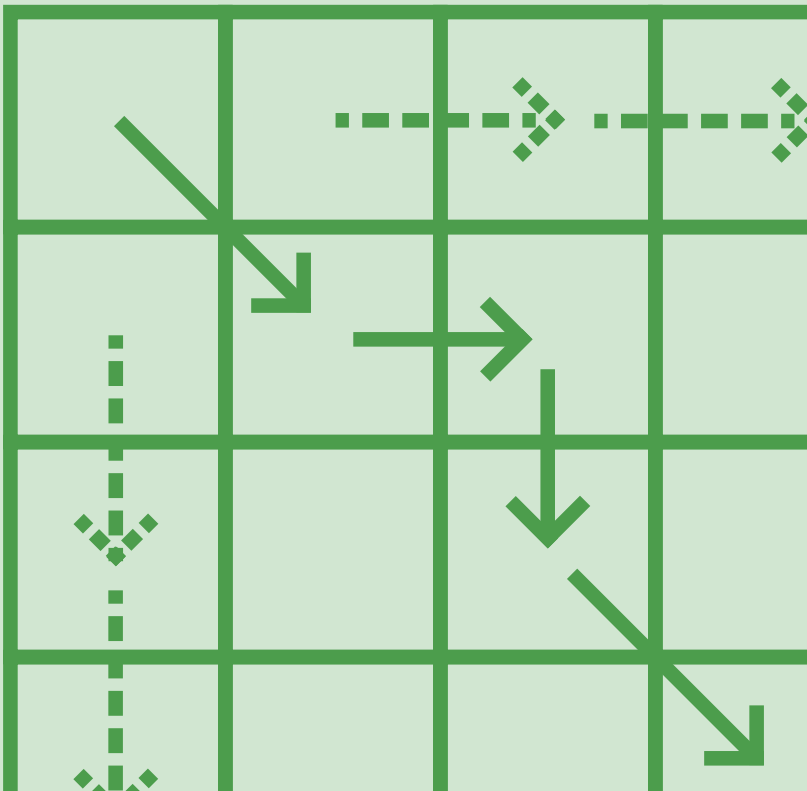
Algorithm Design and Analysis



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.

TERRA-INCOGNITA.DEV

Contents

Appendix

Divide and Conquer

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 (FTT), 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 $f(n) \in O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, 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 $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, 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 = ⌊(1 + L.length)/2⌋
4  (count-a, A) = SORT-AND-COUNT(A[1...mid])
5  (count-b, B) = SORT-AND-COUNT(A[mid + 1...A.length])
6  (count-ab, L') = 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 ≤ A.length and j ≤ B.length
 5       if A[i] ≤ 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 ≤ A.length
14       L[k] = A[i]
15       k = k + 1
16  while j ≤ 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 have 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, ..., 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  δ_1 = CLOSEST-PAIR(P_L)
 3  δ_2 = CLOSEST-PAIR(P_R)
 4  δ = min{δ_1, δ_2}
 5  for p in p_1, p_2, ..., 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, ..., p_n                        // O(n)
10      for i from 1 to 7
11          p_i = ith neighbor of p
12          if DISTANCE(p, p_i) < δ
13              δ = DISTANCE(p, p_i)
14  return δ
```

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 quadradic decision tree model, the closest pair problem requires at least $\Omega(n \log n)$ quadratic tests.

A quadradic decision tree is a comparison tree whose internal nodes are labeled with quadradic 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

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

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 × 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_ij b_ij
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 × n matrix
3  if n == 1
4  else
5      partition A, B, and C each into four sub-matrices
6      C11 = SQUARE-MATRIX-MULTIPLY(A11, B11)+
             SQUARE-MATRIX-MULTIPLY(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY(A11, B12)+
             SQUARE-MATRIX-MULTIPLY(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY(A21, B11)+
             SQUARE-MATRIX-MULTIPLY(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY(A21, B12)+
             SQUARE-MATRIX-MULTIPLY(A22, B22)
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 $S_1, S_2, \ldots, S_{10}$, 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 $P_1, P_2, \ldots, P_7$. 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,

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - B_{21} \\
S_{10} &= B_{11} + B_{12},
\end{aligned}$$

and in Step 3,

$$\begin{aligned}
P_1 = A_{11} \cdot S_1 &= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 = S_2 \cdot B_{22} &= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 = S_3 \cdot B_{11} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 = A_{22} \cdot S_4 &= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 = S_5 \cdot S_6 &= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 = S_7 \cdot S_8 &= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 = S_9 \cdot S_{10} &= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}.
\end{aligned}$$

In Step 4,

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

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

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= \sum_{k=0}^{n-1} a_k x^k \end{aligned}$$

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.

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

3.1.2 Roots

Equivalently,

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

where $r_0, r_1, \ldots, r_{n-1}$ 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):

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

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

By the Foundamental 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:

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

which can be written in pseudocode as follows

EVALUATE$(A = \langle a_0, \ldots a_{n-1}\rangle, x)$
1 $y = 0$
2 **for** $i = n-1$ **to** 0
3 $\quad y = a_j + (x \cdot y)$
4 **return** y

Addition

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

ADD$(A = \langle a_0 \ldots a_{n-1}\rangle, B = \langle b_0 \ldots b_{n-1}\rangle)$
1 **for** $j = 0$ **to** $n-1$
2 $\quad c_j = a_j + b_j$
3 **return** $C = \langle c_0, \ldots, c_{n-1}\rangle$

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 · b_k
6  C = ⟨c_0 ... c_{n+m}⟩
```

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 converts 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 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.

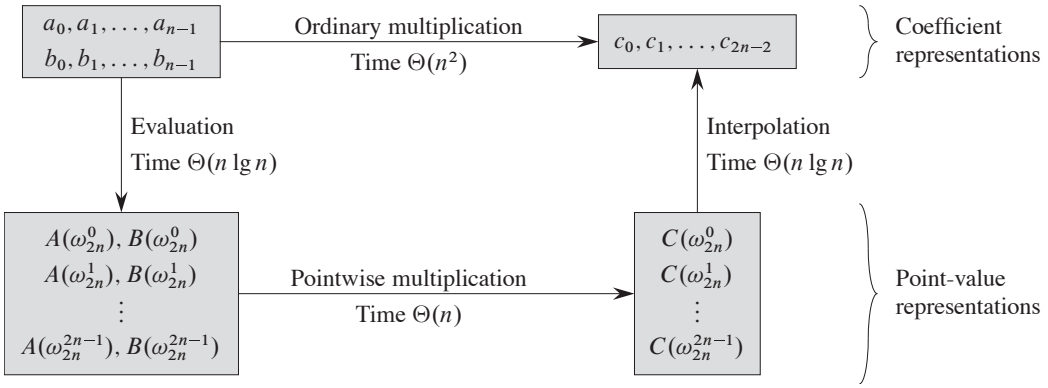


Figure 3.1: Graphical outline of an efficient polynomial multiplication procedure.

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 $x_0, x_1, \ldots, x_{n-1}$ 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 $x_0 \ldots x_{n-1}$" 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)

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

$$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k \quad \leftrightarrow \quad \mathbf{a}_{odd} = \langle a_0, a_2, a_4, \ldots \rangle$$

   Note that the degree of the two resulting polynomials is half of the original polynomial.
2. Conquer: recursive conquer $A_{even}(x)$ and $A_{odd}(x)$ for $x \in X^2$ where $X^2 = \{x^2 \mid x \in X\}$.

3. Combine: combine the terms as follows

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

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

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

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

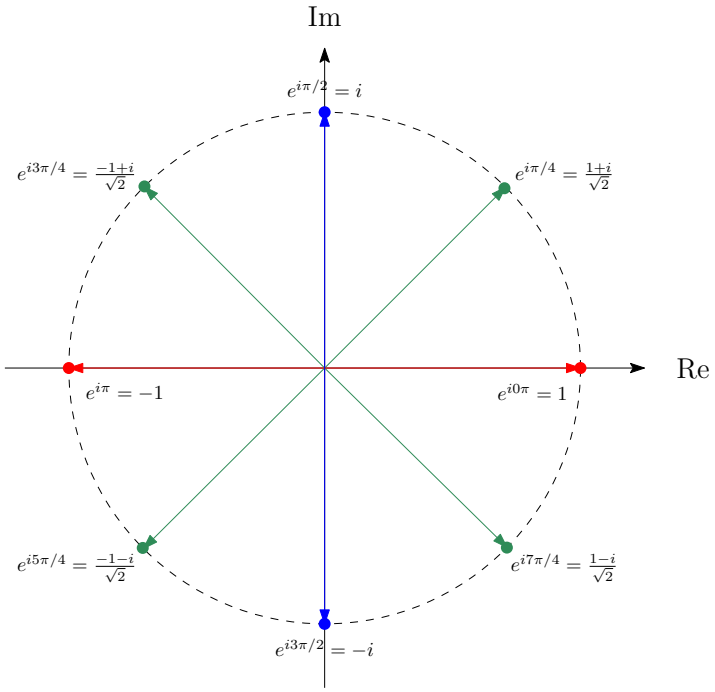
This is no better than the naive approach. The main issue is that we are not halving the size of X. Ideally, we want X to be recursively collapsing.

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 nth root of 1 is called the **complex nth root of unity**. The observation implies that if we start from the nth root of unity and square the elements in the set, the set will collapse every time we square it.

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

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 $\theta = 0, \frac{2}{n}\pi, \frac{4}{n}\pi, \ldots, \frac{2(n-1)}{n}\pi$.



The nth 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 nth 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 nth 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 nth 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 $a_{even} = (a_0, a_2, \ldots, a_{n-2})$
7 $a_{odd} = (a_1, a_3, \ldots, a_{n-1})$
8 $y_{even} =$ FFT-RECURSIVE(a_{even})
9 $y_{odd} =$ 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** $y = (y_0, \ldots, y_{n-1})$

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 fo a polynomial given a set of samples. The transformation is of the form $\mathbf{a}^* \to \mathbf{V}^{-1}\mathbf{a}^* = \mathbf{a}$. 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 nth 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,kth 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 entires 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  ω_n = e^{-2πi/n}
5  ω = 1
6  y_even = (y_0, y_2, ..., y_{n-2})
7  y_odd = (y_1, y_3, ..., y_{n-1})
8  a_even = IFFT-RECURSIVE(y_even)
9  a_odd = IFFT-RECURSIVE(y_odd)
10 for k = 0 to n/2 - 1
11     a_k = (a_even[k] + ω a_odd[k]) / n
12     a_{k+n/2} = (a_even[k] - ω a_odd[k]) / n
13     ω = ω ω_n
14 return a = (a_0, ..., 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 $O(n \log n)$.

II 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 $S = \{a_1, a_2, \ldots, a_n\}$ 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 inverval: 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 ≤ n and S[m].s < S[k].f
3       m = m + 1
4  if m ≤ n
5       return {S[m]} ∪ INTERVAL-SCHEDULING-RECURSIVE(S, m, n)
6  else
7       return ∅
```

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 ith job, and $S[i].f$ denotes the finish time of the ith

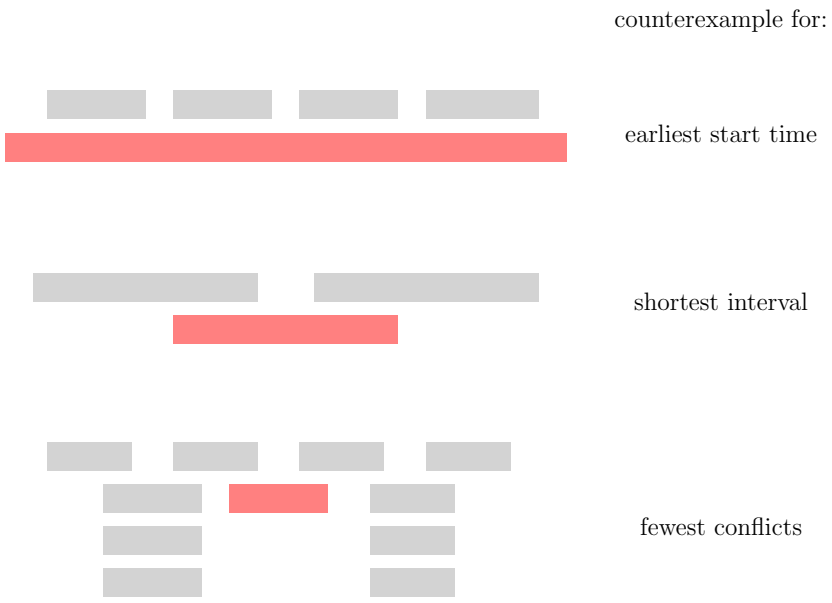


Figure 4.1: Counterexamples for the greedy strategies that do not work for the interval scheduling problem. The interval that satisfies the greedy strategy but leads to incorrect global solution is highlighted in red.

job. k is the job being considered. In each call to the recursive algorithm, we starts 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 INTERVAL-SCHEDULING-RECURSIVE(S, m, n).

The initial call that solves the problem globally is 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.

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 ≥ S[k].f
6          S' = S' ∪ {S[m]}
7          k = m
8  return S'
```

Sorting the array of jobs takes $O(n \log n)$ time, and going over the sorted the list and check 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 $\{a_1, \ldots, a_m\} \subseteq S$

Proof (by contradiction). Suppose, for contradiction, that the greedy algorithm using earlies finish time is not optimal. Let $i_1, i_2, \ldots, i_k$ be the sequence of jobs selected by the algorithm, and let $j_1, j_2, \ldots, j_m$ 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 $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r, i_{r+1}, j_{r+2}, \ldots, j_m$ 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 $T \subseteq \{a_{k+1}, \ldots, a_n\}$ such that $O_k = S_k \cup T$ is optimal.

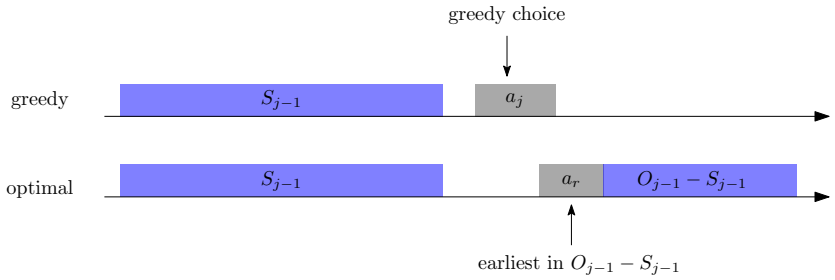
We want to show that for all $k \in \{0, 1, \ldots, n\}$, S_k is promising.

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 $k \in \{0, 1, \ldots, n\}$. ■

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 $S = \{a_1, \ldots, a_n\}$, 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 $h : O \to S'$. 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 exists 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 $h : O \to S'$, 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 know 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 (miminal). 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 that 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 = PRIORITY-QUEUE(key = f)
 3  for i = 1 to S.length
 4      k = EXTRACT-MIN(Q)
 5      if k == NIL
 6          d = d + 1
 7          interval = new INTERVAL(i.f)
 8          INSERT(Q, interval)
 9      if i.s > MIN(Q)
            // if i is compatible with k, allocate i to k
10          k.f = i.f
11          INSERT(Q, k)
12      else
            // otherwise, create new classroom d + 1
13          d = d + 1
14          INSERT(Q, k)
15          interval = new INTERVAL(i.f)
16          INSERT(Q, 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 Algorithms for Minimum Spanning Tree

III

Dynamic Programming

IV

Network Flow

V

Complexity

VI

Linear Programming

VII Approximation Algorithms

VIII Randomized Algorithms

Appendix

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. $((P \vee Q) \wedge (\neg P \vee R))$ IMPLIES $(Q \vee R)$

Laws of Logic

Axiom 9 — Implication Law. $(P$ 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 $d_1, \cdots, d_k$ 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)
- $\varnothing$ 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 $\mathcal{P}(A) = \big\{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\big\}$.
  $S \in \mathcal{P}(A)$ 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 $A \to B$: "the set of all functions from A to B."
- $f : A \to B$ 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 $\lfloor\ \rfloor$ and $\lceil\ \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.

- $|x|$: "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 $\{a_n\} = a_1, a_2, \ldots, a_n, \ldots$ *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 $\lim_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

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

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

In particular, if c is any real number, then

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

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

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

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

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

Examples:

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

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

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

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

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

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 IMPLIES B.

    L1. Assume A.
    ⋮
    L2. B
   A IMPLIES B; direct proof: L1, L2

2. **Implication**: Indirect proof of A IMPLIES B.

    L1. Assume $NOT(B)$.
    ⋮
    L2. $NOT(A)$
   A IMPLIES B; indirect proof: L1, L2

3. **Equivalence**: Proof of A IFF B.

    L1. Assume A.
    ⋮
    L2. B
   L3. A IMPLIES B; direct proof: L1, L2
    L4. Assume B.
    ⋮
    L5. A
   L6. B IMPLIES A; direct proof: L4, L5
   A IFF B; equivalence: L3, L6

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

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

5. **Modus Ponens**.

   ⋮
   L1. A
   ⋮
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   ⋮
   L1. A
   ⋮
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   ⋮
   L1. A AND B
   A; use of conjunction: L1
   B; use of conjunction: L1

8. **Implication with Conjunction**: Proof of (A_1 AND A_2) IMPLIES B.

      L1. Assume A_1 AND A_2.
      A_1; use of conjunction, L1
      A_2; use of conjunction, L1
      ⋮
      L2. B
   (A_1 AND A_2) IMPLIES B; direct proof, L1, L2

9. **Implication with Conjunction**: Proof of A IMPLIES (B_1 AND B_2).

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

10. **Disjunction**: Proof of A OR B and B OR A.

    ⋮
    L1. A
    A OR B; proof of disjunction: L1
    B 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 IMPLIES (B_1 OR B_2).

    L1. Assume A.
    L2. C OR NOT(C) tautology
    L3. Case 1: Assume C.
        ⋮
        L4. B_1
        L5. B_1 OR B_2; disjunction: L4
    L6. C IMPLIES (B_1 OR B_2); direct proof: L3, L5
    L7. Case 2: AssumeNOT(C).
        ⋮
        L8. B_2
        L9. B_1 OR B_2; disjunction: L8
    L10. NOT(C) IMPLIES (B_1 OR B_2); direct proof: L7, L9
    L11. B_1 OR B_2; proof by cases: L2, L6, L10
    A IMPLIES (B_1 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 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.
    ⋮
    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 $\forall x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let x be an arbitrary element of D.
        L2. Assume $P(x)$
        $\vdots$
        L3. $Q(x)$
    L4. $P(x)$ IMPLIES $Q(x)$; direct proof: L2, L3
    Since x is an arbitrary element of D,
    $\forall x \in D.(P(x)$ IMPLIES $Q(x))$; generalization: L1, L4

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

    L1. Assume $\forall x \in D.P(x)$.
    $\vdots$
    L2. $a \in D$
    $P(a)$; specialization: L1, L2
    $\vdots$
    L3. A
    Therefore $(\forall x \in D.P(x))$ IMPLIES A; direct proof: L1, L3

21. **Implication with Universal Quantification**: Proof of A 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 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 $a = \cdots$
    $\vdots$
    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 $\exists x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let $a = \cdots$

    $\vdots$

    L2. $a \in D$

        L3. Suppose $P(a)$.

        $\vdots$

        L4. $Q(a)$

    L5. $P(a)$ IMPLIES $Q(a)$; direct proof: L3, L4

    $\exists x \in D.(P(x)$ IMPLIES $Q(x))$; construction: L1, L2, L5

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

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

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

        $\vdots$

        L2. A

    $(\exists x \in D.P(x))$ IMPLIES A; direct proof: L1, L2

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

    L1. Assume A.

        L2. Let $a = \cdots$

        $\vdots$

        L3. $a \in D$

        $\vdots$

        L4. $P(a)$

    L5. $\exists x \in D.P(x)$; construction: L2, L3, L4

    A 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. $x \in A$ 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. $\forall n \in N.(P(n)$ 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 $\forall j \in N.(j < n$ IMPLIES $P(j))$
            $\vdots$
            L3. $P(n)$
        The following two lines are optional:
        L4. $\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)$; direct proof of implication: L2, L3
    L5. $\forall n \in N.[\forall j \in N.(j < n$ IMPLIES $P(j))$ 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. $C \neq \phi$; definition: L1, L2
        L4. Let e be the smallest element of C; well ordering principle: L2, L3
            Let $e' = \cdots$
            $\vdots$
            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

Bibliography

Courses

[1] Allan Borodin. *CSC364*. University of Toronto. 2004.

[2] Allan Borodin. *CSC373S1 Spring 2011*. University of Toronto. 2011.

[3] Allan Borodin and Sara Rahmati. *CSC373S1 Spring 2020*. University of Toronto. 2020.

[5] Erik Demaine and Srini Devadas. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2011.

[6] Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046J Design and Analysis of Algorithms*. Massachusetts Institute of Technology. 2015.

[7] Faith Ellen. *CSC240S1 Winter 2021*. University of Toronto. 2021.

[8] Faith Ellen. *CSC265F1 Fall 2021*. University of Toronto. 2021.

[10] Mauricio Karchmer, Anand Natarajan, and Nir Shavit. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2021.

Books

[4] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[9] Jeff Erickson. *Algorithms*. 1st. 2019. ISBN: 978-1792644832.

[11] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 1st. Pearson, 2005. ISBN: 978-0321295354.

Journal Articles