CSC373

Algorithm Design, Analysis & Complexity

Sinan Li

2024

CONTENTS

Part I

Notes

CHAPTER

1 INTRODUCTION

1.1 Course Information

- **Instructor**: Nathan Wiebe
  - **Email**: nawibe@cs.toronto.edu
  - **Office**: SF 3318C
- **Text**: [CLRS] *Introduction to Algorithms*: Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford
- **Disclaimer**: Many things are up in the air, so expect a somewhat bumpy ride at the start but hopefully, we will get through together! Use any of the feedback mediums (email, Piazza, ...) to let the instructor know if there are any suggestions for improvement.

1.2 Grading

1.2.1 Assignments

- **4 assignments**, best 3 out of 4
- Group work
  - In groups of *up to three* students
  - Best way to learn is for each member to try each problem
- Questions will be **more difficult**
  - May need to mull them over for several days; do not expect to start and finish the assignment on the same day!
  - May include bonus questions
- Submission on **crowdmark**, more details later. May need to compress the PDF.

1.2.2 Tests

- 2 term tests, one end-of-term test (final exam / assessment)
- Time and Place
  - Fridays during Tutorials
  - In-person

1.2.3 Grading Scheme

Best 3/4 Assignments	×	10%	=	30%
2 Term Tests	×	20%	=	40%
Final Exam	×	30%	=	30%

Note: There is **no** auto-fail policy for the final exam.

1.3 Course Information

1.3.1 What is this course about?

- What if we can't find an efficient algorithm for a problem?
  - Try to prove that the problem is hard
  - Formally establish complexity results
  - NP-completeness, NP-hardness, ...
- We'll often find that one problem may be easy, but its simple variants may suddenly become hard.
  - Minimum spanning tree (MST) vs. bounded degree MST
  - 2-colorability vs 3-colorability

1.3.2 Proofs

In this course you are expected to provide a clear and compelling argument about why you're right about ant claim about an algorithm. We call these argument proofs.
Proof structures used in this course:

- Induction
- Contradiction
- Desperation...

Inductive Proof

Key idea with induction:

- Break the problem into a number of steps, $s(i)$.
- Show that induction hypothesis holds for base case $s(0)$.
- Show that if hypothesis holds for $s(i)$ then it holds for step $s(i+1)$.

Theorem 1.3.1 Principle of Mathematical Induction

Let $P(n)$ be a predicate defined for integers $n \geq 0$. If

1. $P(0)$ is true, and
2. $P(k)$ implies $P(k+1)$ for all integers $k \geq 0$,

then $P(n)$ is true for all integers $n \geq 0$.

Example. Say you want to find the best person to marry in Stardew Valley.

You can apply a single iteration of Bubblesort to find that Sebastian is objectively the best person to marry.

```
1: Function Bubble-Sort(A)
2:     For i = 1 to n − 1 do
3:         For j = 1 to n − i do
4:             If A[j] > A[j + 1] then
5:                 Swap(A[j], A[j + 1])
```

Proof. Proof by induction on the number of iterations of Bubble-Sort.

- **Base case**: $s(1)$

  Then swap doesn't happen and you have a trivially sorted array.

- **Induction Steps**: $s(i) \to s(i+1)$

  Assume that $s(i)$ is sorted by the algorithm for any array s of length i.

  $$s_0, s_1, \ldots, y = s_{i-1}, x = s_i$$

  1. **Case 1**: $x > y$

     Then, comparison between x, y says you should swap them.
  2. **Case 2**: $x \leq y$

     Then, comparison between x, y says you should not swap them. The array is still sorted.

Thus, by induction, the algorithm will sort any array. ■♢

Contradiction

- Assume that the opposite of the hypothesis were true.
- Show that if the opposite were true then the assumptions of the problem would be violated.

This needs more finesse than a proof by induction. There can be a lot more slick when it works.

- Working out small examples of the problem helps.
- Argue about the first/last position where the hypothesis fails to be true.

Example. Assume that BUBBLESORT is does not return the best element (smallest) on right. Let i be first position where in the array s, $s(i+1) > s(i)$.

1. **Case 1**: $s(i) > s(i+1)$

   Then, BUBBLESORT would have swapped them.

2. **Case 2**: $s(i) < s(i+1)$

   Then, BUBBLESORT would have not swapped them.

CHAPTER

2 DIVIDE AND CONQUER

Veni, vidi, vici.

— Gaius Julius Caesar

2.1 Introduction

Divide and Conquer is a general algorithm design paradigm

- **Divide** the problem into smaller subproblems of the same type.
- **Conquer** each subproblems recursively and independently.
- **Combine** solutions from subproblems and/or solve remaining part of the original problem.

Example (Merge Sort)**.** MERGE-SORT is a sorting algorithm that uses the divide and conquer paradigm.

- **Divide**: Split the array into two halves
- **Conquer**: Sort the two halves recursively
- **Combine**: Merge the two sorted halves into a sorted array

♢

Remark

When analyzing divide and conquer algorithms, **constants** matter due to the recursive nature of the algorithm.

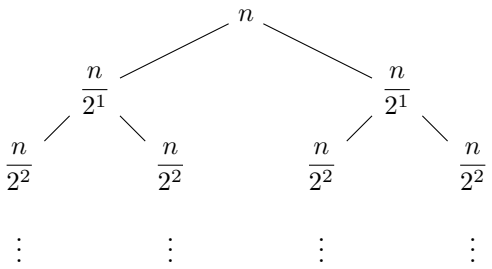
2.1.1 Merge Sort

Merge sort is a sorting algorithm that uses the divide and conquer paradigm. It divides the array into two halves, sorts the two halves recursively, and merges the two sorted halves into a sorted array.

Claim. Two arrays of length m that are sorted can be combined into a sorted string in $\mathcal{O}(m)$ time.

```
1: Function Merge-Sort(A)
2:     If |A| ≤ 2 then
3:         Return BruteForceSort(A)
4:     m ← ⌊|A|/2⌋
5:     L ← Merge-Sort(A[1 ... m])
6:     R ← Merge-Sort(A[m + 1 ... |A|])
7:     Return Merge(L, R)
```

```
1: Function Merge(L, R)
2:     i ← 1
3:     j ← 1
4:     A ← ∅
5:     While i ≤ |L| and j ≤ |R| do
6:         If L[i] ≤ R[j] then
7:             A ← A ∪ {L[i]}
8:             i ← i + 1
9:         Else
10:            A ← A ∪ {R[j]}
11:            j ← j + 1
12:    Return A ∪ L[i ...] ∪ R[j ...]
```



To compute the cost of Merge-Sort, we need to compute the cost of Merge and the cost of the levels.

Claim. The cost of the levels is

$$\left(\frac{n}{2}\right) \cdot \mathcal{O}(2^2) = \mathcal{O}(n)$$

Indeed, in each level j, there are 2^j subproblems of size $\frac{n}{2^j}$, and the cost of each subproblem is $\mathcal{O}(2^j)$. Thus, the cost of each level is

$$\left(\frac{n}{2^j}\right) \cdot \mathcal{O}(2^j) = \mathcal{O}(n).$$

Then, the cost of Merge-Sort is

$$\sum_{j=1}^{\log_2 n - 1} \left(\frac{n}{2^j}\right) \cdot \mathcal{O}(2^j) = \mathcal{O}(n \log n)$$

Claim. Merge-Sort is correct.

Proof. By induction on the number of iterations of Merge-Sort.

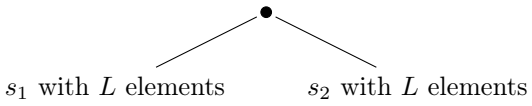
- **Base case**: $s(2)$

  BruteForceSort is correct by construction.

- **Induction Steps**

  Assume Merge-Sort is correct for any array s of length $L \geq 2$.

  Without loss of generality, assume L is a power of 2. For the other cases, some extra work is needed.



  If Merge-Sort is correct, then s_1 and s_2 are sorted.

  For $j = 1$ to L, compare $s_1[j]$ to $s_2[j]$ and insert if $s_1[j] \geq s_2[j]$.

  The algorithm guarantees that inserted $s_1[j] \geq s_2[j]$. Thus, insertion is correct.

  This implies that, in cases of mistake, then the order must have been wrong to start with.

  This is a contradiction, as we assumed that Merge-Sort is correct for L elements.

Thus, Merge-Sort is correct. ■

2.1.2 Counting Inversions

- **Problem**

  Given an array a of length n, count the number of pairs (i, j) such that $i < j$ but $a[i] > a[j]$.

- **Applications**

  - Voting theory
  - Collaborative filtering
  - Measuring the "sortedness" of an array
  - Seneitivity analysis of Google's ranking function
  - ...

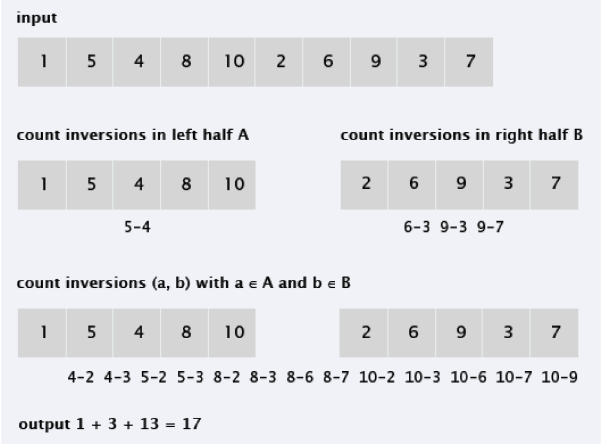
Definition 2.1.1 Inversion

An **inversion** is a pair (i, j) such that $i < j$ but $a[i] > a[j]$.

The brute force algorithm is to check all pairs (i, j) and count the number of inversions. This is $\mathcal{O}(n^2)$. We can do better by using the divide and conquer paradigm.

- **Divide**: Split the array into two equal halves x and y
- **Conquer**: Count the number of inversions in the two halves recursively
- **Combine**:
  - *Solve*: Count the number of inversions where $i \in x$ and $j \in y$
  - *Merge*: Add the three counts together

```
1:  Function Sort-And-Count(L)
2:      If |A| ≤ 1 then
3:          Return (0, L)
4:
5:      Divide the list into two halves A and B
6:      (r_A, A) ← Sort-And-Count(A)
7:      (r_B, B) ← Sort-And-Count(B)
8:      (r_AB, L') ← Merge-And-Count(A, B)
9:
10:     Return (r_A + r_B + r_AB, L')
```



Counting inversions $i \in x$ and $j \in y$ is done by merging the two sorted halves.

- Scan x and y in parallel from left to right
- If $x[i] \leq y[j]$, then $x[i]$ is not an inversion If $x[i] > y[j]$, then $x[i]$ is an inversion with all elements in y that have not been scanned yet
- Append the smaller element to the output array

```
1:  Function Merge-And-Count(L, R)
2:      i ← 1
3:      j ← 1
4:      A ← ∅
5:      r_LR ← 0
6:      While i ≤ |L| and j ≤ |R| do
7:          If L[i] ≤ R[j] then
8:              A ← A ∪ {L[i]}
9:              i ← i + 1
10:         Else
11:             A ← A ∪ {R[j]}
12:             j ← j + 1
13:             r_LR ← r_LR + |L| − i + 1
14:     Return (A ∪ L[i ...] ∪ R[j ...], r_LR)
```

To formally prove correctness of SortAndCount, we can induce on the size of the array, n.
To analyze the running time of SortAndCount,

- Suppose $T(n)$ is the worst-case running time for inputs of size n
- Our algorithm satisfies $T(n) \leq 2T(\frac{n}{2}) + \mathcal{O}(n)$
- Master theorem says this is $T(n) = \mathcal{O}(n \log n)$

2.2 Master Theorem

Theorem 2.2.1 Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.
Let $d = \log_b a$. Then, $T(n)$ has the following asymptotic bounds:

1. If $f(n) = \mathcal{O}(n^{d-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \mathcal{O}(n^d)$.

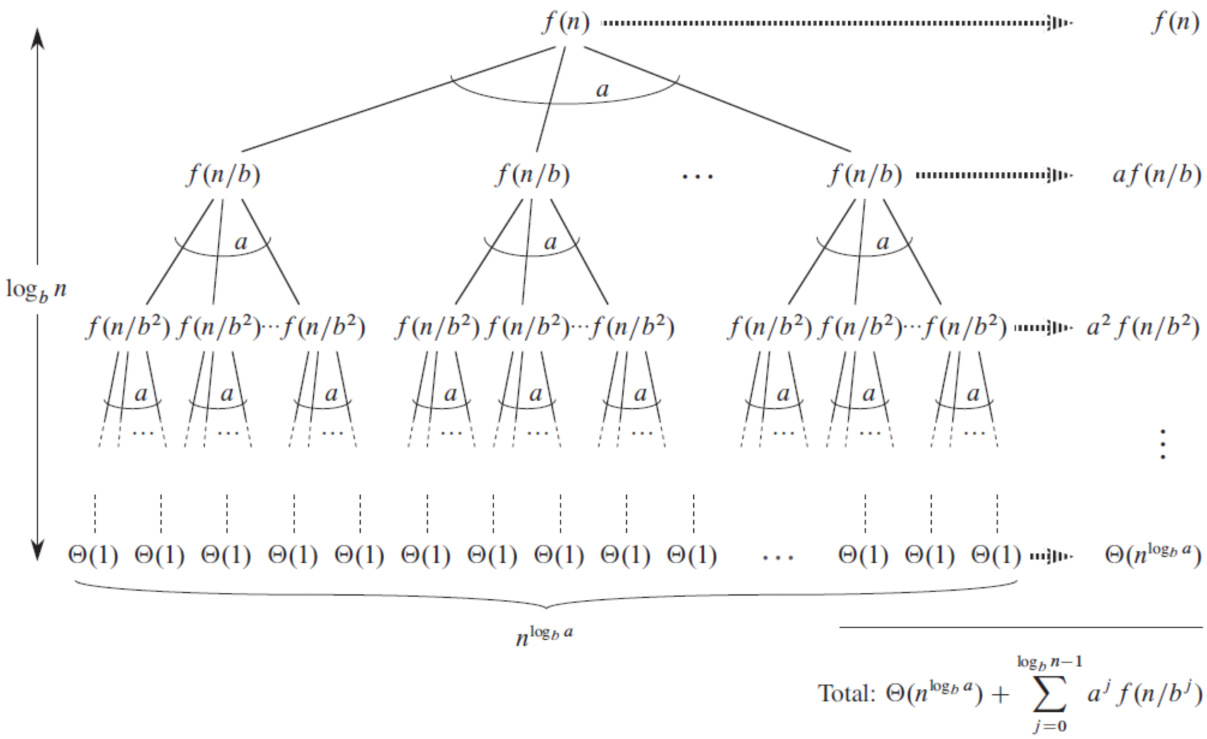
   This is the **merge heavy** case. The cost of merging dominates the cost of recursion.

2. If $f(n) = \mathcal{O}(n^d \log^k n)$, then $T(n) = \mathcal{O}(n^d \log^{k+1} n)$.

   This is the **balanced** case. The cost of merging and recursion are the same.

3. If $f(n) = \mathcal{O}(n^{d+\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \mathcal{O}(f(n))$.

   This is the **leaf (recursion) heavy** case. The cost of recursion dominates the cost of merging.



2.2.1 Closest Pair

- **Problem**

  Given n points of the form (x_i, y_i) in the plane, find the closest pair of points.

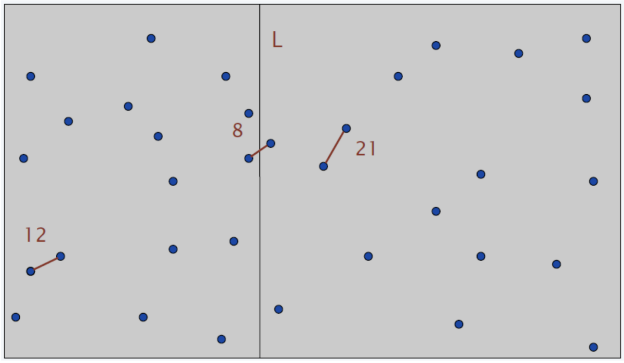
- **Applications**

  - Basic primitive in graphics and computer vision
  - Geographic information systems, molecular modeling, air traffic control
  - Special case of nearest neighbor

- **Brute force** is $\mathcal{O}(n^2)$.

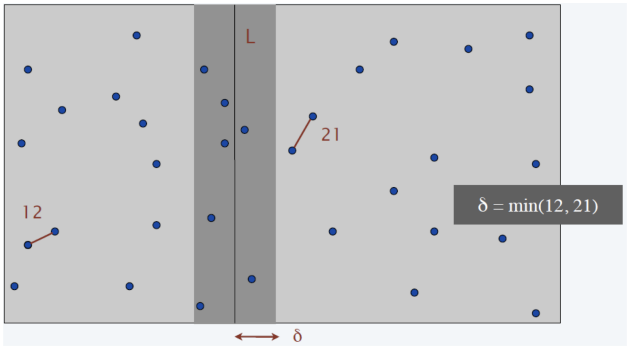
We can use the divide and conquer paradigm to solve this problem.

- **Divide**: Split the points into two equal halves by drawing a vertical line L through the median x-coordinate



- **Conquer**: Find the closest pair of points in each half recursively
- **Combine**: Find the closest pair of points with one point in each half

  We can restrict our attention to points within δ of L on each side, where δ = best of the solutions within the two halves.



  - Only need to look at points within δ of L on each side
  - Sort points on the strip by y coordinate
  - Only need to check each point with next 11 points in sorted list

- Return the best of 3 solutions

Remark

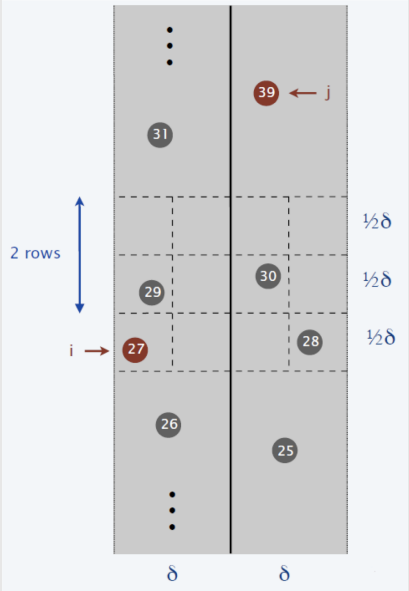
We chose the number 11 on purpose.

Claim. If two points are at least 12 positions apart in the sorted list, their distance is at least δ.

Proof.

- No two points lie in the same $\frac{\delta}{2} \times \delta$ rectangle.
- Two points that are more than two rows apart are at distance δ.

■



Let $T(n)$ be the worst-case running time of the algorithm. To analyze the Running time for the combine operation,

- Finding points on the strip is $\mathcal{O}(n)$
- Sorting points on the strip by their y-coordinate is $\mathcal{O}(n \log n)$
- Testing each point against 11 points is $\mathcal{O}(n)$

Thus, the total running running time is

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \mathcal{O}(n \log n)$$

By the master theorem, this yields $T(n) = \mathcal{O}(n \log^2 n)$.

2.2.2 Multiplication Algorithms

Karatsuba's Algorithm

- **Problem**

  Given two n-bit integers x and y, compute their product xy.

- **Applications**
  - Multiplying large integers
  - Multiplying large polynomials
  - Multiplying large matrices
- **Brute force** is $\mathcal{O}(n^2)$.

Karatsuba's observed that we can divide each integer into two halves,

$$x = x_1 \cdot 10^{\frac{n}{2}} + x_2 \qquad y = y_1 \cdot 10^{\frac{n}{2}} + y_2$$

and then

$$xy = (x_1y_1) \cdot 10^n + (x_1y_2 + x_2y_1) \cdot 10^{\frac{n}{2}} + x_2y_2$$

so four $n/2$-bit integer multiplications can be replaced by three:

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

This would give a running time of

$$T(n) \leq 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.59})$$

Strassen's Algorithm

Strassen's algorithm is a generalization of Karatsuba's algorithm to design a fast algorithm for multiplying two $n \times n$ matrices.

- We call n the "size" of the problem

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

- Natïvely, this requires $2^3 = 8$ matrix multiplications of size $\frac{n}{2}$.
- Strassen's algorithm reduces this to 7 matrix multiplications instead of 8.

$$T(n) \leq 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \implies T(n) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$$

1: Function Strassen(n, A, B)
2: If $n = 1$ **then** Return $A \times B$

3: Partition A and B into 2×2 block matrices
4: $P_1 \leftarrow$ Strassen($\frac{n}{2}, A_{11}, (B_{12} - B_{22})$)
5: $P_2 \leftarrow$ Strassen($\frac{n}{2}, (A_{11} + A_{12}), B_{22}$)
6: $P_3 \leftarrow$ Strassen($\frac{n}{2}, (A_{21} + A_{22}), B_{11}$)
7: $P_4 \leftarrow$ Strassen($\frac{n}{2}, A_{22}, (B_{21} - B_{11})$)
8: $P_5 \leftarrow$ Strassen($\frac{n}{2}, (A_{11} + A_{22}), (B_{11} + B_{22})$)
9: $P_6 \leftarrow$ Strassen($\frac{n}{2}, (A_{12} - A_{22}), (B_{21} + B_{22})$)
10: $P_7 \leftarrow$ Strassen($\frac{n}{2}, (A_{11} - A_{21}), (B_{11} + B_{12})$)

11: $C_{11} \leftarrow P_5 + P_4 - P_2 + P_6$
12: $C_{12} \leftarrow P_1 + P_2$
13: $C_{21} \leftarrow P_3 + P_4$
14: $C_{22} \leftarrow P_1 + P_5 - P_3 - P_7$

15: Return C

2.2.3 Median and Selection

- **Selection**
  - Given an array A of n comparable elements, find the k-th smallest element in A.
  - $k = 1$ is the minimum, $k = n$ is the maximum, and $k = \lfloor \frac{n}{2} \rfloor$ is the median.
  - The running time is $\mathcal{O}(n)$ for minimum and maximum.

- **k-Selection**
  - $\mathcal{O}(nk)$ by modifying bubble sort.
  - $\mathcal{O}(n \log n)$ by sorting.
  - $\mathcal{O}(n + k \log n)$ by using a min-heap.
  - $\mathcal{O}(k + n \log k)$ by using a max-heap.
  - $\mathcal{O}(n)$ by using the divide and conquer paradigm.

QuickSelect

- **Divide**: Pick a pivot p at random from A
- **Conquer**: Partition A into two sub-arrays
  - A_{less} contains all elements less than p
  - A_{more} contains all elements greater than p
- **Combine**:
  - If $|A_{less}| \geq k$, return k-th smallest element in A_{less}
  - Otherwise, return $(k - |A_{less}|)$-th smallest element in A_{more}

However, this algorithm is not guaranteed to be $\mathcal{O}(n)$, as the pivot may be the largest or smallest element in the array. If pivot is close to the min or the max, then we basically get $T(n) \leq T(n-1) + \mathcal{O}(n)$, which is $\mathcal{O}(n^2)$. We want to reduce $n - 1$ to a fraction of n.

CHAPTER

3 GREEDY ALGORITHMS

3.1 Introduction

Greedy algorithms are a class of algorithms that make locally optimal choices at each step in order to find a global optimum. They are often used to solve optimization problems.

- **Goal:** find a solution x maximizing or minimizing some objective function f.
- **Challenge:** space of possible solutions x is too large to search exhaustively.
- **Insight:** x is composed of several parts (e.g., x is a set or a sequence).
- **Approach:** instead of computing x directly,
  - Compute x one part at a time.
  - Select the next part "greedily" to get the most immediate "benefit", which needs to be defined carefully for each problem.
  - Polynomial running time is typically guaranteed.
  - Need to prove that this will always return an optimal solution despite having no global view of the problem.

3.2 Interval Scheduling

3.2.1 Problem Definition

- Job j starts at time s_j and finishes at time f_j.
- Two jobs i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- **Goal:** find a maximum-size subset of mutually compatible jobs.

3.2.2 Greedy Algorithm

The greedy algorithm for interval scheduling follows the template

- Consider jobs in some "natural" order.
- Take a job if it is compatible with the ones already taken.

But, what is the "natural" order?

- **Earliest start time:** ascending order of s_j.
- **Earliest finish time:** ascending order of f_j.
- **Shortest interval:** ascending order of $f_j - s_j$.
- **Fewest conflicts:** ascending order of c_j, where c_j is the number of remaining jobs that conflicts with job j.

However, not all of these orders will yield the optimal solution. Below are some counterexamples.

Earliest Start Time

Shortest Interval

Fewest Conflicts

We can implement the greedy algorithm using **earliest finish time order** (EFT).

- Sort jobs by finish time, say $f_1 \leq f_2 \leq \cdots \leq f_n$.

$$\mathcal{O}(n \log n)$$

- For each job j, we need to check if it's compatible will *all* previously added jobs.
  - Natively, this is $\mathcal{O}(n^2)$, as we need $\mathcal{O}(n)$ for each job.
  - We only need to check if $s_j \geq f_{i^*}$, where i^* is the *last job added*.
    - For any jobs i added before i^*, we have $f_i \leq f_{i^*}$.
    - By keeping track of f_{i^*}, we can check compatibility of job j in $\mathcal{O}(1)$.
- Thus, the total running time is

$$\mathcal{O}(n \log n).$$

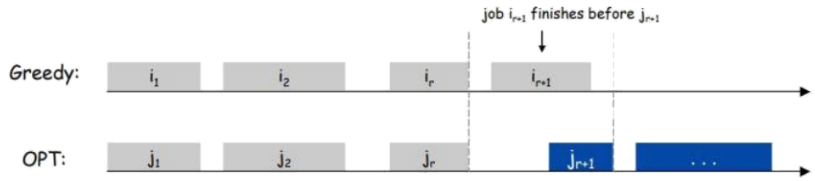
3.2.3 Proof of Optimality

By Contradiction

- Suppose for contradiction that greedy is not optimal
- Say greedy selects jobs $i_1, i_2, \ldots, i_k$ sorted by finish time
- Consider an optimal solution $j_1, j_2, \ldots, j_m$ sorted by finish time which matches greedy for as many indices as possible.

  That is, $j_1 = i_1, j_2 = i_2, \ldots, j_r = i_r$ for the greatest possible r.

- Both i_{r+1} and j_{r+1} must be compatible with the previous selection $i_1, i_2, \ldots, i_r = j_1, j_2, \ldots, j_r$.
- Consider a new solution $i_1, i_2, \ldots, i_r, j_{r+1}, j_{r+2}, \ldots, j_m$.
  - We have replaced j_{r+1} by i_{r+1} in our reference optimal solution
  - This is still **feasible** because $f_{i_{r+1}} \leq f_{j_{r+1}} \leq s_{j_t}$ for $t \geq r+2$.
  - This is still **optimal** because m jobs are sorted.
  - But it matched the greedy solution in $r+1$ indices. This is a contradiction.



By Induction

- Let S_j be the subset of jobs picked by greedy after considering the first j jobs in the increasing order of finish time.

  Define $S_0 = \varnothing$.
- We call this partial solution **promising** if there is a way to extend it to an optimal solution by picking some subset of jobs $j+1, \ldots, n$.

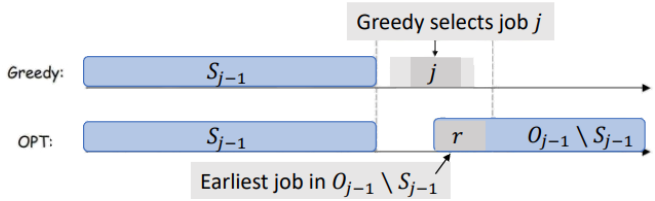
  $\exists T \subseteq \{j+1, \ldots, n\}$ such that $S_j \cup T$ is an optimal solution.
- **Inductive claim:** for all $t \in \{0, 1, \ldots, n\}$, S_t is promising.
  - For $t = n$, if S_n is promising, then it must be an optimal solution.
  - We choose $t = 0$ as our base case since it is trivially promising.
- **Base case:** $t = 0$.

  For $t = 0$, $S_0 = \varnothing$ is promising. Any optimal solution extends it.
- **Inductive step:** $t - 1 \to t$.

  Suppose the claim holds for $t = j - 1$ and optimal solution O_{j-1} extends S_{j-1}.

  At $t = j$, we have two cases:

  1. Greedy did not select job j, so $S_j = S_{j-1}$.
     - Job j must conflict with some job in S_{j-1}.
     - Since $S_{j-1} \subseteq O_{j-1}$, O_{j-1} also include pick job j.
     - $O_j = O_{j-1}$ also extends $S_j = S_{j-1}$.
  2. Greedy selected job j, so $S_j = S_{j-1} \cup \{j\}$.
     - Consider the earliest job r in $O_{j-1} \setminus S_{j-1}$.
     - Consider O_j contained by replacing r with j in O_{j-1}.
     - Prove that O_j is still feasible.
     - O_j extends S_j, as desired.



Contradiction vs Induction

Both methods make the same claim, that

> "The greedy solution after j iterations can be extended to an optimal solution, $\forall j$".

They also use the same key argument.

> "If the greedy solution after j iterations can be extended to an optimal solution, then the greedy solution after $j+1$ iterations can be extended to an optimal solution as well"

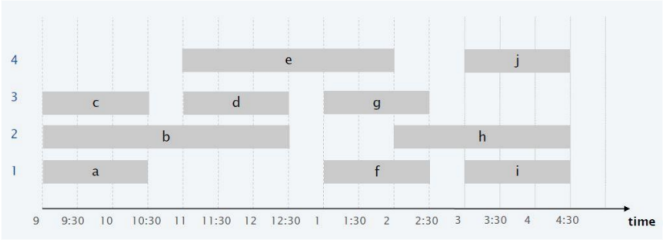
- For proof by induction, this is the key induction step.
- For proof by contradiction, we take the greatest j for which the greedy solution can be extended to an optimal solution, and derive a contradiction by extending the greedy solution after $j+1$ iterations.

3.3 Interval Partitioning

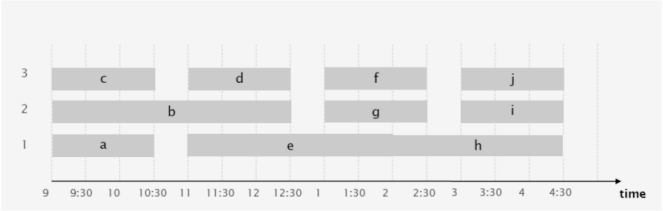
3.3.1 Problem Definition

- Job j starts at time s_j and finishes at time f_j.
- Two jobs are compatible if they do not overlap.
- **Goal:** group jobs into fewest partitions such that jobs in the same partition are compatible.

Example. Think of scheduling lectures for various courses into as few classrooms as possible. This schedule uses **4** classrooms for scheduling 10 lectures, but we can do better.



This schedule uses **3** classrooms for scheduling 10 lectures, which is optimal.



♢

One idea to solve this problem is to find the maximum compatible set using the previously greedy EFT algorithm, and then remove these jobs from the set and repeat. However, this is not optimal.

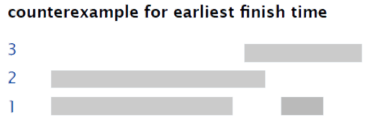
3.3.2 Greedy Algorithm

The greedy algorithm for interval partitioning follows the template

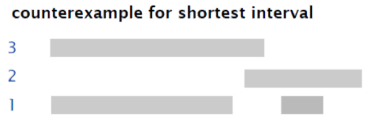
- Go through lectures in some "natural" order.
- Assign each lecture to an *arbitrary* compatible classroom, and create a new classroom if the lecture conflicts with every existing classroom

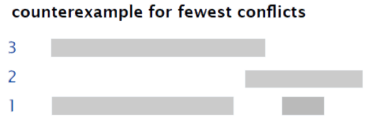
If later we figures that arbitrary assignment is not optimal, we may need to go back to this template, and use a more careful assignment strategy.

It still makes sense to use the same "natural" orders as before: earliest start time, earliest finish time, shortest interval, and fewest conflicts.



Similar to interval scheduling, when we assign each lecture to an arbitrary compatible classroom, three of these heuristics do not work.





FUNCTION EARLIEST-START-TIME-FIRST$(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$
  SORT(lectures) bt start time so that $s_1 \leq s_2 \leq \cdots \leq s_n$
  $d \leftarrow 0$
  FOR $i = 1$ TO n **do**
    IF lecture j is compatible with some classroom **then**
      Schedule lecture j in any such classroom k
    ELSE
      Allocate a new classroom $d + 1$
      Schedule lecture j in classroom $d + 1$
      $d \leftarrow d + 1$
  RETURN schedule

Running Time

- **Key step:** checking if the next lecture can be scheduled at some classroom.
- We can tore classrooms in a priority queue, with the key being the latest finish time of any lecture in the classroom.
- Determining if lecture j is compatible with some classrooms is equivalent to determining whether $s_j \geq f_k$ for some classroom k.
  - If it is compatible, we add jecture j to classroom k with minimum key, and increase its key to f_j.
  - If it is not compatible, we add a new classroom to the priority queue with key f_j.
- There are $\mathcal{O}(n)$ priority queue operations, each of which takes $\mathcal{O}(\log n)$ time.

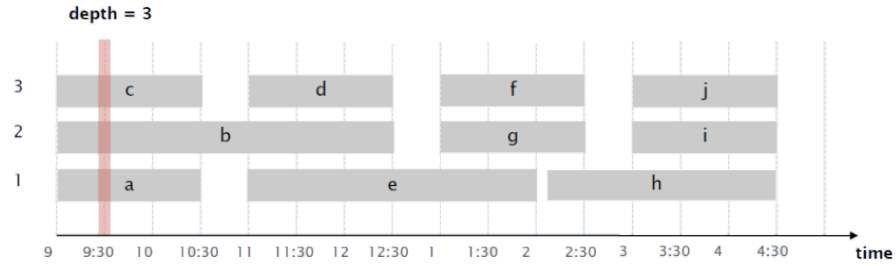
  This gives a total running time of

$$\mathcal{O}(n \log n).$$

3.3.3 Proof of Optimality

Lower Bound

The number of classrooms used by any algorithm is at least the number of lectures running at any time, which we call the "depth". This is because each classroom can only hold one lecture at a time.
We claim that the greedy algorithm uses only these many classrooms.



Upper Bound

- Let d be the number of classrooms used by the greedy algorithm.
- Classroom d was opened because there was a lecture j which was incompatible with some lectures already scheduled in each of $d-1$ other classrooms.
- All these d lectures end after s_j.

  Since we *have sorted the lectures by start time*, they all start at/before s_j.
- So, at time s_j, we have d mutually overlapping lectures.
- Hence, depth $\geq d$, which is the number of classrooms used by the greedy algorithm.

By the lower bound and the upper bound, we have that the greedy algorithm is optimal. ■

3.3.4 Interval Graphs

> **Definition 3.3.1**
> An **interval graph** is a graph whose vertices can be mapped to intervals on the real line such that two vertices are adjacent if and only if their corresponding intervals overlap.

With this definition, we can restate the interval scheduling and partitioning problems as graph problems.

- Interval Scheduling: find a maximum independent set (MIS) in an interval graph.
- Interval Partitioning: find a minimum coloring of an interval graph.

MIS and graph coloring are NP-hard for general graphs, but they are efficiently solvable for interval graphs.

- Graphs which can be obtained from incompatibility of intervals
- In fact, this holds even when we are not given an interval representation of the graph

3.4 Minimizing Lateness

3.4.1 Problem Definition

- We have a single machine.
- Each job j requires t_j units of time and is due by time d_j.
- If it is scheduled to start at time s_j, then it finishes at time $f_j = s_j + t_j$.
- The **lateness** of job j is defined as $\ell_j = \max\{0, f_j - d_j\}$.
- **Goal:** schedule jobs to minimize the maximum lateness, $L = \max_j \ell_j$.

To contrast with interval scheduling problems, we now can decide the start time of each job, and the deadlines are soft.

Example. Below are some jobs given to us.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

An example schedule would be

Note that this schedule is not optimal. ♢

3.4.2 Greedy Algorithm

The greedy algorithm for minimizing lateness follows the template

- Consider jobs one-by-one in some "natural" order.
- Schedule jobs in this order (nothing special to do here, since we have to schedule all jobs and there is only one machine available)

Here, the "natural" order may be

- **Shortest processing time first:** ascending order of processing time t_j.
- **Earliest deadline first:** ascending order of due time d_j.
- **Smallest slack first:** ascending order of $d_j - t_j$.

As expected, some of these orders will not yield the optimal solution.

Shortest processing time first

	1	2
t_j	1	10
d_j	100	10

Smallest slack first

	1	2
t_j	1	10
d_j	2	10

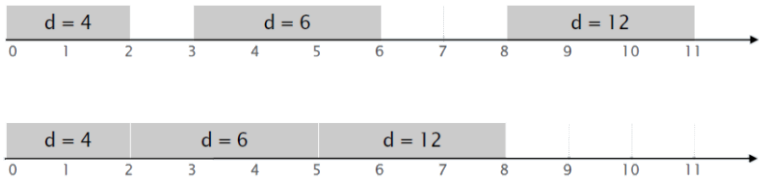
We can implement the greedy algorithm using **earliest deadline first**.

FUNCTION EARLIEST-DEADLINE-FIRST$(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$
  SORT(jobs) by deadline so that $d_1 \leq d_2 \leq \cdots \leq d_n$
  $t \leftarrow 0$
  FOR $j = 1$ TO n **do**
    Assign job j to interval $[t, t + t_j]$
    $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
  RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$

3.4.3 Proof of Optimality

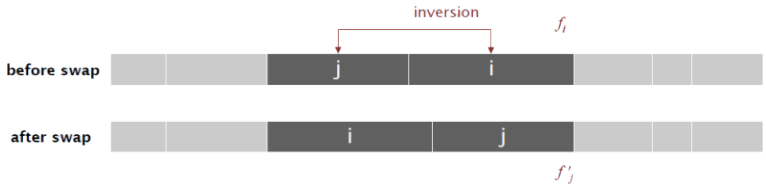
We make the following observations.

- **Observation 1:** There is an optimal schedule with **no idle time**.



- **Observation 2:** Earliest deadline first has no idle time.
- **Observation 3:** By definition, earliest deadline first has no inversions.

  There, an inversion is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i.
- **Observation 4:** If a schedule with no idle time has at least one inversion, it has a pair of inverted jobs scheduled consecutively.
- **Observation 5:** Swapping adjacently scheduled inverted jobs does not increase lateness but reduces the number of inversions by one.



Proof. Let ℓ_k and ℓ'_k denote the lateness of job k before and after the swap.

Let $L = \max_k \ell_k$ and $L' = \max_k \ell'_k$ be the maximum lateness.

Let i and j be the inverted jobs.

- $\ell_k = \ell'_k$ for all $k \neq i, j$.
- $\ell'_i \leq \ell_i$
- $\ell' j = f'_j - d_j = f_i - d_j \leq f_i - d_i = \ell_i$
  This uses the fact that, due to the inversion, $d_j \geq d_i$.
- $L' = \max \left\{ \ell'_i, \ell'_j, \max_{k \neq i,j} \ell'_k \right\} \leq \max \left\{ \ell_i, \ell_j, \max_{k \neq i,j} \ell_k \right\} = L$ ■

Observations 4 and 5 together is the key to the proof of optimality.

Remark

Recall the proof of optimality of the greedy algorithm for interval scheduling

- Take an optimal solution matching greedy for r steps, and produced another optimal solution matching greedy for $r+1$ steps
- "Wrapped" in a proof by contradiction or proof by induction

Observations 4 and 5 provide a similar structure. If optimal solution does not fully match greedy (the number of inversions ≥ 1), we can swap an adjacent inverted pair and reduce the number of inversions by one.

By Contradiction

Proof. Suppose for contradiction that the greedy EDF solution is not optimal

- Consider an optimal schedule S^* with the fewest inversions[1]. Without loss of generality, assume it has no idle time.
- Since the greedy EDF solution is not optimal, there is at least one inversion in S^*.
- By Observation 4, the pair of inversion (i, j) are scheduled consecutively (adjacent).
- By Observation 5, swapping the adjacent pair keeps the schedule optimal but reduces the number of inversions by 1

  This is a contradiction, as S^* has the fewest inversions.

Thus, the greedy EDF solution is optimal. ■

By Induction

Proof. By induction on the number of inversions in an optimal schedule.

Claim. For each $r \in \left\{0, 1, 2, \ldots, \binom{n}{2}\right\}$, there is an optimal schedule with at most r inversions.

- **Base case:** $r = \binom{n}{2}$

  This is trivially true.
- **Inductive step:** $t+1 \to t$

  Suppose the claim holds for $t+1$.

  Take an optimal schedule S^* with at most $t+1$ inversions.

  Without loss of generality, assume it has no idle time.

  - If S^* has at most t inversions, we are done.
  - If S^* has exactly $t+1$ inversions, then there is a pair of inverted jobs (i, j) scheduled consecutively by Observation 4.

    By Observation 5, swapping the adjacent pair keeps the schedule optimal but reduces the number of inversions by 1.

    The number of inversions in the new schedule is at most t, and the claim holds.

Claim for $r = 0$ shows optimality of the greedy EDF solution. ■

[1] Note that this part is a little flawed, as S^* may be different from the greedy EDF solution based on how we break ties. To fix this, we can simply change the way we break ties until we get the same solution as the greedy EDF solution. Since these jobs are adjacent, we can do this by swapping adjacent jobs, and, by Observation 5, this will not increase the lateness, so the solution remains optimal.

Contradiction vs Induction

The favour over contradiction or induction is a matter of taste. It may be the case that for some problems, one method is easier to apply than the other. There is no inherent difference in the two methods, as they both make the same claim and use the same key argument, and one does not need to stick to one method. Meanwhile, as we have seen for the interval partitioning problem, sometimes we may require an entirely different method to prove optimality.

3.5 Lossless Compression

3.5.1 Problem Definition

Example. We have a document that is written using n distinct labels. The naïve encoding would use $\log_2 n$ bits for each label, but it is not optimal. We can assign shorter encodings to more frequent labels.

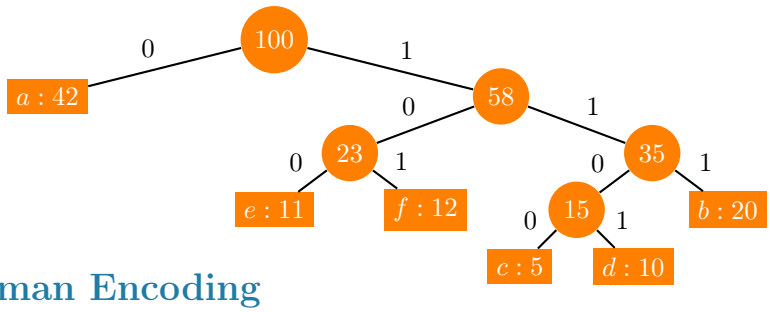
Say we assign $a = 0$, $b = 1$, $c = 01$, ... based on the frequency of the labels. However, now we have created confusion when decoding, as 01 can be decoded as either 'ab' or 'c'.

To avoid conflicts, we need a **prefix-free encoding**, where no label is a prefix of another label. Now, we can read left to right, and whenever the part to the left becomes a valid encoding, we greedily decode it, and continue with the rest. ♢

Definition 3.5.1 Lossless Compression

Given n symbols and their frequencies $(w_1, \ldots, w_n)$, find a prefix-tree encoding with length $(\ell_1, \ldots, \ell_n)$ assigned to the symbols which minimizes $\sum_{i=1}^{n} w_i \ell_i$.

We observe that prefix-free encodings can be represented by a binary tree.

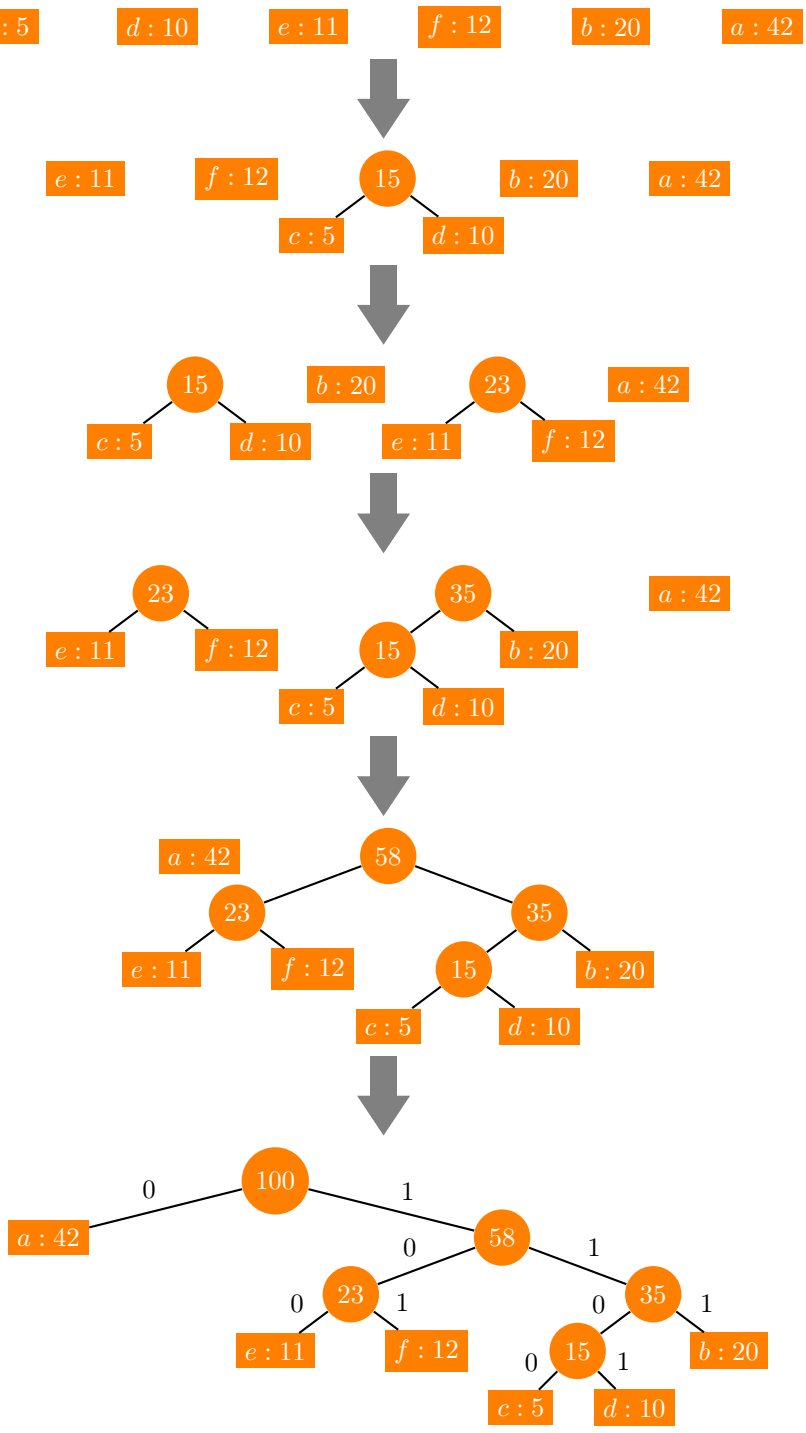


3.5.2 Huffman Encoding

The Huffman encoding algorithm is a greedy algorithm that generates an optimal prefix-tree encoding for a given set of symbols and their frequencies.

FUNCTION HUFFMAN-ENCODING$(n, w_1, w_2, \ldots, w_n)$
  $Q \leftarrow$ Priority-Queue
  FOR each symbol x **do**
    INSERT(Q, x, w_x)
  FOR $i = 1$ TO $n - 1$ **do**
    $(x, w_x) \leftarrow$ Extract-Min(Q)
    $(y, w_y) \leftarrow$ Extract-Min(Q)
    INSERT$(Q, (x, y), w_x + w_y)$
  RETURN the root of the tree

Example. We use the Huffman encoding algorithm to generate the tree for the previous example.



♢

Running Time

The running time of the Huffman encoding algorithm is $\mathcal{O}(n \log n)$. However, it can be made $\mathcal{O}(n)$ if the labels are sorted by frequency, by using two queues.

3.5.3 Proof of Optimality

Proof. By induction on the number of symbols n.

- **Base case:** $n = 2$

  Both encodings which assign 1 bit to each symbol are optimal.

- **Inductive step:** $n - 1 \to n$

  Assume Huffman encoding returns an optimal encoding with $n - 1$ symbols. Consider the case with n symbols.

  > **Lemma 1**
  > If $w_x < w_y$, then $\ell_x \geq \ell_y$ in any optimal tree.

  Proof. Proof of Lemma 1

  Suppose for contradiction that $w_x < w_y$ and $\ell_x < \ell_y$ in an optimal tree.

  Swapping x and y strictly decreases the total encoding length, as

  $$w_x \cdot \ell_y + w_y \cdot \ell_x < w_x \cdot \ell_x + w_y \cdot \ell_y.$$

  This is a contradiction. ■

  Consider the two symbols x and y with lowest frequency which Huffman encoding combines in the first step.

  > **Lemma 2**
  > Exists an optimal tree T in which x and y are siblings.
  > That is, for some p, x and y are assigned encodings of the form $p0$ and $p1$.

  Proof. Proof of Lemma 2

  1. Let T be an optimal tree.
  2. Let x be the label with the lowest frequency in T.
  3. If x does not have the longest encoding in T, swap it with the label with the longest encoding.
  4. Due to optimality, x must have a sibling y' (otherwise, we can swap x with its parent and reduce the total encoding length).
  5. If y' is not y, swap it with y.
  6. We check that Steps 3 and 5 does not change the overall length. ■

  Let x and y be the two least frequency symbols that Huffman combines in the first step into "xy".

  Let H be the Huffman tree produced.

  Let T be an optimal tree in which x and y are siblings.

  Let H' and T' be obtained from H and T bt treating xy as one symbol with frequency $w_x + w_y$.

  By the inductive hypothesis, T' is optimal for the $n - 1$ symbols, so $\textsc{Length}(H') \leq \textsc{Length}(T')$.

  - $\textsc{Length}(H) = \textsc{Length}(H') + (w_x + w_y) \cdot 1$
  - $\textsc{Length}(T) = \textsc{Length}(T') + (w_x + w_y) \cdot 1$

  Therefore, $\textsc{Length}(H) \leq \textsc{Length}(T)$.

Thus, the Huffman encoding algorithm returns an optimal encoding. ■

3.6 Other Greedy Algorithms

Some other greedy algorithms include

- **Shortest Path:** Dijkstra's algorithm
- **Minimum Spanning Tree:** Kruskal's algorithm and Prim's algorithm

Part II

Appendices

BIBLIOGRAPHY

[Cor+22] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th edition. Cambridge, MA: The MIT Press, 2022. ISBN: 978-0-262-04630-5. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

[Huf52] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pages 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[Sha21] Nisarg Shah. *CSC373: Algorithm Design, Analysis & Complexity*. 2021. URL: https://www.cs.toronto.edu/~nisarg/teaching/373f21/.

[Wie24] Nathan Wiebe. *Course recordings, slides, and other materials*. OneDrive shared folder. 2024. URL: https://utoronto-my.sharepoint.com/personal/nathan_wiebe_utoronto_ca/_layouts/15/onedrive.aspx?ga=1&id=%2Fpersonal%2Fnathan%5Fwiebe%5Futoronto%5Fca%2FDocuments%2FCSC373.

[WS24] Nathan Wiebe and Harry Sha. *CSC373H1S 20241*. 2024. URL: https://q.utoronto.ca/courses/337418.

INDEX