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 Bubble-Sort 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.

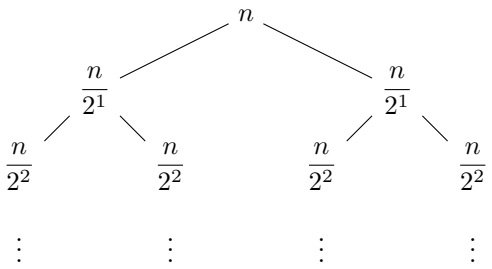
2.1.1 Merge Sort

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.

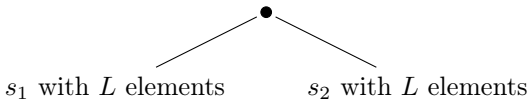
- **Base case**: $s(2)$

  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
  - ...

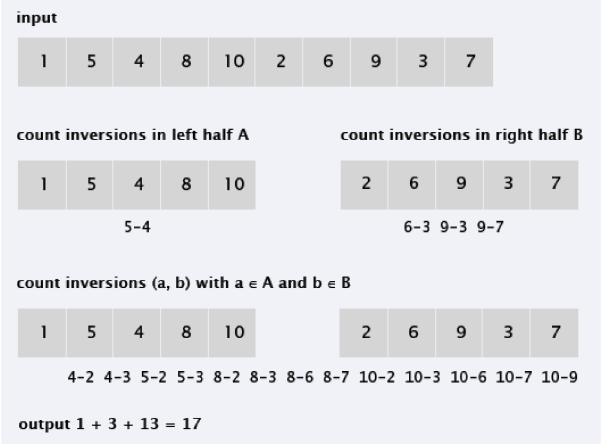
Definition 2.1.1 Inversion

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

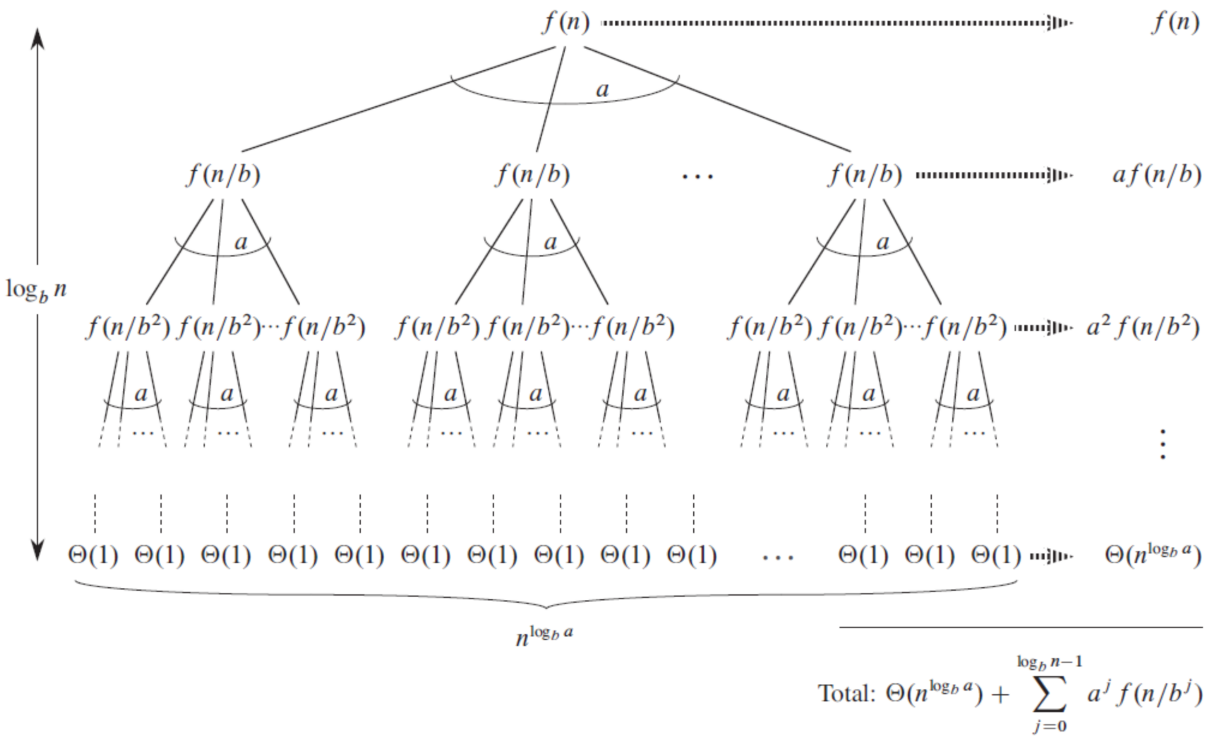
Theorem 2.2.1 Master Theorem

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.

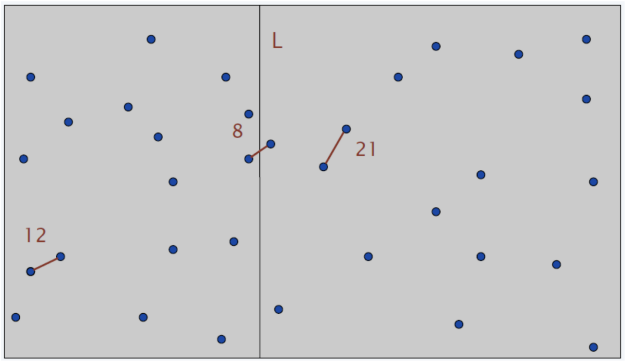
- **Applications**

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

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

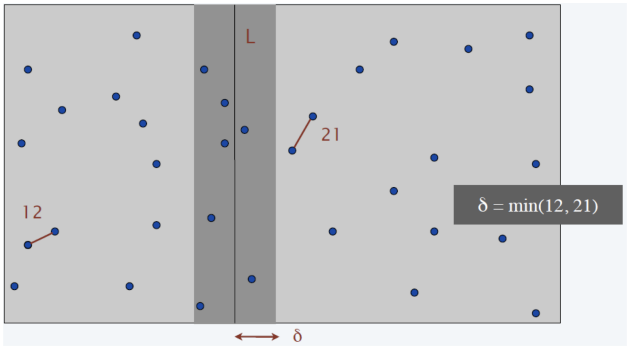
We can use the divide and conquer paradigm to solve this problem.

- **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

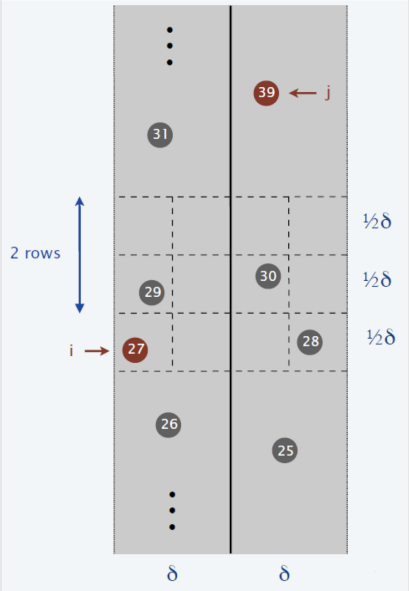
We chose the number 11 on purpose.

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

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

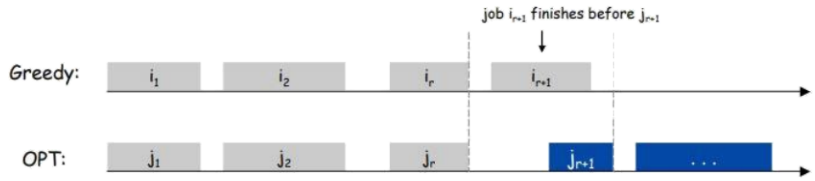
3.2.3 Proof of Optimality

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 $j+1, \ldots, n$.

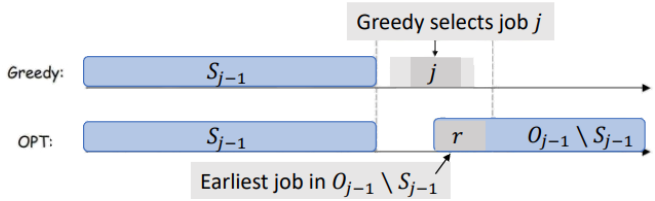
  $\exists T \subseteq \{j+1, \ldots, n\}$ such that $S_j \cup T$ is an optimal solution.
- **Inductive claim:** for all $t \in \{0, 1, \ldots, n\}$, S_t is promising.
  - For $t = n$, if S_n is promising, then it must be an optimal solution.
  - We choose $t = 0$ as our base case since it is trivially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

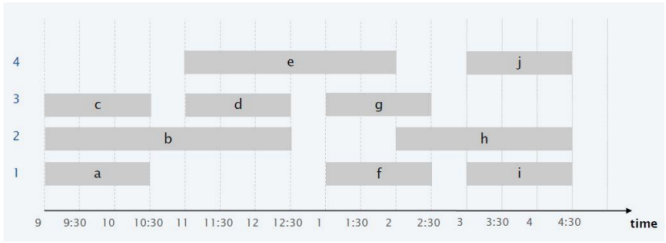
- For proof by induction, this is the key induction step.
- For proof by contradiction, we take the greatest j for which the greedy solution can be extended to an optimal solution, and derive a contradiction by extending the greedy solution after $j+1$ iterations.

3.3 Interval Partitioning

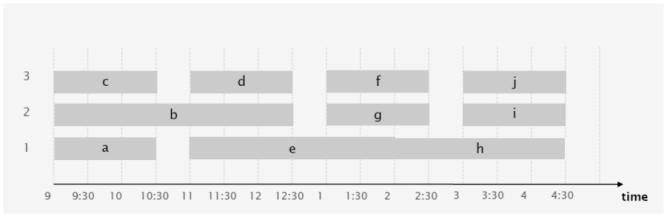
3.3.1 Problem Definition

- 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.

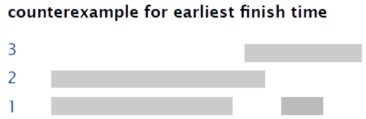
3.3.2 Greedy Algorithm

The greedy algorithm for interval partitioning follows the template

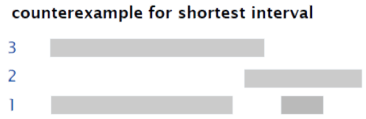
- Go through lectures in some "natural" order.
- Assign each lecture to an *arbitrary* compatible classroom, and create a new classroom if the lecture conflicts with every existing classroom

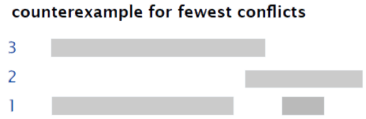
If later we figures that arbitrary assignment is not optimal, we may need to go back to this template, and use a more careful assignment strategy.

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.

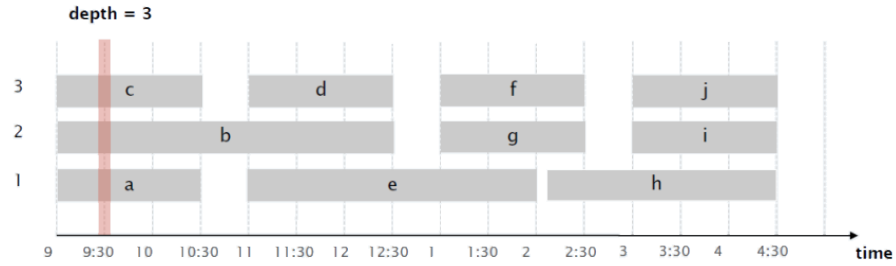
  This gives a total running time of

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

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

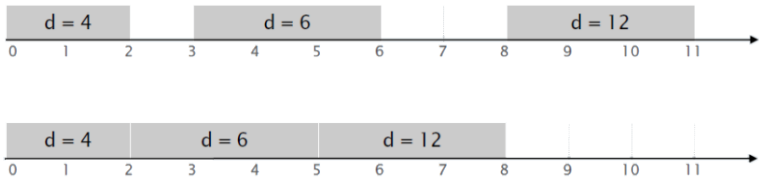
We can implement the greedy algorithm using **earliest deadline first**.

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

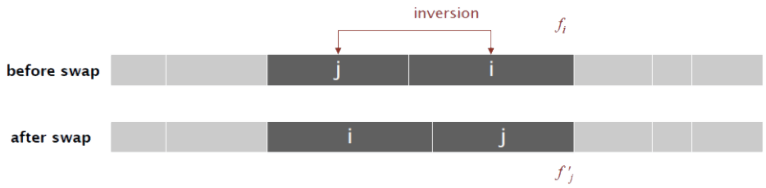
We make the following observations.

- **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.

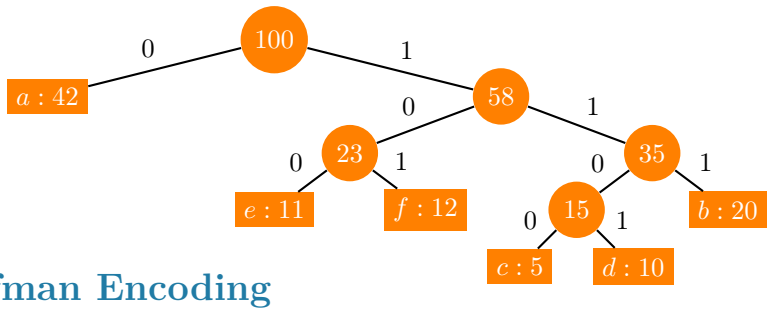
Say we assign $a = 0$, $b = 1$, $c = 01$, ... based on the frequency of the labels. However, now we have created confusion when decoding, as 01 can be decoded as either 'ab' or 'c'.

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.

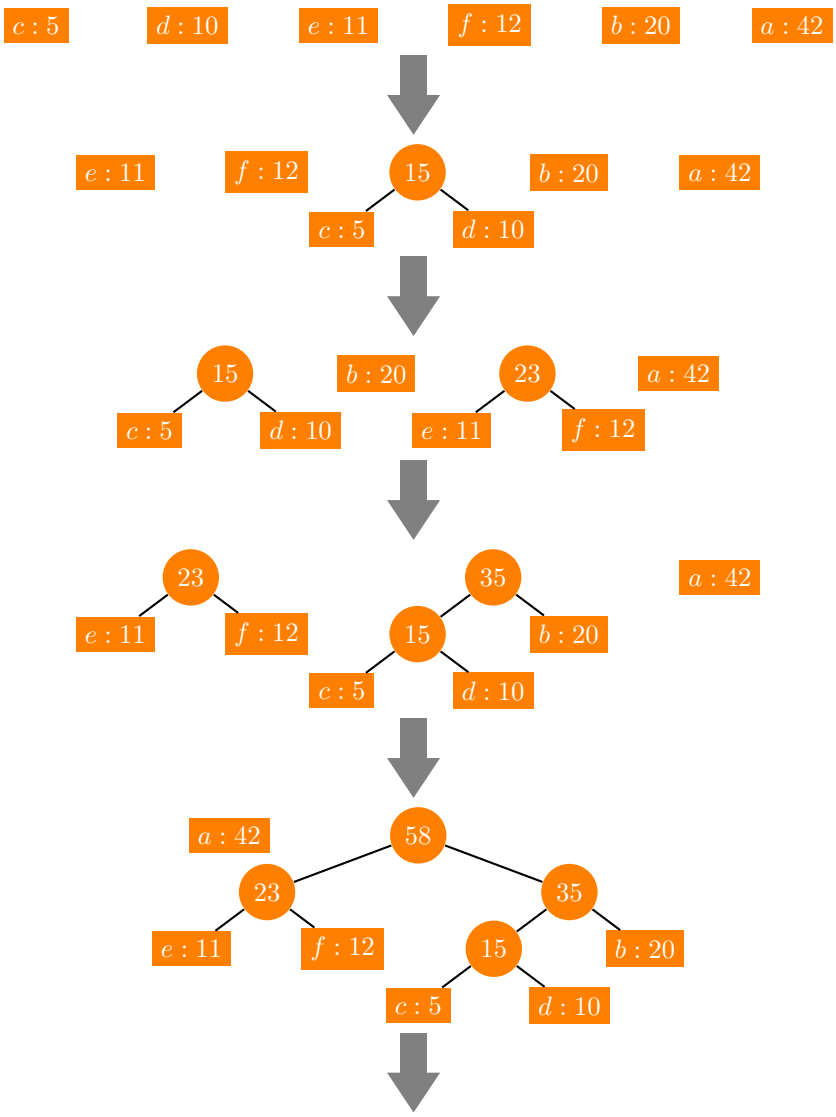


3.5.2 Huffman Encoding

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

3.6.1 Dijkstra's Algorithm

Dijsktra's algorithm is a greedy algorithm that finds the *shortest path* from a single source to all other vertices in a weighted graph.

3.6.2 Kruskal's Algorithm and Prim's Algorithm

Kruskal's algorithm and Prim's algorithm are greedy algorithms that finds a *minimum spanning tree* for a connected, undirected graph.

```
1: Function Dijkstra(G, s)
2:     d[v] ← ∞ for all v ∈ V
3:     d[s] ← 0
4:     Q ← Priority-Queue
5:     Insert(Q, s, 0)
6:     While Q is not empty do
7:         (u, d_u) ← Extract-Min(Q)
8:         For each edge (u, v) do
9:             If d_u + w(u, v) < d[v] then
10:                d[v] ← d_u + w(u, v)
11:                Insert(Q, v, d[v])
12:    Return d
```

CHAPTER

4 DYNAMIC PROGRAMMING

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

— *Jeff Erickson,* ***Algorithms***

4.1 Introduction

Dynamic programming was developed by Richard Bellman in the 1950s, and is both a mathematical optimization method and a computer programming method. It is a method for solving complex problems by breaking them down into simpler subproblems, similar to the divide-and-conquer method, with the added benefit of storing the results of subproblems so that they are not recomputed. It is also more powerful than divide-and-conquer, as it can be used to solve problems where subproblems overlap.

- Breaking the problem down into simpler subproblems, solve each subproblem just once, and store their solutions.
- The next time the same subproblem occurs, instead of recomputing its solution, simply look up its previously computed solution.
- Hopefully, we save a lot of computation at the expense of modest increase in storage space.
- Also called "**memoization**".

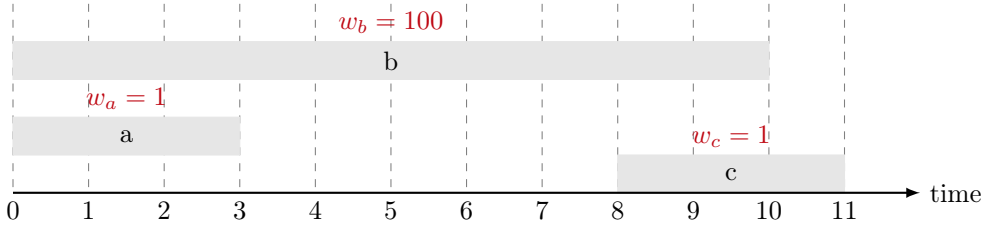
4.2 Weighted Interval Scheduling

4.2.1 Problem Definition

- Job j starts at time s_j and finishes at time f_j.
- Each job j has a weight w_j.
- Two jobs i and j are compatible if $f_i \leq s_j$.

- **Goal:** find a set S of mutually compatible jobs that maximizes $\sum_{j \in S} w_j$.

If all the weights are equal, this is the same as the **interval scheduling** problem. However, if the weights are not equal, the greedy algorithm for interval scheduling fails spectacularly.



What if we use other orderings? We can order the jobs by weight and choose the one with the highest w_j first, or we can order by maximum weight per time, and select jobs with the highest $w_j/(f_j - s_j)$ first. However, none of these orderings work. They are arbitrarily worse than the optimal solution.

Convention

- Jobs are sorted by finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- $p[j] = \max\{i : i < j \text{ and } f_i < s_j\}$, the largest $i < j$ such that job i is compatible with job j.
  - Jobs $1, \ldots, i$ are compatible with j, but jobs $i+1, \ldots j-1$ are not.
  - $p[j]$ can be computed via binary search in $O(\log n)$ time.

4.2.2 Dynamic Programming Solution

- Let OPT be an optimal solution to the problem.
- There are two options for job n:
  - Job n is in OPT
    We cannot use the incompatible jobs $\{p[n]+1, \ldots, n-1\}$
    Must select the optimal solution for the remaining jobs $\{1, \ldots, p[n]\}$.
  - Job n is not in OPT
    Must select the optimal solution for the remaining jobs $\{1, \ldots, n-1\}$.
- OPT is the best of these two options.
  Notice that in both options, we need to solve the problem on a prefix of our ordering.
- Let $OPT(j)$ be the maximum total weight of compatible jobs from $\{1, \ldots, j\}$.
- **Base case:** $OPT(0) = 0$.
- **Recurrence:** $OPT(j) = \max\{w_j + OPT(p[j]), OPT(j-1)\}$.
  - Job j is selected: optimal weight is $w_j + OPT(p[j])$.
  - Job j is not selected: optimal weight is $OPT(j-1)$.

  The **Bellman equation** is

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{w_j + OPT(p[j]), OPT(j-1)\} & \text{if } j > 0 \end{cases}$$

Example (Brute Force Solution)**.** Below is a brute force solution to the problem.

```
1: Function Brute-Force(n, s_1, ..., s_n, f_1, ..., f_n, w_1, ..., w_n)
2:     Sort jobs by finish time and renumber so that f_1 ≤ f_2 ≤ ... ≤ f_n.
3:     Compute p[1], ..., p[n] via binary search.
4:     Return Compute-Opt(n)

5: Function Compute-Opt(j)
6:     If j = 0 then
7:         Return 0
8:     Else
9:         Return max{w_j + Compute-Opt(p[j]), Compute-Opt(j − 1)}
```

Note that Compute-Opt has a time complexity of $O(2^n)$, which is extremely inefficient. This is because some solutions are being computed multiple times. For example, Compute-Opt$(j-1)$ is computed multiple times. For example, if $p[5] = 3$, then Compute-Opt(3) is computed twice: once for $j = 4$ and once for $j = 5$.
To imporve, we can simply remember the results of subproblems and look them up when needed. ♢

Let's store Store-Opt(j) in an array $M[j]$.

Top-Down Dynamic Programming

```
1: Function Top-Down(n, s_1, ..., s_n, f_1, ..., f_n, w_1, ..., w_n)
2:     Sort jobs by finish time and renumber so that f_1 ≤ f_2 ≤ ... ≤ f_n.
3:     Compute p[1], ..., p[n] via binary search.
4:     M[0] ← 0.
5:     Return M-Compute-Opt(n)

6: Function M-Compute-Opt(j)
7:     If M[j] is ininitialized then
8:         M[j] ← max{w_j + Store-Opt(p[j]), Store-Opt(j − 1)}
9:     Return M[j]
```

Claim. This memoized version takes $\mathcal{O}(n \log n)$ time.

- Sorting by finish time takes $\mathcal{O}(n \log n)$ time.
- Computing $p[1], \ldots, p[n]$ takes $\mathcal{O}(n \log n)$ time.
- For each j, at most one of the calls to M-Compute-Opt(j) will make two recursive calls.
  - At most $\mathcal{O}(n)$ total calls to M-Compute-Opt.
  - Each call takes $\mathcal{O}(1)$ time, not considering the time spent in the recursive calls.
  - Hence, the initial call, M-Compute-Opt(n), finished in $\mathcal{O}(n)$ time.

Bottom Up Dynamic Programming

In bottom up dynamic programming, we need to find an order in which to call the functions so that the subsolutions are ready when needed. However, this is generally more efficient, as it avoids the overhead of recursion.

```
FUNCTION BOTTOM-UP(n, s_1, ..., s_n, f_1, ..., f_n, w_1, ..., w_n)
    Sort jobs by finish time and renumber so that f_1 ≤ f_2 ≤ ... ≤ f_n.
    Compute p[1], ..., p[n] via binary search.
    M[0] ← 0.
    FOR j = 1 to n do
        M[j] ← max{w_j + M[p[j]], M[j − 1]}
    RETURN M[n]
```

Top-Down vs. Bottom-Up

- Top-Down may be preferred...
  - ...when not all sub-solutions need to be computed on some inputs
  - ...because one does not need to think of the "right order" in which to compute sub-solutions
- Bottom-Up may be preferred...
  - ...when all sub-solutions will anyway need to be computed
  - ...because it is faster as it prevents recursive call overheads and unnecessary random memory accesses
  - ...because sometimes we can free-up memory early

Optimal Solution Reconstruction

So far, we have only computed the maximum total weight of compatible jobs. We have not yet computed the set of jobs that achieves this maximum. To do so, we can simply store the choices made in the Bellman equation. So, we compute two quantities

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{w_j + OPT(p[j]), OPT(j-1)\} & \text{if } j > 0 \end{cases}$$

and

$$S(j) = \begin{cases} \varnothing & \text{if } j = 0 \\ S(j-1) & \text{if } j > 0 \text{ and } OPT(j-1) \geq w_j + OPT(p[j]) \\ \{j\} & \text{if } j > 0 \text{ and } OPT(j-1) < w_j + OPT(p[j]) \end{cases}$$

This works with both top-down and bottom-up implementations. We can compute OPT and S simultaneously, or compute S after computing OPT.

One may notice that this implementation is wasting a lot of space. We are copying the entire solution of $S(j-1)$ to $S(j)$, which is a by-element array copy. We can avoid this by only storing the change in the solution,

$$S(j) = \begin{cases} \perp & \text{if } j = 0 \\ L & \text{if } j > 0 \text{ and } OPT(j-1) \geq w_j + OPT(p[j]) \\ R & \text{if } j > 0 \text{ and } OPT(j-1) < w_j + OPT(p[j]) \end{cases}$$

where we store only one bit of information for each j: which option yielded the maximum weight.
To reconstruct the optimal solution, start with $j = n$:

- If $S(j) = L$, update $j \leftarrow j - 1$.
- If $S(j) = R$, add job j to the solution and update $j \leftarrow p[j]$.
- If $S(j) = \perp$, stop.

4.2.3 Optimal Substructure Property

Dynamic programming applies well to problems that have optimal substructure property. That is, the optimal solution to a problem can be computed easily given optimal solution to subproblems.

> **Remark**
> Recall that divide-and-conquer also uses this property. It is a special case in which the subproblems do not "overlap", so, there is no need for memoization.
> In dynamic programming, two of the subproblems may in turn require access to solution to the same subproblem.

4.3 Knapsack Problem

4.3.1 Problem Definition

- There are n items, each provides a value $v_i > 0$ and a weight $w_i > 0$.
- There is a knapsack that can hold a maximum weight of W.
- Assume that W, v_i-s, and w_i-s are all integers.
- **Goal:** pack the knapsack with a subset of items with highest total value subject to their total weight being at most W.

4.3.2 Dynamic Programming Solution

- Let $OPT(i, w)$ be the maximum value we can pack using only items $1, \ldots, i$ in s knapsack of capacity w.

  Goal: compute $OPT(n, W)$.
- Consider item i
  - If $w_i > w$, then we can't choose i. Use $OPT(i-1, w)$.
  - If $w_i \leq w$, then we have two options:
    - If we choose i, then the best value is $v_i + OPT(i-1, w-w_i)$.
    - If we don't choose i, then the best value is $OPT(i-1, w)$.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{v_i + OPT(i-1, w-w_i), OPT(i-1, w)\} & \text{otherwise} \end{cases}$$

Running Time

Consider the possible evaluations of $OPT(i, w)$

- $i \in \{1, \ldots, n\}$
- $w \in \{0, \ldots, W\}$
- There are $\mathcal{O}(n \cdot W)$ possible evaluations of $OPT(i, w)$.

  Each is computed in $\mathcal{O}(1)$ time, at most once.
- The total running time is $\mathcal{O}(n \cdot W)$.

This is a **pseudo-polynomial time** algorithm. The time is not polynomial in

$$\log W + \sum_{i=1}^{n} (\log v_i + \log w_i),$$

the number of bits required to represent the input, but it is polynomial in the numeric value of the input in unary representation.

Definition 4.3.1 Pseudo-Polynomial Time Algorithm

An algorithm is **pseudo-polynomial time** if its running time is polynomial in the numeric value of the input in unary representation, but not in the number of bits required to represent the input.

Definition 4.3.2 Unary Representation

A number is in **unary representation** if it is represented as a sequence of 1's. For example, the number 5 is represented as 11111.

Remark

For the knapsack problem, the number of bits required to represent the input is

$$T = \log W + \sum_{i=1}^{n} (\log v_i + \log w_i).$$

Running time of the dynamic programming solution is $\mathcal{O}(n \cdot W)$. If W takes the form 2^n, then $T \in \mathcal{O}(n)$, but the running time is $\mathcal{O}(n \cdot 2^n)$. There is no way to $n \cdot 2^n$ as a polynomial in n.

However, if we consider the unary representation of the input, then

$$T = W + \sum_{i=1}^{n} (v_1 + w_1).$$

We know that $T \geq n$ and $T \geq W$, so the running time $\mathcal{O}(n \cdot W) = \mathcal{O}(T^2)$ is polynomial in T.

Another Dynamic Programming Solution

In this solution, we will focus on the values instead of the weights.

- Let $OPT(i, v)$ be the minimum weight we need to achieve a value of at least v using only items $1, \ldots, i$.

  Goal: compute $OPT(n, V)$, where $V = \sum_{i=1}^{n} v_i$.

- Consider item i.
  - If we choose i, then the best weight is $w_i + OPT(i-1, v-v_i)$.
  - If we don't choose i, then the best weight is $OPT(i-1, v)$.

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, v) & \text{if } v_i > v \\ \min\{w_i + OPT(i-1, v-v_i), OPT(i-1, v)\} & \text{otherwise} \end{cases}$$

This approach has a running time of $\mathcal{O}(n \cdot V)$, which is also pseudo-polynomial time.

4.4 Single-Source Shortest Paths

4.4.1 Problem Definition

- A directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w), and s source vertex s.
- **Goal:** compute the shortest path from s to every other vertex t.

When $\ell_{vw} \geq 0$ for all edges, we can use Dijkstra's algorithm. However, when ℓ_{vw} can be negative, Dijkstra's algorithm does not work.

Example. Consider when we have a cycle of negative length. In this case, we can keep going around the cycle to get a path of arbitrarily small length.
In this case, the shortest paths are not well-defined. ♢

To avoid this issue, we need to restrict the graph to avoid negative cycles.

Claim. With no negative cycles, there is always a shortest path from any vertex to any other vertex that is **simple**.

Proof. Consider the shortest path from s to t with the fewest edges among all shortest $s \rightsquigarrow t$ paths.
If it has a cycle, removing the cycle creates a path with fewer edges that is no longer than the original path ■

4.4.2 Dynamic Programming Solution

Optimal Substructure Property

Consider a simple shortest path P from s to t.

- It could be just a single edge. But if P has more than one edges, consider u which immediately precedes t in the path.
- If $s \rightsquigarrow t$ is the shortest, $s \rightsquigarrow u$ must also be the shortest, and it must use one fewer edge than $s \rightsquigarrow t$.

Let $OPT(t, i)$ be the length of the shortest path from s to t using at most i edges. Then,

- Either this path uses at most $i - 1$ edges, so $OPT(t, i) = OPT(t, i - 1)$, or
- It uses i edges, so $OPT(t, i) = \min_u OPT(u, i - 1) + \ell_{ut}$.

$$OPT(t, i) = \begin{cases} 0 & \text{if } i = 0 \text{ or } t = s \\ \infty & \text{if } i = 0 \text{ and } t \neq s \\ \min\left\{OPT(t, i-1), \min_u OP(u, i-1) + \ell_{ut}\right\} & \text{otherwise} \end{cases}$$

Running Time

There are $\mathcal{O}(n^2)$ entries to evaluate, and each entry takes $\mathcal{O}(n)$ time to evaluate. Hence, the running time is $\mathcal{O}(n^3)$.

4.4.3 All-Pairs Shortest Paths

Problem Definition

- A directed graph $G = (V, E)$ with edge lengths ℓ_{vw} on each edge (v, w).
- **Goal:** compute the shortest path from every vertex s to every other vertex t.

An naïve approach may be to run the previous algorithm for each vertex s. However, this approach has a running time of $\mathcal{O}(n^4)$, while it is possible to solve the problem in $\mathcal{O}(n^3)$ time.
Let $OPT(u, v, k)$ be the length of the shortest simple path from u to v using only vertices in $\{1, \ldots, k\}$ as intermediate vertices. Then,

$$OPT(u, v, k) = \begin{cases} \ell_{uv} & \text{if } k = 0 \\ \min\{OPT(u, v, k-1), OPT(u, k, k-1) + OPT(k, v, k-1)\} & \text{otherwise} \end{cases}$$

4.5 Chain Matrix Product

4.5.1 Problem Definition

- Given matrices $M_1, \ldots, M_n$, where the dimensions of M_i are $d_{i-1} \times d_i$.
- **Goal:** compute the product $M_1 \cdot M_2 \cdot \cdots \cdot M_n$.

We know that matrix multiplication is associative, so the order of multiplication does not matter. However, the number of scalar multiplications required to compute the product does depend on the order of multiplication. Assume we use the brute force approach for matrix multiplication. Multiplying $p \times q$ and $q \times r$ matrices requires $p \cdot q \cdot r$ scalar multiplications.
Note: Our input is simply the dimensions $d_0, d_1, \ldots, d_n$ (such that each M_i is $d_{i-1} \times d_i$) and not the actual matrices

4.5.2 Dynamic Programming Solution

Optimal Substructure

We can use dynamic programming to solve this problem, as it has the optimal substructure property.

- Think of the final product computed, say $A \cdot B$
- A is the product of some prefix, B is the product of the remaining suffix
- For the overall optimal computation, each of A and B should be computed optimally

Let $OPT(i, j)$ be the minimum number of scalar multiplications required to compute the product $M_i \cdot M_{i+1} \cdot \cdots \cdot M_j$. Then,

$$OPT(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{OPT(i, k) + OPT(k+1, j) + d_{i-1} \cdot d_k \cdot d_j\} & \text{if } i < j \end{cases}$$

Running Time

There are $\mathcal{O}(n^2)$ entries to evaluate, and each entry takes $\mathcal{O}(n)$ time to evaluate. Hence, the running time is $\mathcal{O}(n^3)$.

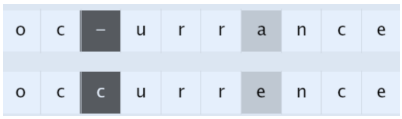
4.6 Edit Distance

4.6.1 Problem Definition

This is also known as the sequence alignment problem. We ask how similar are string $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$.
Suppose we can delete or replace symbols, and can do these operations on any symbol in either string. We want to find the minimum number of operations required to match the two strings.

Example. Consider `ocurrance` and `occurrence`.

(a) 6 replacements, 1 deletion



(b) 1 replacement, 1 deletion

♢

- **input**
  - Strings $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$.
  - Cost $d(a)$ of deleting symbol a.
  - Cost $r(a, b)$ of replacing symbol a with symbol b.
    We assume $r(a, b) = r(b, a)$ and $r(a, a) = 0$, for all a, b.
- **Goal**

  Compute the minimum total cost for matching X and Y.

4.6.2 Dynamic Programming Solution

Optimal Substructure

- **Goal:** match $x_1, \dots, x_m$ with $y_1, \dots, y_n$.
- Consider the last symbols x_m and y_n.
- There are three options:
  - Delete x_m, and optimally match $x_1, \dots, x_{m-1}$ with $y_1, \dots, y_n$.
  - Delete y_n, and optimally match $x_1, \dots, x_m$ with $y_1, \dots, y_{n-1}$.
  - Match x_m and y_n, and optimally match $x_1, \dots, x_{m-1}$ with $y_1, \dots, y_{n-1}$.
    - We increase the cost by $r(x_m, y_n)$ if $x_m \neq y_n$.
    - Recall that $r(a, a) = 0$, so we don't increase the cost if $x_m = y_n$.
- Hence in the dynamic programming, we need to compute the optimal solutions for matching prefixes of X and Y, $x_1, \dots, x_i$ and $y_1, \dots, y_j$.

Let $E[i, j]$ be the distance between $x_1, \dots, x_i$ and $y_1, \dots, y_j$. Then,

$$E[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ B & \text{if } i = 0 \text{ and } j > 0 \\ A & \text{if } i > 0 \text{ and } j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where

$$A = d(x_i) + E[i-1, j]$$
$$B = d(y_j) + E[i, j-1]$$
$$C = r(x_i, y_j) + E[i-1, j-1]$$

Running Time

There are $\mathcal{O}(m \cdot n)$ entries to evaluate, and each entry takes $\mathcal{O}(1)$ time to evaluate. Hence, the running time is $\mathcal{O}(m \cdot n)$.

Space Optimization

The current solution uses $\mathcal{O}(m \cdot n)$ space. However, we can optimize the space complexity of the dynamic programming solution by using a bottom up approach.

- While computing $E[\cdot, j]$, we only need to store $E[\cdot, j]$ and $E[\cdot, j-1]$, so the additional space required is $\mathcal{O}(m)$.
- By storing two rows at a time instead, we can make it $\mathcal{O}(n)$.
- Usually, we include the storage of inputs, so both are $\mathcal{O}(m+n)$.

However, this is not enough if we want to compute the actual solution. Hirschberg's algorithm is a space-optimized version of the dynamic programming solution that can compute the actual solution in $\mathcal{O}(m+n)$ space.

4.7 The Traveling Salesman Problem

4.7.1 Problem Definition

- A complete graph $G = (V, E)$ with distance $d_{i,j}$ from vertex i to vertex j.

  Note that the input needs to satisfy the triangle inequality: $d_{i,j} \leq d_{i,k} + d_{k,j}$ for all i, j, k.
- **Goal:** find the shortest cycle that visits every vertex exactly once. This is called the **Hamiltonian cycle**.

We start at node $v_1 = 1$, and want to visit other nodes in some order, say $v_2, v_3, \ldots, v_n, v_1$. The total distance is

$$d_{1,v_2} + d_{v_2,v_3} + \cdots + d_{v_{n-1},v_n} + d_{v_n,1},$$

and we want to minimize this.
The naïve approach is to consider all possible Hamiltonian cycles and choose the shortest one. However, this approach has a running time of $(n-1)! = \theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$ by Stirling's approximation, which is not feasible for large n.

4.7.2 Dynamic Programming Solution

Consider v_n, the last node before returning to $v_1 = 1$. If v_n is some node c, we find the optimal order of visiting nodes $2, 3, \ldots, n$ that ends at c. To do so, we need to keep track of the subset of nodes visited so far, and the last node visited.

Let $OPT[S, c]$ bethe minimum total travel distance when starting at 1, visiting each node in S exactly once, and ending at $c \in S$. We can find the best ending node c by computing

$$\min_{c \in S}\{OPT[S, c] + d_{c,1}\} \qquad \text{where } S = \{2, 3, \ldots, n\}.$$

The Bellman equation is

$$OPT[S, c] = \begin{cases} d_{1,c} & \text{if } S = \{c\} \\ \min\limits_{m \in S \setminus \{c\}} (OPT[S \setminus \{c\}, m] + d_{m,c}) & \text{if } |S| > 1 \end{cases}$$

which yields the optimal solution

$$\text{Final solution} = \min_{c \in \{2,\ldots,n\}} \{OPT[2, \ldots, n, c] + d_{c,1}\}.$$

Running Time

There are $\mathcal{O}(n \cdot 2^n)$ entries to evaluate, and each entry takes $\mathcal{O}(n)$ time to evaluate. Hence, the running time is $\mathcal{O}(2^n \cdot n^2)$.

4.7.3 Space Optimization

The space complexity of the dynamic programming solution is $\mathcal{O}(n \cdot 2^n)$, which is the same as implementing the solution naïvely. However, we can optimize by using a bottom up approach, as we do not need the entire table at any given time – computing the optimal solution with $|S| = k$ only requires storing the optimal solution with $|S| = k - 1$. By doing so, we can reduce the space complexity to approximately

$$\mathcal{O}\left(n \cdot \binom{n}{n/2}\right) \approx \mathcal{O}\left(\sqrt{n} \cdot 2^n\right).$$

4.8 Remarks

High-level steps in designing a DP algorithm

- Focus on a single decision in optimal solution. Typically, this is the first or the last decision.
- For each possible way of making that decision, [optimal substructure] write the optimal solution of the problem in terms of the optimal solutions to subproblems
- Generalize the problem by looking at the type of subproblems needed.

  For example, in the edit distance problem, we realize that we need to solve the problem for prefixes $(x_1, \ldots, x_i)$ and $(y_1, \ldots, y_j)$ for all (i, j)
- Write the Bellman equation, cover your base cases
- Think about optimizing the running time/space using tricks. This is often easier in the bottom-up implementation

CHAPTER

5 NETWORK FLOW

5.1 Network Flow Problem

- **Input:**
  - A direct graph $G = (V, E)$
  - A capacity function $c : E \to \mathbb{R}_{\geq 0}$
  - The source $s \in V$ and the sink $t \in V$
- **Output:** The maximum flow from s to t

In a network flow problem, we assume

- No edges enter s
- No edges leave t
- Edge capacity $c(e)$ is a non-negative integer

Definition 5.1.1 Flow

An $s-t$ flow in a network is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the following properties:

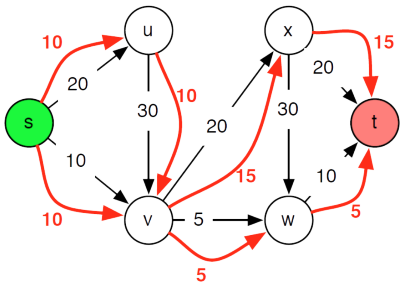
- **Capacity constraint:** For all $e \in E$,

$$0 \leq f(e) \leq c(e)$$

- **Flow conservation:** For all $v \in V \setminus \{s, t\}$,

$$\sum_{(v,u)\in E} f(v, u) = \sum_{(u,v)\in E} f(u, v)$$

Intuitively, $f(e)$ is the amount of flow that is sent through edge e.



Remark **Notation**

We define the function

$$f^{in}(v) = \sum_{(u,v)\in E} f(u,v)$$

to be the total flow into v, and

$$f^{out}(v) = \sum_{(v,u)\in E} f(v,u)$$

to be the total flow out of v.

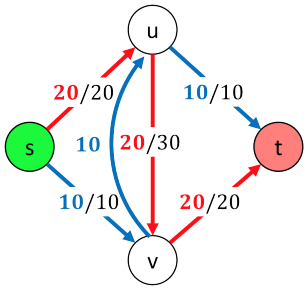
The value of the flow f is defined as

$$v(f) = f^{out}(s) = f^{in}(t)$$

Definition 5.1.2 **Network Flow Problem**

Given a network $G = (V, E)$, a capacity function $c : E \to \mathbb{R}_{\geq 0}$, and two vertices $s, t \in V$, the **network flow problem** is to find an $s - t$ flow f^* of maximum value.

We can try to solve this problem using a greedy algorithm, but it doesn't always work – once it increases the flow on an edge, it is not allowed to decrease it later. We need a way to "undo" the flow on an edge if it turns out to be a bad idea. To do so, we can send some flow **backwards** along the same path.



Definition 5.1.3 **Residual Graph**

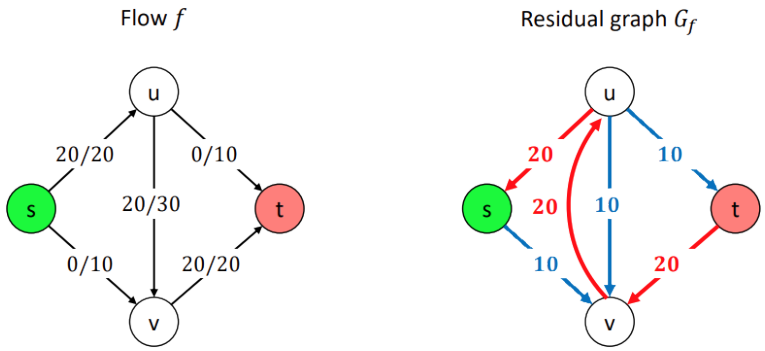
Given a flow f in a network $G = (V, E)$, the **residual graph** $G_f = (V, E_f)$ is a graph with the **same vertices** as G, and edges E_f defined as follows:

- **Forward edges:** $e = (u, v)$ with capacity $c(e) - f(e)$

  This is the amount of additional flow that can be sent along edge e.

- **Reverse edges:** $e^{rev} = (v, u)$ with capacity $f(e)$

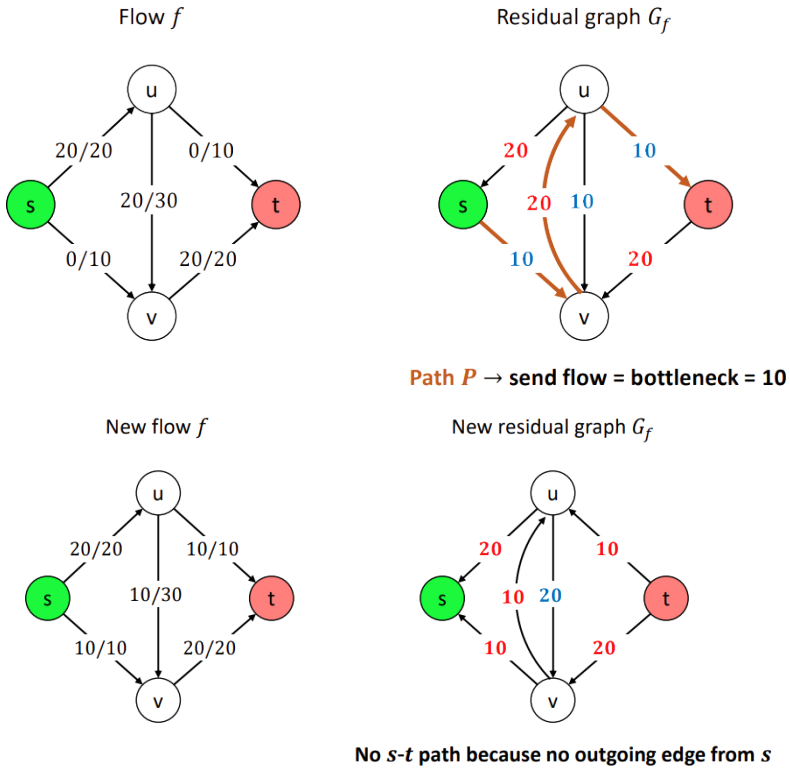
  This is the amount of flow that can be sent backwards along edge e.



Definition 5.1.4 Augmenting Path

Let P be an $s-t$ path in the residual graph G_f.
Let bottleneck(P, f) be the minimum capacity across all edges in P.
We **augment** flow d by sending bottleneck(P, f) units of flow along P.

- For each forward edge $e \in P$, increase the flow on e by x.
- For each reverse edge $e^{rev} \in P$, decrease the flow on e by x.



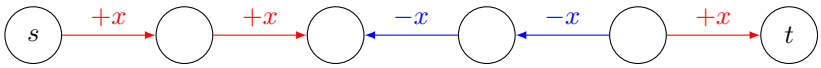
We argue that the new flow is a valid flow.

- **Capacity constraint:**
  - If we **increase** the flow on a forward edge, we can do so by *at most the capacity of forward edge* e in G_f, which is $c(e) - f(e)$.
    So, the new flow can be at most $f(e) + (c(e) - f(e)) = c(e)$.

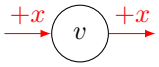
- If we **decrease** the flow on a reverse edge, we can do so by *at most the capacity of reverse edge* e^{rev} in G_f, which is $f(e)$.
  So, the new flow can be at most $f(e) - f(e) = 0$.

- **Flow conservation:**

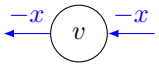
  Each node on the path (except s and t) has exactly two incident edges.



  - Both are forward / reverse edges. Then, one edge is incoming, and the other is outgoing. The flow is increased / decreased by the same amount.

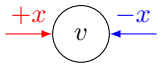


$f^{in}(v) \to +x$ and $f^{out}(v) \to +x$

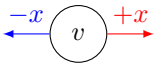


$f^{in}(v) \to -x$ and $f^{out}(v) \to -x$

  - One forward, one reverse edge. Then, both edges are incoming or outgoing. The flow is increased on one edge and decreased on the other by the same amount.



$f^{in}(v)$ and $f^{out}(v)$ are unchanged



$f^{in}(v)$ and $f^{out}(v)$ are unchanged

5.2 Max Flow-Min Cut

5.2.1 Ford-Fulkerson Algorithm

```
1: Function Max-Flow(G)
2:     // Initialize flow to 0:
3:     Set f(e) = 0 for all e ∈ E

4:     // While there is an s − t path in G_f:
5:     While p = Find-Path(s, t, Residual(G, f)) ≠ None do
6:         f ← Augment(f, p)
7:         Update-Residual(G, f)

8:     Return f
```

Running Time Analysis

- **Number of Augmentations**
  - At every step, flow and capacities remain integers.
  - For path P in G_f, bottleneck$(P, f) > 0$ implies bottleneck$(P, f) \geq 1$.
  - Each augmentation increases the flow by at least 1.
  - The maximum flow (hence the number of augmentations) is at most $C = \sum_{e \text{ leaving } s} c(e)$.

- **Preforming an Augmentation**
  - G_f has n vertices and at most $2m$ edges.
  - Finding the path P, computing bottleneck(P, f), and updating G_f all take linear time.

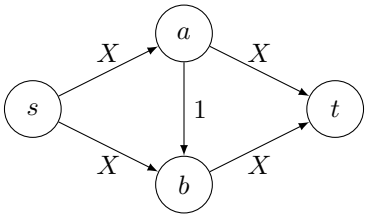
  Thus, the running time of the Ford-Fulkerson algorithm is

$$O((m+n) \cdot C).$$

Edmonds-Karp Algorithm

This algorithm runs in **pseudo-polynomial time** if we choose an *arbitrary* path in G_f at each step. The value of C can be exponentially larger in the input length (the number of bits requires to write down the edge capacities).

Example. In the graph below, we might end up repeatedly sending 1 unit of flow across $a \to b$ and then reversing it. This takes X steps, which can be exponential in the input length.



♢

Remark **Pesudo-polynomial, Weakly Polynomial, and Strongly Polynomial**

- **Pseudo-polynomial time:**
  The running time is polynomial in the unary representation of the input,

$$ops = poly(m, n, X).$$

- **Weakly Polynomial time:**
  The running time is polynomial in the binary representation of the input,

$$ops = poly(m, n, \log X).$$

- **Strong polynomial time:**
  The running time is polynomial in the input length,

$$ops = poly(m, n).$$

To avoid the exponential running time, we need to be more careful about the path we choose.

- Find the **maximum bottleneck capacity** augmenting path
  This makes the algorithm run in *weakly polynomial time*

$$\mathcal{O}(m^2 \cdot \log C)$$

- Find the **shortest augmenting path** using BFS
  This makes the algorithm run in *strongly polynomial time*

$$\mathcal{O}(m^2 \cdot n).$$

  This is known as the **Edmonds-Karp algorithm**.

```
1: Function Edmonds-Karp(G)
2:     // Initialize flow to 0:
3:     Set f(e) = 0 for all e ∈ E

4:     // Find shortest s − t path in G_f:
5:     While p = BFS(s, t, Residual(G, f)) ≠ None do
6:         f ← Augment(f, p)
7:         Update-Residual(G, f)

8:     Return f
```

Proof. Proof of Edmonds-Karp algorithm running time.

Let $d(v)$ be the distance from s to v in the residual graph G_f.

> **Lemma 1**
> During the execution of the algorithm, $d(v)$ does not decrease for any v.

Proof. (Lemma 1)

Suppose augmentation $f \to f'$ decreases $d(v)$ for some v.

Choose the v with the smallest $d(v)$ in $G_{f'}$.

Say $d(v) = k$ in $G_{f'}$, so $d(v) \geq k+1$ in G_f.

We look at node u just before v on a shortest path $s \to v \in G_{f'}$.

- $d(u) = k-1$ in $G_{f'}$
- $d(u)$ didn't decrease, so $d(u) \leq k-1$ in G_f.

$$\begin{array}{cccc} & d(u) & & d(v) \\ G_f & \leq k-1 & & \geq k+1 \\ & \downarrow & & \downarrow \\ G_{f'} & k-1 & & k \end{array}$$

Then, in G_f, (u, v) must be missing, as otherwise $d(v) \leq d(u) + 1 = k$ in G_f.

We must have added (u, v) by selecting (v, u) in augmenting path P.

However, P is a shortest path in $G_{f'}$, so it cannot have edge (v, u) with $d(v) > d(u)$. ∎

We call edge (u, v) **critical** in an augmentation step if

- It is part of the augmenting path P and its capacity is equal to bottleneck(P, f), or
- Augmentation step removes e and adds e^{rev} (if missing).

> **Lemma 2**
> Between any two steps in which (u, v) is critical, the distance $d(v)$ increases by at least 2.

Proof. (Lemma 2)

Suppose (u, v) was critical in G_f. The augmenting path must have removed it.

Let $k = d(u)$ in G_f. Since (u, v) is part of a shortest path, $d(v) = k + 1$ in G_f.

For (u, v) to be critical again, it must be added back at some point.

- Suppose $f' \to f''$ steps adds (u, v) back.
- Augmenting path in f' must have selected (v, u).
- In $G_{f'}$, $d(v) = k + 1 \geq (k + 1) + 1 = k + 2$ by Lemma 1 on v.

■

Each $d(u)$ can go from 0 to n by Lemma 1.
Then, each edge (u, v) can be critical at most $\frac{n}{2}$ times by Lemme 2.
There can be at most $m \cdot \frac{n}{2}$ augmentation steps, and each augmentation takes $O(m)$ time.
Thus, the running time of the Edmonds-Karp algorithm is

$$O(m^2 \cdot n).$$

■

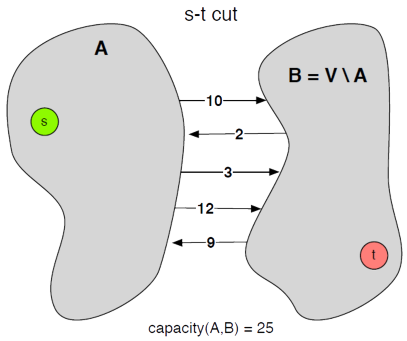
5.2.2 Max Flow-Min Cut Theorem

Definition 5.2.1 $s - t$ **Cut**

An $s - t$ **cut** is a partition of the vertices $V = S \cup T$ such that $s \in S$ and $t \in T$.

The capacity of an $s - t$ cut is the sum of the capacities of edges leaving S and entering T

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



Theorem 5.2.1

For any flow f and any $s - t$ cut (S, T),

$$v(f) = f^{out}(S) - f^{in}(S)$$

Proof. Let $v \in S \setminus \{s\}$. Then,

$$f^{in}(v) = f^{out}(v)$$

$$\sum_{v \in S \setminus \{s\}} \left(\sum_{e \text{ entering } v} f(e) \right) = \sum_{v \in S \setminus \{s\}} \left(\sum_{e \text{ leaving } v} f(e) \right)$$

After rearrangement, we get

$$v(f) = f^{out}(S) - f^{in}(S)$$

■

Theorem 5.2.2

For any flow f, and any $s-t$ cut (S,T),

$$v(f) \leq \text{cap}(S,T)$$

Proof.

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &\leq f^{out}(A) \\ &= \sum_{e \text{ leaving } A} f(e) \\ &\leq \sum_{e \text{ leaving } A} c(e) \\ &= \text{cap}(A,B) \end{aligned}$$

■

Hence,

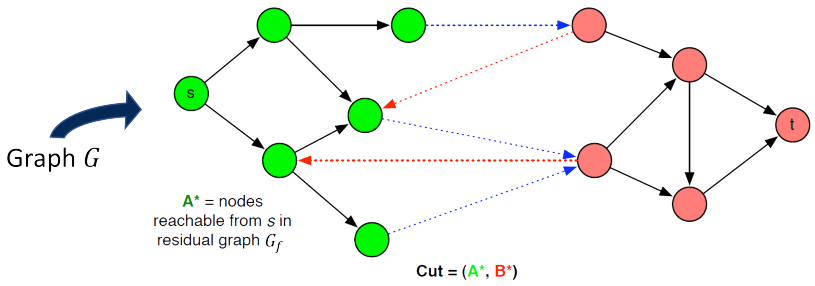
$$\max_f v(f) \leq \min_{(S,T)} \text{cap}(S,T),$$

the maximum flow is at most the minimum cut.

Theorem 5.2.3

Ford-Fulkerson algorithm finds a maximum flow f^*.

Proof. WTS that the flow f found by the Ford-Fulkerson algorithm is a maximum flow.
Let f be the flow found by the Ford-Fulkerson algorithm.
Let G_f be the residual graph after the algorithm terminates.
Let A^* be the nodes reachable from s in G_f, and let $B^* = V \setminus A^*$.



Claim: (A^*, B^*) is an $s-t$ cut.
Indeed, $s \in A^*$ be definition. $t \in B^*$ because when Ford-Fulkerson terminates, there are no $s-t$ paths in G_f, so $s \notin A^*$.

- Let blue edges be the edges going out of A^* **in** G, and

  Each blue edge (u,v) must be saturated.

  Otherwise, G_f would have its forward edge (u,v) and then $v \in A^*$.

  Then, $f^{out}(A^*) = \text{cap}(A^*, B^*)$.

- Let red edges be the edges going out of A^* **in** G_f.

  Each red edge (v, u) must have zero flow.

  Otherwise, G_f would have its reverse edge (u, v) and then $v \in A^*$.

  Then, $f^{in}(A^*) = 0$.

Thus,

$$v(f) = f^{out}(A^*) - f^{in}(A^*) = \text{cap}(A^*, B^*).$$

■

Theorem 5.2.4 Max Flow-Min Cut Theorem

In any flow network, the value of the maximum flow is equal to the capacity of the minimum cut.

Proof. Run Ford-Fulkerson to find a max flow f.
Construct its residual graph G_f.
Le4t A^* be the set of vertices reachable from s in G_f.
Then, $(A^*, V \setminus A^*)$ is a min $s - t$ cut. ■

5.3 Applications of Network Flow

5.3.1 Bipartite Matching

Definition 5.3.1 Bipartite Graph

A graph $G = (V, E)$ is **bipartite** if its vertex set V can be partitioned into two sets U and V such that every edge in E has one endpoint in U and the other in V.

A bipartite matching is when given a bipartite graph $G = (U \cup V, E)$, we want to find a maximum cardinality matching.

CHAPTER

6 LINEAR PROGRAMMING

6.1 Introduction

6.1.1 Linear Programming

Linear programming is a method to achieve the best outcome in a mathematical model whose requirements are represented by linear relationships. It is a special case of mathematical programming (mathematical optimization).

Example (Brewery). A brewery can invest its inventory of corn, hops and malt into producing some amount of ale and some amount of beer. Per unit resource requirement and profit of the two items are as given below.

Beverage	Corn (pounds)	Hops (ounces)	Malt (pounds)	Profit ($)
Ale (barrel)	5	4	35	13
Beer (barrel)	15	4	20	23
constraint	480	160	1190	

Suppose it produces A units of ale and B units of beer. Then, we want to solve this program:

$$\begin{array}{lccccccc} & \text{Ale} & & \text{Beer} & & & & \\ \max & 13A & + & 23B & & & & \text{Profit} \\ \text{s.t.} & 5A & + & 15B & \leq & 480 & & \text{Corn} \\ & 4A & + & 4B & \leq & 160 & & \text{Hops} \\ & 35A & + & 20B & \leq & 1190 & & \text{Malt} \\ & A & , & B & \geq & 0 & & \end{array}$$

♢

Definition 6.1.1 Linear Function

$f : \mathbb{R}^n \to \mathbb{R}$ is a **linear function** if $f(x) = a^T x$ for some $a \in \mathbb{R}^n$.

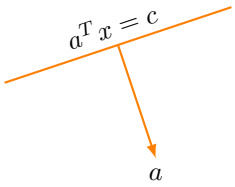
Example. For example,

$$f(x_1, x_2) = 3x_1 - 5x_2 = \begin{pmatrix} 3 & -5 \end{pmatrix}^T \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

We can see that a is the vector of coefficients of the linear function. ♢

- **Linear objective**: f
- **Linear constraints**:
  - $g(x) = c$, where $g : \mathbb{R}^n \to \mathbb{R}$ is a linear function and $c \in \mathbb{R}$
  - Line in the plane (or a hyperplane in higher dimensions $\mathbb{R}^n$)

Geometrically, a is the normal vector of the line(or hyperplane) represented by $a^T x = c$.



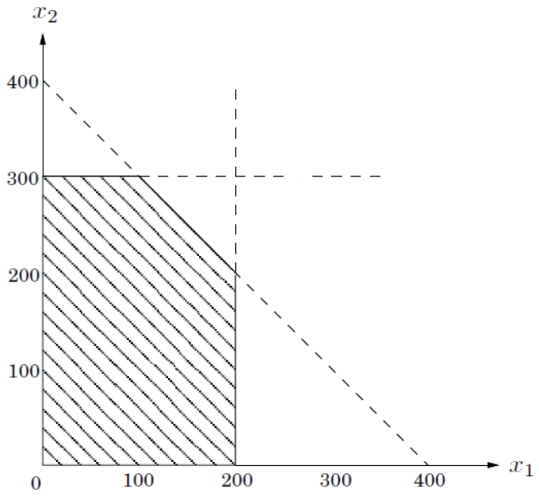
and $a^T x \leq c$ is the **half-space** defined by the line.

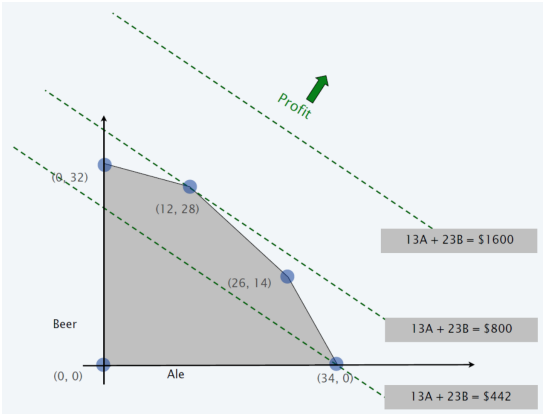
6.1.2 Finding the Optimal Solution

Example. Suppose we want to solve the following linear program:

$$\begin{array}{lrcrcl} \max & x_1 & + & 6x_2 & & \\ \text{s.t.} & x_1 & \leq & 200 & & \\ & x_2 & \leq & 300 & & \\ & x_1 & + & x_2 & \leq & 400 \\ & x_1 & , & x_2 & \geq & 0 \end{array}$$

This is equivalent to finding the **feasible region**, where ant point in the region satisfies all the constraints. The feasible region is the intersection of the half-spaces defined by the constraints.





♢

To find a maximum solution, we push the objective function as far as possible.

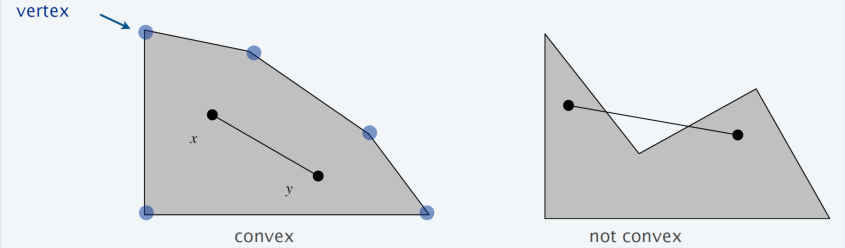
Claim. Regardless of the objective function, there must be a vertex that is an optimal solution

Remark **Convexity**

A **convex set** is a set S cush that

$$\forall x, y \in S, \lambda \in [0,1] \implies \lambda x + (1-\lambda)y \in S$$

A **vertex of a convex set** is a point which cannot be written as a strict convex combination of any two points in the set.



We observe that a feasible region of a linear program is a convex set.

Intuitive Proof. We start at some point x in the feasible region.
If x is not a vertex,

- Find d direction d such that points within a positive distance of ε from x in both d and $-d$ directions are within the feasible region
- Objective must *not decrease* in at least one of the two directions
- Follow that direction until you reach a new point x for which at least one more constraint is "tight"

Repeat until we are at a vertex. ■

6.2 Formatting Linear Programs

6.2.1 Standard Form

- **Input**: $c, a_1, a_2, \ldots, a_m \in \mathbb{R}^n, b \in \mathbb{R}^m$.
- **Goal**:

$$\begin{array}{lrcl} \text{Maximize} & c^T x & & \\ \text{Subject to} & {a_1}^T x & \leq & b_1 \\ & {a_2}^T x & \leq & b_2 \\ & & \vdots & \\ & {a_m}^T x & \leq & b_m \\ & x & \geq & 0 \end{array}$$

We can also combine all of them into a single **matrix equation**:

$$\begin{array}{lrcl} \text{Maximize} & c^T x & & \\ \text{Subject to} & Ax & \leq & b \\ & x & \geq & 0 \end{array}$$

6.2.2 Converting to Standard Form

- Constraints that uses $\geq$:

$$a^T x \geq b \iff -a^T x \leq -b$$

- Constraints that uses $=$:
$$a^T x = b \iff a^T x \leq b, -a^T x \leq -b$$
- Constraints that is a minimization:
$$\text{Minimize } c^T x \iff \text{Maximize } -c^T x$$
- Variable is unconstrained:
$$x_i \text{ is unconstrained} \iff x_i = x_i' - x_i'' \text{ where } x_i' \geq 0, x_i'' \geq 0$$

Example. Consider the following example.

$$\begin{array}{lrcrcl} \text{minimize} & -2x_1 & + & 3x_2 & & \\ \text{subject to} & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & x_1 & & & \geq & 0 \end{array} \implies \begin{array}{lrcrcl} \text{maximize} & 2x_1 & - & 3x_2 & & \\ \text{subject to} & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & x_1 & & & \geq & 0 \end{array}$$

$$\implies \begin{array}{lrcrcrcl} \text{minimize} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\ \text{subject to} & x_1 & + & x_2' & - & x_2'' & = & 7 \\ & x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\ & x_1 & , & x_2' & , & x_2'' & \geq & 0 \end{array}$$

Part II

Appendices

BIBLIOGRAPHY

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th edition. Cambridge, MA: The MIT Press, 2022, ISBN: 978-0-262-04630-5.

[2] J. Erickson, *Algorithms*, 1st edition. self-published online, 2019, ISBN: 978-1-792-64483-2. [Online]. Available: https://jeffe.cs.illinois.edu/teaching/algorithms/.

[3] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, volume 40, number 9, pages 1098–1101, 1952. DOI: 10.1109/JRPROC.1952.273898.

[4] N. Shah. "Csc373: Algorithm design, analysis & complexity." (2021), [Online]. Available: https://www.cs.toronto.edu/~nisarg/teaching/373f21/.

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

[6] N. Wiebe and H. Sha. "Csc373h1s 20241." (2024), [Online]. Available: https://q.utoronto.ca/courses/337418.