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.

```
procedure BUBBLESORT($A$)
    for $i = 1$ to $n - 1$ do
        for $j = 1$ to $n - i$ do
            if $A[j] > A[j+1]$ then
                SWAP($A[j], A[j+1]$)
            end if
        end for
    end for
end procedure
```

Proof. Proof by induction on the number of iterations of Bubblesort.

- **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.

■

◇

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
- **Conquer** the subproblems recursively
- **Combine** the solutions to the subproblems into a solution to the original problem

Example (Merge Sort)**.** MergeSort 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.

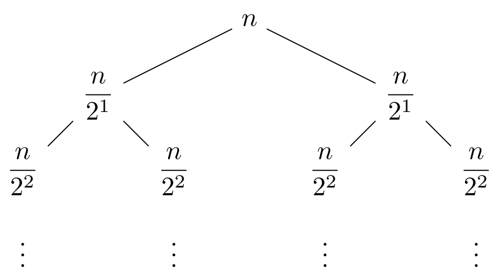
Algorithm 1 Merge Sort

```
function MergeSort($A$)
    if $|A| \leq 2$ then
        return BruteForceSort($A$)
    end if
    $m \leftarrow \lfloor |A|/2 \rfloor$
    $L \leftarrow$ MergeSort($A[1 \ldots m]$)
    $R \leftarrow$ MergeSort($A[m+1 \ldots |A|]$)
    return Merge($L, R$)
end function
```

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

Algorithm 2 Merge

```
function Merge($L, R$)
    $i \leftarrow 1$
    $j \leftarrow 1$
    $A \leftarrow \emptyset$
    while $i \leq |L|$ and $j \leq |R|$ do
        if $L[i] \leq R[j]$ then
            $A \leftarrow A \cup \{L[i]\}$
            $i \leftarrow i + 1$
        else
            $A \leftarrow A \cup \{R[j]\}$
            $j \leftarrow j + 1$
        end if
    end while
    return $A \cup L[i \ldots] \cup R[j \ldots]$
end function
```



To compute the cost of MergeSort, 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 MergeSort 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. MergeSort is correct.

Proof. Proof by contradiction.
Assume that MergeSort is not correct.
TODO: prove on a case by case basis ■

Proof. Proof by induction on the number of iterations of MergeSort.

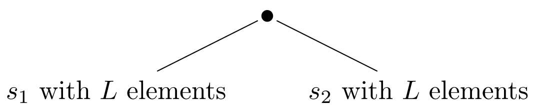
- **Base case**: $s(2)$

  BruteForceSort is correct by construction.

- **Induction Steps**

  Assume MergeSort 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 MergeSort 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 MergeSort is correct for L elements.

  Thus, MergeSort 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**:
  - Count the number of inversions where $i \in x$ and $j \in y$
  - Add the three counts together

Algorithm 3 Sort and Count

```
function SortAndCount($A$)
    if $|A| \leq 1$ then
        return $(A, 0)$
    end if
    $m \leftarrow \lfloor |A|/2 \rfloor$
    $(L, r_L) \leftarrow$ SortAndCount($A[1 \ldots m]$)
    $(R, r_R) \leftarrow$ SortAndCount($A[m+1 \ldots |A|]$)
    $(A', r_{LR}) \leftarrow$ MergeAndCount($L, R$)
    return $(A', r_L + r_R + r_{LR})$
end function
```

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

Algorithm 4 Merge and Count

```
function MergeAndCount($(L, R)$)
    $i \leftarrow 1$
    $j \leftarrow 1$
    $A \leftarrow \emptyset$
    $r_{LR} \leftarrow 0$
    while $i \leq |L|$ and $j \leq |R|$ do
        if $L[i] \leq R[j]$ then
            $A \leftarrow A \cup \{L[i]\}$
            $i \leftarrow i + 1$
        else
            $A \leftarrow A \cup \{R[j]\}$
            $j \leftarrow j + 1$
            $r_{LR} \leftarrow r_{LR} + |L| - i + 1$
        end if
    end while
    return $(A \cup L[i \dots] \cup R[j \dots], r_{LR})$
end function
```

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.

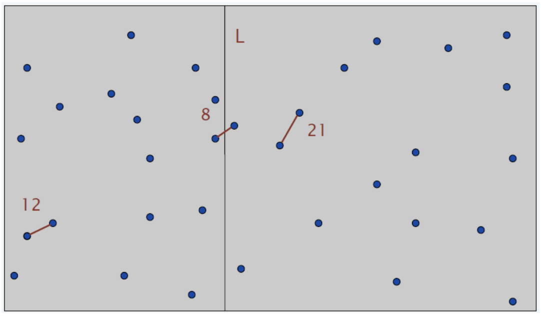
- **Applications**

  - 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.

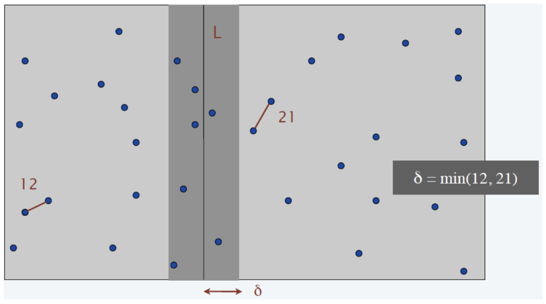
- **Divide**: Split the points into two equal halves by drawing a vertical line L through the median x-coordinate



- **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

Remark

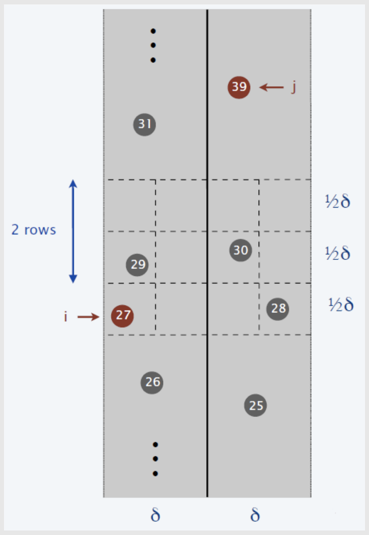
We chose the number 11 on purpose.

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.

∎



- Return the best of 3 solutions

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

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

$$xy = (x_1 y_1) \cdot 10^n + (x_1 y_2 + x_2 y_1) \cdot 10^{\frac{n}{2}} + x_2 y_2$$

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

$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$$

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

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

- Natively, this requires $2^3 = 8$ matrix multiplications of size $\frac{n}{2} \times \frac{n}{2}$.

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.

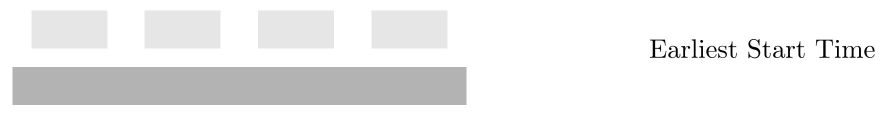
3.2 Interval Scheduling

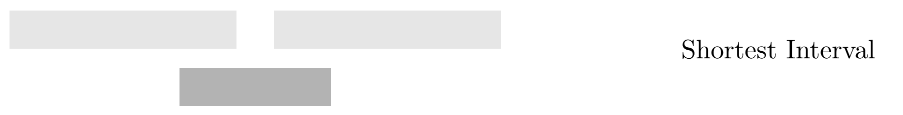
3.2.1 Problem Definition

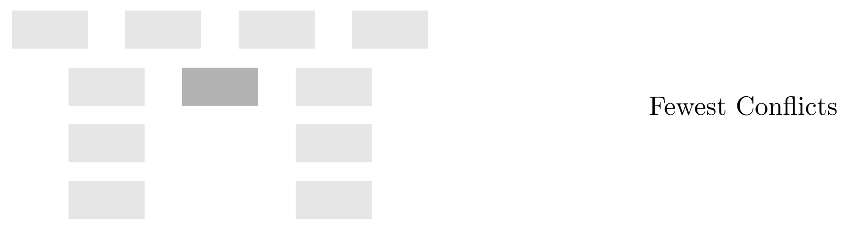
- **Problem**
    - Job j starts at time s_j and finishes at time f_j.

  - 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.

- **Applications**
  - Scheduling jobs on a single machine.
  - Scheduling classes in a classroom.
  - Scheduling packets on a link.







3.2.2 Greedy Algorithm

We can implement greedy with earliest finish time (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} are compatible with $i_1, i_2, \ldots, i_r = j_1, j_2, \ldots, j_r$.
- 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, as greedy is optimal.

By Induction

TODO: SEE SLIDES

Part II

Appendices

BIBLIOGRAPHY

INDEX