# CSC373 Algorithms

Jonah Chen

## 1 Divide and Conquer

- Divide and Conquer algorithm:
  1. Divide problem of size $n$ into $a$ smaller subproblems of size $n/b$ each
  2. Recursively solve each subproblem
  3. Combine the subproblem solutions into the solution of the original problem
- Runtime: $T(1) = c$ and $T(n) = aT(n/b) + cn^d$ for $n > 1$
- Master Theorem: $T(n)$ depends on relation between $a$ and $b^d$.

$$\begin{cases} a < b^d : T(n) = \Theta(n^d) \\ a = b^d : T(n) = \Theta(n^d \log n) \\ a > b^d : T(n) = \Theta(n^{\log_b a}) \end{cases} \tag{1}$$

  - Note that the running time does not depend on the constant $c$
  - In many algorithms $d = 1$ (combining take linear time)
- Examples:
  - Merge sort — sorting array of size $n$ ($a = 2$, $b = 2$, $d = 1 \to a = b^d$) so $T(n) = \Theta(n \log n)$
  - Binary search — searching sorted array of size $n$ ($a = 1$, $b = 2$, $d = 0 \to a = b^d$) so $T(n) = \Theta(\log n)$

### 1.1 Karatsuba Multiplication

- **Add** two binary $n$-bit numbers naively takes $\Theta(n)$ time
- **Multiply** two binary $n$-bit numbers naively takes $\Theta(n^2)$ time
- Divide and Conquer approaches
  1. Multiply $x$ and $y$. We can divide them into two parts

$$x = x_1 \cdot 2^{n/2} + x_0 \tag{2}$$
$$y = y_1 \cdot 2^{n/2} + y_0 \tag{3}$$
$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{4}$$

     - $T(n) = 4T(n/2) + cn; T(1) = c$
     - $a = 4, b = 2, d = 1$ Master Theorem case 3, $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.
     - This is the same complexity of the naive approach, making this approach useless.
  2. Reconsider (4), we may rewrite $(x_1 \cdot y_0 + x_0 \cdot y_1)$ as $(x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0$

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + ((x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{5}$$

- $T(n) = 3T(n/2) + cn; T(1) = c$
- $a = 3, b = 2, d = 1$, Master Theorem case 3, $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
- Minor issue: a carry may increase $x_1 + x_0$ and $y_1 + y_0$ to $\frac{n}{2} + 1$. We can easily prove this by isolating the carry bit and reevaluating the complexity.

- To deal with integers which doesn't have a power of 2 number of bits, we can pad the numbers with 0s to make them have a power of 2 number of bits. This may at most increase the complexity by 3x.
- 1971: $\Theta(n \cdot \log n \cdot \log \log n)$
- 2019: Harvey and van der Hoeven $\Theta(n \log n)$. We do not know if this is optimal.

## 1.2 Strassen's MatMul Algorithm

- Let $A$ and $B$ be two $n \times n$ matrices (for simplicity $n$ is a power of 2), we want to compute $C = AB$.
- The naive approach takes $\Theta(n^3)$ time.
  1. Divide $A$ and $B$ into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}. \tag{6}$$

     Then, $C$ can be calculated with

$$C_1 = A_1B_1 + A_2B_3 \tag{7}$$
$$C_2 = A_1B_2 + A_2B_4 \tag{8}$$
$$C_3 = A_3B_1 + A_4B_3 \tag{9}$$
$$C_4 = A_3B_2 + A_4B_4 \tag{10}$$

     - $T(n) = 8T(n/2) + cn^2; T(1) = c$
     - $a = 8, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$
  2. **Idea:** Compute $C_1, C_2, C_3, C_4$ with only 7 multiplications, not 8.

$$M_1 = (A_2 - A_4)(B_3 + B_4) \tag{11}$$
$$M_2 = (A_1 + A_4)(B_1 + B_4) \tag{12}$$
$$M_3 = (A_1 - A_3)(B_1 + B_2) \tag{13}$$
$$M_4 = (A_1 + A_2)B_4 \tag{14}$$
$$M_5 = A_1(B_2 - B_4) \tag{15}$$
$$M_6 = A_4(B_3 - B_1) \tag{16}$$
$$M_7 = (A_3 + A_4)B_1 \tag{17}$$

     With these we can compute $C_1, C_2, C_3, C_4$ with only additions of the $M$ matrices.

$$C_1 = M_1 + M_2 - M_4 + M_6 \tag{18}$$
$$C_2 = M_4 + M_5 \tag{19}$$
$$C_3 = M_6 + M_7 \tag{20}$$
$$C_4 = M_2 - M_3 + M_5 + M_7 \tag{21}$$

     - $T(n) = 7T(n/2) + cn^2; T(1) = c$
     - $a = 7, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

- If $n$ is not a power of 2, we zero-pad the matrices to have $n$ as a power of two. This may increase the complexity by at most a factor of 7.

## 1.3 Median of Unsorted Arrays

- For an unsorted array $A$, we can find the average, max, min, sum, etc. in linear time.
- The trivial algorithm is to sort $A$ then get the median. That takes $O(n \log n)$ time.
- We will solve a more general problem: Find the $k^{th}$ smallest element in $A$. (e.g. $A = 5, 2, 6, 7, 4$, $\text{Select}(A, 1) = 2, \text{Select}(A, 4) = 6$)
- if $|A| = 1$, then return $A[1]$. Otherwise find a splitter $s$ in arbitrary element of $A$. Partition $A$ into $A^+$ and $A^-$, then divide
- $T(n) = T(\max(|A^-|, |A^+|)) + cn = T(\max(i-1, n-i)) + cn$.
- Worst case: $T(n) = T(n-1) + cn = \Theta(n^2)$
- Best case: $T(n) = T(n/2) + cn = \Theta(n)$. Suppose $b > 1$, by the Master Theorem $T(n) = T(n/b) + cn = \Theta(n)$.

We define $s$ is a good splitter if $s$ is greater than $1/4$ of the elements of $A$ and less than $1/4$ of the elements of $A$. We can make the following observation:

1. With this splitter, we will reduce the size to at most $3n/4$.
2. Half the elements are good splitters.

We should select splitter $s$ uniformly at random.

- $P(\text{splitter is good}) = \frac{1}{2}$
- $P(\text{splitter is bad}) = \frac{1}{2}$
- We can show that the expected number of trials (splitter selections) until obtaining a good splitter is 2.

### 1.3.1 Expected Runtime

$$\underbrace{n_0 \to n_1 \to n_2}_{\text{Phase 0, size} \le n} \to \underbrace{n_3 \to n_4}_{\text{Phase 1, size} \le \frac{3}{4}n} \to \underbrace{n_5 \to n_6}_{\text{Phase 2, size} \le \left(\frac{3}{4}\right)^2 n} \to \dots \tag{22}$$

- Phase $j$: input size $\le (\frac{3}{4})^j n$
- Random variable $y_j = \#$ of recursive calls in phase $j$. Note that $E[y_j] = 2$.
- Random variable $x_j = \#$ of steps to all the recursive calls in phase $j$.
- Total number of steps is $x = x_0 + x_1 + x_2 + \dots$.
- We can compute $E[x] = E[x_0] + E[x_1] + E[x_2] + \dots$.

$$x_j \le c y_j \frac{3}{4}^j n \tag{23}$$

$$E[x_j] \le c E[y_j] \frac{3}{4}^j n \le 2c \frac{3}{4}^j n \tag{24}$$

$$E[x] = \sum_j E[x_j] \le \sum_{j=1}^{\infty} 2c \frac{3}{4}^j n = \frac{2c}{1 - \frac{3}{4}} n = 8cn = \Theta(n) \tag{25}$$

### 1.3.2 Deterministic Algorithm

- If $|A| \leq 5$ then we sort $A$ and return the $k^{th}$ smallest.
- Otherwise, partition $A$ into $n/5$ groups of size $5$ each, then find the median of each group (constant time) and store in list $M$. This takes linear time.
- Select the median of $M$ with the Select algorithm, this is a good splitter.
- the worst case running time is $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\lfloor \frac{3n}{4} \rfloor) + cn$.
- This recursive relation cannot be solved by the Master Theorem. We can prove using induction that $T(n) < 20cn$.

**Question: Why groups of 5?**

- With groups of 5, the total size of subproblems: $\frac{n}{5} + \frac{3n}{4} = \frac{19n}{20} < n$
- With groups of 3, the total size of subproblems: $\frac{n}{3} + \frac{3n}{4} = \frac{13n}{12} > n$, not sufficient.
- So group size of $5, 7, 9, 11, \ldots$ would also work.

# 2 Closest Pair of Points

- Problem: Given a set of $n$ points, find the pair of points that are the closest in $O(n \log n)$.

## 2.1 Closest Pair in 2D

- Divide: points roughly in half by drawing vertical line on midpoint
- Conquer: Find closest pair on each half, recursively.
- Combine: Find the closest pair $(p, q)$, $p \in L$, $q \in R$. However, there may be $\Theta(n^2)$ pairs.

- Claim: Let $p = (x_p, y_p) \in B_L, q = (x_q, y_q) \in B_R$ with $y_p \leq y_q$. If $d(p, q) < \delta$ then there are at most **six** other points (x,y) in $B$ such that $y_p \leq y \leq y_q$.
- Proof:
- $S_L = \{p' = (x, y) : p' \neq p \in B_L \wedge y_p \leq y \leq y_q\}$ (other points on the left of the middle)
- $S_R = \{p' = (x, y) : p' \neq q \in B_R \wedge y_p \leq y \leq y_q\}$ (other points on the right of the middle)
- Assume by contradiction that $|S_L \cup S_R| \geq 7$. WLOG assume $|S_L| \geq 4$.
- In a $\delta \times \delta$ square there are at least $4 + 1 = 5$ points. Divide the square into 4 smaller squares, by Pigeonhole Principle, there is a square with at least 2 points, whose distance is at most $\delta/\sqrt{2}$. This contradicts the assumption that the closest pair on the left is at most $\delta$.

- Then, we can sort everything in the $y$ axis, and check the next seven points by the $y$ coordinate for the minimum distance. This takes linear time.
- We only need to modify the combine step in the algorithm so it's $\Theta(n)$ runtime.

- So $T(n) = 2T(\frac{n}{2}) + cn$, which is $O(n \log n)$.

**Algorithm 1** Closest Pair in 2D

```
1: procedure ClosestPair(P)
2:     P_x := the list of points in P sorted by x-coordinate
3:     P_y := the list of points in P sorted by y-coordinate
4: procedure RCP(P_x, P_y)
5:     if |P_x| ≤ 3 then return brute force(P_x)
6:     L_x := the first half of P_x; R_x := the second half of P_x
7:     m := (max x-coordinate of L_x + min x-coordinate of R_x)/2
8:     L_y := sublist of P_y with points in L_x
9:     R_y := sublist of P_y with points in R_x
10:    (p_L, q_L) := RCP(L_x, L_y); (p_R, q_R) := RCP(R_x, R_y)
11:    δ := min{d(p_L, q_L), d(p_R, q_R)}
12:    if δ = d(p_L, q_L) then
13:        p := p_L; q := q_L
14:    else
15:        p := p_R; q := q_R
16:    B := sublist of P_y with points in [m − δ, m + δ]
17:    for each p in B do
18:        for each next seven q after p in B do
19:            if d(p, q) < d(p*, q*) then (p*, q*) := (p, q)
```

# 3 Greedy Algorithms

- There is an optimization problem: given an input, find a solution that minimize/maximize an objective function $f$ under some constraint.
- Build the solution incrementally in stages
- At each stage, extend the solution greedily and irrevocably.
- For some problems this gives optimal solutions (i.e. MST), but for other problems it does not.
- The order of the stages is very important.

## 3.1 Interval Scheduling

- Input: $n$ intervals, with interval $j$ starts at time $s_j$ and finishes at time $f_j$.
- Output: maximum-size set of intervals that do not overlap
- Naive algorithm is to try each subset of $n$ intervals by brute force $O(2^n)$, way too slow.
- Greedy algorithm sorts interval in some order, then if it doesn't overlap then add it to the solution.
- **What order gives the biggest feasible set?**
  1. Increasing start time: $s_j$
  2. Increasing finish time: $e_j$
  3. Shortest interval: $e_j - s_j$
  4. Fewest conflicts
- The question is which one is optimal? Option 1 has a clear counterexample with one very long interval that overlaps each other interval. Option 3 is also not optimal using a short interval between two long interval. Option 4 is not optimal (not very clear counterexample).
- Option 2 is optimal. The intuition is that chosing these intervals first will leave the most time for the rest of the intervals.
- To find if the interval is compatible, we just need to check if the start time of the new interval is greater than the finish time of the latest scheduled interval.

### 3.1.1 Proof of optimality

- Suppose for contradiction that this greedy algorithm is not optimal
- Say greedy selects interval $i_1, \ldots, i_k$ sorted by increasing finish time.
- Suppose the optimal schedule $j_1, \ldots, j_m$ has $m > k$ intervals, and sort by increasing finish time. Consider an optimal schedule that can match the greedy $j_1 = i_1, \ldots, j_r = i_r$ for the greatest possible $r$.
- By the nature of the greedy algorithm, then $i_{r+1}$ finishes the earliest amongst the compatible intervals remaining. So consider the schedule $S : i_1, \ldots, i_r, \mathbf{i_{r+1}}, j_{r+2}, \ldots, j_m$.
- This is also optimal (contains $m$ intervals) but matches the greedy algorithm by one more position. This is a contradiction.

## 3.2 Interval Partitioning

- Motivation: given a set of lecture time intervals, schedule them into as few classrooms as possible.
- Input: $n$ intervals, interval $j$ starts at $s_j$ and finishes at $f_j$.
- Output: group interval into fewest partitions such that intervals in each partition are compatible.
- An idea is to find the maximum set of compatible intervals using the previous algorithm. This doesn't work.
- We can try the same orders as the previous problem, but only earliest start time is optimal in this case.
- To implement it efficiently, we will use a heap with increase-key operation.

### 3.2.1 Proof of optimality

- We define the **depth** at time $t$ as the number of intervals that contain time $t$. The **maximum depth** $d_{max}$ is the maximum depth over all times.
- Clearly, the number of partitions needed is at least $d_{max}$. We will show that this greedy algorithm create only $d_{max}$ partitions.
- Let $d$ be the number of partition the greedy algorithm opened.
- Partition $d$ was created because there was an interval $j$ that overlaps with some previously scheduled interval in each of the $d-1$ other partitions.
- This means that for $d-1$ intervals, their start times are all before $s_j$ and their finish times are all after $f_j$ (otherwise $j$ must be compatible). Hence, the depth at $s_j$ is exactly $d$.
- Thus, $d_{max} \geq d$ so the greedy algorithm is optimal.
- Warning: this proof rely on the fact that the start time