Divide and Conquer

Textbook reading

Chapter 2
Overview

*Design principle:*  
- Divide and conquer

*Proof technique:*  
- Induction, induction, induction

*Analysis technique:*  
- Recurrence relations

*Problems:*  
- Merge sort  
- Selection  
- Counting inversions  
- Integer multiplication
Merge Sort

\[\text{Merge-Sort}(A, p, r)\]

1. if \( p < r \)
2. then \( q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor \)
3. \text{Merge-Sort}(A, p, q)
4. \text{Merge-Sort}(A, q + 1, r)
5. \text{Merge}(A, p, q, r)\]
Merge Sort

**Merge-Sort**\( (A, p, r) \)
1. if \( p < r \)
2. then \( q \leftarrow \left\lfloor \frac{p+r}{2} \right\rfloor \)
3. Merge-Sort\( (A, p, q) \)
4. Merge-Sort\( (A, q + 1, r) \)
5. Merge\( (A, p, q, r) \)

**Merge**\( (A, p, q, r) \)
1. \( n_1 \leftarrow q - p + 1 \)
2. \( n_2 \leftarrow r - q \)
3. for \( i \leftarrow 1 \) to \( n_1 \)
4. do \( L[i] \leftarrow A[p + i - 1] \)
5. for \( i \leftarrow 1 \) to \( n_2 \)
6. do \( R[i] \leftarrow A[q + i] \)
7. \( L[n_1 + 1] \leftarrow \infty \)
8. \( R[n_2 + 1] \leftarrow \infty \)
9. \( i \leftarrow 1 \)
10. \( j \leftarrow 1 \)
11. for \( k \leftarrow p \) to \( r \)
12. do if \( L[i] < R[j] \)
13. then \( A[k] \leftarrow L[i] \)
14. \( i \leftarrow i + 1 \)
15. else \( A[k] \leftarrow R[j] \)
16. \( j \leftarrow j + 1 \)
Merge Sort: The Micro-View

87 4 17 11 9 13 7 5

87 4 17 11 9 13 7 5

87 4 17 11 9 13 7 5

87 4 17 11 9 13 7 5

4 87 11 17 9 13 5 7

4 87 11 17 9 13 5 7

4 87 11 17 9 13 5 7

4 5 7 9 11 13 17 87

Divide

Divide

Divide

Do nothing

Merge

Merge

Merge
Merge Sort: The Macro-View

Divide

Recurse

Merge

CSCI 3110 • Divide and Conquer • 5/39
The Divide-and-Conquer Paradigm

**Divide** the input instance into one or more smaller instances.

**Recursively** solve these smaller input instances.

**Combine** the solutions produced by the recursive calls into a solution to the original instance.
The Divide-and-Conquer Paradigm

**Divide** the input instance into one or more smaller instances.

**Recursively** solve these smaller input instances.

**Combine** the solutions produced by the recursive calls into a solution to the original instance.

In most algorithms, either the divide or the combine step is trivial:

- The divide step in Merge Sort is trivial
- The combine step in Quick Sort is trivial
Lemma: *Merge Sort correctly sorts any input array.*
Lemma: *Merge Sort correctly sorts any input array.*

*Proof by induction:*

Base case: \((n = 1)\)
Lemma: Merge Sort correctly sorts any input array.

Proof by induction:

Base case: \( n = 1 \)

- A one-element array is already sorted.
Lemma: *Merge Sort correctly sorts any input array.*

**Proof by induction:**

Base case: \((n = 1)\)

- A one-element array is already sorted.

Inductive step: \((n > 1)\)
Lemma: *Merge Sort correctly sorts any input array.*

**Proof by induction:**

**Base case:** \((n = 1)\)

- A one-element array is already sorted.

**Inductive step:** \((n > 1)\)

- The left and right half each have size less than \(n\).
- By the inductive hypothesis, the recursive calls sort them correctly.
- Merge correctly merges two sorted sequences.
Correctness of D&C Algorithms

*Divide-and-conquer algorithms are the algorithmic incarnation of induction:*

**Base case:** Whenever we don’t recurse, but produce the answer directly (often trivially).

**Inductive step:** Reduce the solution of a given instance to the solution of smaller instances, by recursing.
Correctness of D&C Algorithms

*Divide-and-conquer algorithms are the algorithmic incarnation of induction:*

**Base case:** Whenever we don’t recurse, but produce the answer directly (often trivially).

**Inductive step:** Reduce the solution of a given instance to the solution of smaller instances, by recursing.

*Induction is the natural proof method for divide-and-conquer algorithms.*
A *recurrence relation* defines the value of a function $f$ in terms of its values for smaller arguments.
A recurrence relation defines the value of a function $f$ in terms of its values for smaller arguments.

**Examples:**

- Fibonacci numbers:

  $$f(n) = \begin{cases} 
  1 & \text{if } n \leq 2 \\
  f(n - 1) + f(n - 2) & \text{if } n > 2 
  \end{cases}$$
Recurrence Relations

A recurrence relation defines the value of a function $f$ in terms of its values for smaller arguments.

**Examples:**

- **Fibonacci numbers:**

  $$f(n) = \begin{cases} 
  1 & \text{if } n \leq 2 \\
  f(n - 1) + f(n - 2) & \text{if } n > 2 
  \end{cases}$$

- **Binomial coefficients:**

  $$B(n, k) = \begin{cases} 
  1 & \text{if } k = 1 \text{ or } k = n \\
  B(n - 1, k - 1) + B(n - 1, k) & \text{otherwise} 
  \end{cases}$$
A Recurrence for Merge Sort

Analysis:

Recurrence:

\[ T(n) = \begin{cases} \end{cases} \]
A Recurrence for Merge Sort

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.

Recurrence:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 
\end{cases}$$
A Recurrence for Merge Sort

Analysis:
■ If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.
■ If $n > 1$, we

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \text{if } n > 1 \end{cases}$$
A Recurrence for Merge Sort

**Analysis:**
- If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.
- If $n > 1$, we
  - Spend constant time to compute the middle index,

**Recurrence:**

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
 & \text{if } n > 1 
\end{cases}$$
A Recurrence for Merge Sort

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.
- If $n > 1$, we
  - Spend constant time to compute the middle index,
  - Make one recursive call on the left half, whose size is $\lceil n/2 \rceil$.

Recurrence:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(\lceil n/2 \rceil) & \text{if } n > 1 
\end{cases}$$
A Recurrence for Merge Sort

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.
- If $n > 1$, we
  - Spend constant time to compute the middle index,
  - Make one recursive call on the left half, whose size is $\lceil n/2 \rceil$,
  - Make one recursive call on the right half, whose size is $\lfloor n/2 \rfloor$,

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(1) & \text{if } n > 1 \end{cases}$$
A Recurrence for Merge Sort

Analysis:

- If $n = 0$ or $n = 1$, we spend constant time to figure out that there is nothing to do and then exit.

- If $n > 1$, we
  - Spend constant time to compute the middle index,
  - Make one recursive call on the left half, whose size is $\lceil n/2 \rceil$,
  - Make one recursive call on the right half, whose size is $\lfloor n/2 \rfloor$,
  - Spend linear time to merge the two sorted sequences.

Recurrence:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(1) & \text{if } n > 1 
\end{cases}$$
A Recurrence for Merge Sort

Analysis:
- If \( n = 0 \) or \( n = 1 \), we spend constant time to figure out that there is nothing to do and then exit.
- If \( n > 1 \), we
  - Spend constant time to compute the middle index,
  - Make one recursive call on the left half, whose size is \( \lceil n/2 \rceil \),
  - Make one recursive call on the right half, whose size is \( \lfloor n/2 \rfloor \),
  - Spend linear time to merge the two sorted sequences.

Recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 
\end{cases}
\]
A Recurrence for Binary Search

Analysis:

Recurrence:

\[ T(n) = \begin{cases} 
\end{cases} \]
A Recurrence for Binary Search

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to test whether we have found the desired element.

Recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 
\end{cases}
\]
A Recurrence for Binary Search

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to test whether we have found the desired element.
- If $n > 1$, we

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \Theta(1) & \text{if } n > 1 \end{cases}$$
A Recurrence for Binary Search

Analysis:
- If $n = 0$ or $n = 1$, we spend constant time to test whether we have found the desired element.
- If $n > 1$, we
  - Spend constant time to find the middle element of the array and compare it to the search key,

Recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ \Theta(1) & \text{if } n > 1 \end{cases}$$
A Recurrence for Binary Search

Analysis:

- If $n = 0$ or $n = 1$, we spend constant time to test whether we have found the desired element.
- If $n > 1$, we
  - Spend constant time to find the middle element of the array and compare it to the search key,
  - Make one recursive call on one of the two halves.

Recurrence:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(\lceil n/2 \rceil) + \Theta(1) & \text{if } n > 1 
\end{cases}$$
A Recurrence for Binary Search

Analysis:
- If \( n = 0 \) or \( n = 1 \), we spend constant time to test whether we have found the desired element.
- If \( n > 1 \), we
  - Spend constant time to find the middle element of the array and compare it to the search key,
  - Make one recursive call on one of the two halves.

Recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
T(\lceil n/2 \rceil) + \Theta(1) & \text{if } n > 1 
\end{cases}
\]

Observe how we use an inductive description of the running time of an algorithm that operates inductively. This deserves to be called natural.
The recurrences we use to analyze algorithms will have a base case of

\[ T(n) \leq c \quad \forall n \leq n_0. \]

The exact constants only affect the constant factors in the running time.
The recurrences we use to analyze algorithms will have a base case of

\[ T(n) \leq c \quad \forall n \leq n_0. \]

The exact constants only affect the constant factors in the running time.

Floors and ceilings only affect constant factors.
The recurrences we use to analyze algorithms will have a base case of

\[ T(n) \leq c \quad \forall n \leq n_0. \]

The exact constants only affect the constant factors in the running time.

Floors and ceilings only affect constant factors.

So we are lazy and write:

- Merge sort: \( T(n) = 2T(n/2) + \Theta(n) \)
- Binary search: \( T(n) = T(n/2) + \Theta(1) \)
Given two algorithms \( A \) and \( B \) for the same problem with running times

\[
T_A(n) = 2T(n/2) + \Theta(n)
\]
\[
T_B(n) = 3T(n/2) + \Theta(\log n)
\]

Which one is faster?
Given two algorithms $A$ and $B$ for the same problem with running times

\[ T_A(n) = 2T(n/2) + \Theta(n) \]
\[ T_B(n) = 3T(n/2) + \Theta(\log n) \]

Which one is faster?

A recurrence as such tells us very little about the running time of the algorithm?
Solving Recurrences

Given two algorithms \( A \) and \( B \) for the same problem with running times

\[
T_A(n) = 2T(n/2) + \Theta(n)
\]

\[
T_B(n) = 3T(n/2) + \Theta(\log n)
\]

Which one is faster?

A recurrence as such tells us very little about the running time of the algorithm?

We want to “solve” the recurrence, that is, obtain an expression of the form

\( T(n) = \Theta(n^2) \), \( T(n) = \Theta(n \log n) \) or similar.
Solving Recurrences

Given two algorithms $A$ and $B$ for the same problem with running times

$$T_A(n) = 2 T(n/2) + \Theta(n)$$
$$T_B(n) = 3 T(n/2) + \Theta(\log n)$$

Which one is faster?

A recurrence as such tells us very little about the running time of the algorithm?

We want to “solve” the recurrence, that is, obtain an expression of the form $T(n) = \Theta(n^2)$, $T(n) = \Theta(n \log n)$ or similar.

Formally, we want an expression $T(n) = \Theta(f(n))$, where $f(n)$ does not depend on $T(n)$. 
Methods to Solve Recurrences

**Substitution**

- Guess what the right answer is.
  (Intuition, experience, black magic)
- Use induction to prove that the guess is right.
Methods to Solve Recurrences

**Substitution**

- Guess what the right answer is.  
  (Intuition, experience, black magic)
- Use induction to prove that the guess is right.

**Recursion trees**

- Visualize how the recurrence unfolds.
- Use the tree to
  - Obtain a guess, which is then verified using substitution, or
  - Obtain an exact answer if analysis is done sufficiently rigorously.
Methods to Solve Recurrences

Substitution

- Guess what the right answer is. (Intuition, experience, black magic)
- Use induction to prove that the guess is right.

Recursion trees

- Visualize how the recurrence unfolds.
- Use the tree to
  - Obtain a guess, which is then verified using substitution, or
  - Obtain an exact answer if analysis is done sufficiently rigorously.

Master theorem

- Cook book recipe for solving common recurrences.
Lemma: The running time of Merge Sort is $O(n \lg n)$. 
**Lemma:** The running time of Merge Sort is $O(n \lg n)$.

**Recurrence:**

$$T(n) = 2T(n/2) + O(n)$$
Substitution Example: Merge Sort

Lemma: The running time of Merge Sort is $\mathcal{O}(n \lg n)$.

Recurrence:

\[
T(n) = 2T(n/2) + \mathcal{O}(n), \text{ that is,}
\]
\[
T(n) \leq 2T(n/2) + an, \text{ for some } a > 0 \text{ and } n \geq n_0.
\]
Lemma: The running time of Merge Sort is $O(n \lg n)$.

Recurrence:

$$T(n) = 2T(n/2) + O(n), \text{ that is,}$$

$$T(n) \leq 2T(n/2) + an, \text{ for some } a > 0 \text{ and } n \geq n_0.$$ 

Guess:

$$T(n) \leq cn \lg n, \text{ for some } c > 0 \text{ and } n \geq n_1.$$
Lemma: The running time of Merge Sort is $O(n \lg n)$.

Recurrence:

\[ T(n) = 2 T(n/2) + O(n), \text{ that is, } \]
\[ T(n) \leq 2 T(n/2) + an, \text{ for some } a > 0 \text{ and } n \geq n_0. \]

Guess:

\[ T(n) \leq cn \lg n, \text{ for some } c > 0 \text{ and } n \geq n_1. \]

Base case:

- For $2 \leq n < 4$, $T(n) \leq c' \leq c' n \leq c' n \lg n$, for some $c' > 0$.
- Hence, for $c \geq c'$, $T(n) \leq cn \lg n$. 
Inductive step: \((n \geq 4)\)
**Inductive step:** \((n \geq 4)\)

\[ T(n) \leq 2T(n/2) + an \]
**Inductive step:** \( n \geq 4 \)

\[
T(n) \leq 2T(n/2) + an
\]

\[
\leq 2 \cdot \frac{cn}{2} \lg \frac{n}{2} + an
\]
Inductive step: \((n \geq 4)\)

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + an
\]

\[
\leq 2 \cdot \frac{cn}{2} \cdot \log_2 \frac{n}{2} + an
\]

\[
= cn\left(\log n - 1\right) + an
\]
**Inductive step:** \( n \geq 4 \)

\[
T(n) \leq 2T(n/2) + an \\
\leq 2 \cdot \frac{cn}{2} \lg \frac{n}{2} + an \\
= cn(\lg n - 1) + an \\
= cn \lg n + (a - c)n
\]
**Inductive step:** \((n \geq 4)\)

\[
T(n) \leq 2T(n/2) + an
\]

\[
\leq 2 \cdot \frac{cn}{2} \lg \frac{n}{2} + an
\]

\[
= cn(\lg n - 1) + an
\]

\[
= cn\lg n + (a - c)n
\]

\[
\leq cn\lg n, \text{ for all } c \geq a.
\]
**Inductive step:** \((n \geq 4)\)

\[
T(n) \leq 2T(n/2) + an
\]

\[
\leq 2 \cdot \frac{cn}{2} \lg \frac{n}{2} + an
\]

\[
= cn(\lg n - 1) + an
\]

\[
= cn \lg n + (a - c)n
\]

\[
\leq cn \lg n, \text{ for all } c \geq a.
\]

**Notes:**

- Since the base case is valid only for \(n \geq 2\), we can apply the induction hypothesis only to \(n \geq 2\). This is why the inductive step starts at \(n \geq 4\).
**Inductive step:** \((n \geq 4)\)

\[
T(n) \leq 2 \cdot \frac{cn}{2} \lg \frac{n}{2} + an
\]

\[
\leq 2 \cdot \frac{cn}{2} \lg n + an
\]

\[
= cn(\lg n - 1) + an
\]

\[
= cn \lg n + (a - c)n
\]

\[
\leq cn \lg n, \text{ for all } c \geq a.
\]

**Notes:**

- Since the base case is valid only for \(n \geq 2\), we can apply the induction hypothesis only to \(n \geq 2\). This is why the inductive step starts at \(n \geq 4\).

- We only proved the upper bound. The lower bound can be proved similarly, but usually needs to be done separately.
Substitution Example: Binary Search

**Lemma:** The running time of Binary Search is $O(\lg n)$. 
Substitution Example: Binary Search

**Lemma:** The running time of Binary Search is $O(\lg n)$.

**Recurrence:**

$$T(n) = T(n/2) + O(1), \text{ that is, }$$

$$T(n) \leq T(n/2) + a, \text{ for some } a > 0 \text{ and } n \geq n_0.$$
Substitution Example: Binary Search

**Lemma:** The running time of Binary Search is $O(\lg n)$.

**Recurrence:**

$$T(n) = T(n/2) + O(1),$$
that is,

$$T(n) \leq T(n/2) + a,$$
for some $a > 0$ and $n \geq n_0$.

**Guess:**

$$T(n) \leq c \lg n,$$
for some $c > 0$ and $n \geq n_1$.  

Substitution Example: Binary Search

Lemma: The running time of Binary Search is \( O(\lg n) \).

Recurrence:

\[
T(n) = T(n/2) + O(1), \text{ that is, }
T(n) \leq T(n/2) + a, \text{ for some } a > 0 \text{ and } n \geq n_0.
\]

Guess:

\[
T(n) \leq c \lg n, \text{ for some } c > 0 \text{ and } n \geq n_1.
\]

Base case:

- For \( 2 \leq n < 4 \), \( T(n) \leq c' \leq c' \lg n \), for some \( c' > 0 \).
- Hence, for \( c \geq c' \), \( T(n) \leq c \lg n \).
Inductive step: \((n \geq 4)\)

\[ T(n) \leq T(n/2) + a \]
Inductive step: \((n \geq 4)\)

\[
T(n) \leq T(n/2) + a \\
\leq c \lg \frac{n}{2} + a
\]
**Inductive step:** \((n \geq 4)\)

\[
T(n) \leq T\left(\frac{n}{2}\right) + a \\
\leq c \lg \frac{n}{2} + a \\
= c(\lg n - 1) + a
\]
Inductive step: \( n \geq 4 \)

\[
T(n) \leq T(n/2) + a \\
\leq c \log \frac{n}{2} + a \\
= c(\log n - 1) + a \\
= c \log n + (a - c)
\]
**Inductive step:** \((n \geq 4)\)

\[
T(n) \leq T(n/2) + a
\]
\[
\leq c \lg \frac{n}{2} + a
\]
\[
= c(\lg n - 1) + a
\]
\[
= c \lg n + (a - c)
\]
\[
\leq c \lg n, \text{ for all } c \geq a.
\]
**Strategy:** Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
A Recursion Tree for Merge Sort

**Strategy:** Expand the recurrence

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]

\[ \begin{array}{c}
\text{an} \\
\frac{an}{2} & \frac{an}{2} \\
T\left(\frac{n}{4}\right) & T\left(\frac{n}{4}\right)
\end{array} \]
A Recursion Tree for Merge Sort

Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
A Recursion Tree for Merge Sort

**Strategy:** Expand the recurrence

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]
**A Recursion Tree for Merge Sort**

**Strategy:** Expand the recurrence

\[ T(n) = 2 T(n/2) + \Theta(n) \]

![Recursion Tree Diagram](image-url)
Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]

Diagram of the recursion tree for Merge Sort.

- \( T(n) \) at the root.
- Splitting into \( \frac{n}{2} \) at each level.
- Nodes at each level represent \( \Theta(n) \) work.
- Leaf nodes at \( \frac{n}{2^k} \) levels represent base case work.
**A Recursion Tree for Merge Sort**

**Strategy:** Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
A Recursion Tree for Merge Sort

**Strategy:** Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
A Recursion Tree for Merge Sort

**Strategy:** Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
Strategy: Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]
A Recursion Tree for Merge Sort

**Strategy:** Expand the recurrence

\[ T(n) = 2T(n/2) + \Theta(n) \]

**Solution:** \( T(n) = \Theta(n \lg n) \)
A Recursion Tree for Binary Search

\textbf{Recurrence:} \( T(n) = T(n/2) + \Theta(1) \)
**A Recursion Tree for Binary Search**

*Recurrence:* \( T(n) = T(n/2) + \Theta(1) \)
**Recurrence:** \( T(n) = T(n/2) + \Theta(1) \)

**Solution:** \( T(n) = \Theta(\lg n) \)
**Recurrence:** \( T(n) = T(n/3) + T(2n/3) + \Theta(n) \)
**Recurrence:** \( T(n) = T(n/3) + T(2n/3) + \Theta(n) \)
**A Less Obvious Recursion Tree**

**Recurrence:** \( T(n) = T(n/3) + T(2n/3) + \Theta(n) \)

**Solution:** \( T(n) = \Theta(n \log n) \)
Sometimes Only Substitution Will Do

Recurrence: \( T(n) = 2T(n/3) + T(n/2) + \Theta(n) \)
Sometimes Only Substitution Will Do

Recurrence: \( T(n) = 2T(n/3) + T(n/2) + \Theta(n) \)

\( i \)-th level: \( \Theta(n \cdot (7/6)^i) \)

\( \log_2 n \) \hspace{1cm} \log_3 n
\textbf{Sometimes Only Substitution Will Do}

\textit{Recurrence:} \( T(n) = 2T(n/3) + T(n/2) + \Theta(n) \)

\textit{i-th level:} \( \Theta(n \cdot (7/6)^{i}) \)

\textit{Lower bound:} \( T(n) = \Omega(n^{1+\log_{3}(7/6)}) \approx \Omega(n^{1.14}) \)
Sometimes Only Substitution Will Do

**Recurrence:** $T(n) = 2T(n/3) + T(n/2) + \Theta(n)$

**$i$-th level:** $\Theta(n \cdot (7/6)^i)$

**Lower bound:** $T(n) = \Omega(n^{1+\log_3(7/6)}) \approx \Omega(n^{1.14})$

**Upper bound:** $T(n) = O(n^{1+\log_2(7/6)}) \approx O(n^{1.22})$
Master Theorem

**Theorem: (Master Theorem)**

Let \( a \geq 1 \) and \( b > 1 \), let \( f(n) \) be a function over the positive integers, and let \( T(n) \) be given by the following recurrence:

\[
T(n) = a T(n/b) + f(n)
\]

(i) If \( f(n) = \mathcal{O}(n^{\log_b a - \epsilon}) \), for some \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).

(ii) If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a} \lg n) \).

(iii) If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), for some \( \epsilon > 0 \), and \( a f(n/b) \leq c f(n) \), for some \( c < 1 \) and all \( n \geq n_0 \), then \( T(n) = \Theta(f(n)) \).
**Selection**

**Given:**
- An array storing $n$ numbers, $x_1 \leq x_2 \leq \cdots \leq x_n$, in any order
- A parameter $1 \leq k \leq n$

**To compute:** $x_k$

Element $x_k$ is also referred to as the $k$-th order statistic of the set $\{x_1, x_2, \ldots, x_n\}$.

**Example:**

\[
\begin{array}{cccccccc}
16 & 3 & 5 & 21 & 8 & 10 & 7 & 17
\end{array}
\]

The 4-th order statistic is 8 because

\[
\begin{array}{cccccccc}
3 & 5 & 7 & 8 & 10 & 16 & 17 & 21
\end{array}
\]
Repeated Minimum:

- Find and remove the minimum, repeat $k$ times.
Two Simple Solutions

Repeated Minimum:

- Find and remove the minimum, repeat $k$ times.

Running time:
Two Simple Solutions

Repeated Minimum:

- Find and remove the minimum, repeat $k$ times.

Running time: $\Theta(kn)$
Two Simple Solutions

Repeated Minimum:

- Find and remove the minimum, repeat $k$ times.

Running time: $\Theta(kn)$

Sort and select:

- Sort the sequence
- Report the $k$-th item in the sorted sequence
Two Simple Solutions

Repeated Minimum:
- Find and remove the minimum, repeat $k$ times.

Running time: $\Theta(kn)$

Sort and select:
- Sort the sequence
- Report the $k$-th item in the sorted sequence

Running time:
Two Simple Solutions

Repeated Minimum:
- Find and remove the minimum, repeat $k$ times.

Running time: $\Theta(kn)$

Sort and select:
- Sort the sequence
- Report the $k$-th item in the sorted sequence

Running time: $\Theta(n \lg n)$
The Hunt for Intuition: Quicksort

**QUICKSORT**(*A*)

1. if |*A*| ≤ 1
2. then return *A*
3. else \( p \leftarrow \text{median of } *A* \)
4. Partition *A* into three pieces:
   - \( L = \{ x \in *A* \mid x < p \} \)
   - \( \{ p \} \)
   - \( R = \{ x \in *A* \setminus \{ p \} \mid x \geq p \} \)
5. \( L' \leftarrow \text{QUICKSORT}(L) \)
6. \( R' \leftarrow \text{QUICKSORT}(R) \)
7. return \( L' \circ \{ p \} \circ R' \)
The Hunt for Intuition: Quicksort

**Quicksort**

1. **if** $|A| \leq 1$
2. **then return** $A$
3. **else** $p \leftarrow$ median of $A$
4. **Partition** $A$ into three pieces:
   - $L = \{x \in A \mid x < p\}$
   - $\{p\}$
   - $R = \{x \in A \setminus \{p\} \mid x \geq p\}$
5. $L' \leftarrow $ Quicksort($L$)
6. $R' \leftarrow $ Quicksort($R$)
7. **return** $L' \circ \{p\} \circ R'$

**Assumption:** We know how to find the median in $\Theta(n)$ time.

**Running time:** $T(n) =$
**The Hunt for Intuition: Quicksort**

```plaintext
QUICKSORT(A)
1  if |A| ≤ 1
2    then return A
3  else p ← median of A
4      Partition A into three pieces:
5          ■ L = \{x ∈ A | x < p\}
6          ■ \{p\}
7          ■ R = \{x ∈ A \{p\} | x ≥ p\}
8  L' ← QUICKSORT(L)
9  R' ← QUICKSORT(R)
10 return L' ◦ \{p\} ◦ R'
```

**Assumption:** We know how to find the median in $\Theta(n)$ time.

**Running time:** $T(n) = 2T(n/2) + \Theta(n)$
The Hunt for Intuition: Quicksort

Quicksort\( (A) \)

1. if \(|A| \leq 1\) then return \(A\)
2. else \(p \leftarrow \) median of \(A\)
3. Partition \(A\) into three pieces:
   - \(L = \{x \in A \mid x < p\}\)
   - \(\{p\}\)
   - \(R = \{x \in A \setminus \{p\} \mid x \geq p\}\)
4. \(L' \leftarrow \text{Quicksort}(L)\)
5. \(R' \leftarrow \text{Quicksort}(R)\)
6. return \(L' \circ \{p\} \circ R'\)

Assumption: We know how to find the median in \(\Theta(n)\) time.

Running time: \(T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)\)
Start by partitioning $A$ around the median:
Start by partitioning $A$ around the median:

$$L \quad p \quad R$$

- If $k = |L| + 1$, then $p = x_k$. 
Start by partitioning $A$ around the median:

$$L \quad p \quad R$$

- If $k = |L| + 1$, then $p = x_k$.

Return $p$
Start by partitioning $A$ around the median:

- If $k = |L| + 1$, then $p = x_k$.
  
  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$. 
Start by partitioning $A$ around the median:

- If $k = |L| + 1$, then $p = x_k$.
  
  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$.
  
  Recursively find the $k$-th order statistic in $L$ and return it.
Start by partitioning $A$ around the median:

\[
\begin{array}{c|c|c}
L & p & R
\end{array}
\]

- If $k = |L| + 1$, then $p = x_k$.

  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$.

  Recursively find the $k$-th order statistic in $L$ and return it.

- If $k > |L| + 1$, then $x_k \in R$ and $y \leq x_k$, for all $y \in L \cup \{p\}$. 

Start by partitioning $A$ around the median:

$$L \ p \ R$$

- If $k = |L| + 1$, then $p = x_k$.
  
  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$.
  
  Recursively find the $k$-th order statistic in $L$ and return it.

- If $k > |L| + 1$, then $x_k \in R$ and $y \leq x_k$, for all $y \in L \cup \{p\}$.
  
  Recursively find the $(k - |L| - 1)$-st order statistic in $R$ and return it.
Partition & Recurse

Start by partitioning $A$ around the median:

\[
\begin{array}{c|c|c}
L & p & R \\
\end{array}
\]

- If $k = |L| + 1$, then $p = x_k$.
  
  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$.
  
  Recursively find the $k$-th order statistic in $L$ and return it.

- If $k > |L| + 1$, then $x_k \in R$ and $y \leq x_k$, for all $y \in L \cup \{p\}$.
  
  Recursively find the $(k - |L| - 1)$-st order statistic in $R$ and return it.

**Running time:** $T(n) \leq$
Start by partitioning $A$ around the median:

- If $k = |L| + 1$, then $p = x_k$. 
  Return $p$

- If $k < |L| + 1$, then $x_k \in L$ and $y > x_k$, for all $y \in R \cup \{p\}$. 
  Recursively find the $k$-th order statistic in $L$ and return it.

- If $k > |L| + 1$, then $x_k \in R$ and $y \leq x_k$, for all $y \in L \cup \{p\}$. 
  Recursively find the $(k - |L| - 1)$-st order statistic in $R$ and return it.

Running time: $T(n) \leq T(n/2) + O(n)$
Partition & Recurse

Start by partitioning \( A \) around the median:

\[
\begin{array}{c|c|c}
L & p & R \\
\end{array}
\]

- If \( k = |L| + 1 \), then \( p = x_k \).
  
  Return \( p \)

- If \( k < |L| + 1 \), then \( x_k \in L \) and \( y > x_k \), for all \( y \in R \cup \{p\} \).
  
  Recursively find the \( k \)-th order statistic in \( L \) and return it.

- If \( k > |L| + 1 \), then \( x_k \in R \) and \( y \leq x_k \), for all \( y \in L \cup \{p\} \).
  
  Recursively find the \((k - |L| - 1)\)-st order statistic in \( R \) and return it.

**Running time:** \( T(n) \leq T(n/2) + O(n) = O(n) \)
**Problem:** Finding the median of $A$ is selection.
Problem: Finding the median of $A$ is selection.

Have we walked in a circle?
**Problem:** Finding the median of $A$ is selection.

Have we walked in a circle?

**Observation:** An “approximate” median does the job:

If $|L| \leq cn$ and $|R| \leq cn$, for some $c < 1$, then

$$T(n) \leq T(cn) + O(n) = O(n).$$
Finding an Approximate Median

- Partition input into groups of 5 elements.
- Sort each group and add its 3rd element to an array $A'$.
- Find the median of $A'$ (by calling the selection algorithm recursively!) and return as approximate median.
Lemma: There are at least \( \frac{3n}{10} - 6 \) elements on either side of the computed approximate median \( p \).
The Procedure Finds an Approximate Median

**Lemma:** There are at least \( \frac{3n}{10} - 6 \) elements on either side of the computed approximate median \( p \).

**Proof:** (for elements greater than \( p \))

- At least \( \left\lceil \frac{\lceil n/5 \rceil}{2} \right\rceil - 1 \geq \frac{n}{10} - 1 \) groups to the right of \( p \)
- At most one is not full
- Every full group contains at least 3 elements \( > p \)

**Total:** \( 3 \left( \frac{n}{10} - 2 \right) = \frac{3n}{10} - 6 \)
Summary of selection algorithm:

- Find approximate median: linear work + recurse on $\lceil n/5 \rceil$ elements
- Partition: linear work
- Recurse on piece of size at most $7n/10 + 6$
The Final Running Time

Summary of selection algorithm:

- Find approximate median: linear work + recurse on \( \lceil n/5 \rceil \) elements
- Partition: linear work
- Recurse on piece of size at most \( 7n/10 + 6 \)

Recurrence:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq 140 \\
O(n) + T(\lceil n/5 \rceil) + T(7n/10 + 6) & \text{if } n > 140
\end{cases}
\]
Summary of selection algorithm:

- Find approximate median: linear work + recurse on \( \lceil n/5 \rceil \) elements
- Partition: linear work
- Recurse on piece of size at most \( 7n/10 + 6 \)

Recurrence:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq 140 \\
O(n) + T(\lceil n/5 \rceil) + T(7n/10 + 6) & \text{if } n > 140 
\end{cases}
\]

\[= O(n)\]
**The Final Running Time**

**Summary of selection algorithm:**
- Find approximate median: linear work + recurse on $\lceil n/5 \rceil$ elements
- Partition: linear work
- Recurse on piece of size at most $7n/10 + 6$

**Recurrence:**

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq 140 \\
O(n) + T(\lceil n/5 \rceil) + T(7n/10 + 6) & \text{if } n > 140
\end{cases}
\]

\[= O(n)\]

**Theorem:** The $k$-th order statistic of a set of $n$ elements can be found in $O(n)$ time.
Counting Inversions

Given a sequence $S = (x_1, x_2, \ldots, x_n)$ of $n$ numbers, an inversion is a pair $(x_i, x_j)$ such that
- $i < j$ and
- $x_i > x_j$.

Example:

$$5 \ 3 \ 7 \ 8 \ 21 \ 10 \ 17 \ 16$$

Inversions: $(5, 3), (21, 10), (21, 17), (21, 16), (17, 16)$
Given a sequence \( S = (x_1, x_2, \ldots, x_n) \) of \( n \) numbers, an inversion is a pair \((x_i, x_j)\) such that

- \( i < j \) and
- \( x_i > x_j \).

**Example:**

\[
5 \quad 3 \quad 7 \quad 8 \quad 21 \quad 10 \quad 17 \quad 16
\]

Inversions: \((5, 3), (21, 10), (21, 17), (21, 16), (17, 16)\)

**Problem:** Count all inversions in \( S \).
As in Merge Sort, partition array into left half, $L$, and right half, $R$:

- An inversion $(x_i, x_j)$ is **short** if $\{x_i, x_j\} \subseteq L$ or $\{x_i, x_j\} \subseteq R$
- An inversion $(x_i, x_j)$ is **long** if $x_i \in L$ and $x_j \in R
Classifying Inversions

As in Merge Sort, partition array into left half, $L$, and right half, $R$:

- An inversion $(x_i, x_j)$ is **short** if $\{x_i, x_j\} \subseteq L$ or $\{x_i, x_j\} \subseteq R$

- An inversion $(x_i, x_j)$ is **long** if $x_i \in L$ and $x_j \in R$

Since we are talking about divide and conquer:
- Find short recursions recursively.
Observation: Sorting $L$ and $R$ does not affect the number of long inversions.
**Observation:** Sorting $L$ and $R$ does not affect the number of long inversions.

**Procedure:**
- Sort $L$ and $R$.
- Count long inversions by merging $L$ and $R$: 
  - $L \cup R$
  - $L$
  - $R$
Observation: *Sorting* $L$ and $R$ does not affect the number of long inversions.

**Procedure:**
- Sort $L$ and $R$.
- Count long inversions by merging $L$ and $R$:
  - When $y < x$, then $y$ forms an inversion with exactly the elements remaining in $L$. 
Counting Long Inversions

**Observation:** Sorting $L$ and $R$ does not affect the number of long inversions.

**Procedure:**
- Sort $L$ and $R$.
- Count long inversions by merging $L$ and $R$:
  - When $y < x$, then $y$ forms an inversion with exactly the elements remaining in $L$.
  - Increase inversion count by $|L|$.
Analysis

\[ T(n) = \]
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \log n) \]
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \log n) = \Theta(n \log^2 n). \]
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \lg n) = \Theta(n \lg^2 n). \]

**Observation:**
- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \log n) = \Theta(n \log^2 n). \]

**Observation:**
- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
- In particular, the recursive calls on \( L \) and \( R \) return \( L \) and \( R \) in sorted order.
\[ T(n) = 2 T(n/2) + \Theta(n \log n) = \Theta(n \log^2 n). \]

**Observation:**

- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
- In particular, the recursive calls on \( L \) and \( R \) return \( L \) and \( R \) in sorted order.

∴ We can save the sorting step.
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \lg n) = \Theta(n \lg^2 n). \]

**Observation:**
- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
- In particular, the recursive calls on \( L \) and \( R \) return \( L \) and \( R \) in sorted order.
∴ We can save the sorting step.

\[ \therefore T(n) = \]
\[ T(n) = 2T(n/2) + \Theta(n \log n) = \Theta(n \log^2 n). \]

**Observation:**

- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
- In particular, the recursive calls on \( L \) and \( R \) return \( L \) and \( R \) in sorted order.

\[ \therefore \quad T(n) = 2T(n/2) + \Theta(n) \]
Analysis

\[ T(n) = 2T(n/2) + \Theta(n \lg n) = \Theta(n \lg^2 n). \]

**Observation:**

- Counting long inversions produces \( L \cup R \) in sorted order, as a by-product.
- In particular, the recursive calls on \( L \) and \( R \) return \( L \) and \( R \) in sorted order.
- \( \therefore \) We can save the sorting step.

\[ \therefore T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n). \]
Multiplying Large Integers

Problem: Given two $n$-digit numbers $x = x_{n-1}x_{n-2} \ldots x_0$ and $y = y_{n-1}y_{n-2} \ldots y_0$, we want to compute $z = x \cdot y$ using only digit-wise operations.
Multiplying Large Integers

**Problem:** Given two $n$-digit numbers $x = x_{n-1}x_{n-2} \ldots x_0$ and $y = y_{n-1}y_{n-2} \ldots y_0$, we want to compute $z = x \cdot y$ using only digit-wise operations.

**The traditional method:**

\[
\begin{array}{c}
54163 \\ 324978 \\ 162489 \\ 108326 \\ 0 \\
\hline
54163 \\
\hline
3413406423
\end{array}
\]
Multiplying Large Integers

**Problem:** Given two $n$-digit numbers $x = x_{n-1}x_{n-1} \ldots x_0$ and $y = y_{n-1}y_{n-2} \ldots y_0$, we want to compute $z = x \cdot y$ using only digit-wise operations.

**The traditional method:**

\[
\begin{array}{c}
54163 \\
\times \quad 63021
\end{array}
\]

\[
\begin{array}{c}
324978 \\
162489 \\
\quad 0 \\
108326 \\
\quad 54163
\end{array}
\]

\[
\begin{array}{c}
\hline
3413406423
\end{array}
\]

**Cost:**
Multiplying Large Integers

**Problem:** Given two *n*-digit numbers \( x = x_{n-1}x_{n-2} \ldots x_0 \) and \( y = y_{n-1}y_{n-2} \ldots y_0 \), we want to compute \( z = x \cdot y \) using only digit-wise operations.

**The traditional method:**

\[
\begin{array}{c}
54163 \\ \times 63021 \\
\hline
324978 \\
162489 \\
\multicolumn{2}{c}{0} \\
108326 \\
\hline
54163 \\
3413406423
\end{array}
\]

**Cost:** \( \Theta(n^2) \)
Divide-and-Conquer Multiplication

Assumption: $n = 2^k$

A recursive method:
Assumption: $n = 2^k$

A recursive method:

Recurrence: $T(n) =$
Divide-and-Conquer Multiplication

Assumption: \( n = 2^k \)

A recursive method:

Recurrence: \( T(n) = 4T(n/2) + \Theta(n) \)
Divide-and-Conquer Multiplication

**Assumption:** \( n = 2^k \)

**A recursive method:**

\[
\begin{align*}
  x' & \quad x'' \\
  y' & \quad y'' \\
\end{align*}
\]

\[
\begin{align*}
  x'' \cdot y'' & \\
  x'' \cdot y' + x' \cdot y'' & \\
  x' \cdot y' & \\
\end{align*}
\]

\[
\begin{align*}
  x \cdot y & \\
\end{align*}
\]

**Recurrence:** \( T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2) \)

_Bummer!_
Compute recursively:

- $A = x' \cdot y'$
- $B = x'' \cdot y''$
- $C = (x' + x'') \cdot (y' + y'')$
One Less Recursive Call

Compute recursively:

- \( A = x' \cdot y' \)
- \( B = x'' \cdot y'' \)
- \( C = (x' + x'') \cdot (y' + y'') \)

Combine results:
One Less Recursive Call

Compute recursively:

- \( A = x' \cdot y' \)
- \( B = x'' \cdot y'' \)
- \( C = (x' + x'') \cdot (y' + y'') \)

Combine results:

- \( x' \cdot y' = A \)
- \( x'' \cdot y'' = B \)
One Less Recursive Call

Compute recursively:

- $A = x' \cdot y'$
- $B = x'' \cdot y''$
- $C = (x' + x'') \cdot (y' + y'')$

Combine results:

- $x' \cdot y' = A$
- $x'' \cdot y'' = B$
- $x' \cdot y'' + x'' \cdot y' = C - A - B$
One Less Recursive Call

Compute recursively:

- $A = x' \cdot y'$
- $B = x'' \cdot y''$
- $C = (x' + x'') \cdot (y' + y'')$

Combine results:

- $x' \cdot y' = A$
- $x'' \cdot y'' = B$
- $x' \cdot y'' + x'' \cdot y' = C - A - B$

Recurrence: $T(n) =$
One Less Recursive Call

Compute recursively:

- \( A = x' \cdot y' \)
- \( B = x'' \cdot y'' \)
- \( C = (x' + x'') \cdot (y' + y'') \)

Combine results:

- \( x' \cdot y' = A \)
- \( x'' \cdot y'' = B \)
- \( x' \cdot y'' + x'' \cdot y' = C - A - B \)

Recurrence: \( T(n) = 3 T(n/2) + \Theta(n) \)
One Less Recursive Call

Compute recursively:

- $A = x' \cdot y'$
- $B = x'' \cdot y''$
- $C = (x' + x'') \cdot (y' + y'')$

Combine results:

- $x' \cdot y' = A$
- $x'' \cdot y'' = B$
- $x' \cdot y'' + x'' \cdot y' = C - A - B$

Recurrence: $T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{1+\log(3/2)})$
One Less Recursive Call

Compute recursively:
- \( A = x' \cdot y' \)
- \( B = x'' \cdot y'' \)
- \( C = (x' + x'') \cdot (y' + y'') \)

Combine results:
- \( x' \cdot y' = A \)
- \( x'' \cdot y'' = B \)
- \( x' \cdot y'' + x'' \cdot y' = C - A - B \)

Recurrence: \( T(n) = 3T(n/2) + \Theta(n) = \Theta(n^{1+\lg(3/2)}) \approx \Theta(n^{1.58}) \)
One Less Recursive Call

Compute recursively:

- $A = x' \cdot y'$
- $B = x'' \cdot y''$
- $C = (x' + x'') \cdot (y' + y'')$

Combine results:

- $x' \cdot y' = A$
- $x'' \cdot y'' = B$
- $x' \cdot y'' + x'' \cdot y' = C - A - B$

Recurrence: $T(n) = 3 \, T(n/2) + \Theta(n) = \Theta(n^{1+\log(3/2)}) \approx \Theta(n^{1.58})$

Note: This works only because addition has an inverse operation; that is, it does not work over a semi-ring.
**Summary**

**Divide an conquer:**

- **Divide** the problem instance into two or more smaller instances of the same problem.
- Recursively solve (**conquer**) these smaller problem instances.
- **Combine** the solutions obtained for the smaller instances to construct a solution for the original problem instance.

**Divide-and-conquer algorithms are by definition recursive.**

∴ Natural expression of running time using recurrence relations
∴ Natural correctness proofs using induction

**Solving recurrence relations:**

- Substitution
- Recursion trees
- Master theorem