Algorithms, Design & Analysis

Lecture 05: **Asymptotic Notations And Quick Sort**

Hamza Raza And Fouz Ul Azeem

Information Technology University

June 19, 2025

About Your Fellows

- Hi there! We are **Hamza** and **Fouz**.
- We are Associate Students at ITU.

Asymptotic Notations

- Asymptotic Notations
  - There are three rules behind this idea.

Rule 1

Rule 1: We take estimations of runtime in terms of bounds.

- Upper bound
- Lower bound
- Tight bound

- **Upper Bound:** *O (Big Oh)*
  - $T(n) \leq F(n)$.
- **Lower Bound:** Ω *(Big Omega)*
  - $T(n) \geq g(n)$.
- **Tight Bound:** Θ *(Big Theta)*
  - $T(n) \geq F(n)$ & $T(n) \leq F(n)$
- Where F & G are functions of size N.

Rule 1

For every string input function, the size is the length of that string. Makes sense right?

- **Problem:** If the input is an integer. What will be the size?
  - Always remember, in e.g., *F(n)*, **n** is the size for arrays or strings, but for an integer input, the size is O(log n).
  - Therefore, we take the number of bits required to display this number as the size. *(Log(n))*

Rule 1

Example:

Mystery(p):
- if $p < 2$:
- return 1
- else:
- return 1 + Mystery(sqrt(p))

Input Size of this function will be *Log(p)*.

Rule 2

Rule 2: Small Inputs are irrelevant.

When someone gives us a bound on a function, we only need to make sure the bound is valid on a really large n.

How Large is really large?

As large as you want!

Choose a minimum input size n_0.

Upper Bound:

$$T(n) \leq F(n) \quad \forall n > n_0$$

Rule 2

Why is that?

It is because functions can be messy and to actually know what is going on we need a wider picture of it. For example, if we take two graphs, one of $x + 1$ and one of $\log x$, the graph of $x + 1$ will dominate it. Now if we multiply $\log x$ with a large number like fifty thousand, and take cube of $\log x$, visually it will feel as if the graph of $x + 1$ is being dominated, but that is not true as there is a point where $x+1$ will cut $\log x$ graph and increase at a greater rate. This is because $x + 1$ is a linear graph and $\log x$ grows much more slowly.

Graphs

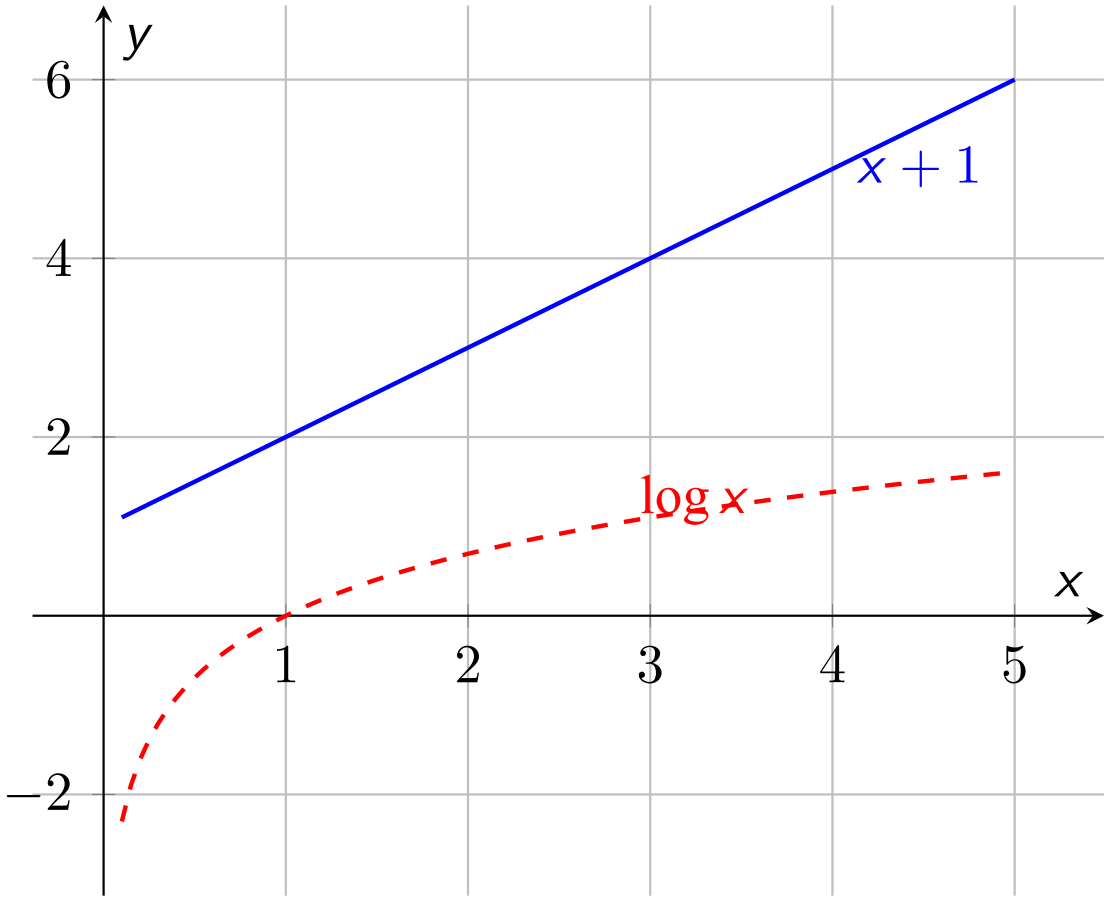


Figure: Graph of $x + 1$ **and** $\log x$

Graphs

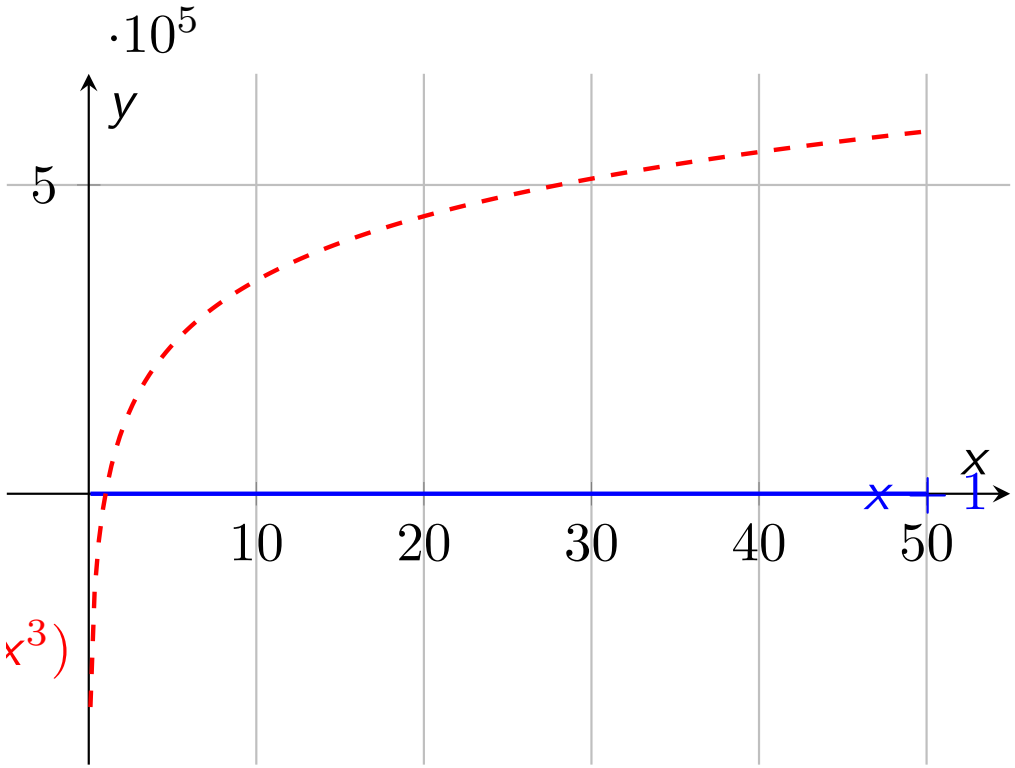


Figure: Graph of $x + 1$ **and** $50000 \cdot \log(x^3)$

Graphs

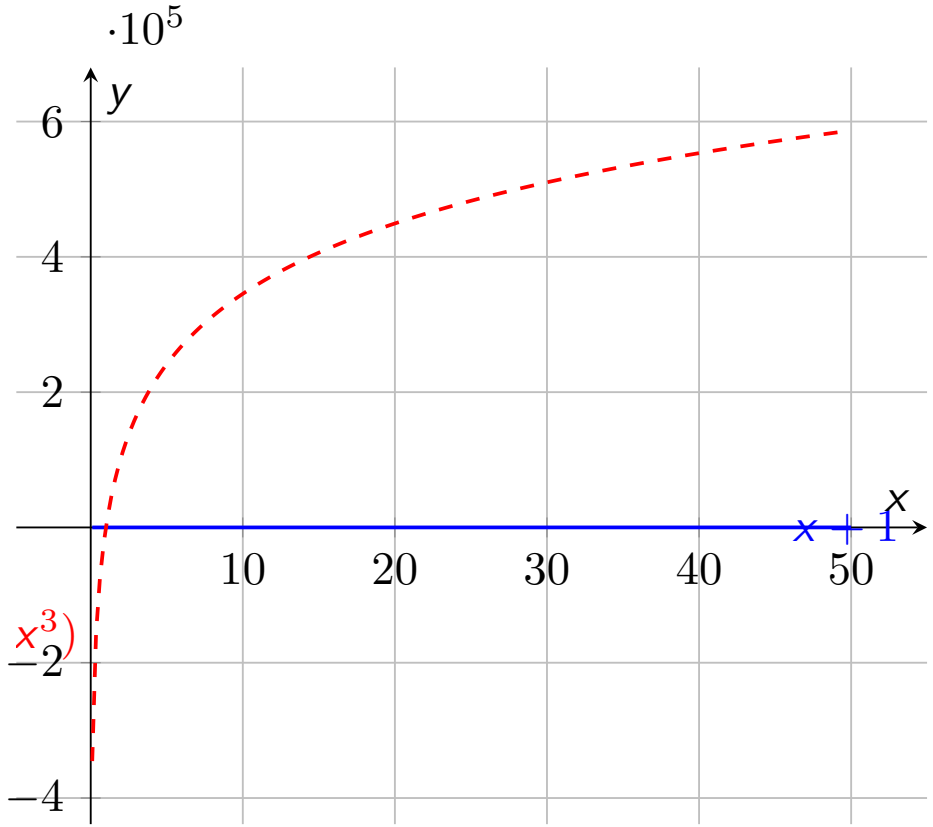


Figure: Graph of $x + 1$ **and** $50000 \cdot \log(x^3)$ in a wider view

- **We can see that the $x + 1$ graph eventually intersects the $50000 \cdot \log(x^3)$ graph.**

Rule 3

Rule 3: Constants do not matter in asymptotics!

- If $T(n)$ is upper bounded by $F(n)$
- 2 * $T(n)$ should also be upper bounded by $F(n)$
- 100 * $T(n)$ should also be upper bounded by $F(n)$

This is achieved by allowing a constant to be multiplied with $F(n)$ when trying to upper bound $T(n)$.

For some $c > 0$, $T(n) \leq c \cdot F(n)$, for $n > n_0$

Examples

Example 1: Let $T(n) = O(g(n))$
This implies there exists a constant c such that:

$$T(n) \leq cg(n)$$

Example 2: Algorithm Complexity

- Algo 0: Time Complexity $= T(n)$
- Algo 1: Loop from $i = 1$ to 100
  $\Rightarrow$ Still $O(T(n))$, since constant factors don't matter
- Algo 2:
  - Loop from $i = 1$ to 10^6
    $\Rightarrow O(T(n))$, as it remains bounded by a constant multiple
  - Loop from $i = 1$ to n
    $\Rightarrow O(nT(n))$, since it scales with n

Conclusion

Therefore, based on these three rules, we have defined that $T(n)$ is Big O of $F(n)$, $T(n) = 0(F(n))$, whenever we can find a constant so that $T(n) \leq c * F(n)$, for really large values of n.

Quick Sort

In Quick sort we use ranks of each element in an array or list to sort them.

- **What is the Rank problem:**
  - Rank means the position of an element in a sorted array/list.
  - Therefore, one easy way of finding the rank is by first sorting the array, and then using index to find the element's rank.
  - Another way is to compare the element with all other elements and count the ones smaller. This is efficient as it is in $O(n)$.

The Rank Problem

- **Problem Statement:**
  - Given: Array A (unsorted, distinct elements)
  - Input: Index i
  - Goal: Find position of A[i] in sorted version of A
- **Definition:**
  - Rank(A, i) = Position of A[i] in sorted array
  - Equivalent to counting elements smaller than A[i]

Rank Algorithm

Algorithm: Rank(A, i)

1. Let count $= 0$
2. For each j i in array A:
   - If A[j] $<$ A[i], increment count
3. Return count $+ 1$ (the rank)

Mathematical Representation

Let L $= \{$A[j] $<$ A[i]$\}$

$$\text{Rank(A, i)} = |L| + 1$$

where $|$L$|$ is the number of elements smaller than A[i]

Time Complexity Analysis

- **Algorithm Cost:**
  - Need to compare A[i] with all other elements
  - Total comparisons = n - 1
  - Time Complexity: O(n)
- **Key Points:**
  - Linear time algorithm
  - No need to sort the entire array
  - Direct comparison approach

Quick Sort

Our Own Version of Quick Sort:

```
QS1(A) {
  B = [ ]
  for i = 0 to |A| − 1
  x = rank(A,i)
  B[x] = A[i]
  return B
}
```

It compares each element with every other element. Therefore, the complexity of this function is $\mathcal{O}(n^2)$.

We can do better!

Quick Sort Example

Initial Array

[5, 9, 7, 0, -15, 4, 19]

Step 1: Choosing Pivot

- Choose 5 as pivot
- Partition array into:
  - Left (smaller): [0, -15, 4]
  - Pivot: [5]
  - Right (larger): [9, 7, 19]

Quick Sort Example (continued)

Step 2: Recursive Partitioning

Left side: [0, -15, 4]

- Choose 0 as pivot
- Left: [-15]
- Pivot: [0]
- Right: [4]

Right side: [9, 7, 19]

- Choose 9 as pivot
- Left: [7]
- Pivot: [9]
- Right: [19]

Quick Sort Example (Final)

Final Result

- Combining all partitions:

$$[-15] \rightarrow [0] \rightarrow [4] \rightarrow [5] \rightarrow [7] \rightarrow [9] \rightarrow [19]$$

Key Observations

- Each partition divides the problem into smaller sub problems
- Elements are naturally sorted during the partitioning process
- More efficient than comparing every element with every other element
- **Mathematical Property:**

$$\forall start \in S, \forall end \in L : \text{rank}(s) < \text{rank}(l) \implies s < l$$

where S is the set of smaller elements and L is the set of larger elements

Improvised Quick Sort complexity

- This function uses extra memory as we create separate lists.
- Best case we get almost equal lengths of both lists so due to recursion it becomes *O(nlogn)*.
- In the worst case, all the elements except one could be on one side and therefore,every element will be compared. So complexity would be $\mathcal{O}(n^2)$.

  Is there a better approach with better worst case complexity and is in place?

  YES!

Improvising

```
QS2(A) {
  if |A| ≤ 1
  return A
  B = [ ]
  S = { A_i | A_i < A[0] }
  L = { A_i | A_i > A[0] }
  NL = QS2(L)
  NS = QS2(S)
  B = [ NS, A[0], NL ]
  return B
}
```

We do not want to use extra memory at first so it becomes an in place algorithm.

Actual Quick Sort

Initial Array:

$$[\mathbf{5}, 9, 7, 0, -15, 4, 19]$$

Pivot: 5

Step 1: compare 5 with 9. As it is lesser we swap 9 with last element and recursively call without 9.

$$[\mathbf{5}, 19, 7, 0, -15, 4, \mathbf{9}]$$

Step 2: compare 5 with 19. As it is lesser we swap 9 with new last element and recursively call without 19,9.

$$[\mathbf{5}, 4, 7, 0, -15, 19, 9]$$

Step 3: We compare 5 with 4 and as 4 is lesser we swap with 5 and recursively call without 4

$$[4, \mathbf{5}, 7, 0, -15, 19, 9]$$

Actual Quick Sort

$$[4, \mathbf{5}, 7, 0, -15, 19, 9]$$

Step 4: We compare 5 with 7 and as 7 is greater we swap with last element which is -15.

$$[4, \mathbf{5}, -15, 0, 7, 19, 9]$$

Step 5: We compare 5 with -15 and as -15 is lesser we swap with 5 and recursively call without -15.

$$[4, -15, \mathbf{5}, 0, 7, 19, 9]$$

Similarly with zero:

$$[4, -15, 0, \mathbf{5}, 7, 19, 9]$$

Now only one element is left so return call would be made and we can see that the pivot is at it's correct position. This is only one iteration. Two recursive calls will be made on each side of the pivot until the base case is reached and whole list is then sorted.

In-Place Algorithm: Does not allocate more than constant space.

QS2 Algorithm: In-Place Quick Sort

- **QS2(A, x, y)** is an in-place variant of Quick Sort.
- It does not allocate more than $O(1)$ extra memory.
- Before executing QS2, shuffle the array A to ensure randomness.

In-Place QuickSort

QS2: In-Place QuickSort

```
def QS2(A, start, end):
 if end < start + 1:
   return A
 i, j = start + 1, end + 1
 p = A[i + 1]
 while i < j - 2:
   if A[i + 2] <= A[i + 1]:
     i += 1
   if A[i + 2] > A[i + 1]:
     A[i + 2], A[j - 1] = A[j - 1], A[i + 2]
     j -= 1
 QS2(A, start, i)
 QS2(A, j, end)
 return A
```

Understanding Time Complexity Recurrence

- The time complexity of QS2:
  - **Worst-case:** $O(n^2)$ (occurs when the pivot selection is poor).
  - **Best-case:** $O(n \log n)$ (occurs when partitions are balanced).
  - **Average-case:** $O(n \log n)$ (expected performance over random inputs).
- The recurrence relation for QS2 is given by:

$$T(n) = aT(n/b) + f(n)$$

- If we assume a randomized pivot selection:

$$a_1, a_2, a_3, \ldots\ldots, a_{n-3}, a_{n-2}, a_{n-1}, a_n$$

- The pivot is expected to lie between $\frac{n}{4}$ and $\frac{3n}{4}$, leading to more balanced partitions.
- If $|S| < \frac{n}{4}$ or $|L| < \frac{n}{4}$, we may need to recompute the partition.

Randomized Quick Sort: Expected Runtime

- Randomized Quick Sort improves partitioning by choosing a random pivot, reducing the likelihood of worst-case behavior.
- The recursive relation for randomized Quick Sort is:

$$T(n) = T(|L|) + T(n - |L| - 1) + cn$$

- Expected time complexity analysis:

$$E(T(n)) = 2cn + E(T(|L|)) + E(T(n - |L| - 1)) \leq c2n$$

- By maintaining a balanced partition, we can ensure an average-case complexity of $O(n \log n)$.

Quick Sort Partitioning Visualization

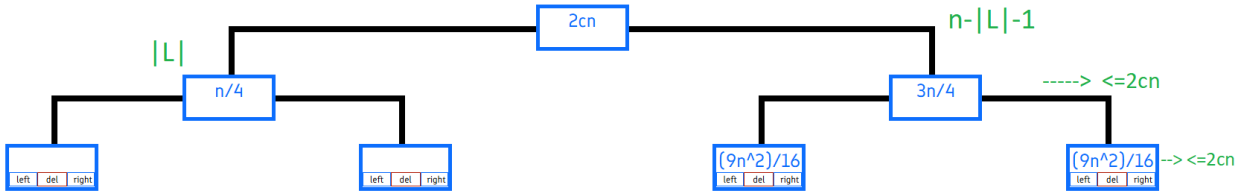


Figure: Recursive Tree of Quick Sort

- The pivot is chosen, and the array is divided into two subarrays.
- Elements smaller than the pivot move to the left; larger ones move to the right.
- The process repeats recursively until the array is sorted.

Quick Sort: Longest Branch and Expected Complexity

- The **right-side** branch in the Quick Sort recursion tree is the **longest branch.**
- The recursion reaches **1** when $k \approx \log n$.
- Given the inequality:

$$(3/4)^k n \leq 1$$

- Solving for k:

$$k \log(3/4) + \log n \leq 1$$

$$k \log(3/4) \geq \log n - 1$$

$$k \approx \log n$$

- Since $k \approx \log n$, the expected time complexity for Quick Sort is:

$$E(T(n)) = O(n \log n)$$

Algorithms, Design & Analysis

Lecture 06: **Quick Sort And Merge Sort**

Quick Sort Recap

- Quick Sort is a divide-and-conquer sorting algorithm that partitions an array and recursively sorts the partitions.

Pivot Selection: Guessing and Median of Medians

- The choice of pivot greatly affects Quick Sort's efficiency.
- One approach is using a random guess or selecting the **median of medians** for improved balance.
- The recurrence relation for expected time complexity is:

$$T(n) = cn + T(|L|) + T(n - |L| - 1)$$

- On average, the time complexity is:

$$E(T(n)) = O(n \log n)$$

Fixing Worst-Case Runtime

- Standard Quick Sort has a worst-case runtime of $\mathbf{O(n^2)}$ when the partitioning is highly unbalanced.

Fixing Worst-Case Runtime

- To ensure worst-case runtime of O(n log n), we can modify the recurrence relation :

$$T(n) = dn + T(n/2) + T((n/2) - 1)$$

- This ensures that the partitions are **always balanced**, leading to consistent performance.

Merge Sort Algorithm

Definition: Merge Sort is a divide-and-conquer sorting algorithm that recursively divides an array into halves, sorts them, and merges them back together.

Important Assumption

The array is 1-based indexed, meaning $A[1]$ is the first element (not $A[0]$).

MergeSort(A) - Recursive Algorithm

- **Input:** An array A of size n
- **Output:** A sorted array

1. **Stopping Condition:** If $|A| \leq 1$, return A.
2. **Divide:** Recursively sort two halves:
   - $B = \text{MergeSort}(A[1 \ldots n/2])$
   - $C = \text{MergeSort}(A[(n/2) + 1 \ldots n])$
3. **Conquer:** Merge sorted subarrays B and C to get D.
4. **Return:** D.

Time Complexity:

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$$

Merging Two Sorted Arrays

Definition: The merge step combines two sorted subarrays, X and Y, into a single sorted array.

Given:

- X is the first half of array A, and it is sorted.
- Y is the second half of array A, and it is sorted.

Time Complexity

Merging two sorted arrays X and Y of sizes n_1 and n_2 respectively takes:

$$S(n_1, n_2) = a(n_1 + n_2) = O(n)$$

since each element is compared at most once.

Merging Two Sorted Arrays

Merge Algorithm

Function: Merge(X, Y)

- If X is empty, return Y.
- If Y is empty, return X.
- If $X[1] \leq Y[1]$, return $(X[1], \text{Merge}(X[2 \ldots |X|], Y))$.
- Otherwise, return $(Y[1], \text{Merge}(Y[2 \ldots |Y|], X))$.

Merging Two Sorted Arrays - Step by Step

Given Arrays:

$$X = [2, 4, 7, 56, 1920, 2025, 2049]$$

$$Y = [0, 5, 9, 1089]$$

Merge Process:

1. Compare $X[1] = 2$ and $Y[1] = 0$. Since $0 < 2$, add 0 first.
2. Compare $X[1] = 2$ and $Y[2] = 5$. Since $2 < 5$, add 2.
3. Compare $X[2] = 4$ and $Y[2] = 5$. Since $4 < 5$, add 4.
4. Compare $X[3] = 7$ and $Y[2] = 5$. Since $5 < 7$, add 5.
5. Compare $X[3] = 7$ and $Y[3] = 9$. Since $7 < 9$, add 7.
6. Compare $X[4] = 56$ and $Y[3] = 9$. Since $9 < 56$, add 9.
7. Compare $X[4] = 56$ and $Y[4] = 1089$. Since $56 < 1089$, add 56.
8. Compare $X[5] = 1920$ and $Y[4] = 1089$. Since $1089 < 1920$, add 1089.
9. Compare $X[5] = 1920$ and Y is empty, so add all remaining elements of X.

Merging Two Sorted Arrays - Step by Step

Final Merged Array:

$$[0, 2, 4, 5, 7, 9, 56, 1089, 1920, 2025, 2049]$$

Time Complexity Analysis of Merge Sort

Recursive Equation:

$$T(n) = T(n/2) + T(n/2) + S(n/2, n/2)$$

Expanding further:

$$= 2T(n/2) + a(n/2 + n/2)$$

$$= 2T(n/2) + an$$

Solving Recurrence using Recursion Tree:

- At level 0: $T(n) = 2T(n/2) + an$
- At level 1: $T(n/2) = 2T(n/4) + a(n/2)$
- At level 2: $T(n/4) = 2T(n/8) + a(n/4)$
- ...
- At level $\log_2 n$, base case: $T(1) = O(1)$

Final Complexity:

$$T(n) = O(n \log n)$$

Merge Function Complexity Analysis

Merge Function Complexity:

$$S(n_1, n_2) = a + \max(S(n_1 - 1, n_2), S(n_1, n_2 - 1))$$

Expanding step by step:

$$S(n_1 + n_2) = a + S(n_1 + n_2 - 1)$$

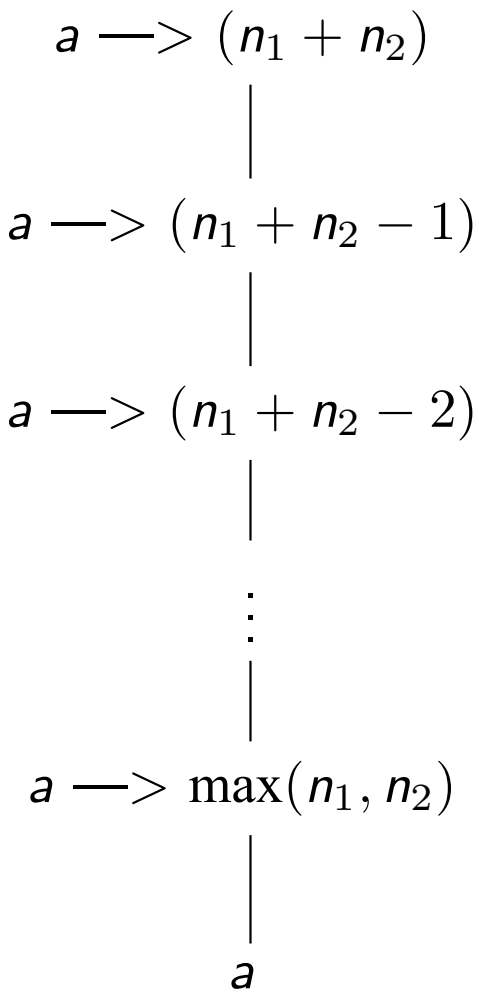
$$S(n_1 + n_2 - 1) = a + S(n_1 + n_2 - 2)$$

$$\vdots$$

$$S(1) = a$$

Merge Function Complexity Analysis

Complexity Tree:



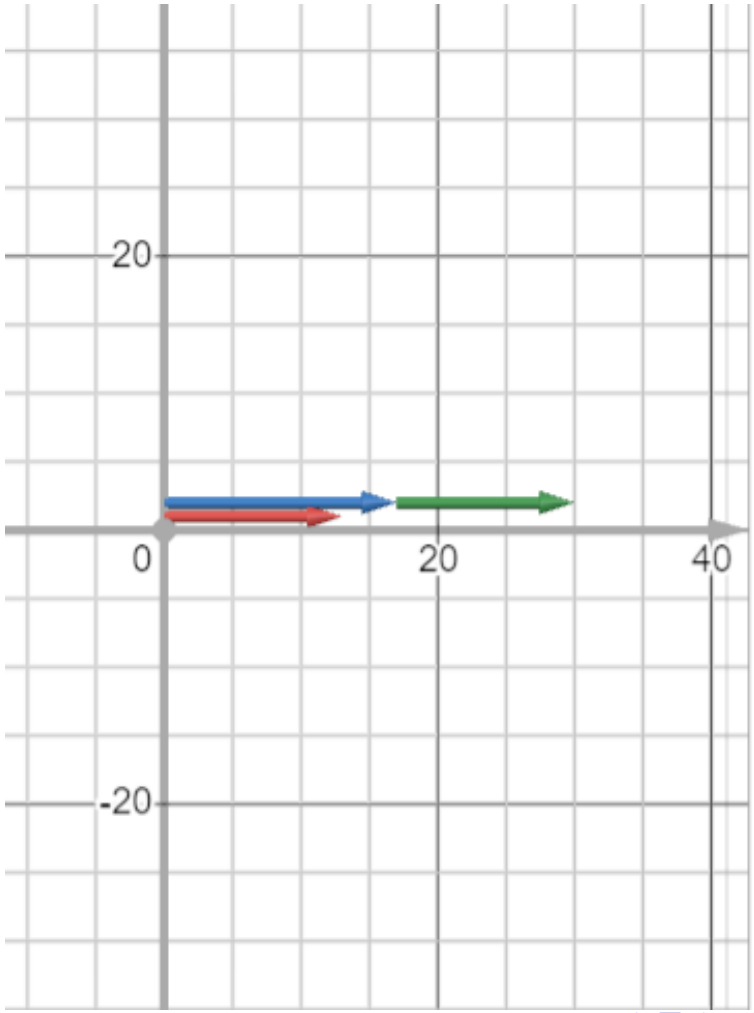
Merge Function Complexity Analysis

Conclusion:

$$\text{Total Cost} \leq a(n_1 + n_2) = O(n)$$

Algorithmic Addition

- **Problem:** Understanding vector addition in dimension 1
- Consider the number line representation:



Algorithmic Addition

→ Vector of magnitude 13
→ Vector of magnitude 17
→ Resultant vector of magnitude 30
Vector of magnitude 13 and 30 are of same length

Example: Addition Steps

- **Adding Numbers:** $13 + 17$

13	(n_1 digits)
17	(n_2 digits)
30	(with carry)

  - Step 1: $3 + 7 = 10$ (carry 1)
  - Step 2: $1 + 1 + 1$ (carried) $= 3$
  - Result: 30

Important Theorem

- **Theorem:** The sum of any three single-digit integers is always at most 2 digits long
- **Proof:**
  - Maximum single digit is 9
  - Maximum sum: $9 + 9 + 9 = 27$
  - Therefore, sum never exceeds 2 digits

Time Complexity Analysis

- **Algorithm:** Add(X,Y)
  Where $|X| = n1$ and $|Y| = n2$
- **Time Complexity:** $O(n_1 + n_2)$
  This is a linear algorithm
- **Optimality:**
  - Question: Does there exist a better algorithm?
  - Answer: No
  - Reason: Must examine each digit at least once
  - Mathematically:

$$E(T(n)) = \Theta(\max(n_1, n_2)) = \Theta(n_1 + n_2)$$

Standard Multiplication Algorithm

- Consider multiplying two numbers:

13	(n_1 digits)
17	(n_2 digits)
91	(7×13)
13_	(1×13)
	221

Multiplication Time Complexity

- **Time Complexity Analysis:**
  - For each digit in n2, multiply with all digits in n1
  - Then add the partial products
  - Time Complexity: $O(n_1 \times n_2)$
  - For numbers of similar size (n1 n2 = n):

$$T(n) = cn_1 n_2 = \Theta(n^2)$$

- **Optimality Question:**
  - Is this the fastest possible algorithm?
  - Answer: No
  - Reason: More efficient algorithms exist.

Summary: Asymptotic Notations

- **Three Fundamental Rules:**
  - Rule 1: Runtime bounds (Upper, Lower, Tight)
  - Rule 2: Small inputs are irrelevant
  - Rule 3: Constants don't matter

- **Key Points:**
  - For integers: size $= \log(n)$ (bits required)
  - Bounds valid for $n > n_0$
  - If $T(n)$ bounded by $F(n)$, then $cT(n)$ also bounded

Summary: Quick Sort Implementations

Three Versions:

- **QS1:** Basic with rank
  - $O(n^2)$ complexity
  - Uses rank comparisons
- **QS2:** Improved with partitioning
  - $O(n \log n)$ average case
  - Extra memory for partitions
- **In-Place QS:**
  - $O(1)$ extra space
  - $O(n \log n)$ average case
  - Randomized pivot selection

Summary: Merge Sort

Algorithm Overview

1. Divide array into halves
2. Recursively sort each half
3. Merge sorted halves

Complexity Analysis

- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Final Complexity: $O(n \log n)$
- Merge Function: $S(n_1, n_2) = O(n_1 + n_2)$

Summary: Elementary Operations

- **Addition:**
  - Time Complexity: $O(n_1 + n_2)$
  - Proven optimal - must examine each digit
- **Multiplication:**
  - Standard: $O(n_1 \times n_2)$

Key Theorems and Results

Important Theorems:

- Sum of three single-digit integers $\leq$ 2 digits
- Randomized QS pivot expected between $\frac{n}{4}$ and $\frac{3n}{4}$

Critical Insights

- Quick Sort randomization improves average case
- Merge Sort always guarantees $O(n \log n)$
- Arithmetic operations can be optimized beyond naive approaches