PDSA Notes

by

Gagneet Kaur

Date January 3, 2022

PAGE NO.
DATE:

WEEK 2

Analysis of Algorithms

→ MEASURING PERFORMANCE

- Examples of validating SIM cards against Aadhaar data
  - → Naive approach takes thousands of years
  - → Smarter solution takes a few minutes

- Two main resources of interest
  - → Running time - how long the algorithm takes
  - → Space - memory requirement

- Time depends on processing power
  - → impossible to change for given hardware
  - → enhancing hardware has only a limited impact at a practical level

- Storage is limited by available memory
  - → Easier to configure / augment

- Typically, we focus on time rather than space

INPUT SIZE

- Running time depends on input size
  - → larger arrays will take longer to sort
- Measure time efficiency as function of input size
  - → Input size (n)
  - → Running time [t(n)]
- Different inputs of size n may take different amounts of time
  - → We will return to this point later

EXAMPLE 1: SIM cards vs Aadhaar cards

→ $n \approx 10^9$ - no. of cards

→ Naive Algorithm : $t(n) \approx n^2$

→ Clever Algorithm : $t(n) \approx n \log_2 n$

→ $\log_2 n$ - no. of times you need to divide n by 2 to reach 1

→ $\log_2(n) = k \Rightarrow n = 2^k$

EXAMPLE 2 : Video Game

→ Several objects on screen

→ Basic step : find closest ~~step~~ pair of objects

→ n objects - naive algorithm is n^2

- for each pair of objects, compute their distance
- Report minimum distance across all pairs

→ There is a clever algorithm that takes time $n \log_2 n$

Shrisoft
PAGE NO.
DATE:

- High resolution gaming console may have 4000 X 2000 pixels
  - → 8×10^6 points — 8 million
- Suppose we have 100,000 = 1×10^5 objects
- Naive algorithm takes 10^{10} steps
  - 1000 seconds, or 16.7 minutes in Python
  - Unacceptable response time!
- $\log_2$ 100,000 is under 20, so $n \log_2 n$ takes a fraction of a second.

ORDERS OF MAGNITUDE

- When comparing $t(n)$, focus on orders of magnitude
  - → ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$
  - → for small values of n, $f(n) < g(n)$
  - → After $n = 5000$, $f(n)$ overtakes $g(n)$
- Asymptotic complexity
  - → What happens in the limit, as n becomes large
- Typical growth functions
  - → Is $t(n)$ proportional to $\log n, \ldots, n^2, n^3, \ldots, 2^n$?
    - → Note: $\log n$ means $\log_2 n$ by default
  - → Logarithmic, polynomial, exponential ...

Input Size	Values of $t(n)$						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
10^2	6.6	10^2	66	10^4	10^6	10^{30}	10^{157}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7	10^{12}			
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

MEASURING RUNNING TIME

- Analysis should be independent of the underlying hardware
  - Don't use actual time
  - Measure in terms of basic operations
- Typical basic operations
  - Compare two values
  - Assign a value to a variable
- Exchange a pair of values!

  $(x, y) = (y, x)$

  $t = x$
  $x = y$
  $y = t$

Shriansh
PAGE NO.
DATE:

- If we ignore constants, focus on orders of magnitude, both are within a factor of 3
- Need not to be very precise about defining basic operations

WHAT IS THE INPUT SIZE?

- Typically a natural parameter
  - Size of a list/array that we want to search or sort
  - No. of objects we want to rearrange
  - No. of vertices & no. of edges in a graph
    - We shall see why these are separate parameters
- What about numeric problems? Is 'n' a prime?
  - Magnitude of 'n' is not the correct measure
  - Arithmetic operations are performed digit by digit
    - addition with carry, subtraction with borrow, multiplication, ~~with~~ long division
  - No. of digits is a natural measure of input size
    - same as $\log_b n$, when we write n in base b

WHICH INPUTS SHOULD WE CONSIDER?

- Performance varies across input instances
  - By luck, the value we are searching for is the first element we examine in an array
- Ideally, want the 'average' behaviour
  - Difficult to compute

→ Average over what? Are all inputs equally likely?
→ Need a probability distribution over inputs

- Instead, worst case input
  - → Input that forces algorithm to take longest possible time
    - search for a value that is not present in an unsorted list
    - must scan all elements
  - → Pessimistic - worst case may be rare
  - → Upper bound for worst case guarantees good performance.

TO SUMMARIZE...

- Two important parameters when measuring algorithm performance : Time & space (we focus mainly on time)
- Running time $t(n)$ is a function of input size n
  - → Interested in orders of magnitude
  - → Asymptotic complexity, as n becomes large
- From running time, we can estimate feasible input sizes
- We focus on worst case inputs
  - → Pessimistic, but easier to calculate than avg case
  - → Upper bound on worst case gives us an overall guarantee on performance

Date
January 4, 2022

PAGE NO.
DATE:

Comparing Orders of Magnitude

ORDERS OF MAGNITUDE

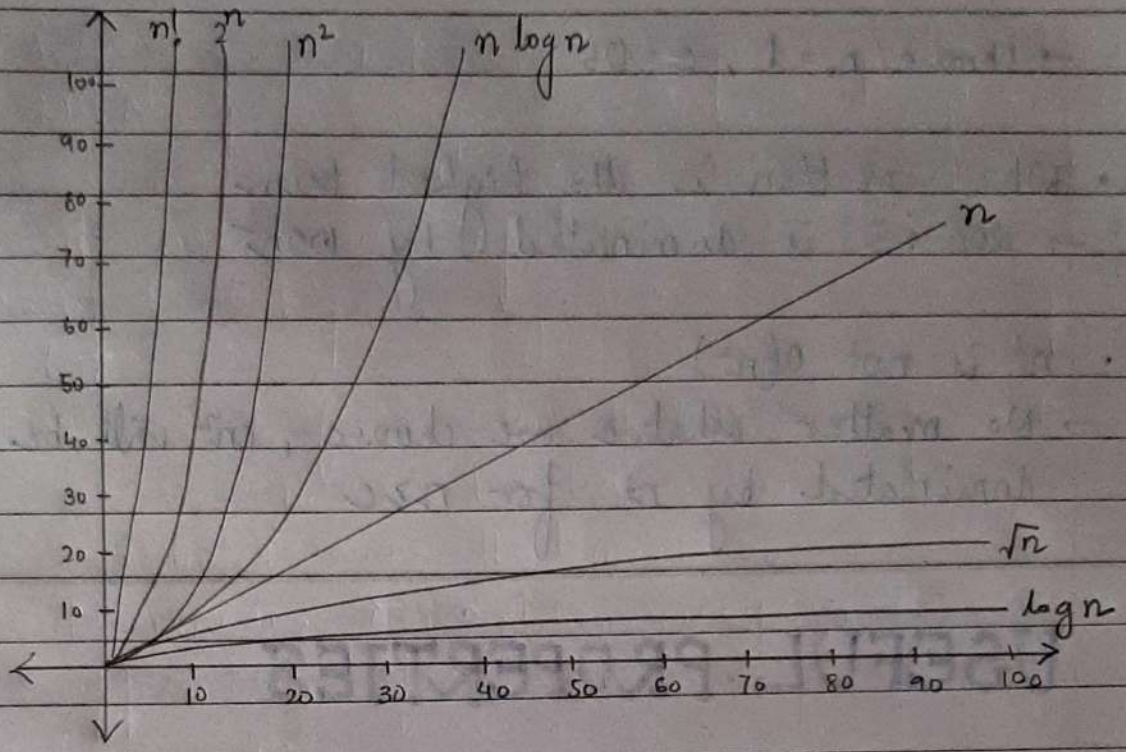
- When comparing $t(n)$, focus on orders of magnitude
  - ignore constant factors
- $f(n) = n^2$ eventually grows faster than $g(n) = 5000\,n^2$
- How do we compare functions with respect to orders of magnitude?

UPPER BOUNDS

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0.
- $f(x) \leq c \cdot g(x)$ for every $x \geq x_0$
- Graphs of typical functions we have seen

Big O — theoretical definition of the complexity of an algorithm as a function of the size.

$O(n)$ (order of magnitude) of complexity → How the algorithm performs in worst case scenario?



EXAMPLES :

(1) $100n + 5$ is $O(n^2)$

→ $100n + 5 \leq 100n + n = 101n$, for $n \geq 5$

→ $101n < 101n^2$

→ Choose $n_0 = 5$, $c = 101$

- Alternatively,

→ $100n + 5 \leq 100n + 5n = 105n$, for $n \geq 1$

→ $105n \leq 105n^2$

→ Choose $n_0 = 1$, $c = 105$

- Choice of n_0, c not unique

(2) $100n^2 + 20n + 5$ is $O(n^2)$

→ $100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$, for $n \geq 1$

→ $100n^2 + 20n + 5 \leq 125n^2$, for $n \geq 1$

→ choose $n_0 = 1$, $c = 125$

- What matters is the highest term
  → $20n + 5$ is dominated by $100n^2$

- n^3 is not $O(n^2)$
  → No matter what c we choose, cn^2 will be dominated by n^3 for $n \geq c$

USEFUL PROPERTIES

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.

- PROOF:
  → $f_1(n) \leq c_1 g_1(n)$, for $n \geq n_1$
  → $f_2(n) \leq c_2 g_2(n)$, for $n \geq n_2$
  → Let $c_3 = \max(c_1, c_2)$, $n_3 = \max(n_1, n_2)$
  → For $n \geq n_3$, $f_1(n) + f_2(n)$
  $\leq c_1 g_1(n) + c_2 g_2(n)$
  $\leq c_3 (g_1(n) + g_2(n))$
  $\leq 2 c_3 (\max(g_1(n), g_2(n)))$

- Algorithm has two phases
  → Phase A takes time $O(g_A(n))$
  → Phase B takes time $O(g_B(n))$

- Algorithm as a whole takes time $\max(O(g_A(n), g_B(n)))$

- Least efficient phase is upper bound for whole algorithm.

Shrinath
PAGE NO.
DATE:

LOWER BOUNDS

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0

  → $f(x) \geq c \cdot g(x)$ for every $x \geq x_0$

- n^3 is $\Omega(n^2)$

  → $n^3 > n^2$ for all n, so $n_0 = 1$, $c = 1$

- Typically we establish lower bounds for a problem rather than an individual algorithm

  → if we ~~start~~ sort a list by comparing elements and swapping them, we require $\Omega(n \log n)$ comparisons

  → This is independent of the algorithm we use for sorting

TIGHT BOUNDS

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$

  → find constants c_1, c_2, x_0 such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for every $x \geq x_0$

- $n(n-1)/2$ is $\Theta(n^2)$

  → Upper Bound

  $n(n-1)/2 = n^2/2 - n/2 \leq n^2/2$ for all $n \geq 0$

  → Lower Bound

  $n(n-1)/2 = n^2/2 - n/2 \gg n^2/2 - (n/2 \times n/2) \geq n^2/4$ for $n \geq 2$

  → Choose $n_0 = 2$, $c_1 = 1/4$, $c_2 = 1/2$

SUMMARY

- $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$
  - → Useful to describe asymptotic worst case running time

- $f(n)$ is $\Omega(g(n))$ means $g(n)$ is lower bound for $f(n)$
  - → Typically used for a problem as whole, rather than an individual algorithm

- $f(n)$ is $\Theta(g(n))$: matching upper & lower bounds
  - → We have found an optimal algorithm for a problem.

Date
January 5, 2022

PAGE NO.
DATE:

Calculating Complexity

Iterative Programs (Focus on loops)

Recursive Programs (write & solve as recurrence)

EXAMPLE 1

Find the maximum element in a list.

→ Input size is length of list
→ Single loop scans all elements
→ Always takes n steps
→ Overall time is O(n)

CODE →

```
def maxElement (L):
    maxval = L[0]
    for i in range (len(L)):
        if L[i] > maxval :
            maxval = L[i]
    return (maxval)
```

EXAMPLE 2

Check whether a list contains duplicates

→ Input size is length of the list
→ Nested loop scans all pairs of elements
→ A duplicate maybe found in the very first iteration
→ Worst Case : no duplicates, both loops run fully
→ Time is $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$
→ Overall time is $O(n^2)$

CODE →

```
def noDuplicates(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[i] == L[j]:
                return(False)
    return(True)
```

EXAMPLE 3

Matrix Multiplication

→ Matrix is represented as list of lists

- $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

- [[1,2,3],[4,5,6]]

→ Input matrices have size mxn , nxp

→ Output matrix is $m \times p$
→ Three nested loops
→ Overall time is $O(mnp) = O(n^3)$ if both are $n \times n$

CODE →

```
def matrixMultiply(A, B):
    (m, n, p) = (len(A), len(B), len(B[0]))

    C = [[

    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i][j] = C[i][j] + A[i][k] * B[k][j]

    return (C)
```

EXAMPLE 4

Number of bits in binary representation of n

→ $\log n$ steps for n to reach 1
→ For number theoretic problems, input size is no. of digits
→ The algorithm is linear in input size

CODE →

```
def numberOfBits(n):
    count = 1
    while n > 1:
        count = count + 1
        n = n // 2
    return (count)
```

PAGE NO.
DATE

EXAMPLE 5

Towers of Hanoi

- Three pegs A, B, C
- Move n disks from A to B, use C as transit peg
- Never put a larger disk on a smaller one

Recursive Solution

- Move (n-1) disks from A to C, use B as transit peg
- Move large disk from A to B
- Move (n-1) disks from C to B, use A as transit peg

Recurrence

- M(n) - no. of moves to transfer n disks
- M(1) = 1
- M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1

Unwind & Solve

$$
\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2)+1) + 1 = 2^2 M(n-2) + (2+1) \\
&= 2^2(2M(n-3)+1) + (2+1) = 2^3 M(n-3) + (4+2+1) \\
&\vdots \\
&= 2^k M(n-k) + (2^k - 1) \\
&\vdots \\
&= 2^{n-1} M(1) + (2^{n-1} - 1) \\
&= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1
\end{aligned}
$$

Searching in a List

SEARCH PROBLEM

- Is value v present in list l ?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when v is not present in l
- Worst case complexity is O(n).

CODE →

```
def naivesearch(v, l):
    for x in l:
        if v == x:
            return (True)
    return (False)
```

SEARCHING A SORTED LIST

- What if l is sorted in ascending order?
- Compare v with the mid point of l
  - → If midpoint is v, the value is found
  - → If v is less than midpoint, search the first half
  - → If v is greater than midpoint, search the second half
  - → Stop when the interval to search become empty.
- Binary Search

BINARY SEARCH

CODE :

```
def binarysearch (v, l) :
    if l == [] :
        return (False)

    m = len (l) // 2

    if v == l[m] :
        return (True)

    if v < l[m] :
        return ( binarysearch (v, l[:m]))
    else :
        return ( binarysearch (v, l[m+1 :]))
```

→ How long does it take ?
- Each call halves the internal to search
- Stop when the internal become empty

→ log n - no. of times to divide n by 2 to reach 1
- 1//2 = 0, so next call reaches empty interval

→ O (log n) steps (Logarithmic Time)

Alternative Calculation :

→ T(n) : the time to search a list of length n
- If n = 0, we exit, so T(n) = 1

- If $n>0$, $T(n) = T(n//2) + 1$

- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n//2) + 1$, $n>0$

- Solve by "unwinding"

- $T(n) = T(n//2) + 1$
  $= (T(n//4) + 1) + 1 = T(n//2^2) + \overbrace{1+1}^{2}$
  $\vdots$
  $= T(n//2^k) + \underbrace{1 + \ldots + 1}_{k}$
  $= T(1) + k$, for $k = \log n$
  $= (T(0) + 1) + \log n$
  $= 2 + \log n$

SUMMARY

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list

- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time

- In a sorted list, we can determine that v is absent by examining just '$\log n$' values.

Date
January 6, 2022

Shrinath
PAGE NO.
DATE:

Selection Sort

SORTING A LIST

- Sorting a list makes many other computations easier
  - → Binary Search
  - → Finding the median
  - → Checking for duplicates
  - → Building a frequency table of values

- How do we sort a list ?

Example : You are the TA for a course
- → Instructor has a pile of evaluated exam papers
- → Papers in random order of marks
- → Your task is to arrange the papers in descending order of marks

STRATEGY 1 :

- Scan the entire pile and find the paper with min. marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - → Add the paper with next min. marks to the second pile each time
- Eventually, the new pile is sorted in descending order.

SELECTION SORT

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - → Swap the minimum element into the first position
  - → Swap the second minimum element into second position
  - → ....
- Eventually the list is rearranged in place in ascending order

CODE →

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos : position of minimum in L[i:]
        for j in range(i+1, n):
            if L[j] < L[mpos]:
                mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i], L[mpos]) = (L[mpos], L[i])
        # now L[:i+1] is sorted
    return(L)
```

Shrinidhi
PAGE NO.
DATE:

ANALYSIS OF SELECTION SORT

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates n times
  - Inner loop : $n-i$ steps to find minimum in $L[i:]$
  - $T(n) = n + (n-1) + \ldots + 1$
  - $T(n) = n(n+1)/2$
- $T(n)$ is $O(n^2)$

→ SUMMARY ←

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list
- Worst case complexity is $O(n^2)$
  - Every input takes this much time
  - No advantage even if list is arranged carefully before sorting

---

Insertion Sort

SORTING A LIST

Example : You are the TA for a course
- → Instructor has a pile of evaluated exam papers
- → Papers in random order of marks
- → Your task is to arrange the papers in desc. order of marks

STRATEGY 2 :

- Move the first paper to a new pile
- Second paper
  - → Lower marks than first paper? Place below first paper in new pile
  - → Higher marks than first paper? Place above the first paper in new pile
- Third paper
  - → Insert into correct position with respect to first two
- Do this for the remaining papers
  - → Insert each one into correct position in the second pile

INSERTION SORT

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - → Assume L[:i] is sorted
  - → Insert L[i] in L[:i]
- A recursive formulation
  - → Inductively sort L[:i]
  - → Insert L[i] in L[:i]

CODE →

```
def Insert (L,v):
    n = len(L)
    if n==0:
        return ([v])
    if v>=L[-1]:
        return (L+[v])
    else:
        return (Insert(L[:-1],v) + L[-1:])

def ISort(L):
    n = len(L)
    if n<1:
        return(L)
    L= Insert(ISort(L[:-1]),L[-1])
    return(L)
```

(Recursive Insertion Sort)

ANALYSIS OF INSERTION SORT

(1) Iterative Insertion Sort

→ Correctness follows from the invariant
→ Efficiency
- Outer loop iterates n times
- Inner loop : i steps to insert L[i] in L[:i]
- $T(n) = 0+1+\ldots+(n-1)$
- $T(n) = n(n-1)/2$

→ $T(n)$ is $O(n^2)$

CODE:
```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return (L)
    for i in range (n):
        j = i
        while (L[j] < L[j-1]):
            (L[j], L[j-1]) = (L[j-1], L[j])
            j = j-1
    return (L)
```

(2) Recursive Insertion Sort

→ For input size of n, let
- TI(n) be the time taken by Insert
- TS(n) be the time taken by ISort

→ First calculate $TI(n)$ for 'Insert'
- $TI(0) = 1$
- $TI(n) = TI(n-1) + 1$
- Unwind to get $TI(n) = n$

→ Setup a recurrence for $TS(n)$
- $TS(0) = 1$
- $TS(n) = TS(n-1) + TI(n-1)$

→ Unwind to get $1+2+\ldots+(n-1)$

SUMMARY

- Insertion Sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list
- Worst case complexity is $O(n^2)$
  - Unlike selection sort, not all cases take time n^2
  - If list is already sorted, Insert stops in step 1
  - Overall time can be close to $O(n)$

Merge Sort

BEATING THE $O(n^2)$ BARRIER

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

STRATEGY 3 :

→ Divide the list into two halves
→ Separately sort the left and right half
→ Combine the two sorted halves to get a fully sorted list

COMBINING TWO SORTED LISTS

- Combining two sorted lists A and B into a single sorted list C
  → Compare first elements of A and B
  → Move the smaller of the two to C
  → Repeat till you exhaust A & B
- Merging A and B.

Shrinath
PAGE NO.
DATE:

MERGE SORT

- Let n be the length of L
- Sort A[: n//2]
- Sort A[n//2 :]
- Merge the sorted halves into B
- How do we sort A[: n//2] and A[n//2 :] ?
  → Recursively, same strategy !

DIVIDE AND CONQUER
→ Break up the problem into disjoint parts
→ Solve each part separately
→ Combine the solutions efficiently

MERGING SORTED LISTS

- Combine two sorted lists A & B into C
  → If A is empty, copy B into C
  → if B is empty, copy A into C
  → Otherwise compare first elements of A & B
    → Move the smaller of the two to C
  → Repeat till all elements of A & B have been moved

CODE →

```
def merge(A,B):
    (m,n) = (len(A), len(B))
    (C,i,j,k) = ([],0,0,0)
    while k < m+n:
        if i == m:
            C.extend(B[j:])
            k = k + (n-j)
```

```
        elif j == n :
            C.extend ( A[i:] )
            k = k + (m-i)
        elif A[i] < B[j] :
            C.append (A[i])
            (i, k) = (i+1, k+1)
        else :
            C.append (B[j])
            (j, k) = (j+1, k+1)
    return (C)
```

- To sort A into B, both of length n
- If $n \leq 1$, nothing to be done
- otherwise
  - → sort A[:n//2] into L
  - → sort A[n//2:] into R
  - → Merge L and R into B

CODE →

```
def mergesort (A) :
    n = len (A)
    if n <= 1 :
        return (A)
    L = mergesort ( A[ :n//2] )
    R = mergesort ( A[n//2 :] )
    B = merge (L, R)
    return (B)
```

ANALYSIS OF MERGE SORT

(1) Analysing Merge

→ Merge A of length m, B of length n

→ Output list C has length $m+n$

→ In each iteration we add (at least) one element to C

→ Hence 'merge' takes time $O(m+n)$

→ Recall that $m+n \leq 2(\max(m,n))$

→ If $m \approx n$, 'merge' takes time $O(n)$

(2) Analysing Merge Sort

→ Let $T(n)$ be the time taken for input of size n
- For simplicity, assume $n = 2^k$ for some k

→ Recurrence
- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$
  - Solve 2 subproblems of size $n/2$
  - Merge the solutions in time $n/2 + n/2 = n$

→ Unwind the recurrence to solve

→ Recurrence
- $T(0) = T(1) = 1$
- $T(n) = 2T(n/2) + n$

→ $T(n) = 2T(n/2) + n$

$= 2[2T(n/4) + n/2] + n$

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

$= 2^2[2T(n/2^3) + n/2^2] + 2n = 2^3 T(n/2^3) + 3n$

$\vdots$

$= 2^k T(n/2^k) + kn$

→ When $k = \log n$, $T(n/2^k) = T(1) = 1$

→ $T(n) = 2^{\log n} T(1) + (\log n) n = n + n \log n$

→ Hence $T(n)$ is $O(n \log n)$

Merge needs to create a new list to hold the merged elements
→ No obvious way to efficiently merge 2 lists in place
→ Extra storage can be costly

Inherently recursive
→ Recursive calls and returns are expensive