Algorithm analysis

Primitive operations:

	Primitive Operation	Example
(1)	Assigning a value to a variable	x ←5
(2)	Following an object reference	x ←obj.pointer
(3)	Performing an arithmetic operation	x + y
(4)	Comparing two numbers or variables	x < y
(5)	Accessing a single element of an array by index	A[i]
(6)	Calling a method	mycalculator.sum()
(7)	Returning from a method	**return** result

ex 1

Ln #	Algorithm my_test-1	Cost	Times	Comments
1	`if (n > 0)`	c1 = 1	1	• 1 comparison
2	`  call print("value of n is", n);`	c2 = 1	1	• 1 method call
3	`  call my_test-1(n-1);`	c3 = 1	f(n-1)	• 1 method call (recursive)

Starting with the Recurrence Relation:

$T(n) = 2 + f(n-1)$

Substitute f(n-1) using the definition of f(n)

$T(n) = 2 + [2 + f((n-1)-1)] = 2 + [2 + f(n-2)] = 4 + f(n-2)$
$T(n) = 4 + [2 + f((n-2)-1)] = 4 + [2 + f(n-3)] = 6 + f(n-3)$
$T(n) = 6 + [2 + f((n-3)-1)] = 6 + [2 + f(n-4)] = 8 + f(n-4)$
$T(n) = 8 + [2 + f((n-4)-1)] = 8 + [2 + f(n-5)] = 10 + f(n-5)$
$T(n) = 10 + [2 + f((n-5)-1)] = 10 + [2 + f(n-6)] = 12 + f(n-6)$
$T(n) = 12 + [2 + f((n-6)-1)] = 12 + [2 + f(n-7)] = 14 + f(n-7)$
(...)

Continue for *k* times...

$f(n) = (2 * k) + f(n-k)$

... until we reach the **BASE** case.

In the BASE case, let us assume $n - k = 0 \therefore n = k$

$f(n) = (2 * n) + f(n-n) = (2 * n) + f(0) = (2 * n) + 1$

$f(n) = 2n + 1$

just sub n with n-1

Recursion

Pros

1. **Clarity and Readability:** Recursive algorithms often mirror the problem's mathematical or structural definition, making them easier to understand.
2. **Simplifies Complex Problems:** Recursive solutions can simplify problems by breaking them down into smaller, more manageable subproblems.
3. **Natural for Tree and Graph Structures:** Recursive functions are well-suited for problems involving tree and graph structures.

Cons

1. **Memory Overhead**: Recursive calls consume memory for each function call, which can lead to a stack overflow for deep recursive calls.
2. **Performance Overhead**: Recursive calls generally have more function call overhead compared to loops, impacting performance.
3. **Potential for Infinite Recursion**: If not handled properly, recursion may lead to infinite loops.

When to Use Recursive Approach?

1. When the problem has a natural recursive structure – e.g., PA-1
2. When clarity and readability are essential.
3. When working with **tree** or **graph** structures.
4. When the problem involves *backtracking*
   - Backtracking is used to systematically explore possible solutions, making choices and backtracking when needed to efficiently search for a solution.

Iterative approach

Pros

1. **Efficiency**: Iterative solutions often have lower memory and runtime overhead compared to recursive solutions.
2. **Tail Optimization**:
   - Some languages and compilers can optimize *tail-recursive calls* into *iterative* form. The recursive call is the last operation performed by the function before returning its result. It is significant because it enables certain programming languages or compilers to optimize recursive functions into iterative loops, thereby avoiding **excessive stack usage** and **potential stack overflow errors** for deep recursion.
3. **Avoids Stack Overflow**: Since there is no function call overhead, iterative solutions are less prone to stack overflow errors due to deep recursion (i.e., many recursive calls).

Cons

1. **Complexity for Certain Problems:** Some problems might be more naturally expressed using recursion and could be less intuitive in an iterative form.
2. **Code Length:** In some cases, the iterative version of an algorithm may require more lines of code.

When to Use Iterative Approach?

1. When efficiency and performance are critical.
2. When the problem can be expressed more naturally using loops.
3. When there is a concern about stack overflow in deep recursive calls.

ex 2

Ln #	Algorithm my_test-1	Cost	Times	Comments
1	`if (n > 0)`	c1 = 1	1	• 1 comparison
2	`  call print("value of n is", n);`	c2 = 1	1	• 1 method call
3	`  call my_test-2(n/2);`	c3 = 1	f(n/2)	• 1 method call (recursive)

$f(n) = 2 + f(n/2) = (2 * 1) + f(n/2^1)$
$f(n) = 4 + f(n/4) = (2 * 2) + f(n/2^2)$
$f(n) = 6 + f(n/8) = (2 * 3) + f(n/2^3)$
$f(n) = 7 + f(n/16) = (2 * 4) + f(n/2^4)$
$f(n) = 10 + f(n/32) = (2 * 5) + f(n/2^5)$
$f(n) = 12 + f(n/64) = (2 * 6) + f(n/2^6)$
$f(n) = 14 + f(n/128) = (2 * 7) + f(n/2^7)$
$f(n) = 16 + f(n/256) = (2 * 8) + f(n/2^8)$
⋮

Continuing for *k* times, we have:

$f(n) = (2 * k) + f(n/2^k)$

... until we reach the **BASE** case, where we assume:

$n/2^k = 1 \therefore n = 2^k \rightarrow k = \log_2 n$

$f(n) = (2 * k) + f(n/2^k)$
$f(n) = (2 * k) + f(1)$
$f(n) = (2 * \log_2 n) + 1$

Reminder

examples:

If-statement

Line #	Algorithm	Cost	Times	Comments
1	`if (n < 0)`	c1 = 1	1	• One op: comparing two numbers
2	`  absval = -n;`	c2 = 2	1	• Two ops: assign + arithmetic ops
3	`else`	-	-	-
4	`  absval = n;`	c3 = 1	1	• One op: assign

while loop

Line #	Algorithm	Cost	Times	Comments
1	`i = 1;`	c1 = 1	1	• One op: assign
2	`sum = 0;`	c2 = 1	1	• One op: assign
3	`while (i <= n) {`	c3 = 1	n+1	Loop termination test: • a comparison i <= n each time • n successes and one failure
4	`  i = i + 1;`	c4 = 2	n	• Two ops per iteration (i.e., add, assign).
5	`  sum = sum + i;`	c5 = 2	n	• Two ops per iteration (i.e., add, assign).
6	`}`	-	-	-

$f(n) = (c1 * 1) + \max((c2 * 1), (c3 * 1))$
$f(n) = c1 + \max(c2, c3)$
$f(n) = 1 + \max(2, 1) = 1 + 2 = 3$

* take max

Line #	Algorithm	Cost	Times	Comments
1	`i = 1;`	c1 = 1	1	One op: assign
2	`sum = 0;`	c2 = 1	1	One op: assign
3	`while (i <= n) {`	c3 = 1	n+1	Loop termination test: • a comparison i <= n each time; • n successes and one failure
4	`  j = 1;`	c4 = 1	n	One op per iteration (i.e., assign).
5	`  while (j <= n) {`	c5 = 1	n * (n+1)	Loop termination test: • a comparison i <= n each time; • n successes and one failure
6	`  sum = sum + i;`	c6 = 2	n * n	Two ops per iteration (i.e., add, assign).
7	`  j = j + 1;`	c7 = 2	n * n	Two ops per iteration (i.e., add, assign).
8	`  }`	-	-	-
9	`  i = i + 1;`	c8 = 2	n	Two ops per iteration (i.e., add, assign).
10	`}`	-	-	-

* term will always be 1 greater than inc

for loop

Line #	Algorithm	Cost	Times	Comments
1	`prod ← 0`	c1 = 1	1	• Assigning a value
2	`for i ← 0 to n-1 do`	c2 = 1	1	Loop initializing: assigning a value
		c3 = 2	n	Loop incrementation: • Two ops: an addition and an assignment • done n times
		c4 = 1	n + 1	Loop termination test: • a comparison i < n each time • n successes and one failure
3	`prod ← prod + A[i]*B[i]`	c5 = 5	n	• Five ops per iteration (i.e., mult, add, 2 array refs, assign). • executed n times
4	`return prod`	c6 = 1	1	• One primitive op: returning from a method

i = 0

Line #	Algorithm	Cost	Times	Comments
1	`prod ← A[0]*B[0]`	c1 = 4	1	Four ops: mult, 2 array refs, assign
2	`for i ← 1 to n-1 do`	c2 = 1	1	Loop initializing (assigning a value).
		c3 = 2	n-1	Loop incrementation: • Two ops: an addition and an assignment • done (n-1) times
		c4 = 1	n	Loop termination test: • a comparison i <= (n-1) each time • (n – 1) successes and one failure i.e., (n - 1) + 1 = n
3	`prod ← prod + A[i]*B[i]`	c5 = 5	n-1	• Five ops per iteration (i.e., mult, add, 2 array refs, assign). • executed (n – 1) times
4	`return prod`	c6 = 1	1	• One primitive operation

i = 1

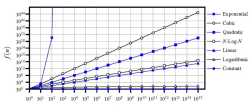
* always match inc with what's inside loop

Growth rates:

1. Comparing Growth Rates

Example #2: If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?

Solution:



* notice the division

Growth Rate	Time	
$f(n) = constant$	$T(n) = 1$	1 sec
$f(n) = \log_2 n$ (within constant factors)	$T(n) = (1 * \log_2 16)/\log_2 8 = 4/3$	1.33 sec
$f(n) = n$ (within constant factors)	$T(n) = (1 * 16)/8$	2 sec
$f(n) = n * \log_2 n$ (within constant factors)	$T(n) = (1 * 16 * \log_2 16)/8 * \log_2 8 = 8/3$	2.67 sec
$f(n) = n^2$ (within constant factors)	$T(n) = (1 * 16^2)/8^2$	4 sec
$f(n) = n^3$ (within constant factors)	$T(n) = (1 * 16^3)/8^3$	8 sec
$f(n) = 2^n$ (within constant factors)	$T(n) = (1 * 2^{16})/2^8 = 2^8$	256 sec = 4.27 min

Suppose it is known that the running time (i.e., the growth rate function f(n)) of an algorithm is (1/3)n² + 6n, and that the running time of another algorithm for solving the same problem is 111n – 312. Which one would you prefer, assuming all other factors equal?

Solution: We have:

- f(n) of algorithm #1 = (1/3)n² + 6n (i.e., proportional to quadratic time)
- f(n) of algorithm #2 = 111n – 312 (i.e., proportional to linear time)

To see for which input these two algorithms take the same time, we find **n** such that

f(n) of algorithm #1 = f(n) of algorithm #2 (i.e., intersection between the two functions)

(1/3)n² + 6n = 111n – 312

(1/3)n² + 6n – 111n + 312 = 0

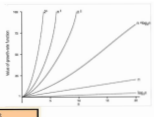
(1/3)n² – 105n + 312 = 0

Solving for n, the intersection between the two f(n)s occurs at n = 3 and n = 312.

```
if (n >= 3 AND n <= 312)
    CALL Alg-1; // Quadratic Growth Rate
else
    CALL Alg-2; // Linear Growth Rate
```

1. Comparing Growth Rates

Example #3: Now, assume we implement our algorithms in a computer that executes a million instructions a second. The chart below summarizes the amount of time required to execute f(n) instructions on this machine for various values of n (i.e., one thousand, one hundred thousand, and one million data items)



f(n)	n=10³	n=10⁵	n=10⁶
$\log_2(n)$	10^{-6} sec ①	$1.7 * 10^{-6}$ sec	$2 * 10^{-6}$ sec
n	10^{-3} sec	0.1 sec	1 sec
$n*\log_2(n)$	0.01 sec	1.7 sec	20 sec
n^2	1 sec	3 hr	12 days
n^3	17 min ②	32 yr	317 centuries
2^n	10^{285} centuries	10^{10000} years	10^{100000} years

① $\frac{\log_2(10^3)}{10^6}$

② $\frac{(10^3)^3}{10^6}$

ex of O(n)

f(n)	Growth Rate Analysis			Asymptotic Analysis		
	Time Units	Prop to	Rate	Big-Oh	c	$n \geq n_0 =$
$f(n) = 5n + 3$	$5n + 3$	$5n$	Linear	$O(n)$	8	1
$f(n) = 5n^2 + 3n\log n + 2n + 5$	$5n^2 + 3n\log n + 2n + 5$	$5n^2$	Quadratic	$O(n^2)$	15	1
$f(n) = 20n^3 + 10n\log n + 5$	$20n^3 + 10n\log n + 5$	$20n^3$	Cubic	$O(n^3)$	35	1
$f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$	$5n^4 + 3n^3 + 2n^2 + 4n + 1$	$5n^4$	4th Degree Polynomial	$O(n^4)$	15	1
$f(n) = 3\log n + 2$	$3\log n + 2$	$3\log n$	Logarithmic	$O(\log n)$	5	2
$f(n) = 2^{n+2}$	2^{n+2}	2^{n+2}	Exponential	$O(2^n)$	4	1
$f(n) = 2n + 100\log n$	$2n + 100\log n$	$2n$	Linear	$O(n)$	102	1

when n=1 log n = 0

why worst case > average case?

Average Case Analysis? i.e., average over all possible inputs of the same size.

- ✗ Typically, quite challenging.
- ✗ Requires a probability distribution on the set of inputs (difficult task)

Worst Case Analysis?

- ✓ Much easier than average-case analysis
- ✓ Identify the worst-case input, which is often simple.
- ✓ Typically leads to better algorithms.

  i.e., an algorithm performing well in the worst case requires it will do well on ***every*** input.

Sorting algorithms

Selection Sort:

- Search through entire array and find largest
- exchange largest with last spot
- Search through rest of unsorted array and repeat
- in place

worst case: $O(n^2)$

comparisons: $\frac{n^2}{2}$

exchanges: n (x2 in worst scenario)

Ln#	Algorithm	Cost	Times	Comments
1	for(i = arr.length-1; i > 0; i--){	c1 = 1	1	INIT: 1 assign + 1 access + 1 arith.
		c2 = 2	n-1	DEC: 1 assign + 1 arith.
		c3 = 1	n	TERM: comp. → n-1 success + 1 fail
2	pos_greatest = 0;	c4 = 1	n-1	1 assign exec. n-1 times (success)
3	for(j = 0; j <= i; j++){	c5 = 1	n-1	INIT: 1 assign → n-1
		c6 = 2	n-1 x n	INC: 1 assign + 1 arith.
		c7 = 1	n-1 x n+1	TERM: comp. → n success + 1 fail
4	if(arr[j] > arr[pos_greatest]){	c8 = 3	n-1 x n	1 comp. + 2 array access
5	pos_greatest = j;	c9 = 1	n-1 x n	1 assign
8	temp = arr[i];	c10 = 2	n-1	1 assign. + 1 array access
9	arr[i] = arr[pos_greatest];	c11 = 3	n-1	1 assign. + 2 array access
10	arr[pos_greatest] = temp;	c12 = 2	n-1	1 assign. + 1 array access

good for small input

good for fixed Na of elements

Bubble Sort:

- compare 1st and 2nd, 2nd and 3rd.....
- Flip when needed
- keep repeating until sorted

Ln#	Algorithm	Cost	Times	Comments
1	n = MAX - 1;	2	1	arithmetic + assign
2	k = n; sorted = FALSE;	2	1	assign
3	while ((k > 0) && (sorted == FALSE))	3	n+1	last term
4	{ sorted = TRUE;	1	n	assign
5	for (int j = 0; j < k; j++)	1	1 x n	loop init
		2	n x n	loop inc
		1	(n+1) n	loop terminate
6	{ if (list[j] > list[j+1])	4	n x n	
7	{ tmp = list[j];	2	n x n	
8	list[j] = list[j+1];	4	n x n	
9	list[j+1] = tmp;	3	n x n	
10	sorted = FALSE;	1	n x n	
13	k--;	2	n	

- Bubble up process repeated n-1 times
- has ability to stop when list is sorted

worst case: $O(n^2)$

comparisons: $\frac{n^2}{2}$

exchanges: $\frac{n^2}{2}$

good when data is nearly or mostly sorted

Insertion Sort:

- keeps making left side of array sorted until whole array is sorted
- shifts numbers
- in place sort, so doesn't require extra memory (requires constant amount of additional memory)
- stable sort
- very efficient when input is almost sorted (linear)

-low memory overhead

best
comparisons: N } already sorted
moves: 0

average:
comparisons: $N^2/4$
moves: $N^2/4$

worst:
comparisons: $N^2/2$
moves: $N^2/2$

Mergesort:

- Divide and conquer
  - cut the problem in half until the problem is trivial
  - solve for both halves
  - combine the solutions
  - Splits partitions in half
  - Merges smaller lists back into larger list
  - Requires overhead when sorting arrays
  - Use extra space
  - Is guaranteed to have O(n log n) performance in the worst case

$O(n \log n)$

- Mergesort is selected over Quick sort in real applications where
  - ✓stability,
  - ✓predictable performance, and
  - ✓worst-case time complexity

  are more important than space efficiency or cache performance.

1. External Sorting
2. Stable Sorting
3. Sorting Linked Lists
4. Applications with Predictable Performance

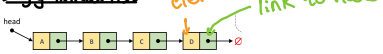
mergesort used where preserving the relative order of equal elements is necessary (comes from divide and conquer)

Merge Sort: Suitable for:
- External sorting when data is too large to fit into memory all at once.
- Stable sorting, preserving the relative order of equal elements.
- Sorting linked lists efficiently due to its divide-and-conquer approach.
- Applications requiring predictable performance, as it consistently performs at O(n log n) regardless of initial data order.
- Provides consistent performance crucial for real-time systems, financial trading systems, gaming applications, telecommunication systems, embedded systems, database systems, content delivery networks (CDNs), and high-performance computing (HPC) applications.
- Offers advantages in scenarios where stability, predictable performance, and worst-case time complexity are critical considerations.

Linear data structures:

singly linked list:

element, link to node



a data structure consisting of a sequence of nodes, linked to each other by pointers, starting from a head pointer.

-can grow and shrink dynamically at runtime

✓ Each link in a linked list is an object (also called an element, node, etc.)

✓ Each node holds a pointer (a reference, an address) to the location of the next node.

✓ The last link in a singly linked list points to null, indicating the end of the list.

PROS

- **Insertions and deletions are quick.**
- **Grows and shrinks as needed.**

CONS

- **Random access is slow.** Nodes in a linked list must be accessed sequentially (i.e., it can be slow to access a specific object).
- **Memory is a concern.** Each object in a singly linked list requires data (i.e., element) as well as one pointer (i.e., reference) to other nodes in the singly linked list. 1-D arrays use significantly less memory, each entry [i] in an array only requires memory to store its data.

When to Use (Singly) Linked Lists?

- You do not need random access to any specific node.
- You need to do constant insertions and deletions.
- You are not sure how many items will be in the list (i.e., dynamic data).

- **The heap is memory not used by the stack**
- **Dynamic variables live in the heap**
- **We need a pointers to access our list in the heap**

4 main operations

- insert at head
- remove head
- insert at tail
- remove tail

Insert at head

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node

remove tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node
- We need an auxiliary pointer (reference) to traverse the list from head until the node previous to the tail node

array methods

Methods	Description
`size()`	Return the number of elements in the list.
`isEmpty()`	Return true if the list is empty, otherwise return false.
`isFull()`	Return true if the list is full, otherwise return false.
`get(i)`	Return an element from the list at any given position i.
`set(i,e)`	Replace an element at any position i by another element e.
`add(i,e)`	Insert an element e at any position i of the list.
`remove(i)`	Remove the element at a specified location from a non-empty list.

stored consecutively in memory

An ***array*** is a sequenced collection of variables all of the same type.

Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell.

The cells of an array, A, are numbered 0, 1, 2, and so on.

Each value stored in an array is often called an ***element*** of that array.

good for large input

instability comes from partitioning - can be modified to be stable

Quicksort:

Pick a "pivot". Divide into less-than & greater-than pivot. Sort each side recursively. ↖ recur and conquer

- Relies on a pivot point for sorting
- Smaller sets are sorted based on pivot point
- Can perform slowly if a bad pivot point is used
- Does not use extra space
- Is not guaranteed to have O(n log n) performance in the worst case, unlike merge sort

also average

Best case: $O(n \log_2 n)$

worst case: $O(N^2)$ ← terrible pivot is picked

- faster than mergesort on average, especially when good pivot is picked

1. In-memory Sorting
2. Sorting Arrays with Random Data
3. Real-time Systems
4. Sorting Data with Limited Overhead
5. Sorting Large Datasets with Parallelism

- Quicksort is selected over Mergesort in real applications where
  - ✓memory usage optimization,
  - ✓cache performance,
  - ✓average-case time complexity,
  - ✓real-time processing requirements,
  - ✓simplicity, and
  - ✓parallel processing capabilities

  are critical considerations.

Quick Sort: Suitable for:
- Sorting data with limited overhead due to its minimal memory requirements and in-place partitioning.
- Parallel processing, as it can be parallelized effectively due to its divide-and-conquer nature.
- Chosen in real applications for memory usage optimization, cache performance, average-case time complexity, real-time processing requirements, simplicity, and parallel processing capabilities.
- Used in low-level programming, embedded systems development, network packet processing, database systems, graphics processing, financial services, genomics, and bioinformatics for efficient sorting of large datasets and real-time analysis.

Dynamic vs. Static

Static (fixed in size)

- **Sometimes we create data structures that are "fixed" and don't need to grow or shrink.**

Dynamic (change in size)

- **Other times, we want the ability to increase and decrease the size of our data structures to accommodate changing needs.**

Dynamic Data

- **Dynamic data refers to data structures which can grow and shrink to fit changing data requirements.**
- **We can allocate (create) additional dynamic variables whenever we need them.**
- **We can de-allocate (erase) dynamic variables whenever we are done with them.**
- **A key advantage of dynamic data is that we can always have a exactly the number of variables required - no more, no less.**
- **For example, with pointer variables to connect them, we can use dynamic data structures to create a chain of data structures called a linked list.**

Static Data

- **Static data is data declared "ahead of time."**
- **It is declared in a module (or main algorithm) and "lives" for as long as that module is active.**
- **If we declare more static variables than we need, we waste space.**
- **If we declare fewer static variables than we need, we are out of luck.**
- **Often, real world problems mean that we don't know how many variables to declare, as the number needed will change over time.**