Algorithms

Definition	unambiguous procedure executed in a finite number of steps
What makes a good algorithm?	Correctness, Speed, Space, Simplicity
Speed:	time it takes to solve problem
Space:	amount of memory required
Simplicity:	easy to understand, easy to implement, easy to debug, modify, update

Running Time

Definition	measurement of the speed of an algorithm	
Dependent variables:	size of input & content of input	
Best Case:	time on the easiest input of fixed size	meaningless
Average Case:	time on average input	good measure, hard to calculate

Running Time (cont)

Worst Case:	time on most difficult input	good for safety critical systems, easy to estimate

Proofs by Induction (Examples)

Claim: $for\ any\ n \geq 1, \quad 1+2+3+4+\cdots+n = \frac{n \cdot (n+1)}{2}$

Proof:

- Base case: $n = 1 \quad 1 = \frac{1 \cdot 2}{2}$ ✔
- Induction step:

$for\ any\ k \geq 1, \quad if \quad 1+2+3+4+\cdots+k = \frac{k \cdot (k+1)}{2}$

$then \quad 1+2+3+4+\cdots+k+(k+1) = \frac{(k+1) \cdot (k+2)}{2}$

Loop Invariants

Definition	loop property that holds before and after every iteration of a loop.
Steps:	
1. Initialization	If it is true prior to the iteration of the loop
2. Maintenance	If it is true before an iteration of the loop, it remains true before the next iteration
3. Termination	When the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct

QuickSort

Divide:	choose an element of the array for pivot
	divide into 3 sub-groups; those smaller, those larger and those equal to pivot
Conquer	recursively sort each group
Combine	concatenate the 3 lists

QuickSort

```
Algorithm partition(A, start, stop)
Input: An array A, indices start and stop.
Output: Returns an index j and rearranges the elements of A such that for all i<j, A[i] ! A[j] and for all k>i, A[k] " A[j].
pivot # A[stop]
left # start
right # stop - 1
while left ! right do
while left ! right and A[left] ! pivot) do
left # left + 1
 while (left ! right and A[right] " pivot)
do right # right -1
if (left < right )
then exchange A[left] $ A[right]
```

QuickSort (cont)

```
exchange A[stop] $ A[left]
return left
```

Time Complexities:
• Worse case:
– Already sorted array (either increasing or decreasing)
– T(n) = T(n-1) + c n + d
– T(n) is O(n2)
• Average case: If the array is in random order, the pivot splits the array in roughly equal parts, so the average running time is O(n log n)
• Advantage over mergeSort:
– constant hidden in O(n log n) are smaller for quickSort. Thus it is faster by a constant factor
– QuickSort is easy to do “in-place”

In Place Sorting

Definition:	Uses only a constant amount of memory in addition of that used to store the input
Importance:	Great for large data sets that take up large amounts of memory
Examples:	Selection Sort, Insertion Sort (Only moving elements around the array)
MergeSort:	Not in place: new array required

Object Orientated Programming

Definition:	User defined types to complement primitive types
	Called a class
Contains:	Data & methods
Static members:	members shared by all objects of the class

Recursion Programming

Definition	using methods that call themselves
Structure:	
base case	a simple occurrence that can be answered directly
recursive case	A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.

Divide & Conquer

Divide	the problem into sub problems that are similar to the original but smaller in size
Conquer	the sub-problems by solving them recursively. If they are small enough, solve them in a straightforward manner

Divide & Conquer (cont)

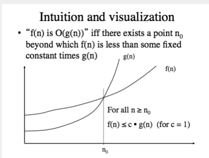
Combine	the solutions to create a solution to the original problem

BIG O Definition

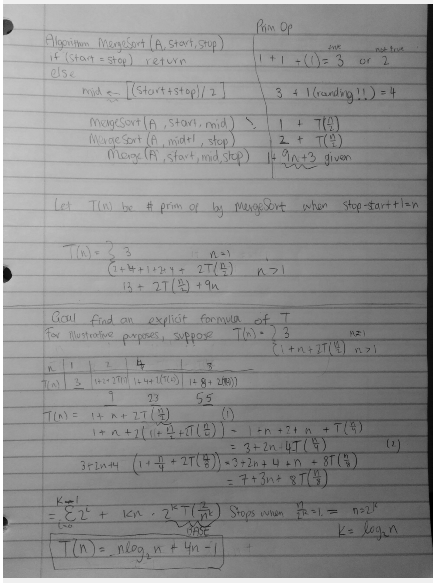
f(n) & g(n) are two non negative functions defined on the natural numbers N

f(n) is O(g(n)) if and only if:	there exists an integer n0 and a real number c such that ∀n>= n0 f(n) <= c * g(n)
	N.B. The constant c must not depend on n

Big O Visualization



Big O Recurrence



Sum of a^i from 0 to n = $(a^{(n+1)} - 1)/(a-1)$

Limitations of Arrays

Size has to be known in advance
memory required may be larger than number of elements
inserting or deleting an element takes up to O(n)

ADT: Queues

FIFO(First in first out)	
Any first come first serve service	
Operations	enqueue() - add to rear
	dequeue() - removes object at front
	front() - returns object at front
	size() - returns number of objects O(n)
	isEmpty() - returns true if empty

Double Ended Queues(deque): Allows for insertions and removals from front and back - By adding reference to previous node - removals occur in O(1)

ATD: Stacks

Def:	Operations allowed at only one end of the list (top)
	LIFO: (Last in first out)
Operations:	push() - inserts element at top
	pop() - removes object at top

ATD: Stacks (cont)

	top() - returns top element without removing it
	size() - returns number of elements
	isEmpty() - returns True if empty
Applications	page visited history in web browser
	JVM - keeps track of chain of active elements (allows for rec)
Perfor-mance:	space used: O(n)
	operations: O(1)
Limitations	max size must be defined prior
	pushing to a full stack causes implementation specific error

BinarySearch

```
BinarySearch(A[0..N-1], value) {
      low = 0
      high = N - 1
      while (low <= high) {
        mid = (low + high) / 2
          if (A[mid] > value)
              high = mid - 1
          else if (A[mid] < value)
              low = mid + 1
          else
              return mid
      }
```

BinarySearch (cont)

```
      return not_found
// value would be
inserted at index "low"
  }
```

Invariants:

```
value > A[i] for all i <
low value < A[i] for all
i > high
```

Worst case performance: O(log n)

Best case performance: O(1)

Average case performance: O(log n)

BinarySearch (Recursive)

```
int bsearch(int[] A, int
i, int j, int x) {
if (i<j) {
int e = [(i+j)/2];
if (A[e] > x) {
return bsearch(A,i,e-1);
} else if (A[e] < x) {
return
bsearch(A,e+1,j);
} else {
return e;
}
} else { return -1; }
}
```

Time Complexity: log(base2)(n)

Insertion Sort (Iterative)

```
For i ← 1 to length(A) -
1
 j ← i
 while j > 0 and A[j-1]
> A[j]
 swap A[j] and A[j-1]
 j ← j - 1
 end while
end for
```

Time complexity: $O(n^2)$

Merge-then-sort

```
Algorithm
ListIntersection (A,m,
B,n)
Input: Same as before
Output: Same as before
inter ← 0
Array C[m+n];
for i ← 0 to m-1 do C[i]
← A[i];
for i ← 0 to n-1 do C[
i+m ] ← B[i];
C ← sort( C, m+n );
ptr ← 0
while ( ptr < m+n-1 ) do
{
 if ( C[ptr] = C[ptr+1]
) then {
 inter ← inter+1
 ptr ← ptr+2
 }
 else ptr ← ptr+1
}
return inter
```

Time Complexity: (m+n) * (⌈log(m+n)⌉) + m + n -1

MergeSort (Recursive)

```
MergeSort (A, p, r) //
sort A[p..r] by divide &
conquer
if p < r
   then q ← ⌊(p+r)/2⌋
      MergeSort (A, p,
q)
      MergeSort (A, q+1,
r)
      Merge (A, p, q, r)
```

Time Complexity: 2T(n/2) + k • n + c’

Primitive Operations

assignment
calling method
returning from method
arithmetic
comparisons of primitive types
conditional
indexing into array
following object reference

Assume each primitive operation holds the same value = 1 primitive operation

Prove Big - Oh