Asymptotic Notation

OmG LoGs:

(b > 1, also possible for 0<b<1 but not b=1)

Space complexity:

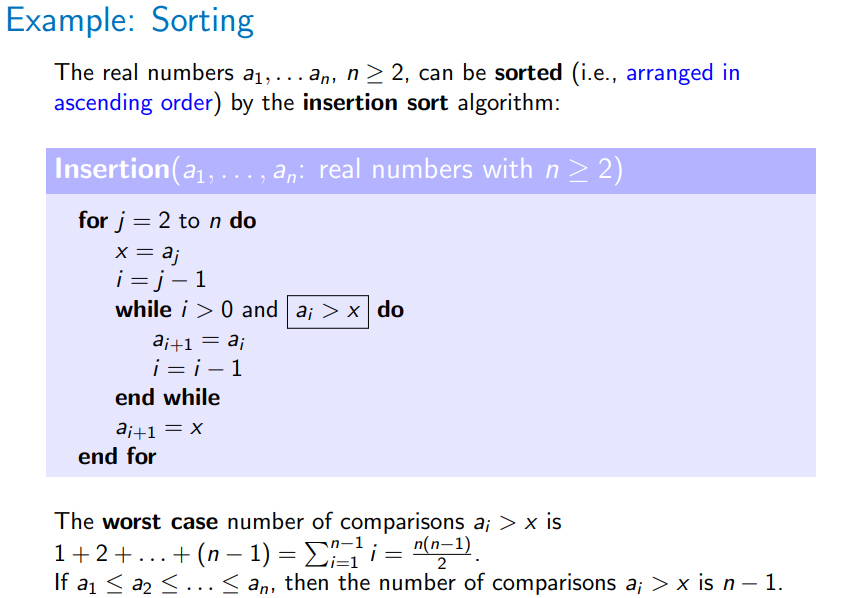
The space complexity of an algorithm is expressed in terms of the memory required by the algorithm for an input of a particular size.

(space complexity is important, but generally irrelevant ☹)

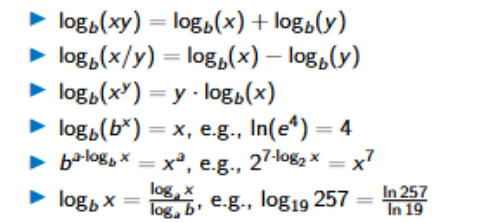
Time Complexity:

The time complexity of an algorithm can be expressed in terms of the number of basic operations used by the algorithm when the input has a particular size. Basic operations include addition, multiplication, assignments etc. (anything that can be done in constant time)

The worst-case time complexity of an algorithm can be expressed in terms of the largest number of basic operations used by the algorithm for a particular sized input. Usually, time complexity means worst-case time complexity.

Worst case time complexity is used as computing the exact number of operations is difficult, and so estimations for the upper bound for the worst-case time complexity are used instead. These give us the possibility to estimate the growth of number of operations as the input size increases.

a



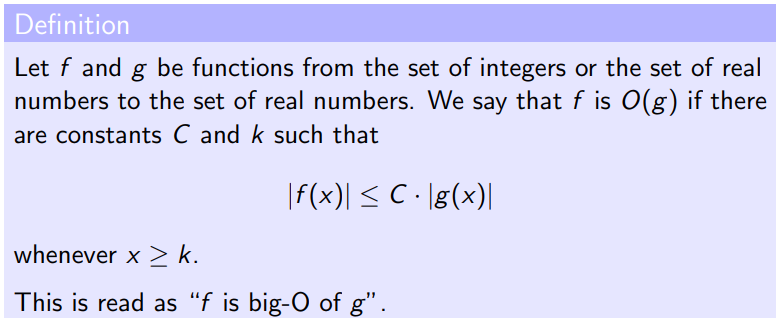
(this worst case occurs when the list is in reverse order)

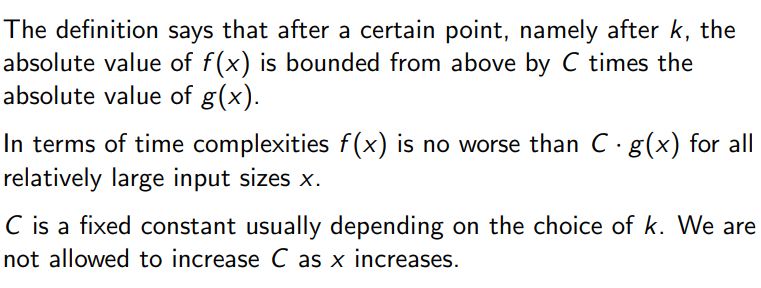
(n-1 is the best case, i.e. already sorted)

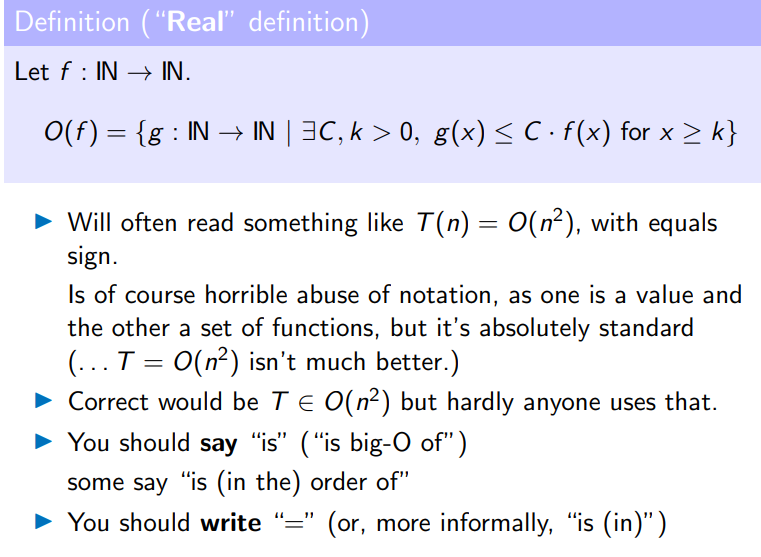
Average-case complexity analysis can also be used, where the average number of operations over all inputs is used.

It is difficult to compute the exact number of operations, so we usually use an upper bound for worst case analysis, and the estimation of the growth of the number of operations compared to the growth of the input size.

Big-O Notation

(this is for general functions, for time complexity will always be positive so don’t need || )

The constants C and k are “witnesses” to the relationship f(x) = O(g(x)). If a pair of witnesses exists we can establish that f(x) is O(g(x)). There are infinitely many pairs of C and k and the smallest value is not needed.



Show f(x) is O(g(x)):

F(x) = x2+2x+1, g(x) = x2

Need: x2+2x+1 <= C \* x2 for all x >= k

x2+2x+1

x2 <= x2 for all x

2x <= 2x2 for x >= 1

1 <= x2 for x >= 1

So f(x) <= 4x2  for x >= 1

C = 4, k = 1

Therefore x2+2x+1 is O(x2)

We use big-O notation as we don’t want to say a whole expression for the exact time complexity of an algorithm, instead we only care about the fastest-growing (most dominant) term, removing the coefficient of this term as it is less important.

Show 3x is not O(2x)

Proof by contradiction (uh oh):

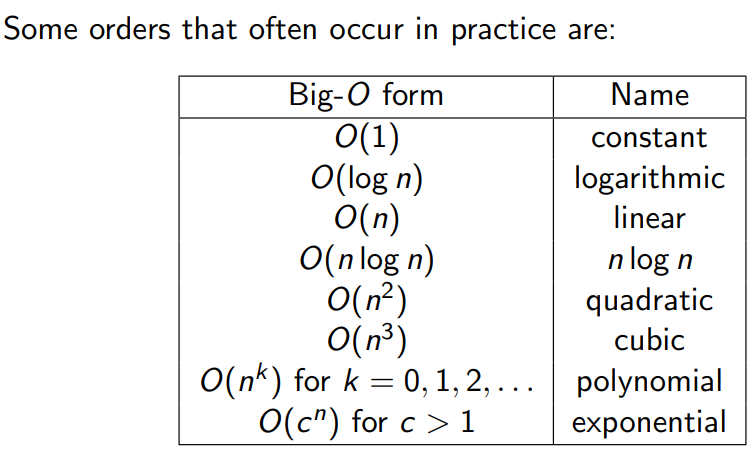
Assume 3x is O(2x), i.e. there exists C,k >0: 3x <= C\*2x for all x >= k

3x/2x <= C for all x >= k

1.5x <= C for all x >= k

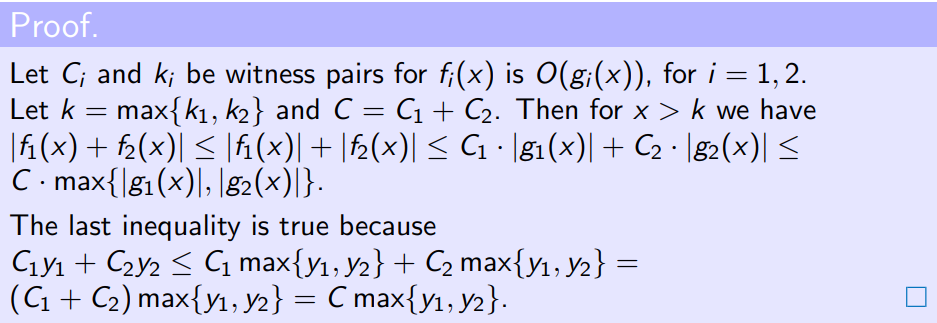
1.5x goes to infinity while C remains constant for increasing x, so cannot hold for all x >= k. This is a contradiction, so our assumption is false.

As a result, 3x is not O(2x).

Big-O Orders:

For O(logx) no base needs to be specified for the logarithms, since logax = logbx/logba and logba is a constant.

Sum rule: if functions f1(n) is O(g1(n)) and f2(n) is O(g2(n)), then f1(n) + f2(n) is O(max{g1(n),g2(n)})

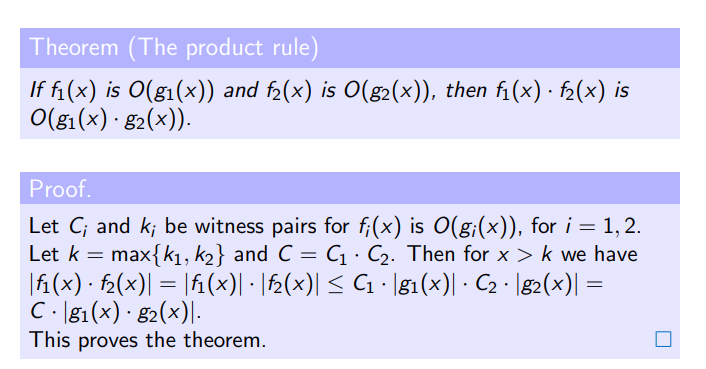
i.e. the maximum of the time complexities of f1 and f2. Used when one function runs after another.

Witness pair: C = C1+C2 , k = max{k1,k2}

Corollary: if both f1 and f2 are O(g), f1 + f2 is also O(g)

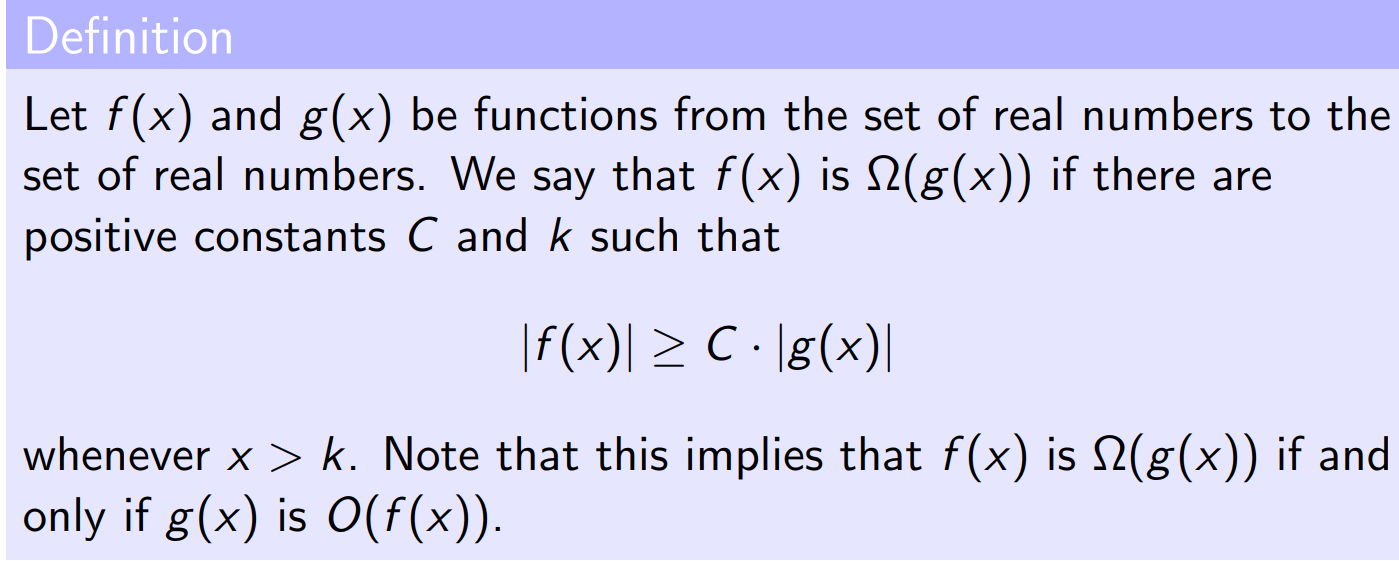
(The sum rule can only be applied a constant number of times)

^ i.e just multiply the time complexities. Used when one function iteratres through a block of code a set number of times.

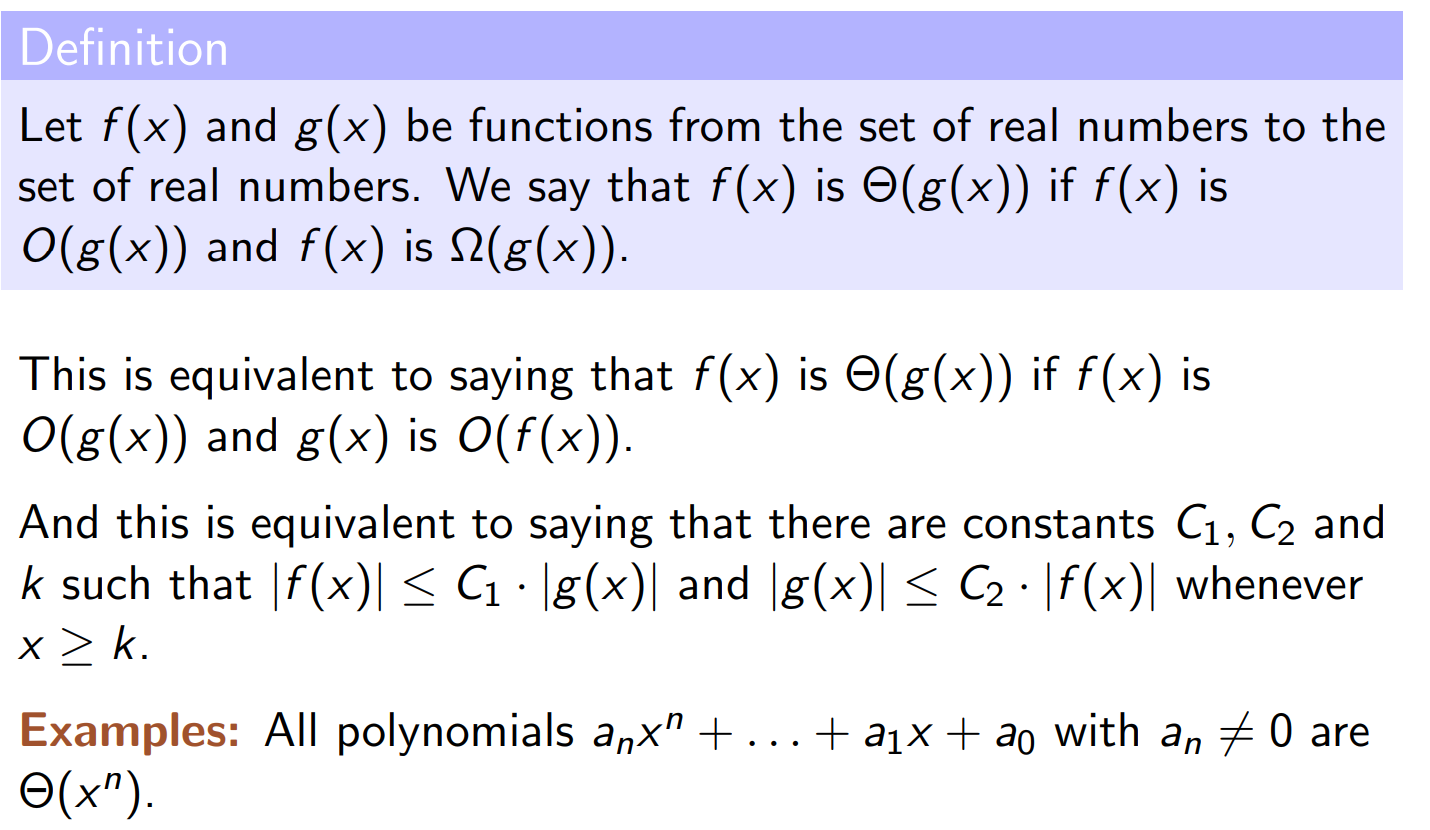
With big-O notation we lose information about the exact number of basic operations for a given n, but we gain a simple way of expressing how the time complexity grows with the input size. In most contexts, using big-O is more practical than expressing the exact time complexity.

Big-O notation is used to find a reasonable upper bound for growth rates, but does not help if we want the best function to match the growth rate. A similar definition for lower bounds is called Big-Omega notation.

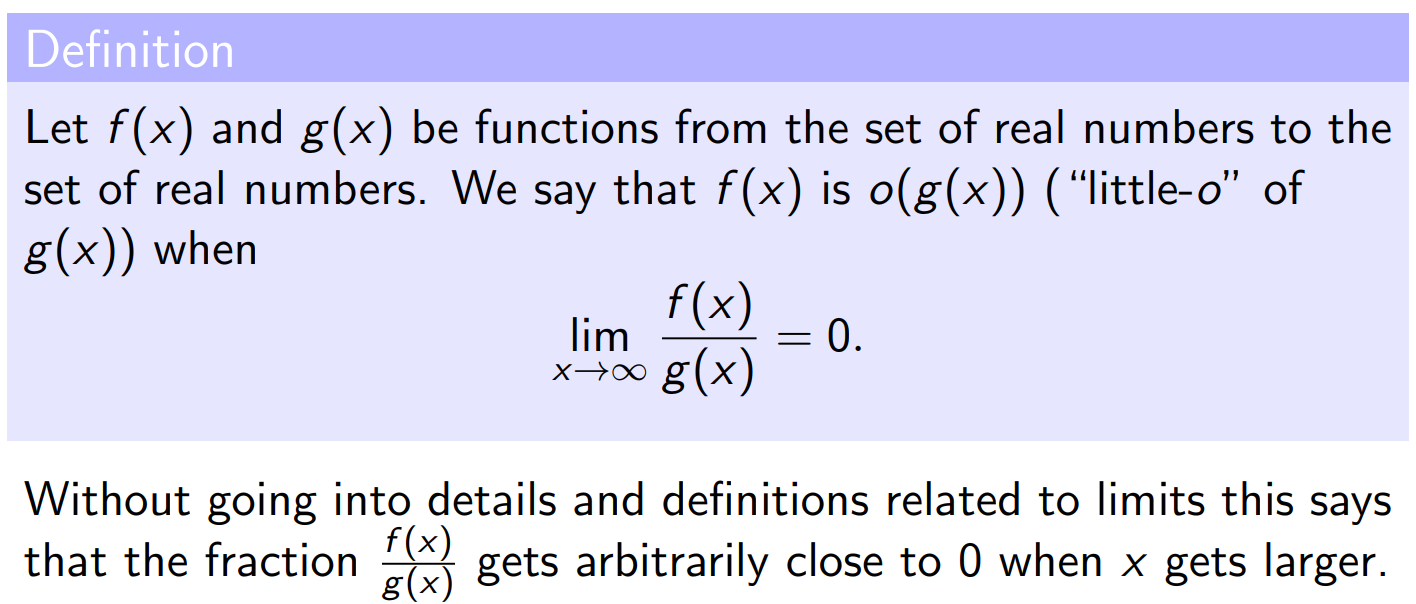
(Prove f is big omega of g by proving g is big O of f)

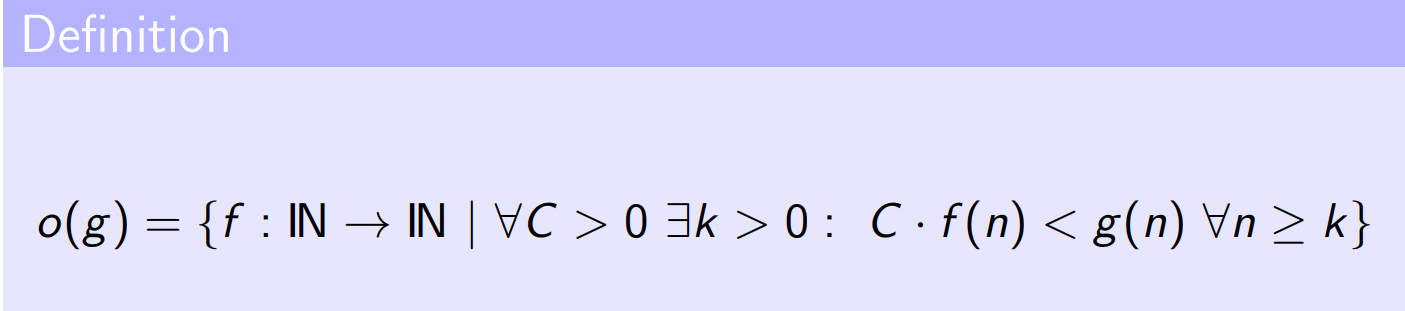


Same order growth rates:

^This is Theta.

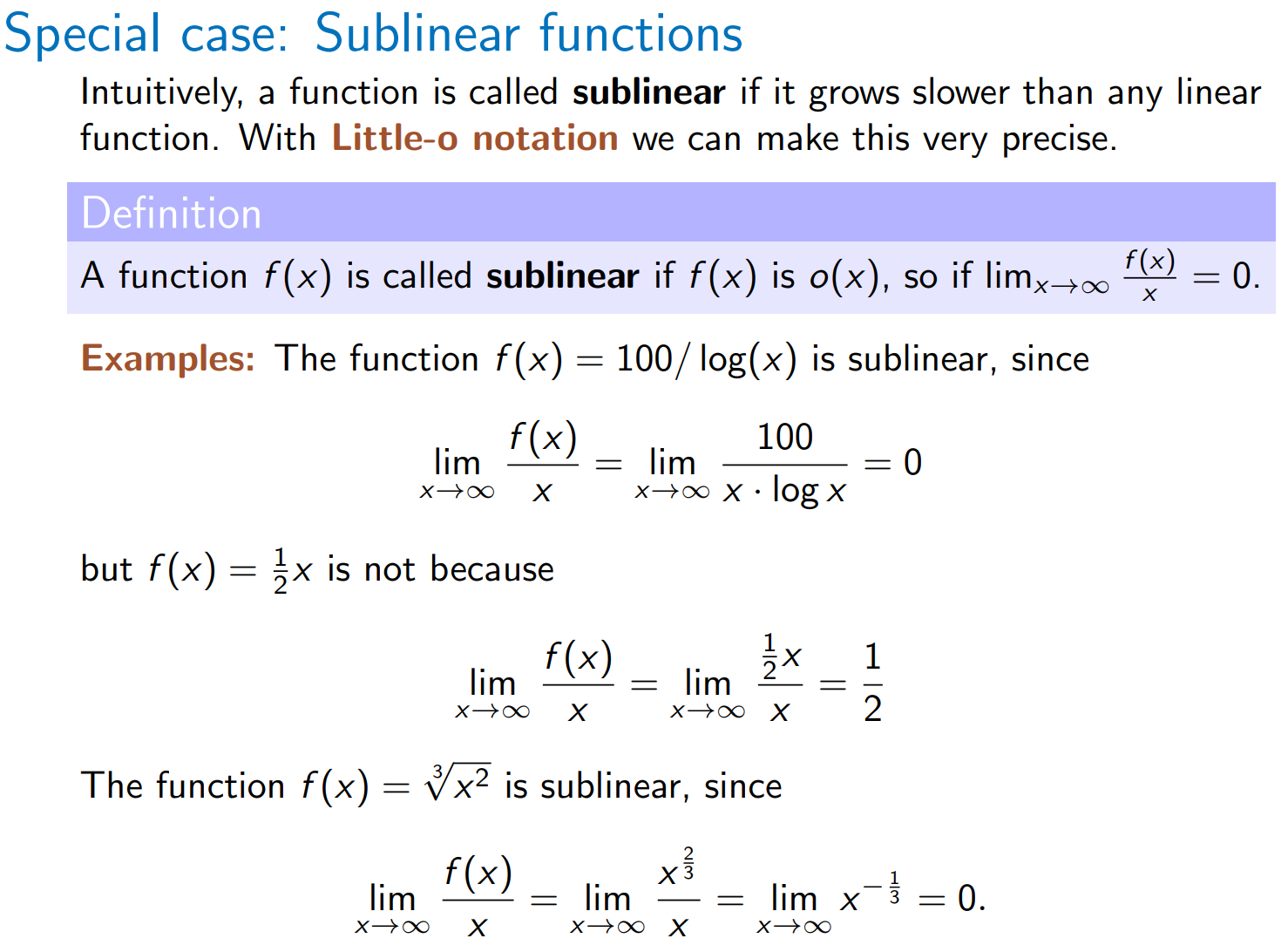
Little-o notation gives us a tool for neglecting or disregarding “smaller order” terms.

(uh oh)

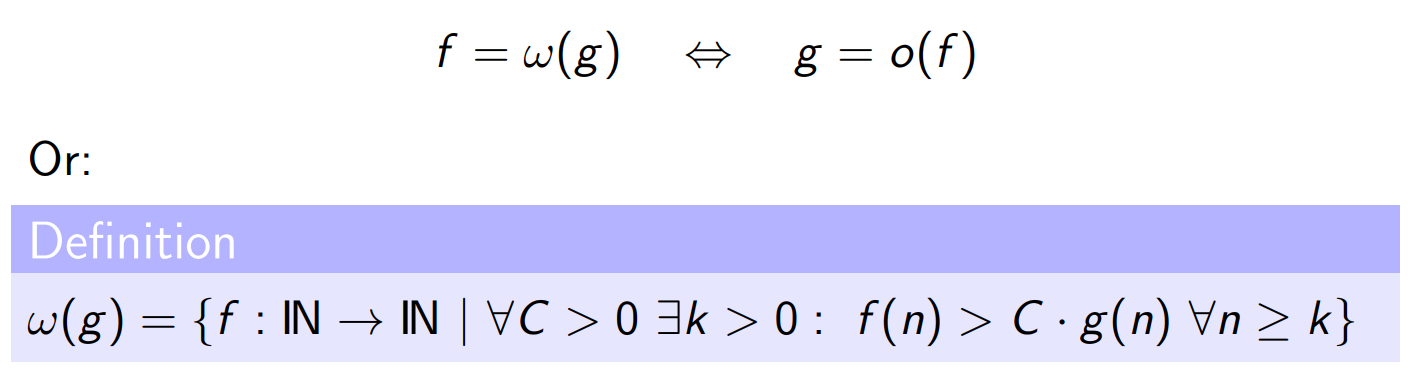
^Upside down A means “for all”

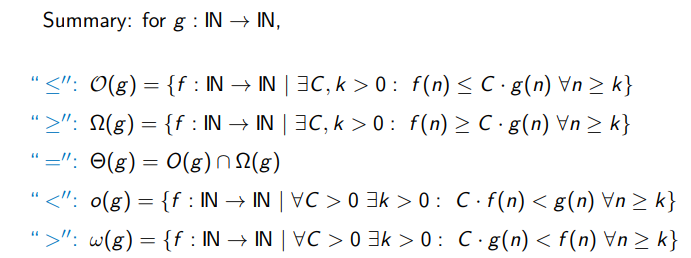
If f(x) is o(g(x)), then f(x) is also O(g(x)). If f(x) is not O(g(x)) then f(x) > C \* g(x) for all C and k and for at least one value of x, so the limit as x -> infinity will not go to zero, meaning f(x) is not o(g(x))

Example:



Little omega is to o what big-omega is to big-O.

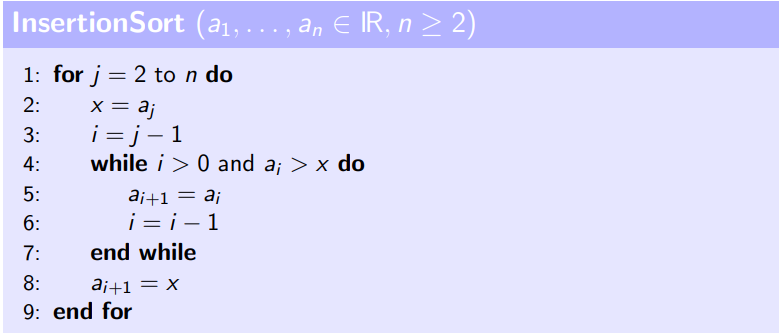
(prove that f is little omega of g by proving g is little o of f using limits)

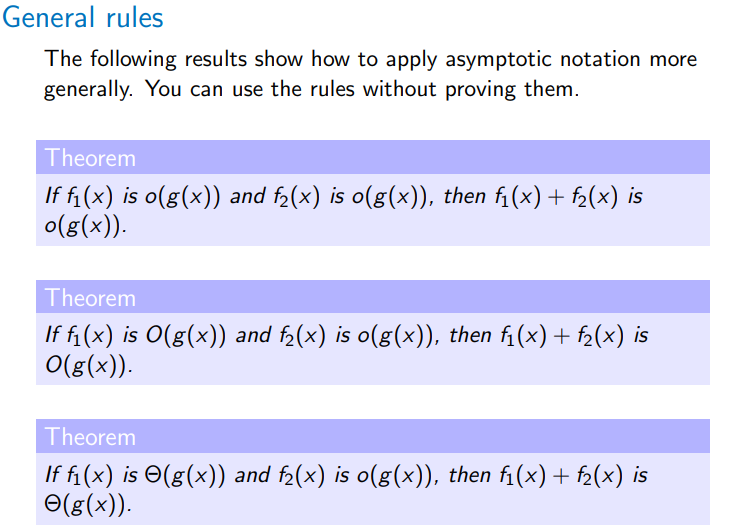
a

Sorting Algorithms

We will see three iterative algorithms – Insertion sort, Bubble sort and Selection sort, as well as two recursive algorithms – Merge sort and Quicksort. We will also see randomised Quicksort.

Insertion sort:



When j has a certain value, it inserts the jth element into an already sorted sequence, yielding sorted

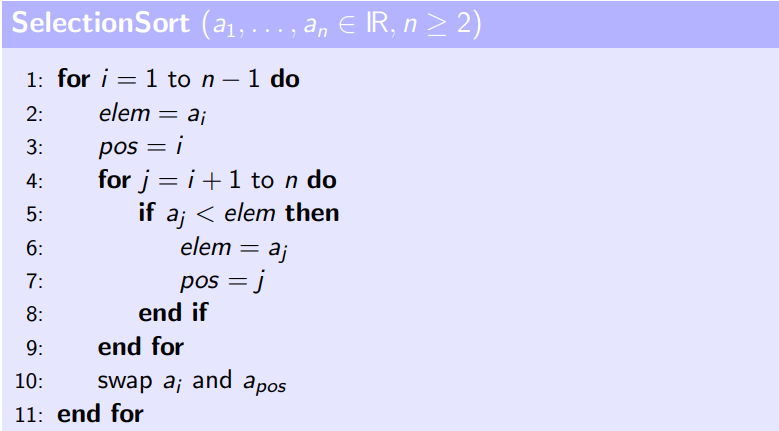
a1, … , aj.

Running time between n-1 and n(n-1)/2 – worst case O(n2)

Can be proved correct by using invariant “after jth iteration, first j+1 elements are in order.

Insertion sort is theta(n) in best case and theta(n2) in worst case. Is Omega(n) and O(n2) on all inputs. Typically, we say it is O(n2).

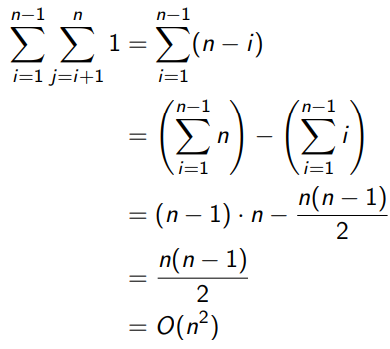
Selection sort:

(searches the unsorted array for the smallest element, and swaps it with the element at the front position of the unsorted part of the list. Repeat until list is sorted)

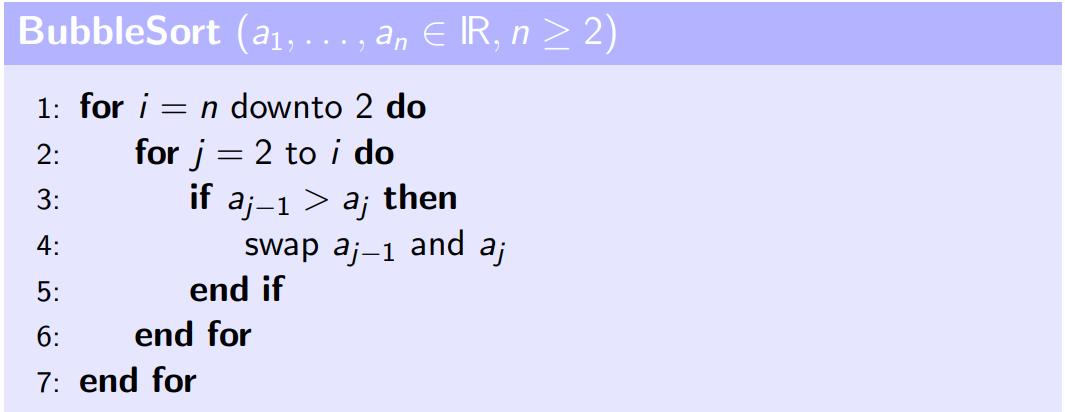
(see <https://visualgo.net/en/sorting> for animation 😊)

Invariant: after the ith iteration, positions 1, …, I contain the overall i smallest elements in order. In ith iteration, we search ith smallest element in remainder and move it to the front.

Time complexity:

^ both best and worst cases are n2 since the inner for loop still executes n times regardless of input.

Bubble Sort:

^the outer for loop is backwards – going down after each iteration instead of up (why???)

Invariant: after the iteration for i of the outer for loop, positions i, …, n contain the overall n-i+1 many largest elements in order.

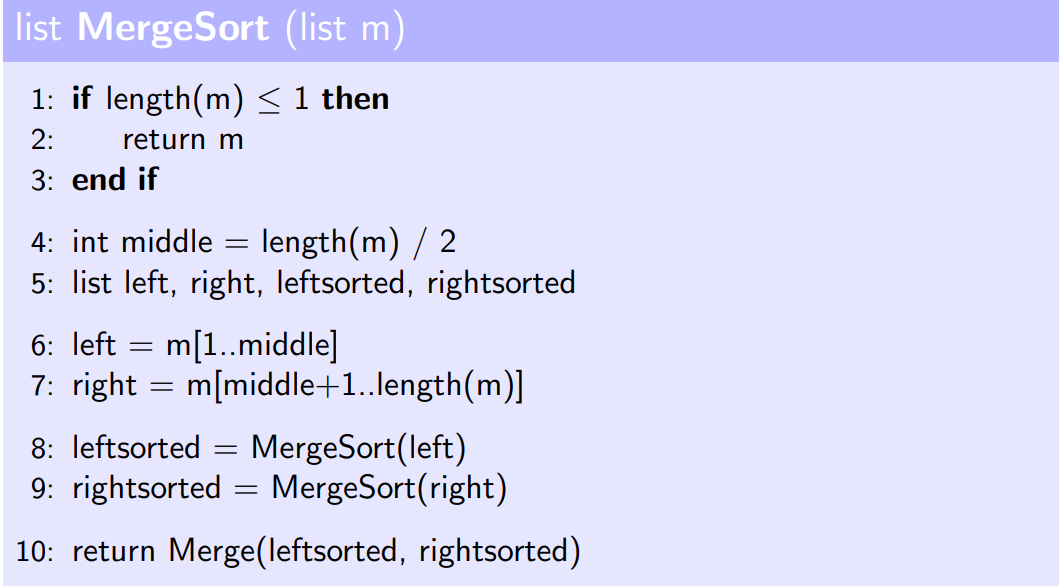
Time complexity: (n-1) + (n-2) + … + 1 -> O(n2)

(there is a variation of bubble sort which terminates if no swaps are made in a given iteration, i.e. when all elements are sorted.)

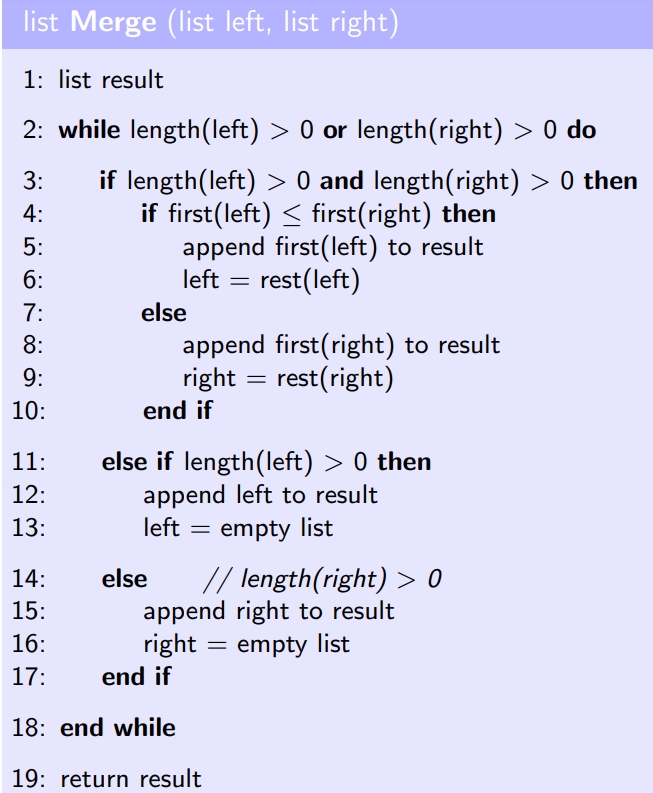
Recursive Sorting Algorithms

Merge Sort:

^scary

If a given sequence is no more than 1 element, we are done (base case for recursion). Otherwise, split sequence into two shorter sequences recursively. Sort them and merge the resulting sorted sequences recursively to build up a final sorted list.

Assume length of input sequence is a power of 2 to allow nice splitting into equal sized sequences 😊

The Merge() function used in the above pseudocode for MergeSort:

^input lists left and right are already sorted. To merge them, compare the first element of left to the first element of right and add the smallest of these to the result, removing it from its original list (left or right). Repeat the process with the new first element of the lists until either left or right is empty. At this point, you know the non-empty list contains the largest elements of the result in order, so this list can just be added to the result, and made empty so the program will know to exit the while loop. The result will be returned as the final sorted list.

Merge sort is the simplest recursive sorting algorithm, with a much better worst case time complexity than Insertion, Bubble or Selection sort ( merge sort has time complexity O(nlogn) ). Its good cases may be worse than for some of the other algorithms – Merge sort always needs nlogn steps, whereas the best case for Insertion sort is roughly n steps, however Insertion sort usually needs n2 steps which is considerably worse than nlogn.

Each recursive call has running time proportional to the length of the list input into the call, i.e. each is O(n). Total running time of merge sort is O(n \* depth of recursion tree) = O(n\*logn) = O(nlogn)

^this will be proved later 😊

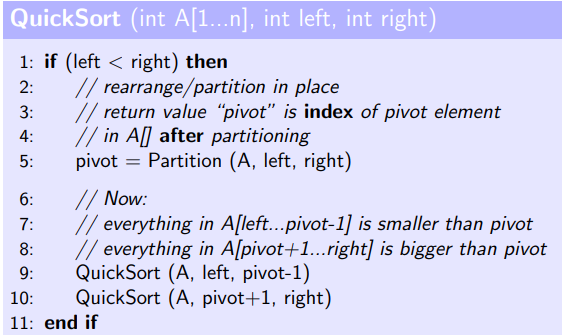
Quicksort:

Good/average case is good, but worst case is poor (similar to Selection sort).

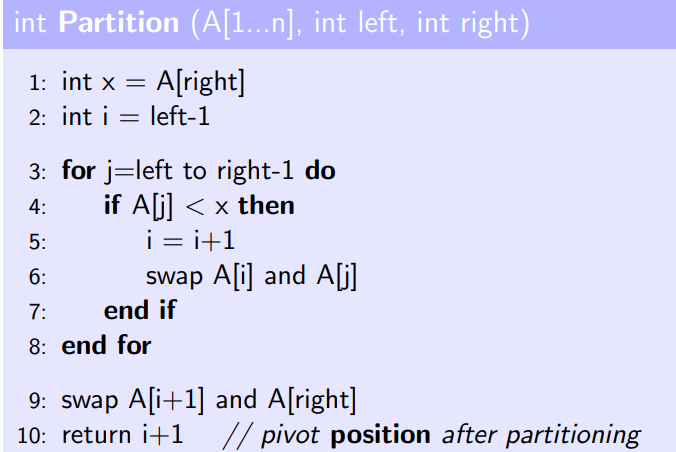
* Split input into two parts (but differently to merge sort)
* Recursively sort them
* Concatenate the resulting sorted subsequences.

^quicksort guarantees that any element in the left sublist is smaller than any element in the right sublist so they can just be concatenated together rather than merged as with Mergesort.

At the beginning of a recursive call, the algorithm chooses a pivot. The partitioning is done wrt this pivot. Each element smaller than the pivot goes to the left, elements larger go to the right part (parts may have very different sizes).

(Mergesort does the complicated bit after the recursive calls, i.e. the merging, whereas Quicksort does the complicated bit before recursing, i.e. when splitting the list into parts, meaning simple concatenation instead of a complex merge algorithm is OK).

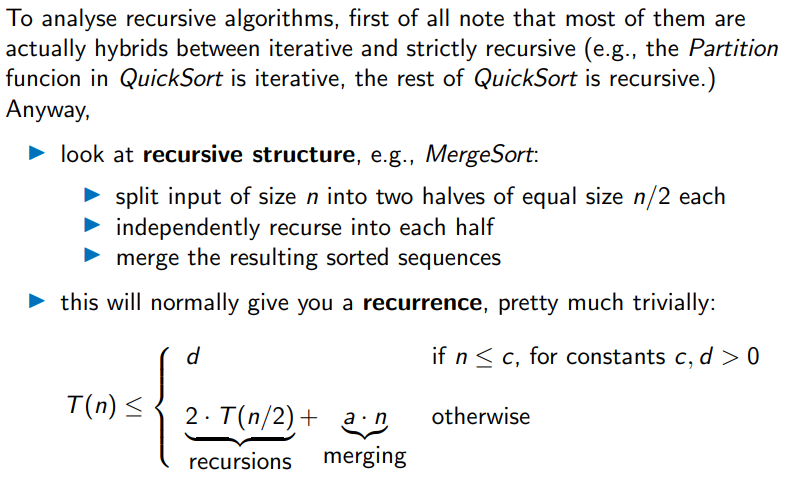
The pivot is selected by the partition function, which then moves everything smaller than the pivot to the left of it and everything larger to the right. Note – partition function does not sort these elements – only compares them to the pivot and places them on the correct side of the pivot. Simplest example of partition function chooses either the leftmost or rightmost element in the unsorted list as a pivot. (ints left and right store the indices for these values). Recursive calls to Quicksort will gradually sort the list.

^this implementation chooses the rightmost position as the pivot, assigning it to variable x.

The element at position i+1 is the first element in the list that is greater than the pivot. If an element is discovered to be less than the pivot, swap this one with the i+1th element and increase i and j by 1. This means that this new smaller element is at position i, i.e. in the area of elements smaller than pivot, and the element it was swapped with has been moved to a position between i+1 and j, which is in the area for elements larger than the pivot.

Invariants: elements between left and i are smaller than pivot, elements between i+1 and j are larger than pivot, elements between j and right at any point during the execution are those which have yet to be put into the correct position.

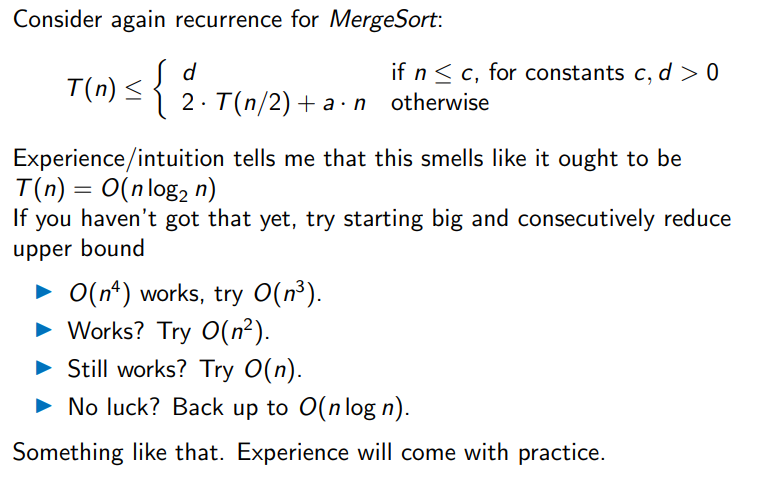
Recurrences

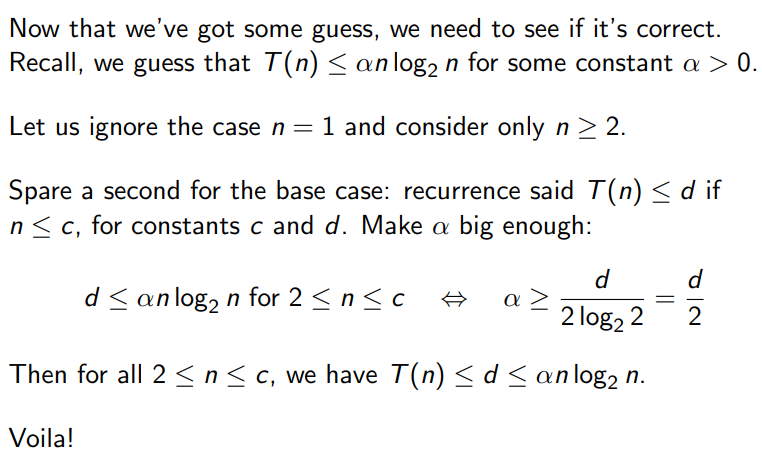
^since each Mergesort call for a list of size n produces 2 Mergesort calls for a list of size n/2 plus a call to Merge which has linear time complexity (i.e. running time a\*n for a constant a), T(n) = 2 \* T(n/2) + a\*n where T(n) is the running time for input size n.

Methods for solving these recurrences:

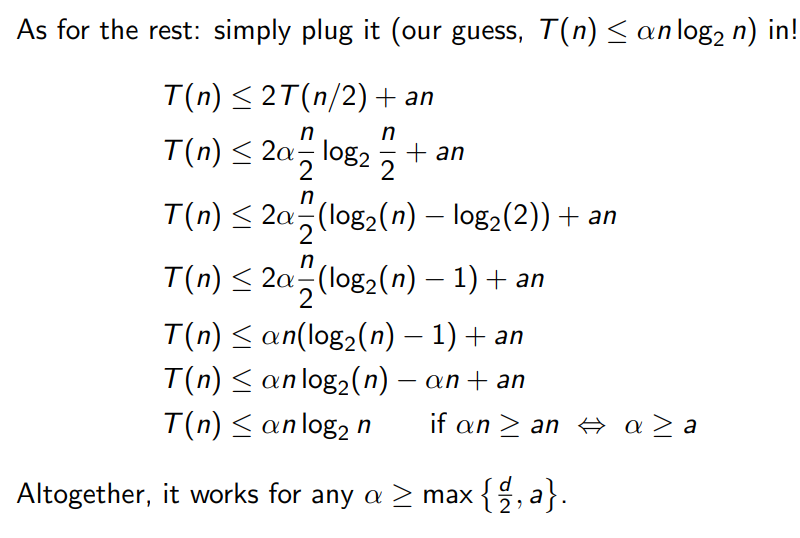
1. Induction – guess, substitute and verify
2. Master Theorem.

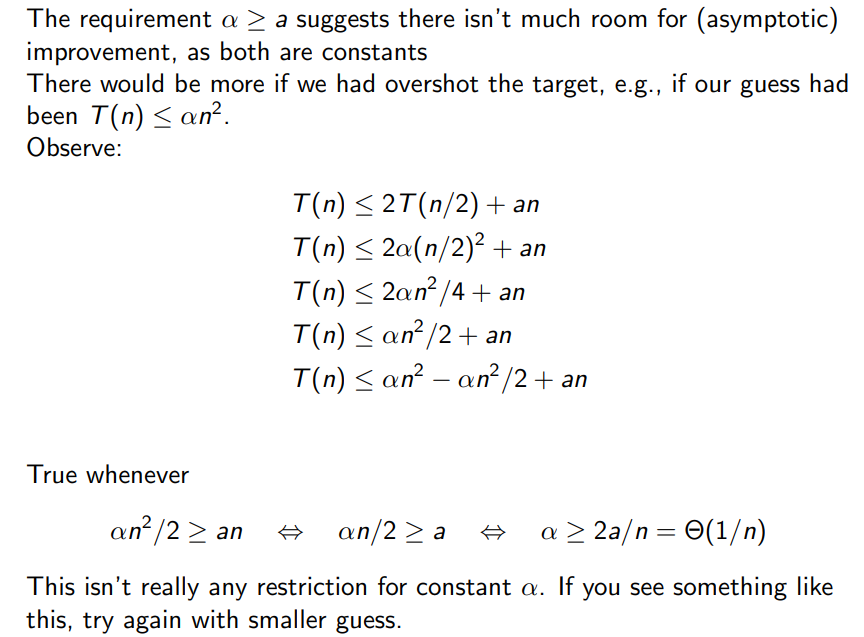
By induction:

* Guess correct solution, verify base case and step.

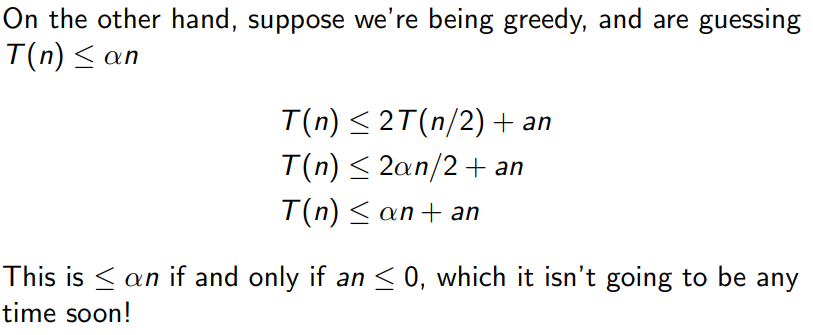
^just keep trying until you have an idea of what the lower bound could be

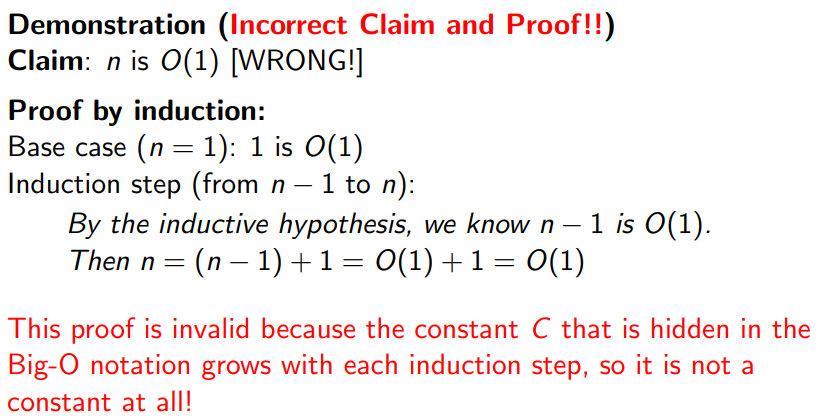
^ignore n = 1 since log(1) = 0

^ do not use big O in the proof as using the sum rule can break things 😊

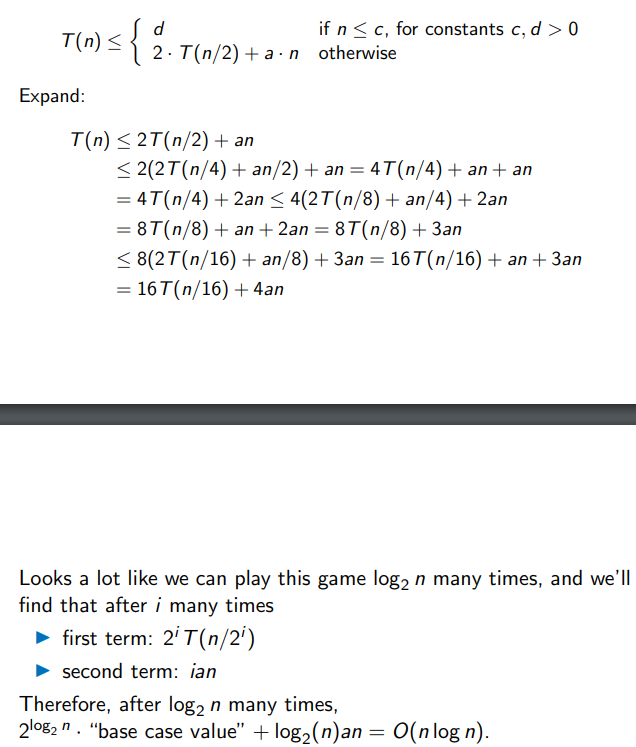
^

i.e. if induction proof creates a result of the form alpha >= constant, this is good. If alpha >= constant / n then bound is too large. If all alphas cancel and you get 0 <= constant\*n then the bound is too low and the proof is not true.

Why not to use Big-O notation within a proof:

^sum rule breaks things

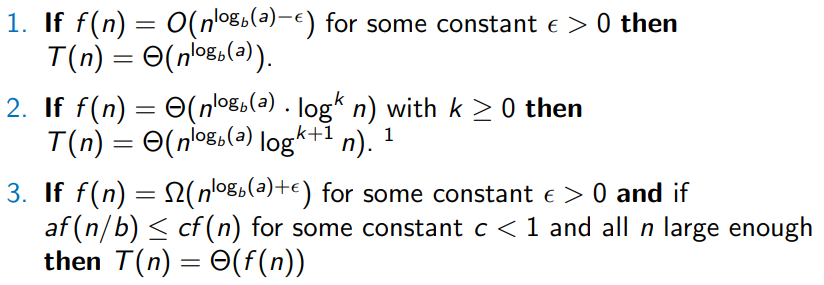
Solving recurrences by iterative substitution:

.

Master Theorem:

Can use if recurrence is in the form T(n) = aT(n/b) + f(n) for constants a >=1, b>1

For example Mergesort has a = 2, b = 2, f(n) = a\*n

Three cases:

^ for case 3, af(n/b) <= cf(n) is true for all polynomials but not for all weird functions.

e.g. with Mergesort, nlogb(a) = nlog2(2) = n and f(n) = a\*n, thus case 2 applies with k = 0, so T(n) = theta(nlogn)

If recurrence not in the required form, the Master Theorem does not apply, e.g. if T(n) = T(n-1) + dn it does not apply.

Analysing running time of Quicksort:

What do we do with elements equal to the pivot?

Options:

* Some copies of the pivot to the left, some to the right -> probably best overall.
* All copies of the pivot added to the end of the left list
* All copies of the pivot added to the right pivot-> these two involving putting them all on one side not good if many copies of the pivot, as will cause imbalance in the recursive calls.
* Put all in the middle, not included in either recursive call. -> adds a lot of extra logic to Partition function but can save time in the recursive calls if there are many copies of the pivot, but otherwise makes running time worse due to this added complexity.

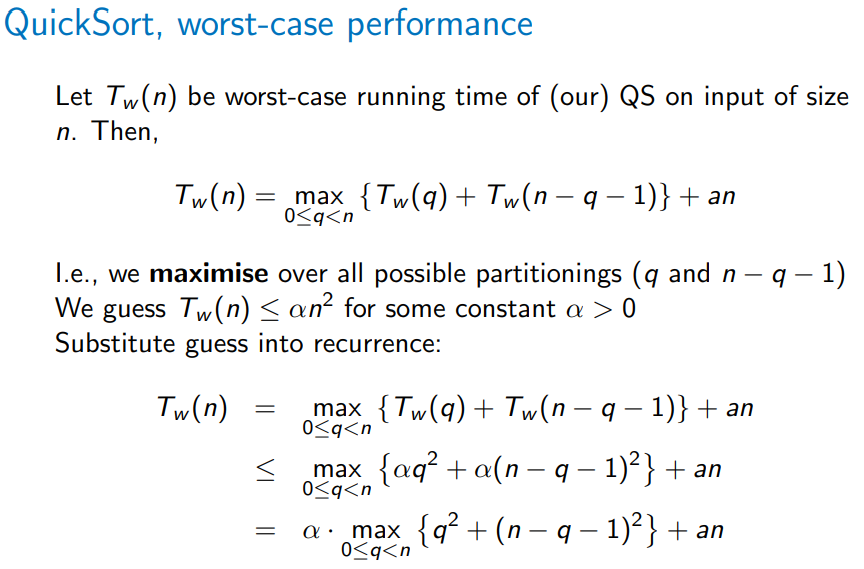
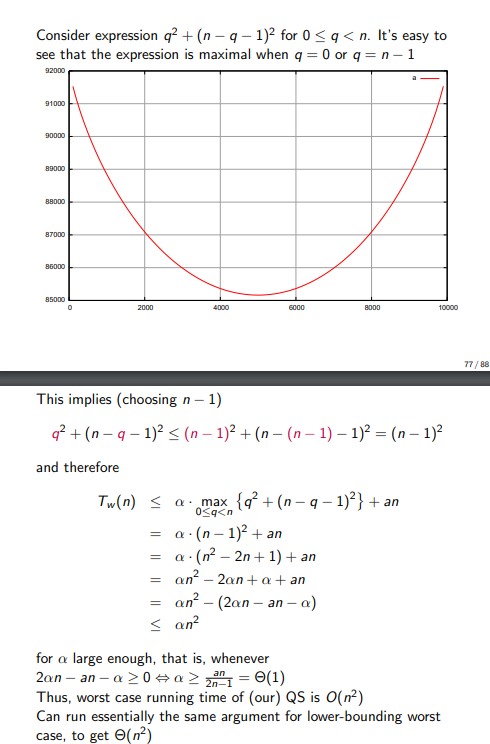
Solving QuickSort recurrence:

T(n) = d if n <= c, constants c,d > 0

T(n) = T(q) + T(n-q-q) + a\*n for some 0 <= q < n , i.e. dependent on pivot chosen.

Master Theorem does not apply here as not in the correct form.

.

^Induction – assuming Tw(x) <= alpha(x2) and plugging in

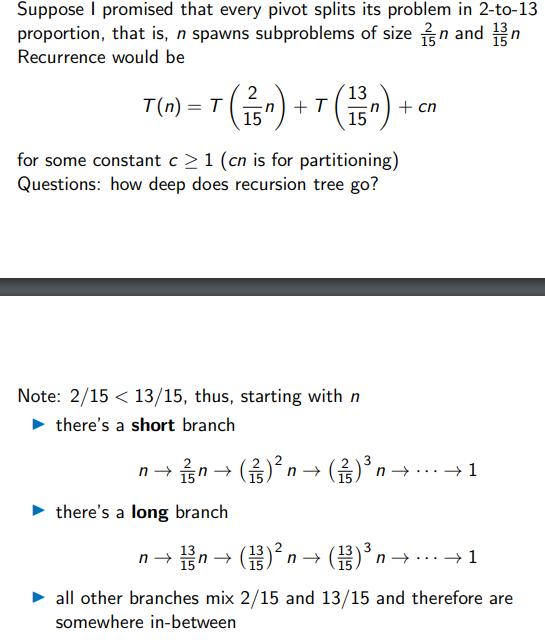
^finding the maximum of the value in the bracket and simplifying to show that Tw(n) <= alpha(n2), using the induction performed on the recursive calls.

(not very “quick” then is it?)

How to make QuickSort quick?

* Roll some dice
* Use a clever algorithm to find a good pivot (discussed later in module) – not usually done in practice due to extra computational time in finding the median element to use as pivot -> can find pivot in O(n) time but there is a large constant factor.

If the pivot is balanced, QuickSort is theta(nlogn), but if extremely unbalanced then theta(n2).

At depth i in the rightmost (13/15) branch, we have input size n \* (13/15)n.

For which i is n\* (13/15)i <= 1?

N <= (15/13)i

Log2n <= i\*log2(15/13)

I >= 1/(log2(15/13) \* log2n

Therefore deepest branch has depth O(logn) since 1/(log2(15/13) is constant.

Time complexity is <= O(logn) \* cn -> O(nlogn)

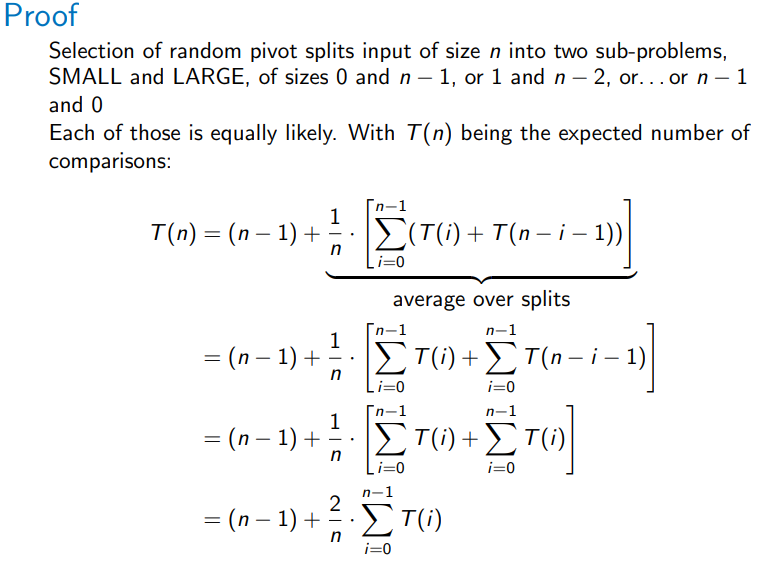
^Would still be nlogn for any pivot which splits the problem into two parts, each of size >= 1, as only the constant factor will be affected.

We can use randomisation algorithms to “eliminate” bad input instances. Randomised Quicksort (RQS) chooses the pivot uniformly at random from all possible elements in the list in the hope that it produces reasonably good splits most of the time – i.e. reduce the possibility of choosing the minimum or maximum.

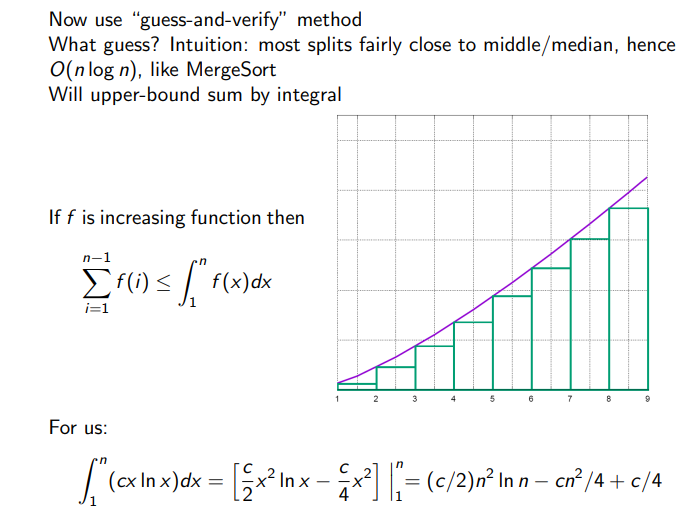
How would we analyse? – Analysing RQS not examinable 😊

If X is a random variable that denotes the number of comparisons during a run of RQS then

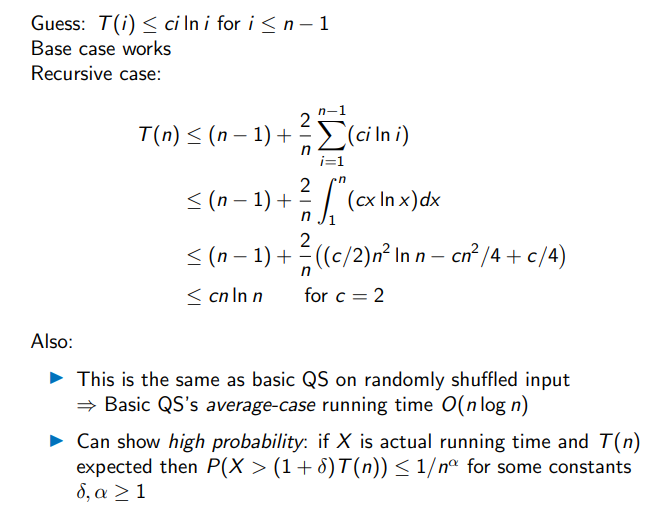
E[X] = O(nlogn)

^each possible pivot has a chance of 1/n of being chosen

Note – E[X] = sum probabilities that X = x multiplied by the value of x for all possible values of x that X can take. i.e. probability of each split is 1/n, the value of x for each possible X = i (expected number of comparisons) is T(i) + T(n-i-1), representing the T() for each split for that particular i, and we are summing them up to get the E[X].

^sum of f(i) represented by “staircase” of bars, integral represented by area under curve – useful trick if f(i) can be integrated easily but not summed easily. This is also why n-1 becomes 1 because discrete vs continuous.

(How do you think to do that tho)

Induction:

nn

