Notes for the course Algorithms and Data Structures CSE1305

(G. Bertalan)

Week 2.1

- Introduction to Algorithms and Data Structures.

- Complexity analysis and big-Oh notation [Sections 4.1, 4.2, 4.3].

- Recursion [Sections 5.1, 5.3, 5.4].

- Complexity analysis: proof methods [Section 4.4].

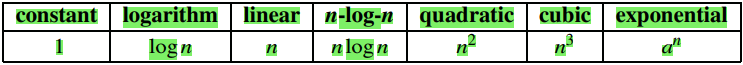
- Recursive analysis [Section 5.2].

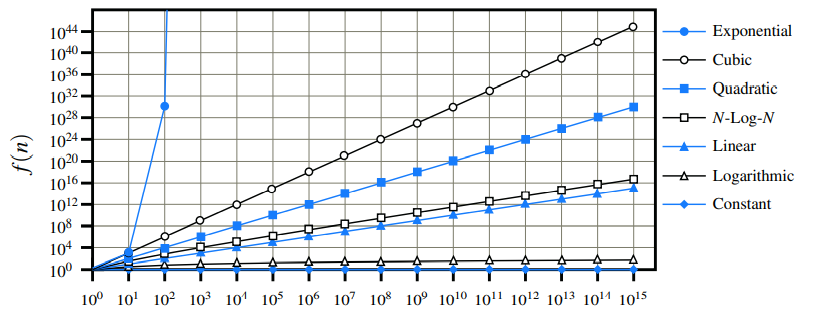
- Space complexity.

- Tail recursion and pitfalls of recursion [Section 5.5].

**Complexity analysis and big-Oh notation**

**Seven functions:**



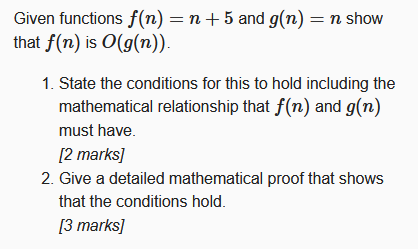


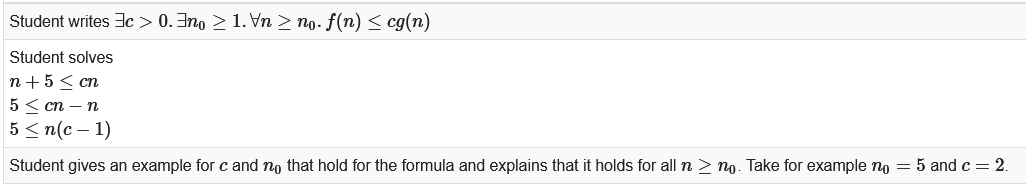
**The big-O notation:**

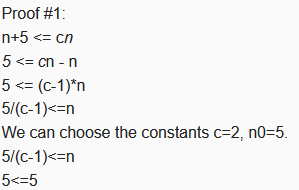
Let f(n) and g(n) be functions mapping positive integers to positive real numbers. We say that f(n) is O(g(n)) if there is a real constant c > 0 and an integer constant n0 ≥ 1 such that f(n) ≤ c · g(n), for n ≥ n0.

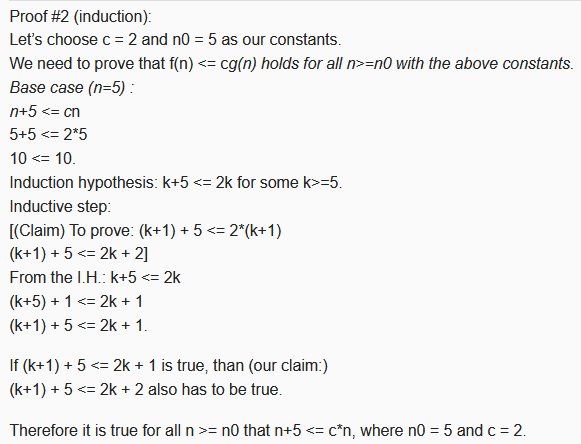
Short:  
∃c>0.∃n0≥1.∀n≥n0.f(n)≤cg(n)

Exercise:









Exercise:Prove that 5n^4 +3n^3 +2n^2 +4n+1 is O(n4).

Proof: Note that 5n^4 +3n^3 +2n^2 +4n+1 ≤ (5+3+2+4+1)n^4 = cn^4, for c = 15, when n ≥ n0 = 1.

Exercise(We rely on the mathematical fact that logn ≤ n for n ≥ 1):

Prove that 5n2 +3nlog n+2n+5 is O(n2).

Proof: 5n2 +3nlog n+2n+5 ≤ (5+3+2+5)n2 = cn2, for c = 15, when n ≥ n0 = 1.

Exercise:

Prove that 20n3 +10nlog n+5 is O(n3).

Proof: 20n3 +10nlog n+5 ≤ 35n3, for n ≥ 1.

Exercise:

Prove that 3log n+2 is O(logn).

Proof: 3logn+ 2 ≤ 5log n, for n ≥ 2. Note that logn is zero for n = 1. That is why we use n ≥ n0 = 2 in this case.

Exercise:

Prove that 2^(n+2) is O(2n).

Proof: 2n+2 = 2n ·22 = 4·2n; hence, we can take c = 4 and n0 = 1 in this case.

Exercise:

Prove that 2n+100log n is O(n).

Proof: 2n+100log n ≤ 102n, for n ≥ n0 = 1; hence, we can take c = 102 in this case.

**Big-Omega:**

We say that f(n) is Ω(g(n)), pronounced “ f(n) is big-Omega of g(n),” if g(n) is O(f(n)), that is, there is a real constant c > 0 and an integer constant n0 ≥ 1 such that f(n) ≥ cg(n), for n ≥ n0.

Short:  
∃c>0.∃n0≥1.∀n≥n0.f(n)>=cg(n)

Exercise:

Prove that 3n log n − 2n is Ω(nlog n).

Proof: 3nlog n− 2n = nlog n+ 2n(logn− 1) ≥ nlog n for n ≥ 2; hence, we can take c = 1 and n0 = 2 in this case.

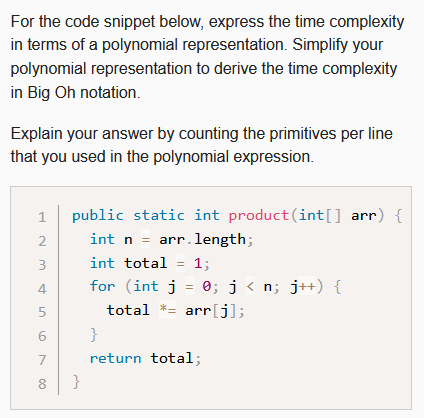
**Big-Theta:**

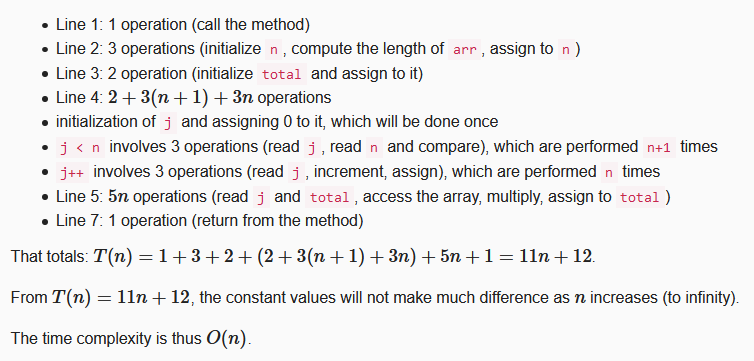
We say that f(n) is Θ(g(n)), pronounced “ f(n) is big-Theta of g(n),” if f(n) is O(g(n)) and f(n) is Ω(g(n)), that is, there are real constants c’ > 0 and c” > 0, and an integer constant n0 ≥ 1 such that c’g(n) ≤ f(n) ≤ c”g(n), for n ≥ n0.

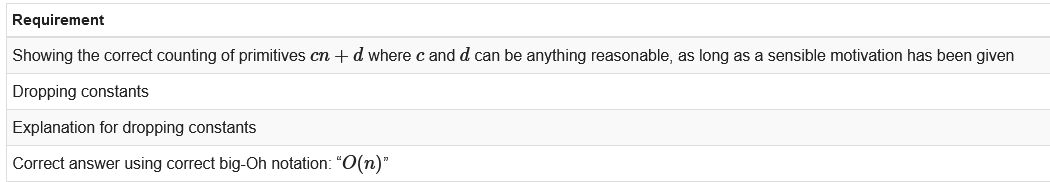
Exercise:

Prove that 3n log n + 4n + 5logn is Θ(nlog n).

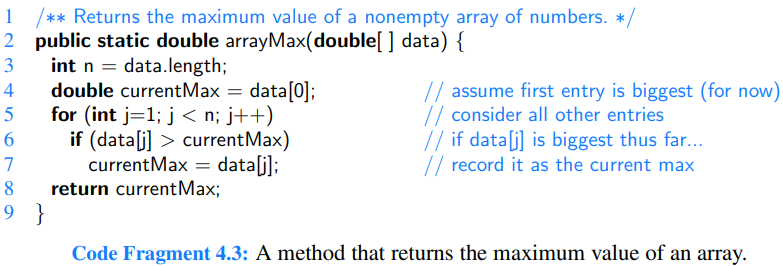
Proof: 3nlog n ≤ 3nlog n + 4n + 5logn ≤ (3+4+5)nlogn for n ≥ 2.







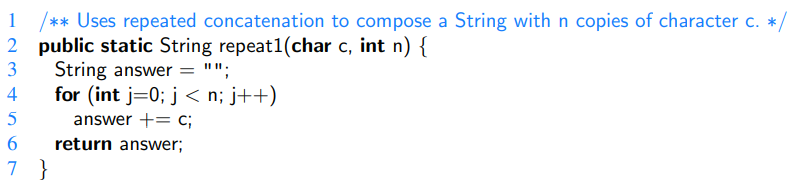
ArrayMax



The *arrayMax* is O(n).

A more interesting question about arrayMax is how many times we might update the current “biggest” value. In the worst case, if the data is given to us in increasing order, the biggest value is reassigned n − 1 times. But what if the input is given to us in random order, with all orders equally likely; what would be the expected number of times we update the biggest value in this case? To answer this question, note that we update the current biggest in an iteration of the loop only if the current element is bigger than all the elements that precede it. If the sequence is given to us in random order, the probability that the jth element is the largest of the first j elements is 1/ j (assuming uniqueness). Hence, the expected number of times we update the biggest (including initialization) is Hn = ∑ 1/ j (j goes from 1 to n), which is known as the nth **Harmonic number.** It can be shown that Hn is **O(logn)**. Therefore, the expected number of times the biggest value is updated by arrayMax on a randomly ordered sequence is O(logn).

Composing Long Strings



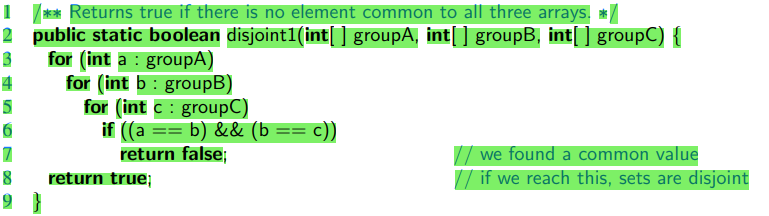
Strings in Java are immutable objects. Once created, an instance cannot be modified. The command, answer += c does not cause a new character to be added to the existing String instance; instead it produces a new String with the desired sequence of characters, and then it reassigns the variable, answer, to refer to that new string.

The creation of a new string as a result of a concatenation, requires time that is proportional to the length of the resulting string. The first time through this loop, the result has length 1, the second time through the loop the result has length 2, and so on, until we reach the final string of length n. Therefore, the overall time taken by this algorithm is proportional to 1+2+···+n, which is (n\*(n+1)) / 2. Therefore the time complexity of the repeat1 algorithm is O(n^2).

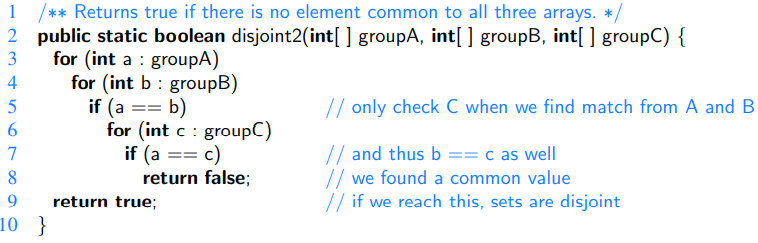
With Java’s StringBuilder class it would be O(n).

Three-Way Set Disjointness

The three-way set disjointness problem: determine if the intersection of the three sets is empty, namely, that there is no element x such that x ∈ A, x ∈ B, and x ∈ C.



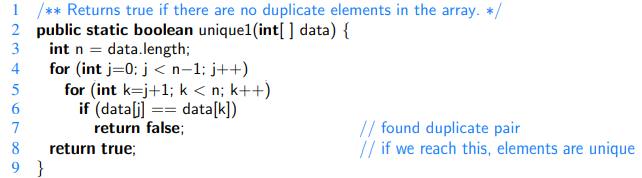
The time complexity of disjoint1() is O(n^3). We can improve this: Once inside the body of the loop over B, if selected elements a and b do not match each other, it is a waste of time to iterate through all values of C looking for a matching triple. An improved solution:



This has time complexity O(n^2). The for-loop over A requires O(n) time. The for-loop over B accounts for a total of O(n^2) time, since that loop is executed n different times. The test a == b is evaluated O(n^2) times. The rest of the time spent depends upon how many matching (a,b) pairs exist. There are at most n such pairs; therefore, the management of the loop over C and the commands within the body of that loop use at most O(n^2) time.

Element Uniqueness

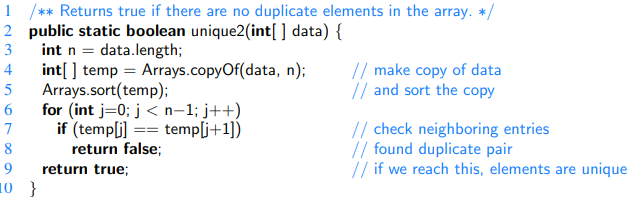
In the element uniqueness problem, we are given an array with n elements and asked whether all elements of that collection are distinct from each other.



(n−1) + (n−2) +···+2+1 -> O(n^2)

Using Sorting as a Problem-Solving Tool

By sorting the array of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, to determine if there are any duplicates, all we need to do is perform a single pass over the sorted array, looking for consecutive duplicates.



The best sorting algorithms guarantee a worst-case running time of O(n log n). Once the data is sorted, the subsequent loop runs in O(n) time, and so the entire unique2 algorithm runs in O(n log n) time.

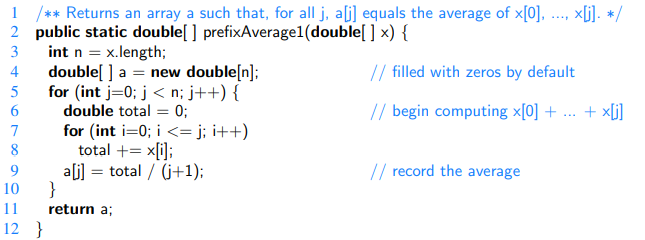
Prefix Averages

Prefix averages of a sequence of numbers. Namely, given a sequence x consisting of n numbers, we want to compute a sequence a such that aj is the average of elements x0,...,xj, for j = 0,...,n−1, that is,



For example, given the year-by-year returns of a mutual fund, ordered from recent to past, an investor will typically want to see the fund’s average annual returns for the most recent year, the most recent three years, the most recent five years, and so on.

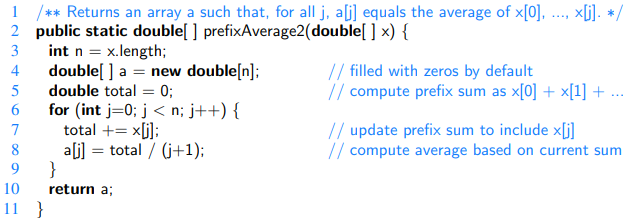
A Quadratic-Time Algorithm



The running time of prefixAverage1 is O(n^2).

A Linear-Time Algorithm

An intermediate value in the computation of the prefix average is the prefix sum x0 + x1 + ··· + xj, denoted as total in our first implementation; this allows us to compute the prefix average a[j] = total / (j + 1). In our first algorithm, the prefix sum is computed anew for each value of j. That contributed O(j) time for each j, leading to the quadratic behavior. For greater efficiency, we can maintain the current prefix sum dynamically, effectively computing x0 +x1 +···+xj as total + xj, where value total is equal to the sum x0+x1+···+xj−1, when computed by the previous pass of the loop over j.



The running time of prefixAverage2 is O(n), which is much better than the quadratic time of algorithm prefixAverage1.

Recursion

The Factorial Function

Definition:

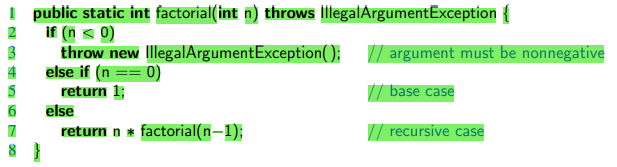
For any integer n ≥ 0:



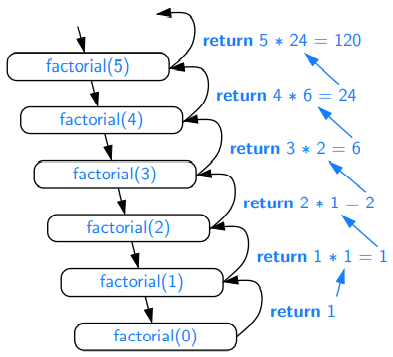
Recursive definition:



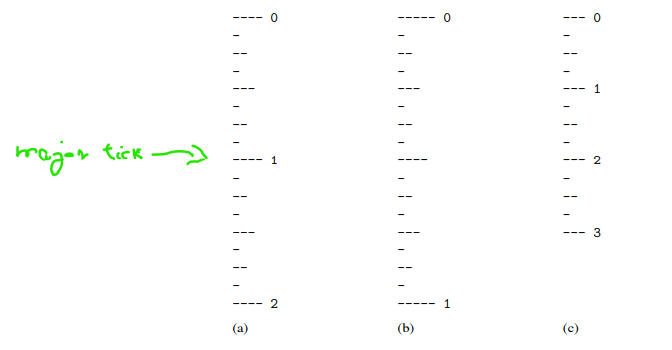
Recursive Implementation of the Factorial Function:

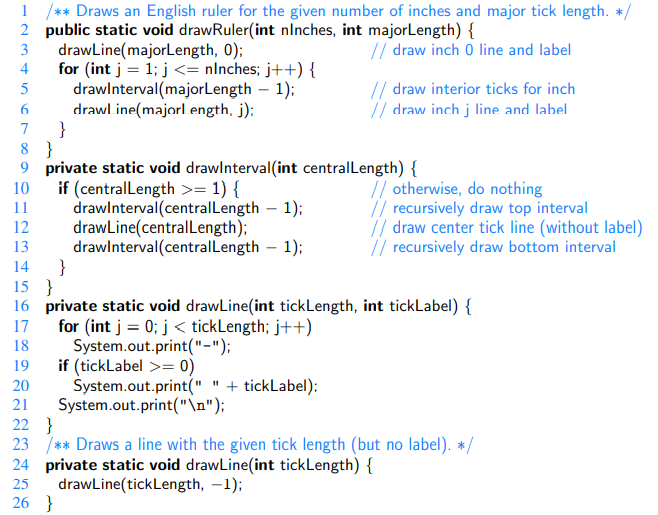


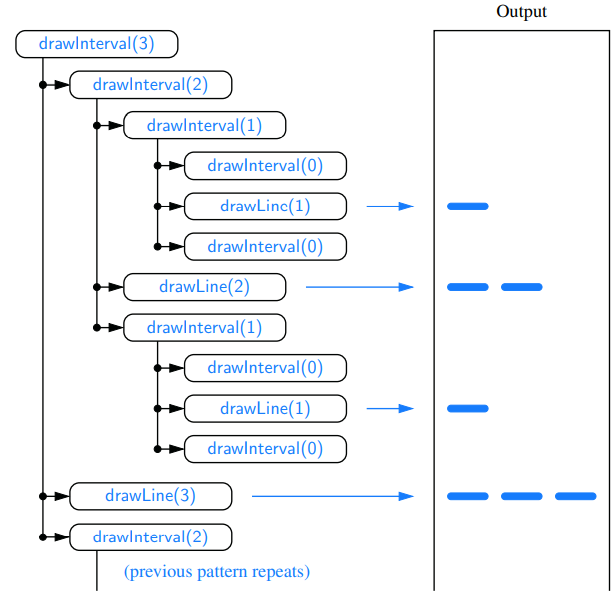
Recursive Trace of the Factorial Function:



Drawing an English Ruler







Linear Search

Use a loop to examine every element, until either finding the target or exhausting the data set. O(n) time.

Binary Search

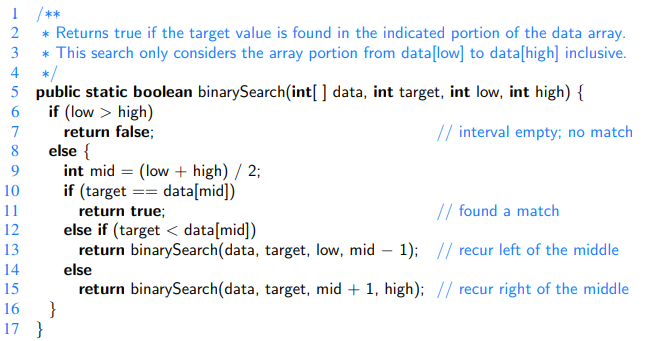
Variables: high, low, median.

Initially, low = 0 and high = n− 1.

The median candidate: mid = ⌊(low +high)/2⌋ .

We consider three cases:   
 • If the target equals the median candidate, then we have found the item we are looking for, and the search terminates successfully.   
 • If the target is less than the median candidate, then we recur on the first half of the sequence, that is, on the interval of indices from low to mid−1.   
 • If the target is greater than the median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high.

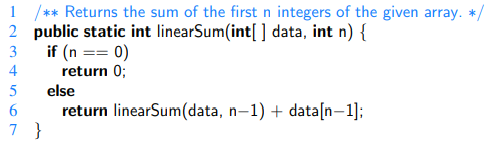
An unsuccessful search occurs if low > high, as the interval [low,high] is empty.



Further Examples of Recursion

Linear Recursion

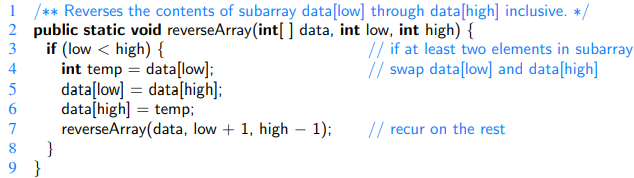
Summing the Elements of an Array Recursively



For an input of size n, the linearSum algorithm makes n+ 1 method calls. Hence, it will take O(n) time, because it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that the memory space used by the algorithm (in addition to the array) is also O(n), as we use a constant amount of memory space for each of the n+1 frames in the trace at the time we make the final recursive call (with n = 0).

Reversing a Sequence with Recursion

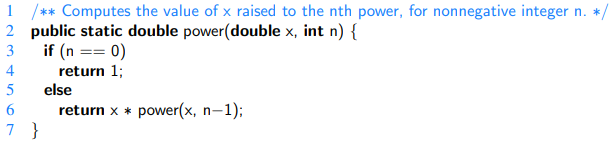
Swapping the first and last elements. Initial call: reverseArray(data, 0, n−1).

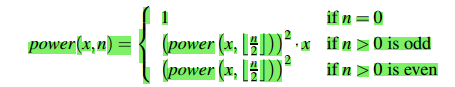
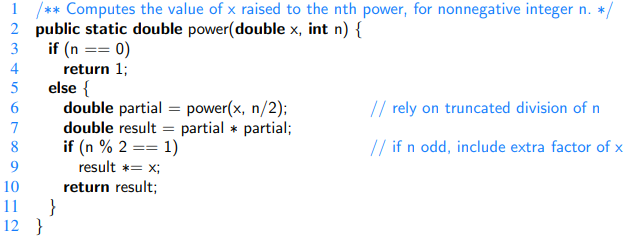


This recursive algorithm is guaranteed to terminate after a total of 1+ Floor(n/2)\_x0005\_ recursive calls. Because each call involves a constant amount of work, the entire process runs in O(n) time.

Recursive Algorithms for Computing Powers

Two different recursive formulations, with very different performance:

  
  
   
  
  
A recursive call to this version of power(x,n) runs in O(n) time.

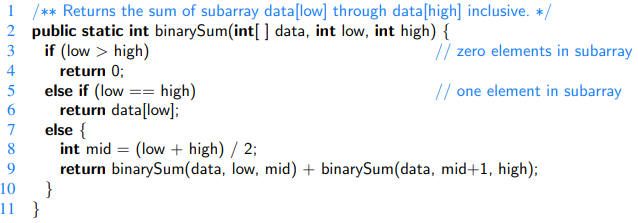
Alternative definition:  
  
  
  


To analyze the running time of the revised algorithm, we observe that the exponent in each recursive call of method power(x,n) is at most half of the preceding exponent. As we saw with the analysis of binary search, the number of times that we can divide n by two before getting to one or less is O(logn). Therefore, our new formulation of power results in O(logn) recursive calls. Each individual activation of the method uses O(1) operations (excluding the recursive call), and so the total number of operations for computing power(x,n) is O(logn). This is a significant improvement over the original O(n)-time algorithm.

The improved version also provides significant saving in reducing the memory usage. The first version has a recursive depth of O(n), and therefore, O(n) frames are simultaneously stored in memory. Because the recursive depth of the improved version is O(log n), its memory usage is O(logn) as well.

Binary Recursion

Summing the n integers of an array: binarySum(data, 0, n−1).



BinarySum uses O(logn) amount of additional space, which is a big improvement over the O(n) space used by the linearSum method.

However, the running time of binarySum is O(n), as there are 2n−1 method calls, each requiring constant time.

Designing Recursive Algorithms

An algorithm that uses recursion typically has the following form:

• Test for base cases. We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls will eventually reach a base case, and the handling of each base case should not use recursion.

• Recur. If not a base case, we perform one or more recursive calls. This recursive step may involve a test that decides which of several possible recursive calls to make. We should define each possible recursive call so that it makes progress towards a base case.

Parameterizing a Recursion

To design a recursive algorithm for a given problem, it is useful to think of the different ways we might define subproblems that have the same general structure as the original problem.

A successful recursive design sometimes requires that we redefine the original problem to facilitate similar-looking subproblems. Often, this involved reparameterizing the signature of the method. For example, when performing a binary search in an array, a natural method signature for a caller would appear as binarySearch(data, target). However we defined our method with calling signature binarySearch(data, target, low, high), using the additional parameters to demarcate subarrays as the recursion proceeds.

Complexity analysis: proof methods

By Counterexample

Example: Professor Amongus claims that every number of the form 2i − 1 is a prime, when i is an integer greater than 1. Professor Amongus is wrong.

Proof: To prove Professor Amongus is wrong, we find a counterexample. Fortunately, we need not look too far, for 24 −1 = 15 = 3 · 5.

By Contrapositive

“if p is true, then q is true,”  
 “if q is not true, then p is not true”

Example 4.18: Let a and b be integers. If ab is even, then a is even or b is even.

Proof: To justify this claim, consider the contrapositive, “If a is odd and b is odd, then ab is odd.” So, suppose a = 2 j+1 and b = 2k+1, for some integers j and k. Then ab = 4 jk+2 j +2k+1 = 2(2 jk+ j +k) +1; hence, ab is odd.

De Morgan’s Laws:  
 The negation of a statement of the form “p or q” is “not p and not q.”

The negation of a statement of the form “p and q” is “not p or not q.”

By Contradiction

Example: Let a and b be integers. If ab is odd, then a is odd and b is odd

Proof: Let ab be odd. We wish to show that a is odd and b is odd. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is even or b is even. In fact, without loss of generality, we can assume that a is even (since the case for b is symmetric). Then a = 2 j for some integer j. Hence, ab = (2 j)b = 2(jb), that is, ab is even. But this is a contradiction: ab cannot simultaneously be odd and even. Therefore, a is odd and b is odd.

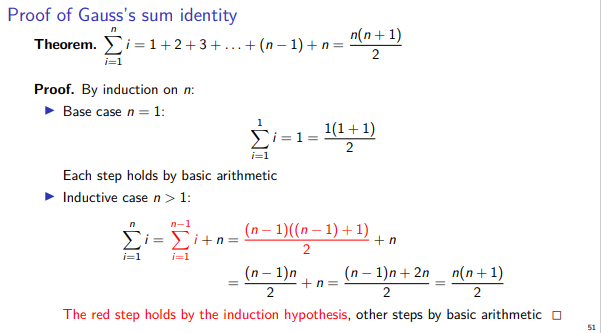
By Induction

Proposition: Consider the Fibonacci function F(n), which is defined such that F(1) = 1, F(2) = 2, and F(n) = F(n − 2) + F(n − 1) for n > 2. (See Section 2.2.3.) We claim that F(n) < 2n.

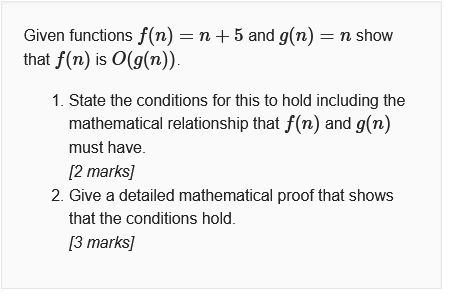
Justification: We will show our claim is correct by induction. Base cases: (n ≤ 2). F(1) = 1 < 2 = 21 and F(2) = 2 < 4 = 22. Induction step: (n > 2). Suppose our claim is true for all j < n. Since both n− 2 and n−1 are less than n, we can apply the inductive assumption (sometimes called the “inductive hypothesis”) to imply that F(n) = F(n−2) +F(n−1) < 2n−2 +2n−1 . Since 2n−2 +2n−1 < 2n−1 +2n−1 = 2 · 2n−1 = 2n, we have that F(n) < 2n, thus showing the inductive hypothesis for n.

Proposition:  
 

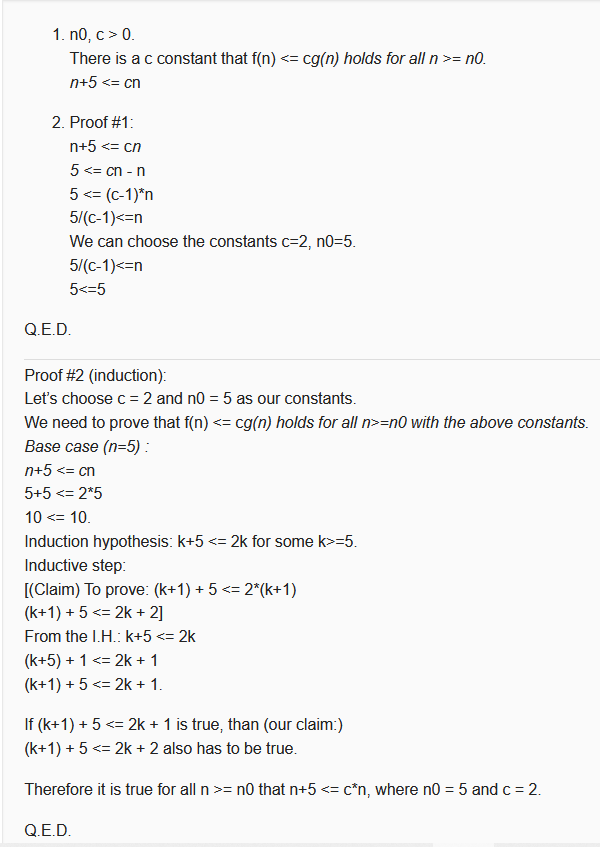
Justification: We will justify this equality by induction. Base case: n = 1. Trivial, for 1 = n(n+1)/2, if n = 1. Induction step: n ≥ 2. Assume the inductive hypothesis is true for any j < n. Therefore, for j = n−1, we have n−1 ∑ i=1 i = (n−1)(n−1+1) 2 = (n−1)n 2 . Hence, we obtain n ∑ i=1 i = n+ n−1 ∑ i=1 i = n+ (n−1)n 2 = 2n+n2 −n 2 = n2 +n 2 = n(n+1) 2 , thereby proving the inductive hypothesis for n.



Induction Weblab Example:







Loop Invariants

Book page 181(199)

Recursive Analysis

Computing Factorials

To compute factorial(n), we see that there are a total of n+1 activations, as the parameter decreases from n in the first call, to n−1 in the second call, and so on, until reaching the base case with parameter 0.

It is also clear, given an examination of the method body, that each individual activation of factorial executes a constant number of operations. Therefore, we conclude that the overall number of operations for computing factorial(n) is O(n), as there are n+1 activations, each of which accounts for O(1) operations.

Recurrence Equation

[Slide-ok az elozo evbol]

Drawing an English Ruler

Proposition: For c ≥ 0, a call to drawInterval(c) results in precisely 2c − 1 lines of output.

[Ezt probaljuk meg a recurrence equation-nel igazolni.]