Data Structures

Augustin Cosse.

Spring 2021

March 2, 2021

Algorithm Analysis

- The big-Oh notation is widely used to characterize running times and space bounds in terms of some parameters n which is defined as a chosen measure of the size of the problem
- Suppose two algorithms solving the same problem are available. An algorithm A which has a running time $O(n)$ and an algorithm B which has running time $O(n^2)$. Which algorithm is the better?
- We know that n is $O(n^2)$ which implies that algorithm A is asymptotically better than algorithm B
- Using this idea, we can use the big-Oh notation to order classes of functions by asymptotic growth rate

Algorithm Analysis

- Our seven functions are ordered by increasing growth rate as follows

$$1, \quad \log n \quad n, \quad n \log n \quad , n^2, \quad n^3, \quad 2^n$$

- The importance of good algorithm design goes beyond what can be solved effectively on a given computer
- Even if we achieve a dramatic speedup in harware, we still cannot overcome the handicap of an asymptotically slow algorithm

Algorithm Analysis

- A few words of caution about asymptotic notation are in order at this point.
- First, note that the use of the big-Oh notation can be somewhat misleading and the hidden constant factor s might be very large
- As an illustration of this, if we know that an algorithm has complexity $10^{100}n$ (hence O(n)) while another one is $10n\log n$ we might want to favor the second one as 10^{100} (one googol) in itself represents an intractable number of computations

Algorithm Analysis

- Second, given the big-Oh notation, one could naturally wonder what represents a fast (or efficient) algorithm. Generally speaking, any algorithm running in $O(n \log n)$ should be considered efficient. Even a $O(n^2)$ algorithm may be fast enough in some context.

- An algorithm whose running time is an exponential function however such as $O(2^n)$ should almost never be considered efficient

- To understand how bad an exponential running time is, consider the story of the inventor of the game of chess who asked a king to pay him one grain of rice for the first square, 2 grains for the second, 4 grains for the fourth and so on.

- The total number of grains for the 64^{th} square then reached

$$2^{63} = 9,223,372,036,854,775,808$$

Algorithm Analysis

- It seems natural to draw a line in terms of efficiency between algorithms running in polynomial time $O(n^c)$ with $c > 1$ and exponential time (i.e. $O(b^n)$ with $b > 1$)
- This should however be taken with a grain of salt as an algorithm running in $O(n^{100})$ should not really be considered as efficient

A few examples

- All of the constant operations have by definition constant running time :
    - Assigning a value to a variable
    - Following an object reference
    - Performing one arithmetic operation (e.g. adding two numbers)
    - Comparing two numbers, calling a method, returning from a method
    - Accessing a single element of an array by index
- The expression `A.length` in Java is evaluated in constant time because arrays are represented internally with an explicit variable that records the length of the array

A few examples

- For any valid index `j` the element `A[j]` can be accessed in constant time as well
- This is because an array uses a consecutive block of memory. I.e. The j^{th} element can be found, not by iterating through the array one element at a time, bu by validating the index and using it as an offset from the beginning of the array in determining the appropriate memory address

A few examples

- As a classic example of an algorithm with a running time that grows proportional to n, we consider the goal of finding the largest element of an array

- A typical strategy is to loop through the elements of the array while maintaining as a variable the largest element seen so far

```java
/** Returns the max of an array */
public static double arrayMax(double[ ] data) {
  int n = data.length;
  double currentMax = data[0];
  // assume first entry is biggest (for now)
  for (int j=1; j < n; j++)
    if (data[j] > currentMax)
      currentMax = data[j];
  return currentMax;
}
```

A few examples

- Using the big-Oh notation, we can write the following statement regarding the running time of the algorithm `arrayMax`

Proposition

The algorithm `arrayMax`*, for computing the maximum element of an array of n numbers runs in $O(n)$ time*

```java
/** Returns the max of an array */
public static double arrayMax(double[ ] data) {
  int n = data.length;
  double currentMax = data[0];
  // assume first entry is biggest (for now)
  for (int j=1; j < n; j++)
    if (data[j] > currentMax)
      currentMax = data[j];
  return currentMax;
}
```

A few examples

Proposition

The algorithm `arrayMax`*, for computing the maximum element of an array of n numbers runs in $O(n)$ time*

- To see this, first note that the initialization and the return statement both only involve a constant number of primitive operations
- Each iteration of the loop also only requires a constant number of primitive operations and the loop runs $n-1$ times
- The total number of primitive operations is thus $c'(n-1)+c''$ for appropriate constants c' and c''

A few examples

Proposition

The algorithm `arrayMax`*, for computing the maximum element of an array of n numbers runs in $O(n)$ time*

- One could wonder how many operations are need for a given ordering of the entries.
- In the worst case, the largest value is re-assigned $n-1$ times
- But what if the entries are in random order. For a random ordering, the probability that the j^{th} element is the largest of the first j elements is $1/j$. The expected number of times we update the variable `currentMax` is thus given by $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$ which is known as the n^{th} Harmonic number

A few examples

Proposition

The algorithm `arrayMax`*, for computing the maximum element of an array of n numbers runs in $O(n)$ time*

- One could wonder how many operations are need for a given ordering of the entries.

- In the worst case, the largest value is re-assigned $n-1$ times

- But what if the entries are in random order. For a random ordering, the probability that the j^{th} element is the largest of the first j elements is $1/j$. The expected number of times we update the variable `currentMax` is thus given by $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$ wihch is known as the n^{th} Harmonic number which can be shown to be $O(\log n)$

- On a randomly ordered array, the expected number of times the `currentMax` value is updated is thus $O(\log n)$

A few examples

- As a second example, we consider the repeated use of the string concatenator to create long strings.

```java
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;}
```

- The most important aspect of this example is that strings are immutable objects in java. That is to say, once created they cannot be modified

A few examples

- As a consequence of this, the command `answer+=c` is shorthand for `answer = answer+c` does not cause a character to be added to the string, 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.
- In terms of efficiency, the problem with this first approach is that the creation of a new string now requires time that is proportional to the length of the resulting string
- The first time through the loop, the result has length 1, the second time, the result has length 2 and so on until we reach the final length n. The overall time taken by the algorithm is thus given by

$$1 + 2 + \ldots + n = O(n^2)$$

A few examples

- Suppose that we are given 3 sets A, B and C which are stored in three distinct integers arrays
- We suppose that no individual set contains duplicate values but that there might be numbers that appear in two or three of the sets.
- The three-way set disjointness problem is to determine if the intersection of the three sets is empty, i.e. if there is no element x such that $x \in A$, $x \in B$ and $x \in C$.

```java
public static boolean disjoint1(int[ ] groupA,
                                int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false; // we found a common value
  return true; // if we reach this, sets are disjoint}
```

A few examples

- It is clear that if each original set has size n, the worst case complexity of the algorithm is $O(n^3)$

```
public static boolean disjoint1(int[ ] groupA,
                                int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      for (int c : groupC)
        if ((a == b) && (b == c))
          return false; // we found a common value
  return true; // if we reach this, sets are disjoint}
```

A few examples

- We can improve upon the asymptotic performance with a simple observation
- Once inside the body of the B loop, if selected elements a and b do not match each other, it is a waster of time to iterate through the values of C looking for a matching triple.
- Following this idea, we can come up with an improved solution

```
public static boolean disjoint2(int[ ] groupA,
                           int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b) // only check if match
        for (int c : groupC)
          if (a == c) // and thus b == c as well
            return false;
  return true; }
```

A few examples

- In this improved version, the worst case complexity is now $O(n^2)$
- To see this, note that there are quadratically many pairs (a, b) to consider. However if A and B are sets of distinct elements there are at most n pairs with $a = b$.
- The third loop on C hence executes at most n times.

```
public static boolean disjoint2(int[ ] groupA,
                                int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b) // only check if match
        for (int c : groupC)
          if (a == c) // and thus b == c as well
            return false;
  return true; }
```

A few examples

- In short the total number of operations for the A and B loops is $O(n^2)$

- The test a==b is evaluated n^2 times but since that condition is satisfied only $O(n)$ times, the inner loop on c is executed only $O(n)$ times and the commands within the body of the C loop are therefore executed at most $O(n^2)$ times.

- All in all we thus have a global complexity of $O(n^2)$

```
public static boolean disjoint2(int[ ] groupA,
                               int[ ] groupB, int[ ] groupC) {
  for (int a : groupA)
    for (int b : groupB)
      if (a == b) // only check if match
        for (int c : groupC)
          if (a == c) // and thus b == c as well
            return false;
  return true; }
```

A few examples

- A problem closely related to the three way set disjointness is the element uniqueness problem
- In the element uniqueness problem we are given an array with n elements and we wnat to know whether all the elements of the array are distinct from each other

```java
public static boolean unique1(int[ ] data) {
  int n = data.length;
  for (int j=0; j < n-1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false; // found duplicate pair
  return true;
}
```

A few examples

- A simple solution to the problem can be obtained by looping through all the distinct pairs of indices $j < k$ checking if those pairs refer to equivalent elements

- The algorithm relies on two nested loops. The first iteration of the outer loop leads to $n-1$ iterations of the inner loop, the second one leads to $n-2$ iterations,..

- The total complexity in this case is thus $(n-1)+(n-2)+\ldots+2+1 = O(n^2)$

```
public static boolean unique1(int[ ] data) {
  int n = data.length;
  for (int j=0; j < n-1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false; // found duplicate pair
  return true; }
```

A few examples

- A better algorithm for the element uniqueness problem is based on using sorting as a problem solving tool
- By sorting the array elements we are guaranteed that any duplicate elements will be placed next to each other
- To determine if there are any duplicates, all we need to do now is do a single pass over the array, looking for consecutive elements

```java
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n); // make copy
  Arrays.sort(temp); //  sort the copy
  for (int j=0; j < n1; j++)
    if (temp[j] == temp[j+1]) // check neighbors
      return false; // found duplicate pair
  return true; }
```

A few examples

- The best sorting algorithms (including those used by `Array.sort` in Java) garantee a worst complexity in $O(n \log n)$.
- Once the data is sorted, the loop runs in $O(n)$ iterations.
- Altogether, the entire algorithm is thus $O(n \log n)$

```
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n); // make copy
  Arrays.sort(temp); //  sort the copy
  for (int j=0; j < n1; j++)
    if (temp[j] == temp[j+1]) // check neighbors
      return false; // found duplicate pair
  return true; }
```

A few examples

- As our next problem, we want to compute prefix averages of a sequence of numbers
- Given a sequence x consisting of n numbers, we want to compute a sequence a such that a_j is the average of elements x_j for $j = 0, \ldots, n-1$. That is

$$a_j = \frac{\sum_{i=0}^{j} x_i}{j+1}$$

- Prefix averages have many applications in economics and statistics. As an example, given the year by year return of a mutual fund, and investor might want to have access to the fund's annual return for the most recent year, the most recent three years, the most recent five years and so on.

A few examples

- As a first algorithm for computing prefix averages, consider the algorithm given below

```
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n]; // filled with zeros
  for (int j=0; j < n; j++) {
    double total = 0; // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1); // record the average}
  return a;}
```

- The first line in this algorithm (i.e. the initialization) as well as the return line at the end both execute in $O(1)$

- Initializing and creating the array can be done in $O(n)$.

A few examples

- There are two nested for loops. The body of the outer for loop is executed n times and so is the statement `a[j] = total/(j+1)` as well as the management of the j variable in the statement of the loop

- Finally the body of the inner loop gets executed $j + 1$ times depending on the outer loop j. In total the statement `total+=x[i]` gets executed $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

```java
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n]; // filled with zeros
  for (int j=0; j < n; j++) {
    double total = 0; // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1); // record the average}
  return a;}
```

A few examples

- Altogether, our analysis shows a running time of $O(n^2)$ for this first algorithm

```
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n]; // filled with zeros
  for (int j=0; j < n; j++) {
    double total = 0; // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1); // record the average}
  return a;}
```

A few examples

- Let us now consider the following alternative
- An intermediate value in the computation of the prefix average is the prefix sum $x_0 + x_1 + \ldots + x_j$
- In the previous algorithm, the prefix average was computed anew for each new value of j. In this second algorithm, you see that we maintain a version of the prefix sum dynamically

```java
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];
  double total = 0;
  for (int j=0; j < n; j++) {
    total += x[j]; // update prefix sum to include x[j]
    a[j] = total / (j+1); }
  return a;}
```

A few examples

- Initializing the variables `n` and `total` both takes $O(1)$ operations
- Initialization of the array a takes $O(n)$ operations
- There is now a single for loop controled by j whose body runs $O(n)$ times. Since both of the `total+=x[j]` and `a[j] = total/(j+1)` take $O(1)$ time, total contribution of the loop is $O(n)$

```java
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];
  double total = 0; //
  for (int j=0; j < n; j++) {
    total += x[j]; // update prefix sum to include x[j]
    a[j] = total / (j+1); }
  return a;}
```

A few examples

- Total running time of the `prefixAverage2` algorithm is thus $O(n)$

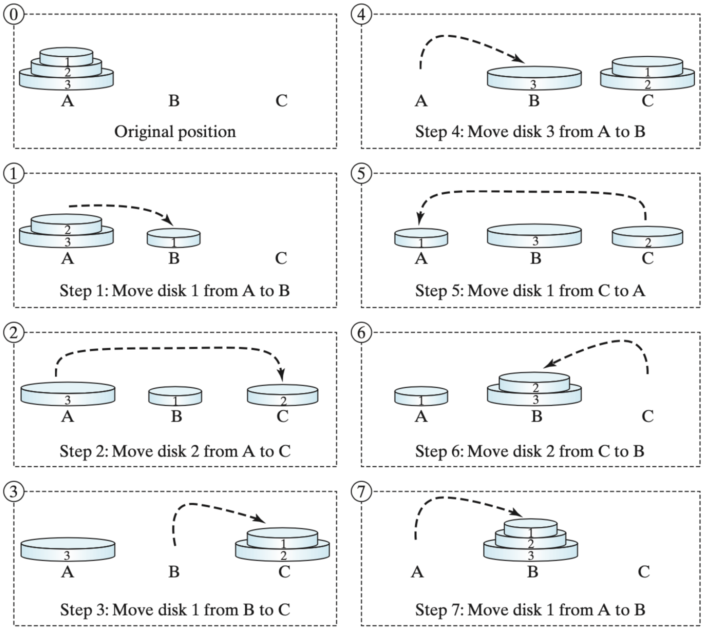
```
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];
  double total = 0; //
  for (int j=0; j < n; j++) {
    total += x[j]; // update prefix sum to include x[j]
    a[j] = total / (j+1); }
  return a;}
```

Exponential time: The tower of Hanoi problem

- The Tower of Hanoi problem is a classic problem that can be solved easily using recursion but is difficult to solve otherwise
- The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:
  - There are n disks labeled $1, 2, 3, \ldots, n$ and three towers labeled A, B and C
  - No disk can be on top of smaller disk at any time
  - All the disks are initially placed on tower A
  - Only one disk can be moved at a time and it must be the smaller disk on a tower
- The objective of the problem is to move al the disks from A to B with the assistance of C.

The tower of Hanoi problem

- In the case of three disks, the solution can easily be found manually
- In the case of 4 disks however, the problem is already much more involved. Fortunately the problem has an inherent recursive nature

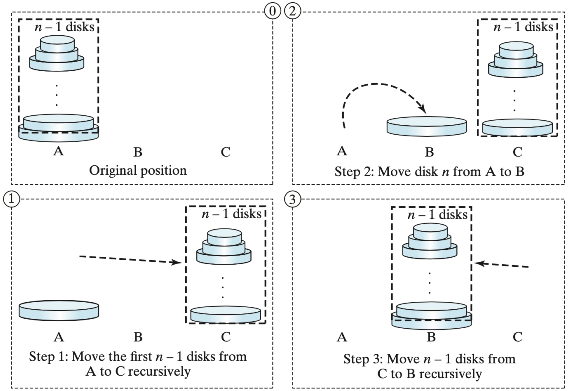

0
1
2
3
A
B
C
Original position
1
2
3
A
B
C
Step 1: Move disk 1 from A to B
3
1
2
A
B
C
Step 2: Move disk 2 from A to C
3
1
2
A
B
C
Step 3: Move disk 1 from B to C
4
3
1
2
A
B
C
Step 4: Move disk 3 from A to B
5
1
3
2
A
B
C
Step 5: Move disk 1 from C to A
6
1
2
3
A
B
C
Step 6: Move disk 2 from C to B
7
1
2
3
A
B
C
Step 7: Move disk 1 from A to B

The tower of Hanoi problem

- The base case is $n = 1$. In this case, one can simply move the disk from A to B.
- When $n > 1$, one can split the original problem into the following three subproblems which can then be solved sequentially

The tower of Hanoi problem

1. Move the first $n-1$ disks from A to C recursively with the assistance of tower B
2. Move disk n from A to B
3. Move $n-1$ disks from C to B recursively with the assistance of tower A



0
n – 1 disks
A
B
C
Original position
2
n – 1 disks
A
B
C
Step 2: Move disk n from A to B
1
n – 1 disks
A
B
C
Step 1: Move the first n – 1 disks from A to C recursively
3
n – 1 disks
A
B
C
Step 3: Move n – 1 disks from C to B recursively

The tower of Hanoi problem

- If we introduce the method below which moves n disks from the fromTower to the toTower,

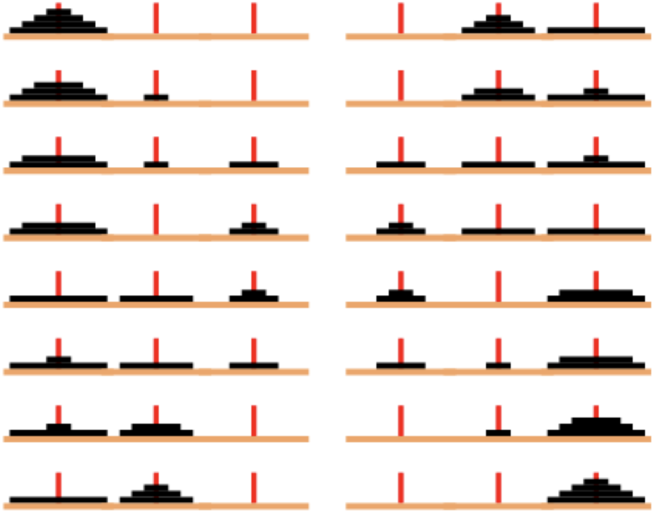
```
void moveDisk(int n, char fromTower, char toTower, char auxTower)
```

- We can then introduce the pseudo code for this method as follows

```
if(n==1) // Stopping condition
  move disk 1 from fromTower to toTower
else {
  moveDisks(n-1, fromTower, auxTower, toTower);
  Move disk n from the fromTower to the toTower;
  moveDisks(n-1, auxTower, toTower, fromTower);
}
```

The tower of Hanoi problem

- For 4 pieces, how many moves (iterations) do you count?



The tower of Hanoi problem

- The complexity of the Tower of Hanoi algorithm is measured by the number of moves
- Let $T(n)$ denote the number of moves for the algorithm to move n disks from tower B with $T(1) = 1$.
- From this, it is easy to see that

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2(2(2T(n-3) + 1) + 1) + 1 \\ &= \ldots \\ &= 2^{n-1} + 2^{n-2} + \ldots + 1 = O(2^n) \end{aligned}$$

The tower of Hanoi problem

- The algorithm for the tower of Hanoi problemhas thus exponential time complexity
- Suppose that a disk is moved at a rate of 1 per second, it would then take $2^{32}/(365 * 24 * 60 * 60) = 135$ years to move 32 disks and $2^{64}/(365 * 24 * 60 * 60) = 585$ billion years to move 64 disks.

Simple proof techniques

- Sometimes we will want to make claims regarding a particular algorithm (i.e. prove that it runs fast or that it is correct)
- In order to backup our claims, we will need to prove our statements. There are several approaches to do this:
  - Through the use of (counter-)example.
  - Through the use of contrapositives or contradictions
  - Through the use of inductions or loop invariants

By example

- Some claims are of the generic form 'There is an element x in a Set S that has property P'
- To justify such a claim we only need to produce a particular x in S that has the property.
- Likewise, some hard to believe claims are of the generic form 'Every element in a Set S has property P'
- To justify that such a claim is false, we only need to produce a single x from S that does not have property P. Such an instance is called a counterexample
- As an example, consider the claim: *'Every number of the form $2^i - 1$ is a prime, when i is an integer greater than 1'. Taking $2^4 - 1 = 15 = 3.5$ proves the claim wrong.*

Contrapositive and Contradiction

- To justify the statement 'if p is true than q is true', we can establish the statement 'if q is false then p is false'
- Logically those two statements are the same bu the second one which is called the contrapositive may sometimes be easier to think about.
- Consider the following statement: 'Let a and b be integers. If ab is even, then a is even or b is even'.
- To prove this statement we can consider the contrapositive and consider the statement 'If a is odd and b is odd then ab is odd'. To prove this we write $a = 2k + 1$, $b = 2\ell + 1$ and develop $ab = 4\ell k + 2\ell + 2k + 1$ which is odd.
- We have relied on de Morgan's law to define the negation of a conditional or: not(A or B)]≡ (not A) and (not B)

Contrapositive and Contradiction

- Another negative justification is the justification by contradiction which also involves de Morgan Law.
- In order to apply the justification by contradiction we establish that a statement q is true by assuming that q is false and showing that this assumption leads to a contradiction
- As an example of this, consider the following statement: 'Let a and b be integers, if ab is odd then a is odd or b is odd'
- When ab is odd, we want to show that a is odd or b is odd. With the hope of leading to a contradiction, we might assume the opposite, namely if ab is odd a is even and b is even. Let us take a even, we then have $a = 2k$, from which $ab = 2kb$ which is even and leads to a contradiction with respect to our original assumption on ab. Hence a must be odd.

Induction and loop invariant

- Most of the claims we make about a running time or a space bound involve a integer parameter n. Moreover, most of these claims are equivalent to saying that some statement $q(n)$ is true for all $n \geq 1$
- Since this is making a claim about an infinite set of numbers, we cannot justify this exhaustively in a direct fashion
- Those claims can however be justified using the notion of induction
- If we want to show that a statement $q(n)$ is true for all values of n, induction start by showing that $q(n)$ is true for a particular value of n, let us say $n = 1$, then it justifies that the inductive step is true for all $n > 1$ namely that if $q(j)$ is true for all $j < n$ then $q(n)$ is true.

Induction and loop invariant

- An example of this, 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 $n > 2$. We claim that $F(n) < 2^n$
- We can show that claim by induction. We first show the base case $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$
- We then show the induction step. Suppose that the claim is true for all $j < n$, we then have

  $$F(n) = F(n-2) + F(n-1) < 2^{n-2} + 2^{n-1}$$

  Now use

  $$2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1} = 22^{n-1} = 2^n$$

Induction and loop invariant

- Consider the statement

$$\sum_{i=1}^{n} i = \frac{(n+1)n}{2}$$

- Again we can easily prove this statement by induction

- We start with the base case, $n = 1$ for which we have $1 = 1 \cdot 2/2 = 1$

- We then prove the induction step. Assuming the statement holds for every $j < n$, we have

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2}$$

  From which we get

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n(n+1)}{2}$$

Induction and loop invariant

- A last technique that is often useful is the notion of loop invariant. To prove that a statement L about a loop is correct, we can define L in terms of a series of smaller statements $L_0, L_1, \ldots L_k$ where

  1. The initial claim L_0 is true before the loop begins
  2. If L_{j-1} is true before iteration j then L_j will be true after iteration j
  3. The final statement L_k implies the desired statement to be true

Induction and loop invariant

- To illustrat the idea of loop invariant, we consider the algorithm `arrayFind` below which finds the smallest index at which element `val` occurs in array A. The algorithm also returns -1 if there is no such element

```
public static int arrayFind(int[ ] data, int val) {
  int n = data.length;
  int j = 0;
  while (j < n) {
    if (data[j] == val){return j; }
    j++;
  return -1; }
```

- To show that the algorithm is correct, we inductively define a list of statements L_j that lead to the conclusion of the algorithm. Let L_j define the fact that `val` is not equal to any of the first j elements of data.

Induction and loop invariant

```
public static int arrayFind(int[ ] data, int val) {
  int n = data.length;
  int j = 0;
  while (j < n) {
    if (data[j] == val){return j; }
    j++;
  return -1; }
```

- The claim is true at the beginning of the algorithm since $j = 0$ and no element can be equal to `val`
- In iteration j we compare element `val` to element `data[j]`. if those elements are the same we return the index j which is correct since no other element of `data` was equal to `val`. If `data[j]` and `val` are not equal we have found one more element not equal to `val` and we increment j

Induction and loop invariant

```
public static int arrayFind(int[ ] data, int val) {
  int n = data.length;
  int j = 0;
  while (j < n) {
    if (data[j] == val){return j; }
    j++;
  return -1; }
```

- The claim L_j will then be true for this value of j and L_{j-1} being true thus implies L_j being true
- If the while loop terminates without returning an index in `data`, we have $j = n$ and L_n is true, there is no element in `data` equal to `val` and the algorithm returns -1 which verifies the statement.

Induction and loop invariant

- One way to describe repetition within a computer program is through the notion of loops. Another completely different way is through the notion of recursion
- Recursion is a technique through which a method makes one or more calls to itself during execution or by which a data structure relies upon smaller instances of the very same type of data structure in its representation
- There are many examples of recursion in art and nature. One such example are Russian Matryoshka dolls.
- In computing, recursion provides an elegant and powerful alternative to perform repetitive tasks

Induction and loop invariant

- Most modern programming languages support functional recursion using the same mechanism that is used to support traditional forms of method calls.
- When one invocation of the method makes a recursive call, the invocation is suspended until the recursive call completes
- We consider four illustrative examples: The factorial function, the English ruler, the binary search and the file system

The factorial function

- The factorial of a positive integer n denoted $n!$ is defined as the product of the integer from 1 to n

- If $0!$ is set to 1 by convention, for any integer n, we write
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot \ldots 3 \cdot 2 \cdot 1, & \text{otherwise} \end{cases}$$

- The factorial is important because it is known to equal the number of ways in which n distinct numbers can be aranged into a sequence, that is the number of permutations of n items. For example, the three characters a, b and c can be arranged in $3! = 6$ ways abc, acb, bac, bca cab and cba.

- There is a natural recursion for the factorial function as we can write
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) & \text{if } n \geq 1 \end{cases}$$

The factorial function

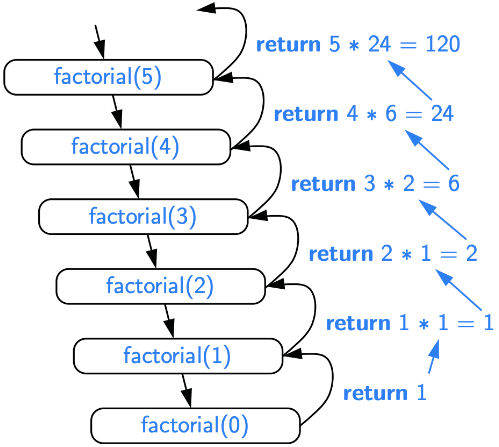
- As you can see with the implementation below, recursion does not require any explicit loop but is instead achieved
- The process is finite because each time the method is invoked, its argument is reduced by 1

```java
public static int factorial(int n)
                   throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );
  else if (n == 0)
    return 1; // base case
  else
    return n  factorial(n-1); // recursion}
```

The factorial function

- We can illustrate the recursion process using a recursion Trace.
- Each entry of the trace corresponds to a recursive call to the function.
- Each new recursive method call is indicated by a downward arrow to the new invocation.

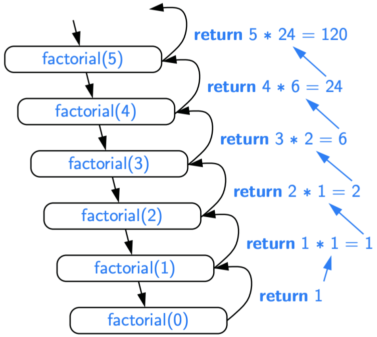
The factorial function



The factorial function

- When the method returns, an arrow showing this return is drawn and the return value is indicated alongside the arrow

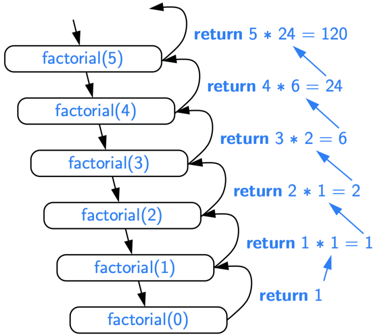
- A recursive trace closely mirrors a programminng language's execution of the recursion . In Java, each time a function is called, an activation record or activation frame is created to store information about the progress of that method



The factorial function

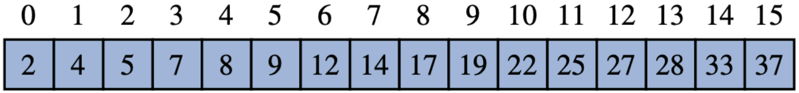
- When the execution of a method leads to a nested method call, the execution of the former call is suspended and its frame stores the place in the source code where the flow of control should return unpon completion of the nested call

- This process is used both for simple function calls as for nested calls. The key idea is that Java maintains a separate frame for each active call



Binary search

- As an additional application of recursion, we consider the classic binary search algorithm.
- Binary search is used to locate a target value within a sorted sequence of n elements stored in an array
- Binary search is among the most important computer algorithms



Binary search

- When the array is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the dataset
- This algorithm is known as linear search or sequential search and it runs in $O(n)$
- When the sequence is sorted and indexable, we can design a more efficient algorithm
- If we consider an rbitrary element in the sequence with value v, we can be sure that all elements prior to this particular element have value less than v and that all elements that come after have value greater than v

Binary search

- We call an element of the sequence a candidate if at a particular moment of the search we cannot rule out that this item matches the target
- The binary search algorithm maintains two parameters `low` and `high` such that all elements have index at least `low` and at most `high`.
- We then compare our target value to the median candidate

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$$

  and we consider three cases

  1. If the target equals the median, then we have found the element we are looking for
  2. If the target is less than the median, we recur on the first half of the sequence (low, mid-1)
  3. If the target is larger than the median then we recur on the second half of the sequence (mid+1, high)

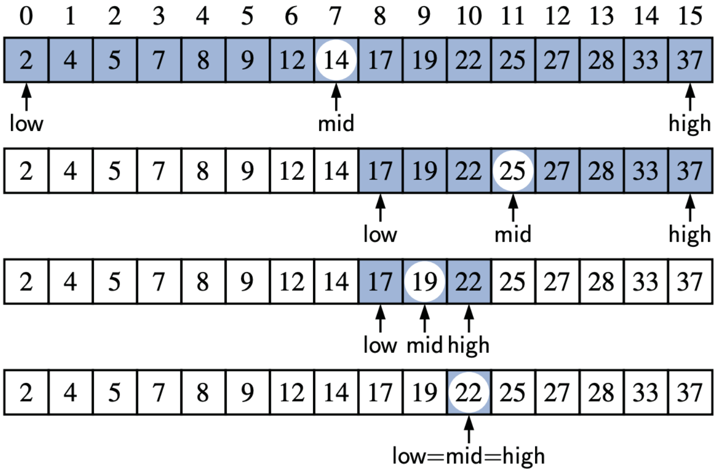
Binary search

- All in all, this leads to the following algorithm known as binary search

```
public static boolean
    binarySearch(int[ ] data, int target, int low, int high)
  if (low > high)
    return false; // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true; // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid  1);
   else
     return binarySearch(data, target, mid + 1, high);
}}
```

Binary search

Example of binary search for a target equal to 22


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
low
mid
high
low
mid
high
low=mid=high