W3203
Discrete Mathematics

Induction, Recursion, & Algorithms

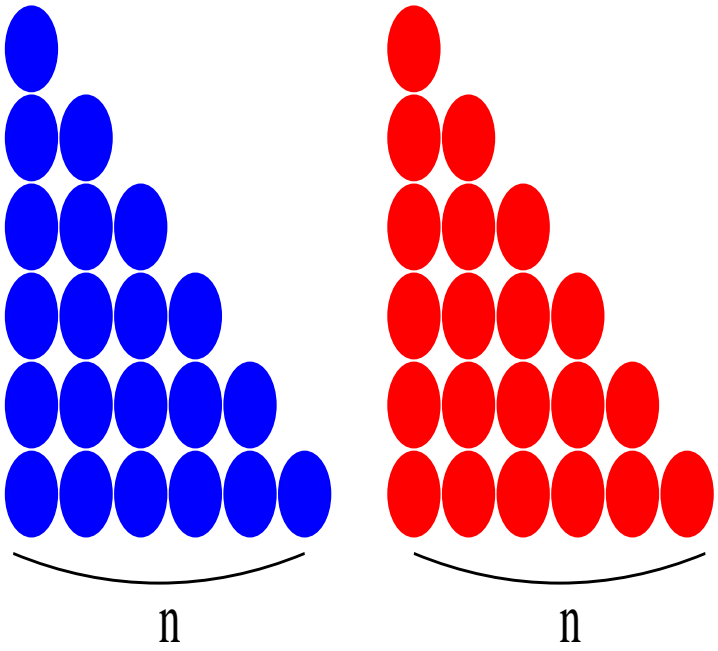
Spring 2015
Instructor: Ilia Vovsha

http://www.cs.columbia.edu/~vovsha/w3203

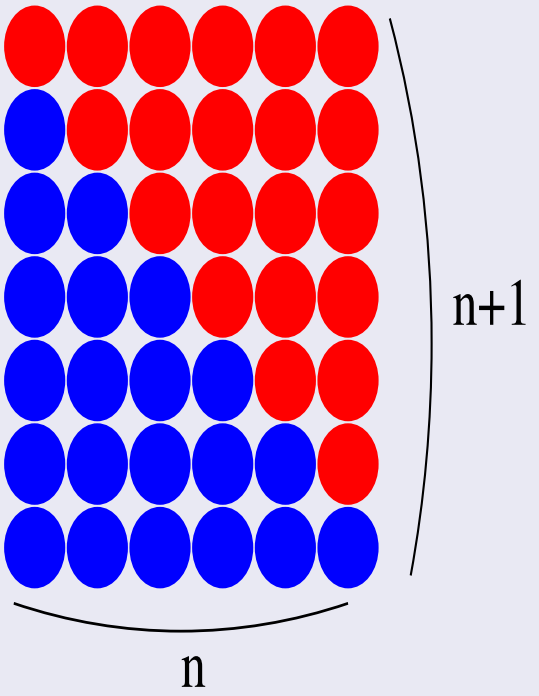
Outline

- Induction Principle
- Strong Induction
- Recursive definitions
- Structural Induction
- Simple algorithms
- Big-O notation, complexity
- Recursive algorithms
- Text: Rosen 3, 5.1 – 5.4,
- Text: Lehman 5.1-5.3, 6.1-6.3

“P(r)oof” by Picture



T_n blue dots and T_n red dots for a grand total of $2T_n$ dots



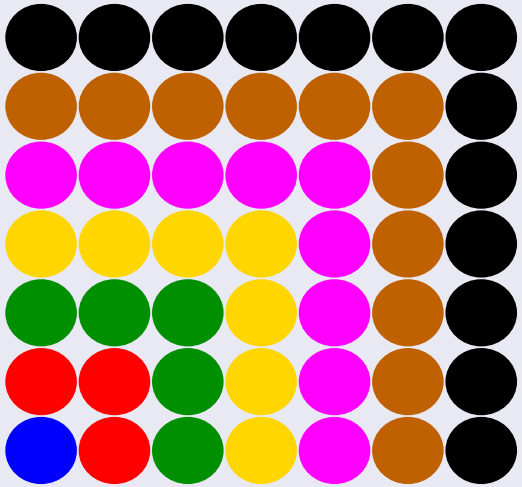
$2T_n = \text{Total Dots} = n(n+1)$

Sum of Odd Integers

Theorem

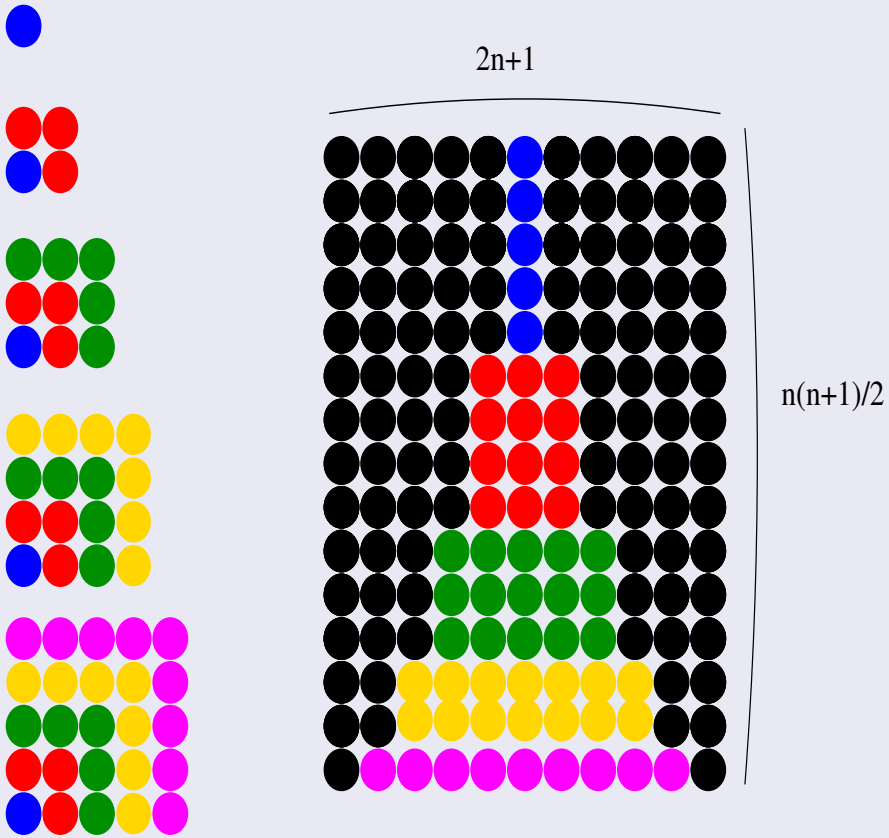
$$1 + 3 + 5 + \cdots + (2k - 1) = k^2$$

Proof.



□

Sum of Squares



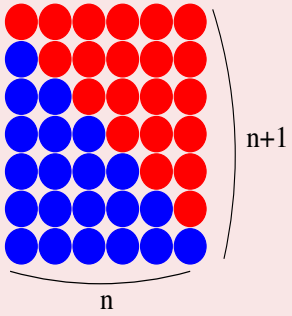
$$2S_n \text{ Black Dots } + S_n \text{ Color Dots } = 3S_n \text{Total Dots} = \frac{n(n+1)(2n+1)}{2}$$

Guidelines

Guidelines

In general, a picture is a proof only if:

1. The picture represents an abstract idea



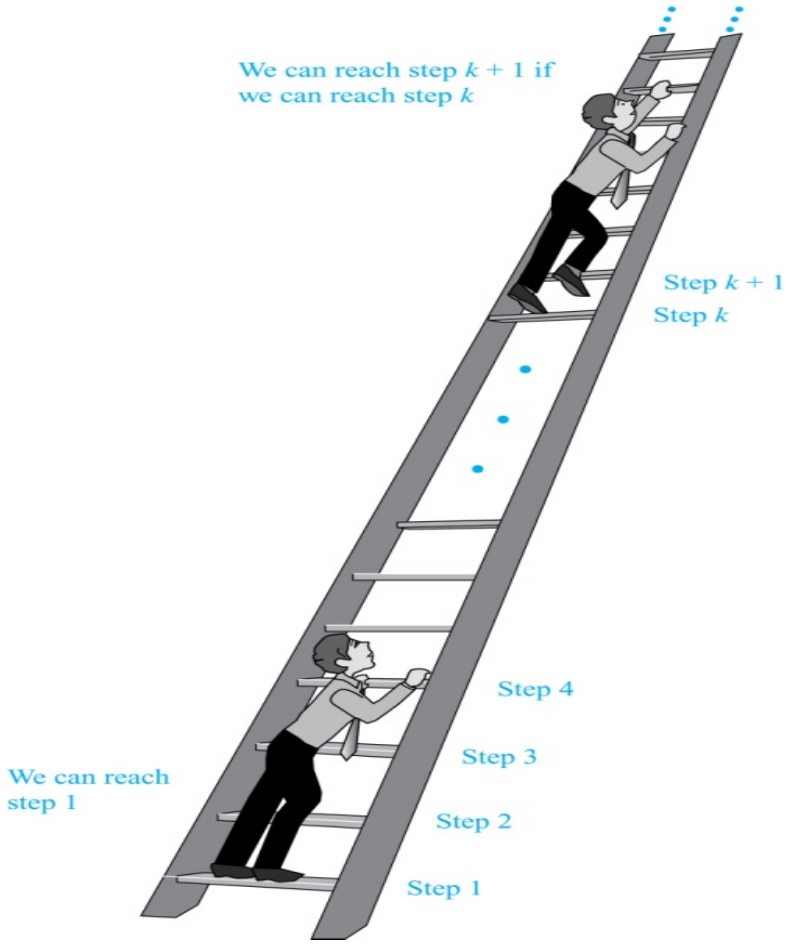
$$(1+2+\cdots+n)+(n+(n-1)+\cdots+1)=$$
$$(n+1)+(n+1)+\cdots(n+1)=n(n+1)$$

2. The specific drawing of the picture isn't actually important
3. The picture can be "scaled up" to as big an n as necessary

Remember: it's not the picture that's the proof–it's the **idea** that the picture is representing that really counts

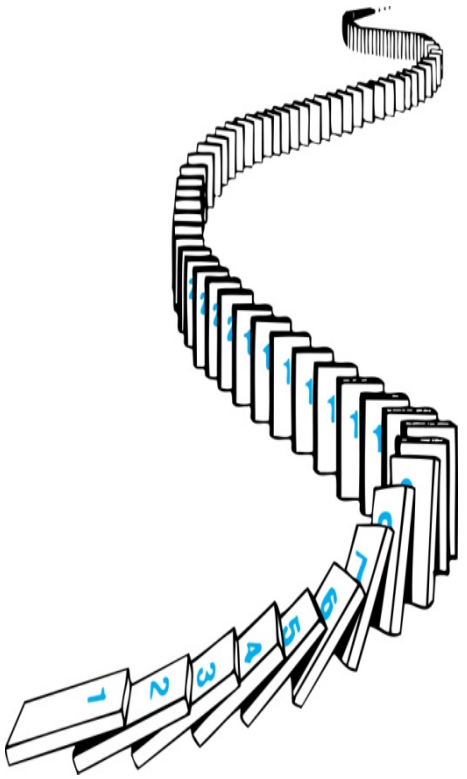
Mathematical Induction (idea)

- Suppose we have an infinite ladder:
  1. We can reach the first rung of the ladder
  2. If we can reach a particular rung of the ladder, then we can reach the next rung



Ordinary Induction (principle)

- Goal: prove that $P(n)$ -- predicate on nonnegative integers -- is true for all n
  1. *Basis step*: show that P(0) is true
  2. *Inductive hypothesis*: assume that P(k) holds for an arbitrary (integer) k
  3. *Inductive step*: show that P(k) → P(k + 1) holds for all k



Induction (rule)

- Rule of inference:
  1. Premise 1: $P(0)$
  2. Premise 2: $\forall k\,[\,P(k) \rightarrow P(k+1)\,]$
  3. Conclusion: $\forall n\,P(n)$
- Note: in a proof by mathematical induction, we don't assume that $P(k)$ is true for all positive integers!

Sum of Integers (proof)

Ind. Step.

Ind Hyp. $\sum_{j=1}^{k} j = \frac{k(k+1)}{2}$ when $k = n$.

$$\sum_{j=1}^{n+1} j = \sum_{j=1}^{n} j + (n+1)$$

$$= \frac{n(n+1)}{2} + (n+1) \text{ by ind hyp}$$

$$= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \text{ by arithmetic}$$

$$= \frac{n(n+1)+2(n+1)}{2} \text{ by arithmetic}$$

$$= \frac{(n+2)(n+1)}{2} \text{ distrib in numerator}$$

$$= \frac{(n+1)(n+2)}{2} \text{ commutativity} \quad \diamondsuit$$

Sum of Odd Integers (proof)

Basis Step. $\left[\sum_{j=1}^{k}(2j-1) = k^2\right]$ when $k = 0$

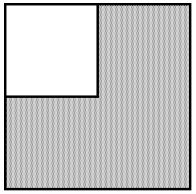
Ind Hyp. $\left[\sum_{j=1}^{k}(2j-1) = k^2\right]$ when $k = n$

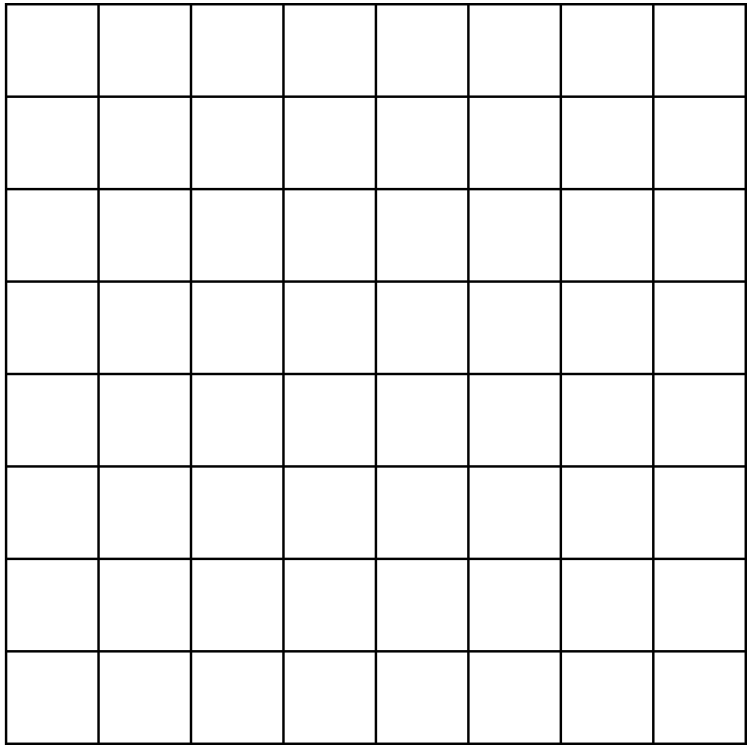
Ind Step. Consider the case $k = n+1$.

$$\begin{aligned}\sum_{j=1}^{n+1}(2j-1) &= \sum_{j=1}^{n}(2j-1) + [2(n+1)-1] \\ &= \sum_{j=1}^{n}(2j-1) + 2n + 1 \\ &= n^2 + 2n + 1 \text{ by ind. hyp.} \\ &= (n+1)^2 \text{ by factoring } \diamondsuit\end{aligned}$$

Tiling Boards

- Problem: can we tile a 2^k – by – 2^k board with one covered square with L-shaped tiles?





Different Base Case

Example 5.2.2: $2^n > n^2$ for all $n \geq 5$.

Basis Step. $2^5 > 5^2$

Ind Hyp. Assume $2^k > k^2$ for $k \geq 5$

Ind. Step.

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k && \text{arithmetic} \\ &= 2^k + 2^k && \text{arithmetic} \\ &> k^2 + k^2 && \text{ind. hyp.} \\ &> k^2 + (2k+1) && \text{by Example 5.2.1} \\ &= (k+1)^2 && \text{arithmetic} \end{aligned}$$

$\diamondsuit$

Postage Example

Example 5.2.3: Prove that any postage of $8¢$ or more can be created from nothing but $3¢$ and $5¢$ stamps.

Basis Step. $8 = 1 \cdot 3¢ + 1 \cdot 5¢$

Ind Hyp. Assume $n¢$ possible from 3's and 5's.

Ind. Step. Try to make $(n+1)¢$ postage.

Suppose that $n = r \cdot 3¢ + s \cdot 5¢$

Case 1: $s \geq 1$. Then $n+1 = \ldots$

Case 2: $s = 0$. Then $n+1 = \ldots$

Strong Induction (rule)

- Goal: prove that $P(n)$ -- predicate on nonnegative integers -- is true for all n
  1. *Base case*: show that P(0) is true
  2. *Inductive hypothesis*: assume that P(k) holds for all integers less than an arbitrary (integer) k
  3. *Inductive step*: show that [P(0),P(1), ... ,P(k)] $\rightarrow$ P(k + 1) holds for all k
- Rule of inference:
  1. Premise 1: $P(0)$
  2. Premise 2: $\forall k\ [\ [\forall j \leq k, P(j)] \rightarrow P(k+1)\ ]$
  3. Conclusion: $\forall n\ P(n)$

Product of Primes Example

- Theorem: every integer > 1 is a product of prime numbers
  1. *Define predicate*: P(n) ::= "n is a product of primes"
  2. *Base case*: P(2) is true since 2 is prime (product of length 1)
  3. *Inductive hypothesis*: assume that for all integers less than an arbitrary (integer) k ≥ 2, k is a product of primes
  4. *Inductive step*: show that k + 1 must be a product of primes
- Proof idea:
  1. If k+1 is itself prime, then it is a product of length 1 by definition
  2. If k+1 is not prime, then by definition k+1 = a*b. By Ind. Hyp. {a,b} are products of primes

Recursion

Recursively Defined Functions

- Problem: given a sequence (a_0, a_1,…,a_k) construct a consistent rule to determine any (nth) term:
  - By recursion
  - Closed form (can be difficult!)
  - A function $f(n)$ is the same as a sequence where $f(i) = a_i$
- Recursive definition:
  1. *Basis step*: specify the value of the function at zero
  2. *Recursive step*: give a rule for finding its value at an integer from its values at smaller integers.

Recursively Defined Functions (examples)

Example 2.4.5: $1, 3, 5, 7, 9, 11, \ldots$

recursion: $a_0 = 1; \quad a_n = a_{n-1} + 2$ for $n \geq 1$

closed form: $a_n = 2n + 1$

Example 2.4.6: $1, 3, 7, 13, 21, 31, 43, \ldots$

recursion: $b_0 = 1; \quad b_n = b_{n-1} + 2n$ for $n \geq 1$

closed form: $b_n = n^2 + n + 1$

Example 2.4.7: $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$

recursion: $c_0 = 1, c_1 = 1;$

$c_n = c_{n-1} + c_{n-2}$ for $n \geq 1$

closed form: $c_n = \frac{1}{\sqrt{5}} \left[G^{n+1} - g^{n+1} \right]$ where $G = \frac{1+\sqrt{5}}{2}$ and $g = \frac{1-\sqrt{5}}{2}$

More Functions (examples)

Example 5.3.1: n-factorial $n!$

$$(B) \qquad 0! = 1$$

$$(R) \qquad (n+1)! = (n+1) \cdot n!$$

Example 5.3.5: partial sums of sequences

$$\sum_{j=0}^{n} a_j = \begin{cases} a_0 & \text{if } n = 0 \\ \sum_{j=0}^{n-1} a_j \; + \; a_n & \text{otherwise} \end{cases}$$

Example 5.3.3: Hanoi sequence $0, 1, 3, 7, 15, \ldots$

$$h_0 = 0$$

$$h_n = 2h_{n-1} + 1 \quad \text{for} \quad n \geq 1$$

Recursively Defined Sets

- To define a set recursively:
  1. *Basis step*: specify initial collection of elements
  2. *Recursive (constructor) step*: give a rule for forming new elements from old ones
- Example: the natural numbers **N**
  - *(B) Basis step:* $0 \in \mathbf{N}$
  - *(R) Recursive step*: If n is in **N**, then $n + 1$ is in **N**

Strings (definition)

- Definition: a set of characters/letters/symbols is called an *alphabet*
- Definition: a sequence in an alphabet is a *string*

Example 2.4.3: Some common alphabets:

- $\{0, 1\}$ the binary alphabet
- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ the decimal digits
- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ the hexadecimal digits
- $\{A, B, C, D, \ldots, X, Y, Z\}$ English uppercase
- ASCII

Strings (recursive definition)

- Given alphabet ***A***, define recursive data type $\boldsymbol{A}^*$ of strings over ***A***:
  - *(B)*: $\lambda \in \boldsymbol{A}^*$ (λ is the empty string)
  - *(R)*: If $a \in \boldsymbol{A}$ and $s \in \boldsymbol{A}^*$, then $sa \in \boldsymbol{A}^*$
- Example:
  - A = {0,1}
  - A^*: all bit strings, λ,0,1, 00,01,10, 11, etc.
- Example:
  - A = {a,b} show that $aab \in A^*$

1) $\lambda \in A^*$ *and* $a \in A \rightarrow a \in A^*$

2) $a \in A^*$ *and* $a \in A \rightarrow aa \in A^*$

3) $aa \in A^*$ *and* $b \in A \rightarrow aab \in A^*$

Strings (examples)

Example 5.3.12: binary strings of even length

(B) $\lambda \in S$

(R) If $b \in S$, then $b00, b01, b10, b11 \in S$.

Example 5.3.13: binary strings of even length that start with 1

(B) $10, 11 \in S$

(R) If $b \in S$, then $b00, b01, b10, b11 \in S$.

Example 5.3.16: set of binary palindromes

(B) $\lambda, 0, 1 \in S$

(R) If $x \in S$ then $0x0, 1x1 \in S$.

String Concatenation

- Given alphabet $\boldsymbol{A}$, and a set of strings $\boldsymbol{A}^*$, define the *concatenation* of two strings, denoted by $\cdot$, recursively
  - *(B)*: If $w \in \boldsymbol{A}^*$ then $w \cdot \lambda = w$
  - *(R)*: If $w_1 \in \boldsymbol{A}^*$, $w_2 \in \boldsymbol{A}^*$, and $x \in \boldsymbol{A}$, then

$$w_1 \cdot (w_2 \cdot x) = (w_1 \cdot w_2)\, x$$

String Length

- Given alphabet $\boldsymbol{A}$, and a set of strings $\boldsymbol{A}^*$, recursively define the *length* of string w denoted by $|w|$:
  - (B): $|\lambda| = 0$
  - (R): $|wx| = |w| + 1$, where $w \in \boldsymbol{A}^*$, $x \in \boldsymbol{A}$

Structural Induction

- Goal: prove that *P*(*r*) -- predicate on recursively defined set R -- is true for all elements of the set r ∈ R

1. *Base case*: show that P(b) is true for base case elements b ∈ R
2. *Inductive hypothesis*: assume that P(k) holds for all elements used to construct new elements in the recursive step
3. *Inductive step*: show that the result holds for the newly constructed elements

Algorithm (definition)

- Definition: an *algorithm* is a finite set of precise instructions for performing computation or solving a problem
- General considerations:
  - Running time, resources
  - Average/worst/best case scenarios
  - When do we terminate the algorithm?
  - How do we compare algorithms?

Pseudocode (definition)

- Pseudocode: representation of an algorithm in prose + code. Preparation step before implementation

Algo 3.1.1: Find Maximum

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
Output: largest integer in array

$\{Initialize\}$ $max := a_1$

For $i := 2$ **to** n
  If $max < a_i$ **then** $max := a_i$
  Continue with next iteration of for-loop.

Return (max)

Binary Search (idea)

- Problem definition:
  - Given sorted list (vector) of numbers V, and a number X. Find the index k such that V(k) = X. Return k = -1 if X is not in V.
- Solution approach:
  - Scan through the list and compare X to each element
  - Linear running time, does not take advantage of the list being sorted
- Better approach:
  - *Divide and conquer* algorithms
  - Break problem into subproblems of the same type
  - Constant time to cut problem size by a fraction (usually ½)

Binary Search (algorithm)

- Pseudocode:
  - **Input**: *sorted* (ascending order) vector V with N elements, element X
  - **Goal**: find index k where V(k) = X or return -1
  - **Assumption**: > < operators exist
  - **Algorithm**:

1. low_k = 1, high_k = N
2. while low_k <= high_k % we still have indices to check
   m = (low_k + high_k) / 2 % middle element
   compare X to V(m)
   If X == V(m) ➔ stop, return k = m
   If X > V(m) ➔ search right half: low_k = m + 1;
   if X < V(m) ➔ search left half: high_k = m – 1;
3. Return k = -1 % we failed to find X in V

Sorting (selection sort algorithm)

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:
    1. Set k = 1
    2. Locate minimum element in (sub)vector V(k..N)
    3. Switch (swap) that element with element at index k
    4. Increment k (k = k+1) and go to step 2, stop when k = N-1

Selection Sort (analysis)

- Estimate efficiency of sorting algorithm:
  - Number of element comparisons
  - Number of element exchanges
- Selection sort:
  - First iteration of the loop: N-1 comparisons, 1 exchange
  - Some iteration: N-k comparisons, 1 exchange
  - Last iteration: 1 comparison, 1 exchange
  - Total comparisons: (N-1) + (N-2) + …..+ 2 + 1 = N x (N-1) / 2
  - Total exchanges: N-1 (at most)

Big-O Notation (idea)

- Estimate efficiency of algorithm, relative to other algorithms for identical task
  - Difficult to get precise measure
  - Approximate effect on change of number of items (N) processed
  - Compare growth rates
  - Order of magnitude (O)

Big-O Notation

- Comparing growth rates:

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is ***asymptotically dominated*** by g if

$$(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq g(x)]$$

NOTATION: $f \preceq g$.

- Function classes:

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ ("***big-oh of g***") if

$$(\exists C \in \mathbb{R})\,[f \preceq Cg]$$

NOTATION: $f \in \mathcal{O}(g)$.

Witnesses

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ ("***big-oh of g***") if

$$(\exists C \in \mathbb{R})\,(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq Cg(x)]$$

DEF: In the definition above, the multiplier C and the location K on the x-axis after which $Cg(x)$ dominates $f(x)$ are called the ***witnesses*** to the relationship $f \in \mathcal{O}(g)$.

Big-O Notation (example)

Example 3.2.1: $4n^2 + 21n + 100 \in \mathcal{O}(n^2)$

Pf: First suppose that $n \geq 0$. Then

$$\begin{aligned} 4n^2 + 21n + 100 &\leq 4n^2 + 24n + 100 \\ &\leq 4(n^2 + 6n + 25) \\ &\leq 8n^2 \text{ which holds whenever} \end{aligned}$$

$n^2 \geq 6n + 25$, which holds whenever

$n^2 - 6n + 9 \geq 34$, which holds whenever

$n - 3 \geq \sqrt{34}$, which holds whenever $n \geq 9$.

Thus,

$$(\forall n \geq 9)[4n^2 + 21n + 100 \leq 8n^2] \qquad \diamond$$

Witnesses (example 1)

$$(\forall n \geq 9)\left[4n^2 + 21n + 100 \leq 8n^2\right] \qquad \Diamond$$

$$C = 8 \quad \text{and} \quad K = 9$$

are witnesses to the relationship

$$4n^2 + 21n + 100 \in \mathcal{O}(n^2)$$

Larger values of C and K could also serve as witnesses. However, a value of C less than or equal to 4 could not be a witness.

Witnesses (example 2)

Example 3.2.4: $2^n \in \mathcal{O}(n!)$.

Pf:

$$\overbrace{2 \cdot 2 \cdots 2}^{n \text{ times}} = 2 \cdot 1 \cdot \overbrace{2 \cdot 2 \cdots 2}^{n-1 \text{ times}}$$

$$\leq 2 \cdot 1 \cdot 2 \cdot 3 \cdots n = 2n!$$

We have used the witnesses $C = 2$ and $K = 0$. $\Diamond$

Algorithmic Complexity

- Complexity is a measure of resource (time/space) consumption
- Time complexity: number of computational steps required to execute an algorithm as a function of input size
- Estimate running time of algorithm:
  - Worst case scenario: bound
  - Average case scenario: hard to compute
  - Analysis pinpoints bottlenecks
  - No particular units of time
  - Analyze inside out

P vs NP

- "P": class of problems which can be solved with polynomial time algorithms
- "NP": (nondeterministic polynomial time, exponential) class of problems whose solution can be verified in polynomial time
- Implications of P = NP:
  - Complete chaos
  - Can solve problems as quickly as we can verify the solution
  - Cryptography breaks
  - Mathematicians replaced by machines

Factorial

Algo 5.4.5: factorial

recursive function: factorial(n)
Input: integer $n \geq 0$
Output: $n!$

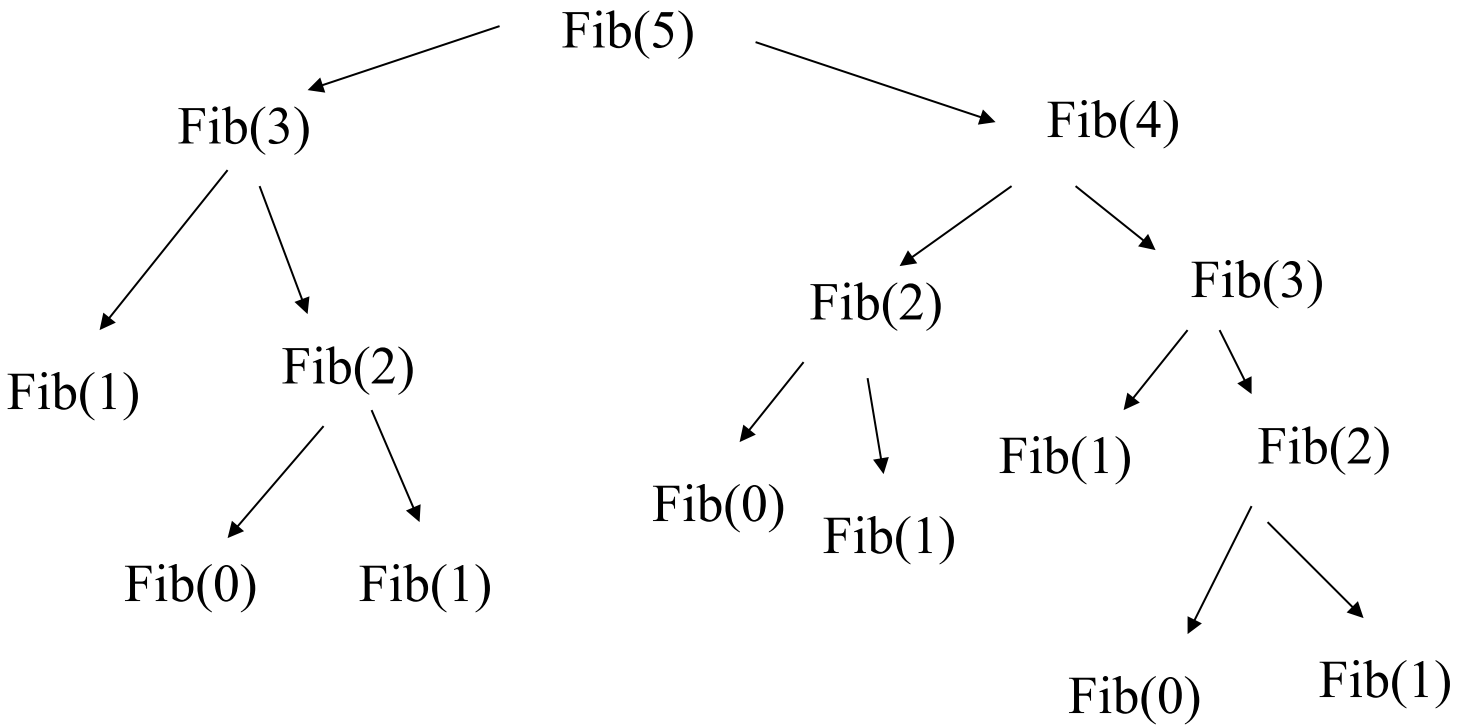
If $n = 0$ **then return** (1)
else return $\big(\text{prod}(n, \text{factorial}(n-1))\big)$

Fibonacci Numbers

- $F_0 = 1, F_1 = 1, F_2 = 2, F_3 = 3, F_4 = 5, \ldots,$ $\mathbf{F_i = F_{i-1} + F_{i-2}}$
- Clever use of recursion?

```
function  F = fib(N)
if  N  <=  1
      F = 1;
else
      F = fib(N-1) + fib(N-2);
end
```

Fibonacci Function (call tree)



Binary Search (analysis)

- Binary search is a recursive algorithm, we can define and solve a *recurrence relation*:
  - Base case: T(0) = constant
  - Recursive case: T(N) = T(subproblems) + T(combine solutions)
- Running time depends on:
  - Number of subproblems
  - Size of subproblems
  - Cost of combining solutions

Mergesort (idea)

- Classic divide and conquer strategy:
  - Divide list into 2 halves (each half = subproblem)
  - Apply algorithm recursively to sort each half
  - Merge the two sorted lists
- Merging two sorted lists:
  - One pass through the input (N elements)
  - Linear running time, at most N-1 comparisons
  - Requires a temporary array (additional resource)

Mergesort (algorithm)

- Pseudocode:
  - **Input**: vector V with N elements
  - **Goal**: sort vector in ascending order
  - **Assumption**: comparison based sorting (> < operators exist)
  - **Algorithm**:

```
1. left = 1, right = N
2. if  left < right                    % we still need to sort
       m  =  (left + right) / 2        % middle element
       mergeSort(V, left, m, T)        ➔ sort left half
       mergeSort(V, m+1, right, T)     ➔ sort right half
       merge(V, left, m+1, right, T)   ➔ merge sorted halves
3. Return V                            % finished
```