# MAT 7003 : Mathematical Foundations

## (for Software Engineering)

**J Paul Gibson, A207**

paul.gibson@it-sudparis.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/MAT7003/

# Complexity & Algorithm Analysis

…/~gibson/Teaching/MAT7003/L9-Complexity&AlgorithmAnalysis.pdf

# Complexity

To analyze an algorithm is to determine the resources (such as time and storage) necessary to execute it.

Most algorithms are designed to work with inputs of arbitrary length/size.

Usually, the *complexity* of an algorithm is a function relating the input length/size to the number of *fundamental steps* (time complexity) or *fundamental storage locations* (space complexity).

The fundamental steps and storage locations are, of course, dependent on the « physics of the underlying computation machinery ».

# Complexity

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense: estimate the complexity function for arbitrarily large input.

Big O notation, omega notation and theta notation are often used to this end.

For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(n)) ("in logarithmic time“)

Usually asymptotic estimates are used because different implementations of the same algorithm may differ in complexity.

However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

Exact (not asymptotic) measures of complexity can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation.

## Complexity

Time complexity estimates depend on what we define to be a *fundamental step.*

For the analysis to correspond usefully to the actual execution time, the time required to perform a *fundamental step* must be guaranteed to be **bounded above by a constant**.

One must be careful here; for instance, some analyses count an addition of two numbers as one step.

This assumption will be false in many contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

# Complexity

Space complexity estimates depend on what we define to be a *fundamental storage location.*

Such storage must offer reading and writing functions as *fundamental steps*

Most computers offer interesting relations between time and space complexity.

For example, on a Turing machine the number of spaces on the tape that play a role in the computation cannot exceed the number of steps taken.

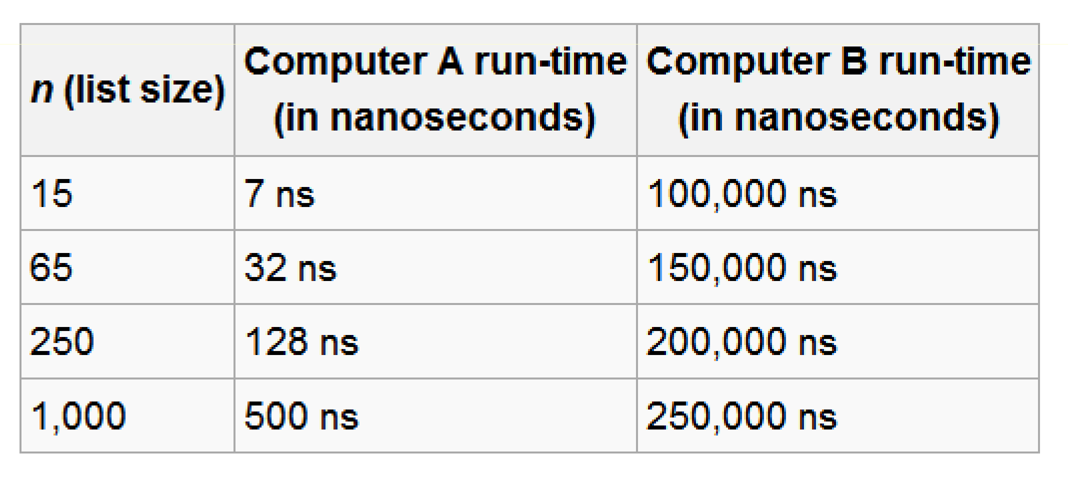
In general, space complexity is bounded by time complexity.

Many algorithms that require a *large time* can be implemented using *small space*

# Complexity: why not just measure empirically?

Because algorithms are platform-independent, e.g. a given algorithm can be implemented in an arbitrary programming language on an arbitrary computer (perhaps running an arbitrary operating system and perhaps without unique access to the machine's resources)

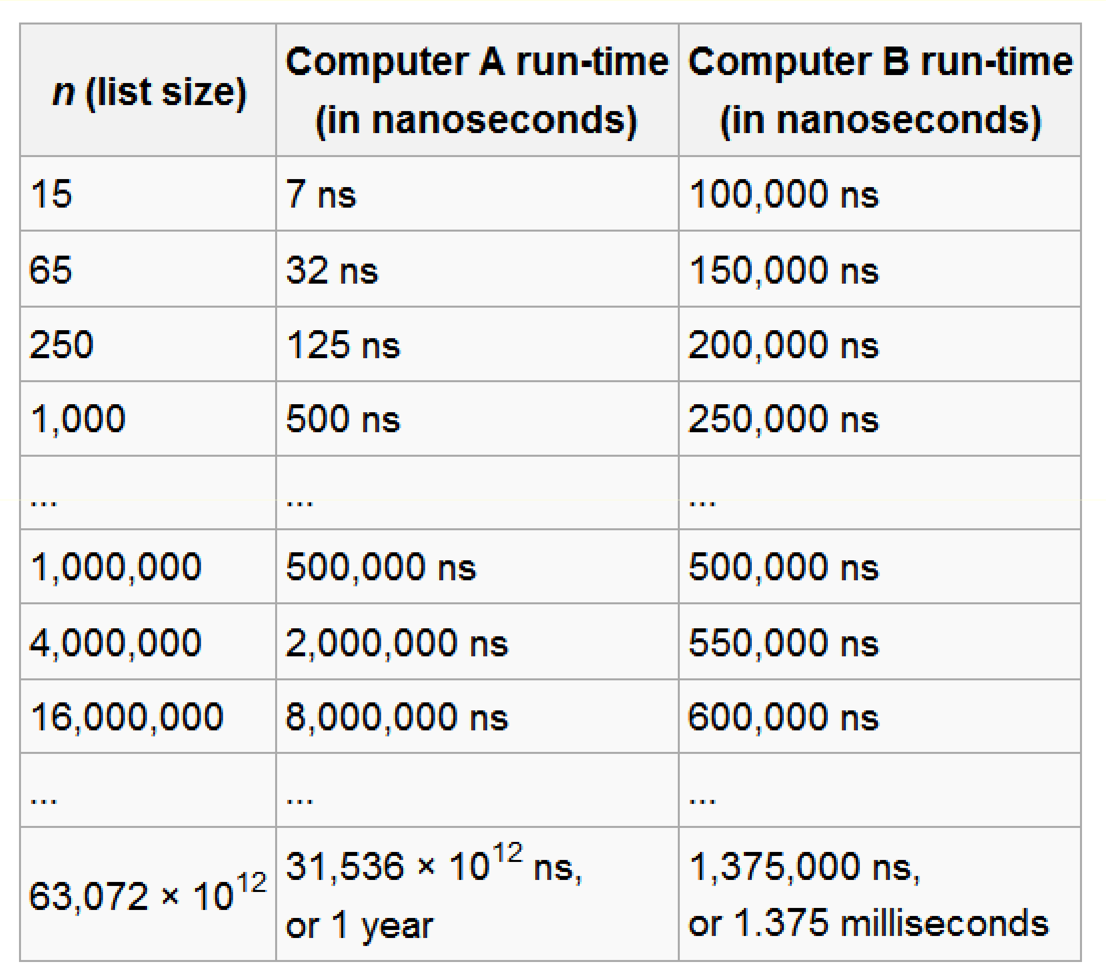
For example, consider the following run-time measurements of 2 different implementations of the same function on two different machines



| *n* (list size) | Computer A run-time (in nanoseconds) | Computer B run-time (in nanoseconds) |
|---|---|---|
| 15 | 7 ns | 100,000 ns |
| 65 | 32 ns | 150,000 ns |
| 250 | 128 ns | 200,000 ns |
| 1,000 | 500 ns | 250,000 ns |

Based on these metrics, it would be easy to jump to the conclusion that Computer A is running an algorithm that is far superior in efficiency to what Computer B is running. However, …

# Complexity: why not just measure empirically?



| *n* (list size) | Computer A run-time (in nanoseconds) | Computer B run-time (in nanoseconds) |
|---|---|---|
| 15 | 7 ns | 100,000 ns |
| 65 | 32 ns | 150,000 ns |
| 250 | 125 ns | 200,000 ns |
| 1,000 | 500 ns | 250,000 ns |
| ... | ... | ... |
| 1,000,000 | 500,000 ns | 500,000 ns |
| 4,000,000 | 2,000,000 ns | 550,000 ns |
| 16,000,000 | 8,000,000 ns | 600,000 ns |
| ... | ... | ... |
| 63,072 × $10^{12}$ | 31,536 × $10^{12}$ ns,<br>or 1 year | 1,375,000 ns,<br>or 1.375 milliseconds |

Extra data now shows us that our original conclusions were false.

Computer A, running the linear algorithm, exhibits a linear growth rate.

Computer B, running the logarithmic algorithm, exhibits a logarithmic growth rate

B is a much better solution for large input

# Complexity: Orders of growth – Big O notation

Informally, an algorithm can be said to exhibit a *growth rate* on the order of a mathematical function if beyond a certain input size n, the function f(n) times a positive constant provides an upper bound or limit for the run-time of that algorithm.

In other words, for a given input size n greater than some $n_o$ and a constant c, an algorithm can run no slower than c × f(n). This concept is frequently expressed using **Big O notation**

For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order **$O(n^2)$.**

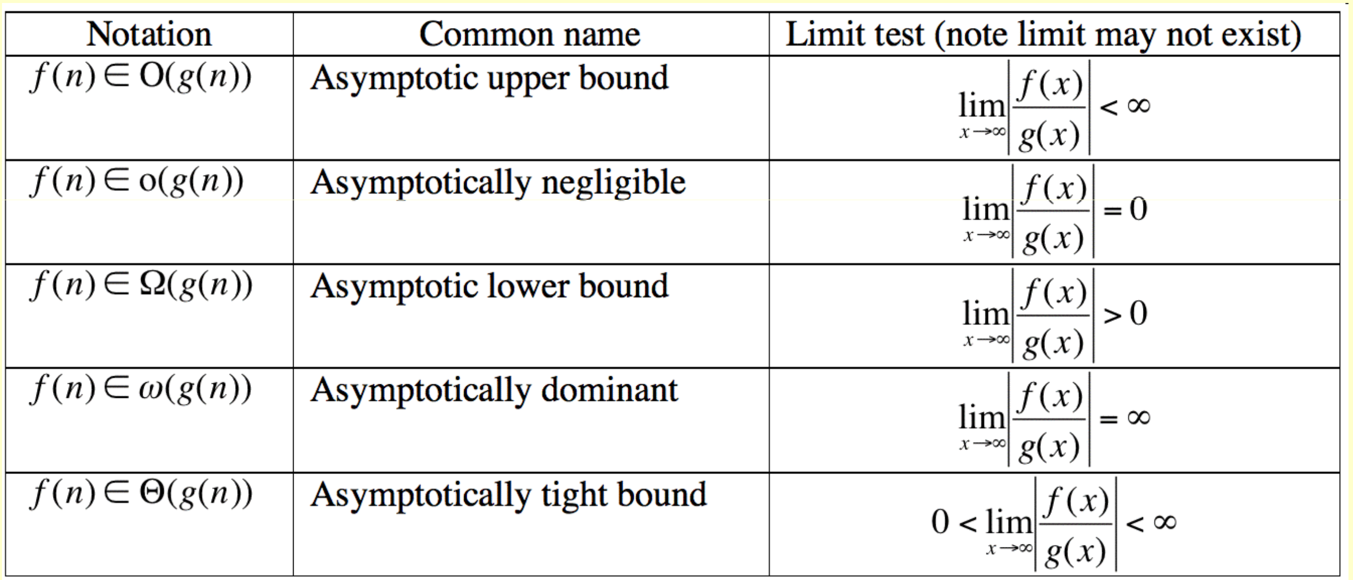
Big O notation is a convenient way to express the *worst-case* scenario for a given algorithm, although it can also be used to express the *average-case* — for example, the worst-case scenario for *quicksort* is $O(n^2)$, but the average-case run-time is O(n lg n).

**Note**: Average-case analysis is much more difficult that worst-case analysis

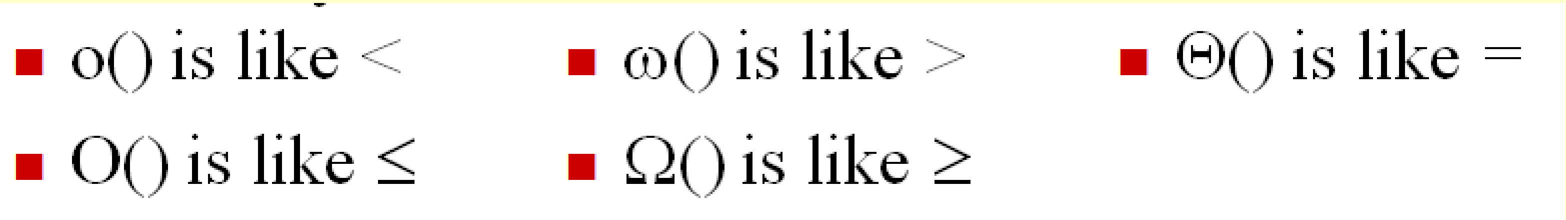
# Complexity: Orders of growth –big/little-omega, big theta, little-o,

Just as *Big O* describes the upper bound, we use Big Omega to describe the lower bound

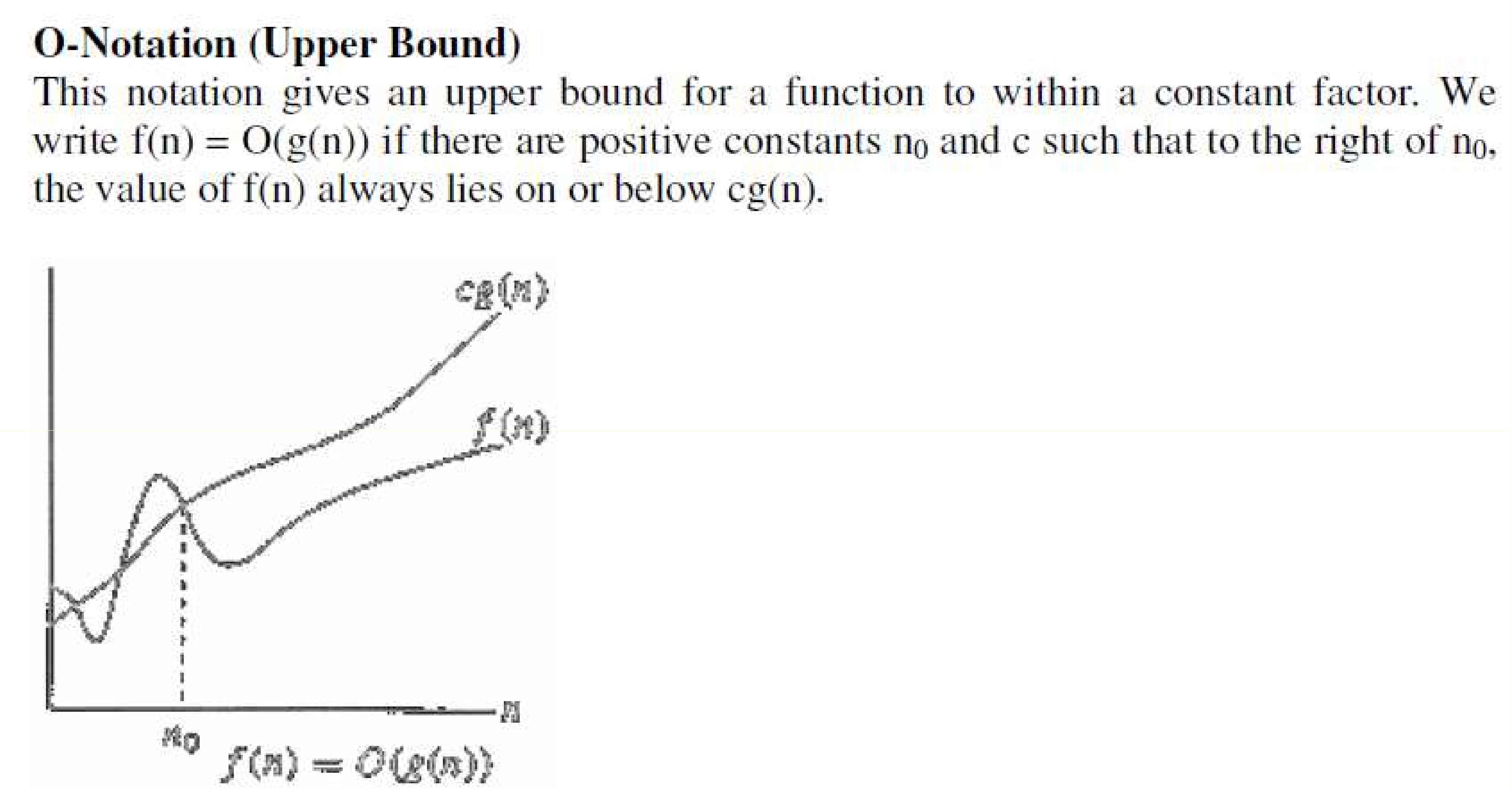
Big Theta describes the case where the upper and lower bounds of a function are on the same order of magnitude.



| Notation | Common name | Limit test (note limit may not exist) |
|---|---|---|
| $f(n) \in \mathrm{O}(g(n))$ | Asymptotic upper bound | $\lim\limits_{x\to\infty}\left\lvert\frac{f(x)}{g(x)}\right\rvert < \infty$ |
| $f(n) \in \mathrm{o}(g(n))$ | Asymptotically negligible | $\lim\limits_{x\to\infty}\left\lvert\frac{f(x)}{g(x)}\right\rvert = 0$ |
| $f(n) \in \Omega(g(n))$ | Asymptotic lower bound | $\lim\limits_{x\to\infty}\left\lvert\frac{f(x)}{g(x)}\right\rvert > 0$ |
| $f(n) \in \omega(g(n))$ | Asymptotically dominant | $\lim\limits_{x\to\infty}\left\lvert\frac{f(x)}{g(x)}\right\rvert = \infty$ |
| $f(n) \in \Theta(g(n))$ | Asymptotically tight bound | $0 < \lim\limits_{x\to\infty}\left\lvert\frac{f(x)}{g(x)}\right\rvert < \infty$ |



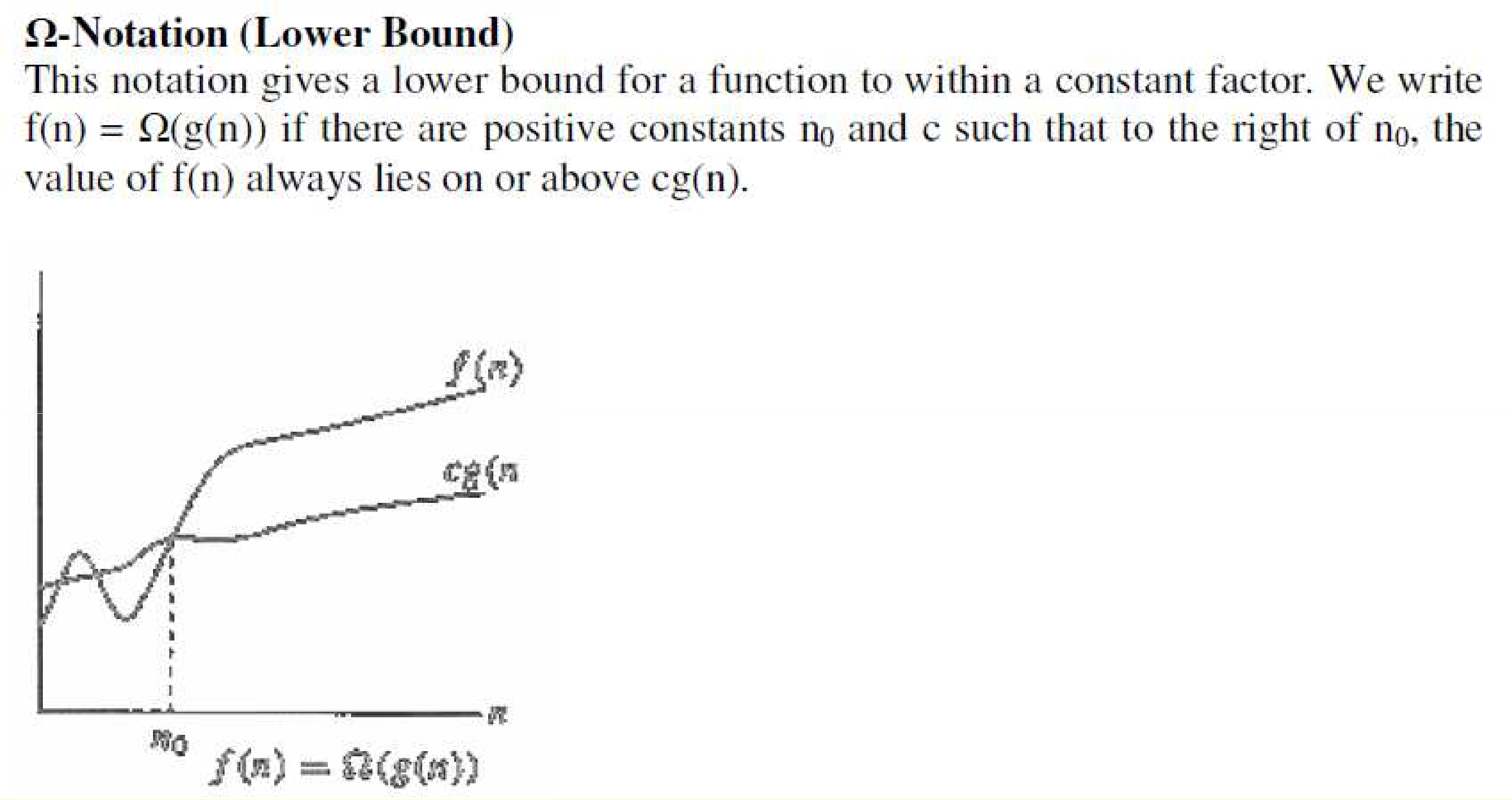
- o() is like $<$
- O() is like $\leq$
- ω() is like $>$
- Ω() is like $\geq$
- Θ() is like $=$



**O-Notation (Upper Bound)**

This notation gives an upper bound for a function to within a constant factor. We write f(n) = O(g(n)) if there are positive constants $n_0$ and c such that to the right of $n_0$, the value of f(n) always lies on or below cg(n).

$f(n) = O(g(n))$

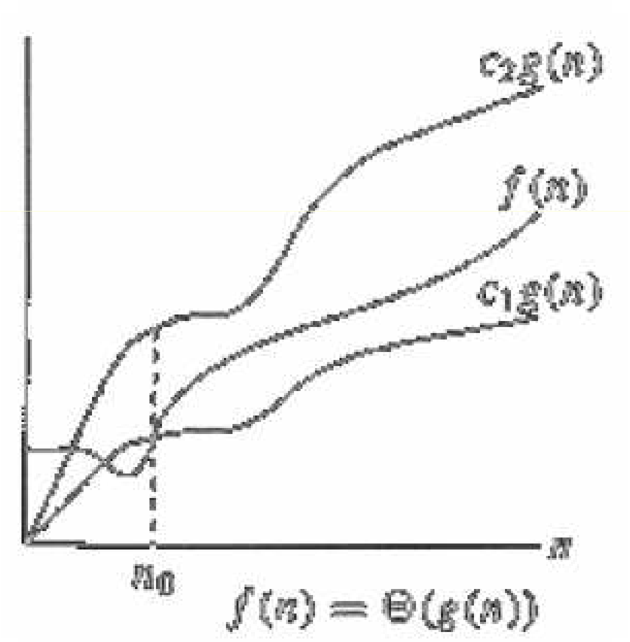


**Ω-Notation (Lower Bound)**

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and c such that to the right of $n_0$, the value of f(n) always lies on or above cg(n).

## Θ-Notation (Same order)

This notation bounds a function to within constant factors. We say f(n) = Θ(g(n)) if there exist positive constants $n_0$, $c_1$ and $c_2$ such that to the right of $n_0$ the value of f(n) always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



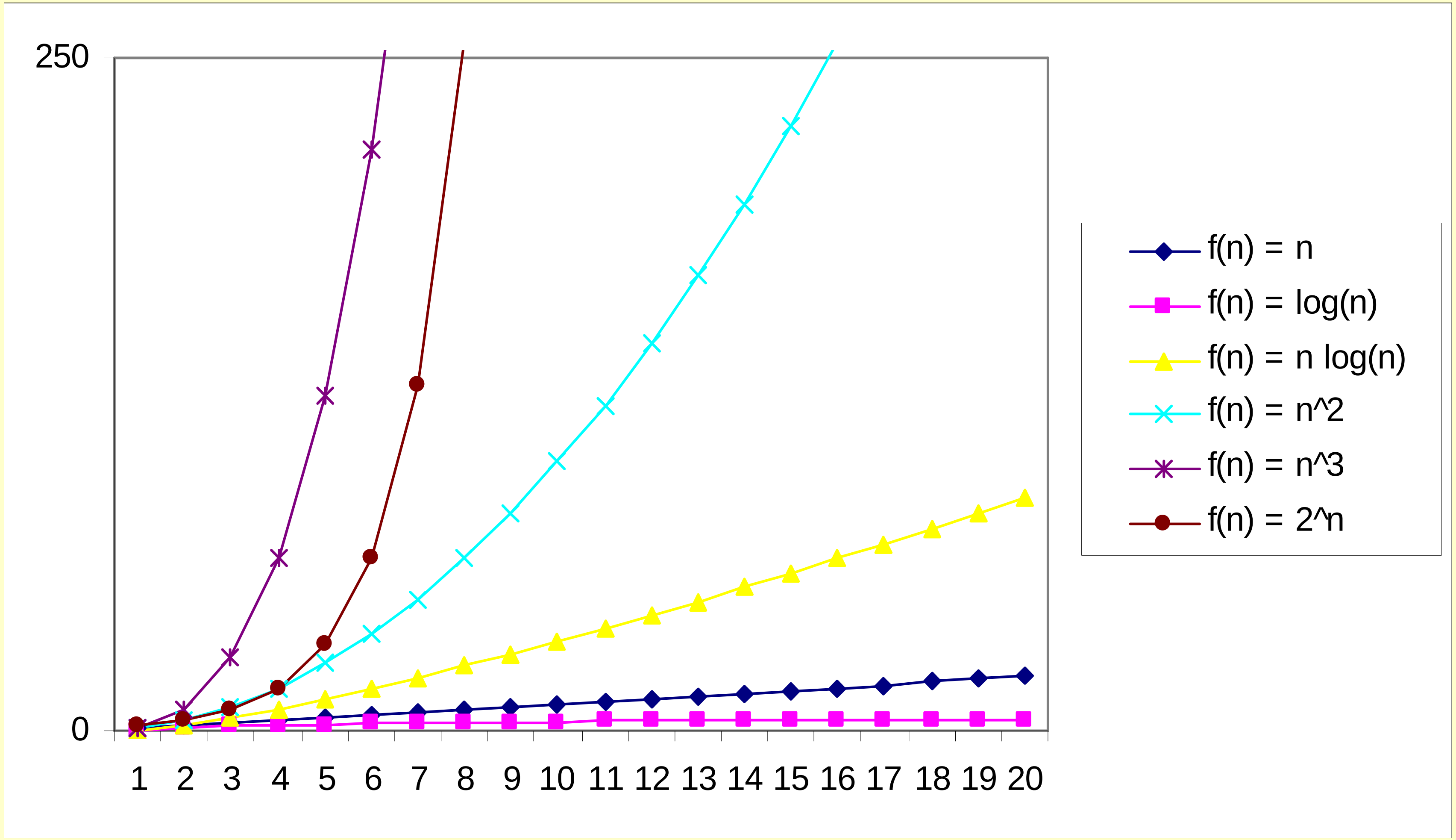
## Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem.

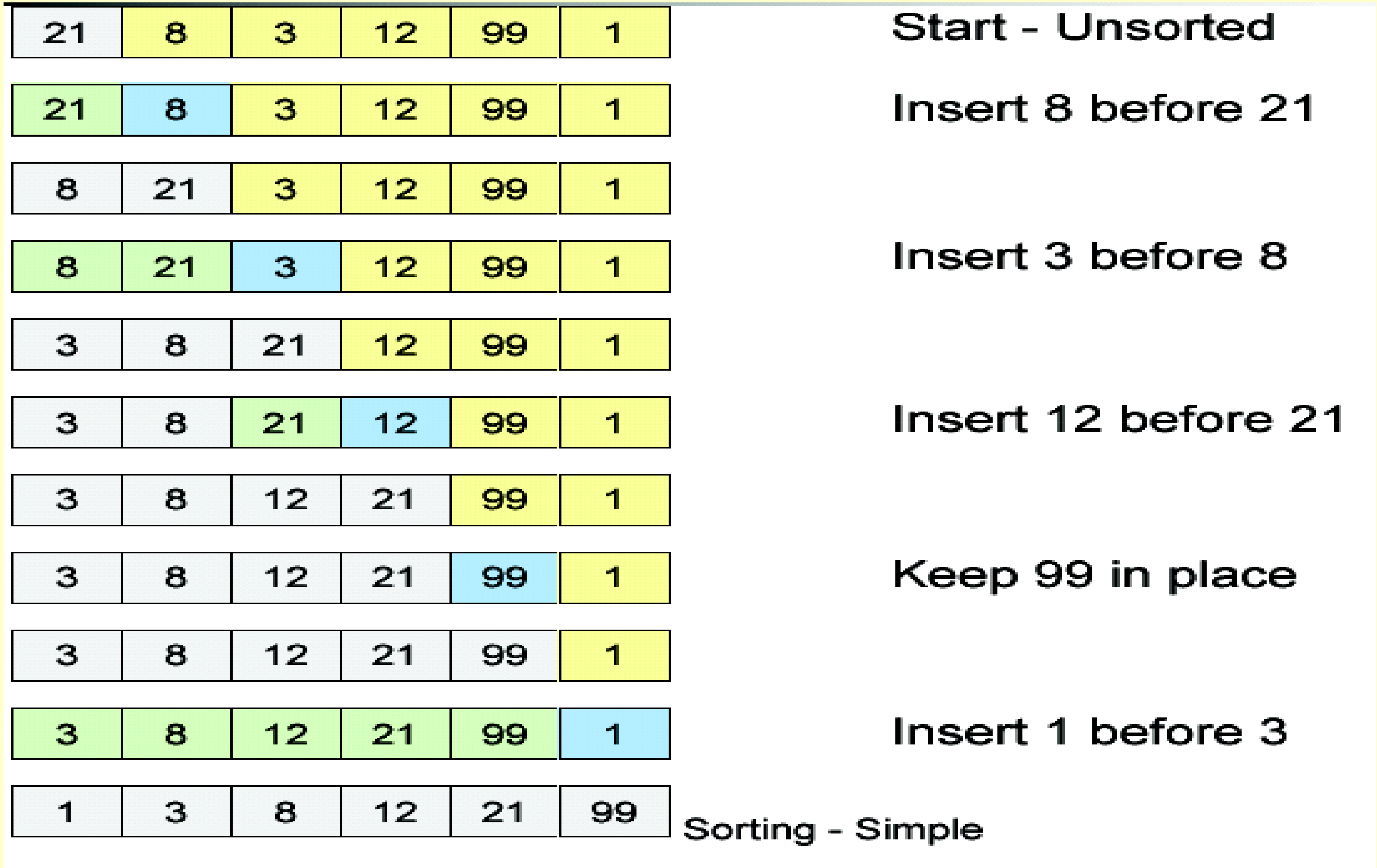
## Reduction

Another technique for estimating the complexity of a problem is the transformation of problems, also called problem reduction. As an example, suppose we know a lower bound for a problem A, and that we would like to estimate a lower bound for a problem B. If we can transform A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

# Classic Complexity curves



# Iterative Algorithm Analysis Example : Insertion Sort



| | | | | | | |
|---|---|---|---|---|---|---|
| 21 | 8 | 3 | 12 | 99 | 1 | Start - Unsorted |
| 21 | 8 | 3 | 12 | 99 | 1 | Insert 8 before 21 |
| 8 | 21 | 3 | 12 | 99 | 1 | |
| 8 | 21 | 3 | 12 | 99 | 1 | Insert 3 before 8 |
| 3 | 8 | 21 | 12 | 99 | 1 | |
| 3 | 8 | 21 | 12 | 99 | 1 | Insert 12 before 21 |
| 3 | 8 | 12 | 21 | 99 | 1 | |
| 3 | 8 | 12 | 21 | 99 | 1 | Keep 99 in place |
| 3 | 8 | 12 | 21 | 99 | 1 | |
| 3 | 8 | 12 | 21 | 99 | 1 | Insert 1 before 3 |
| 1 | 3 | 8 | 12 | 21 | 99 | Sorting - Simple |

# Insertion Sort

```
for (int i = 1; i < a.length; i++)
{
            // insert a[i] into a[0:i-1]
            insert(a, i, a[i]);
}

public static void insert
                        (int[] a, int n, int x)
{
      // insert t into a[0:i-1]
      int j;
      for (j = i - 1;
                        j >= 0 && x < a[j]; j--)
      a[j + 1] = a[j];
      a[j + 1] = x;
}
```

## Insertion Sort - Simple Complexity Analysis

How many compares are done?

1+2+…+(n-1), O(n^2) *worst case*

(n-1)* 1 , O(n) *best case*

How many element shifts are done?

1+2+...+(n-1), O(n2) *worst case*

0 , O(1) *best case*

How much space/memory used?

In-place algorithm

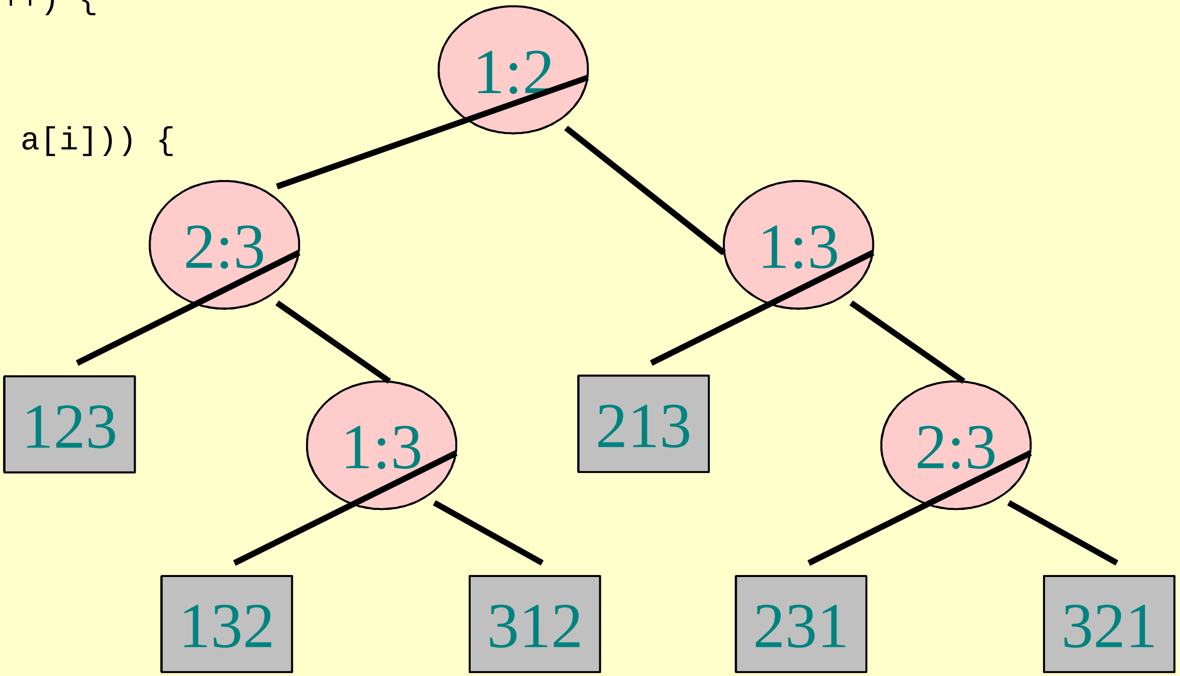
# Proving a Lower Bound for *any comparison based algorithm* for the Sorting Problem

*A decision tree can model the execution of any comparison sort:*

- One tree for each input size *n*.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

# Any comparison sort can be turned into a Decision tree

```
class InsertionSortAlgorithm  {

     for (int i = 1; i < a.length; i++) {

         int j = i;

         while ((j > 0) && (a[j-1] > a[i])) {

                 a[j] = a[j-1];

                  j--;  }

          a[j] = B;  }}
```



**Theorem.** Any decision tree that can sort $n$ elements must have height $\Omega(n \lg n)$.

*Proof.* The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-$h$ binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$\begin{aligned} \therefore\ h &\geq \lg(n!) && \text{(}\lg\text{ is mono. increasing)} \\ &\geq \lg\left((n/e)^n\right) && \text{(Stirling's formula)} \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n)\,. \quad \square \end{aligned}$$

## Divide-and-conquer algorithms

The *divide-and-conquer strategy solves a problem by:*

1. Breaking it into *subproblems that are themselves smaller instances of the same type of* problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

The real work is done piecemeal, in three different places: in the partitioning of problems into subproblems; at the very tail end of the recursion, when the subproblems are so small that they are solved outright; and in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.

# Analysis of Merge Sort – a typical divide-and-conquer algorithm

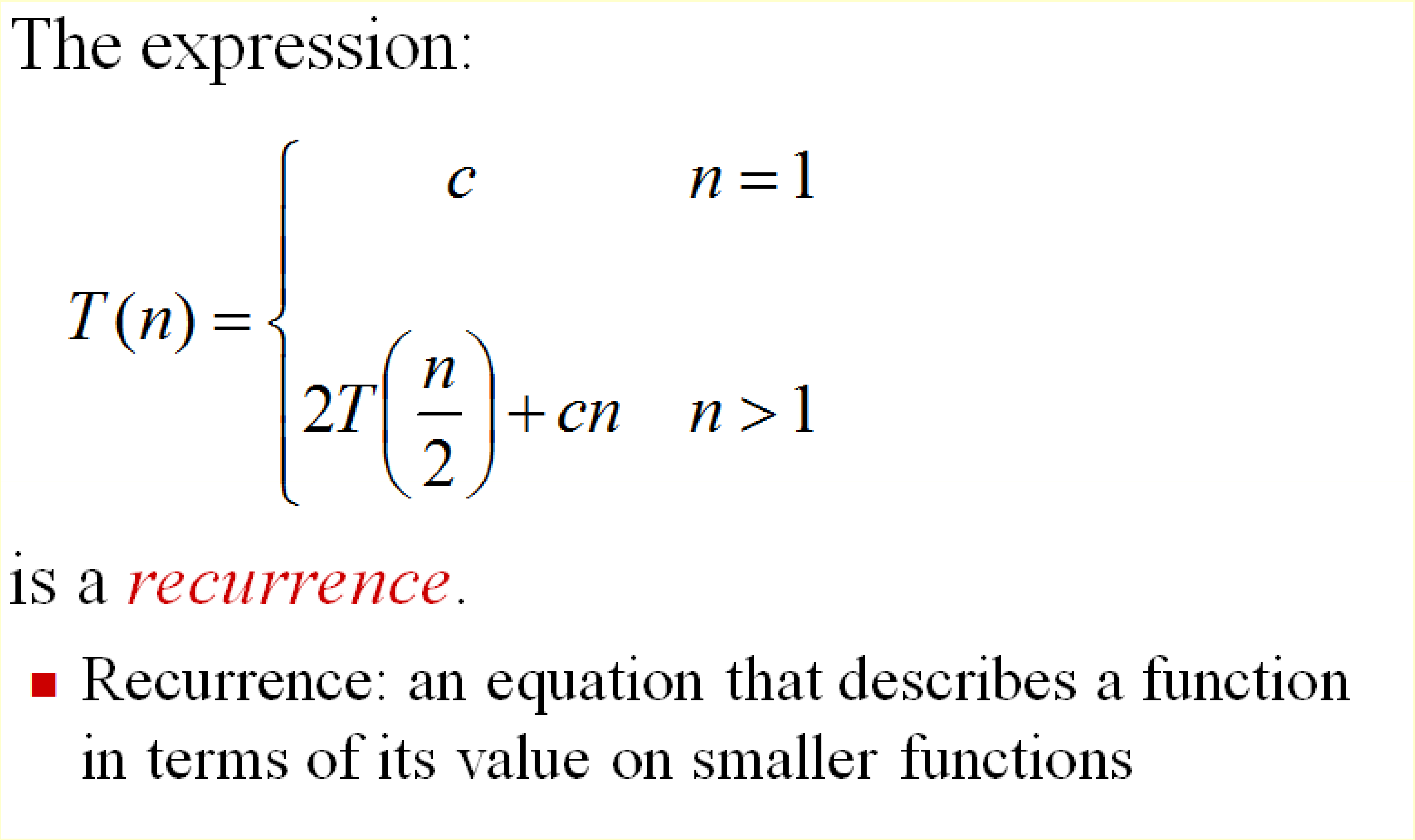
Algorithm Effort

```
MergeSort(A, left, right) {                             T(n)
   if (left < right) {                                  Θ(1)
      mid = floor((left + right) / 2);                     Θ(1)
      MergeSort(A, left, mid);                             T(n/2)
      MergeSort(A, mid+1, right);                          T(n/2)
      Merge(A, left, mid, right);                          Θ(n)
   }
}
```

So $T(n) =$ $\Theta(1)$ when $n = 1$, and

$2T(n/2) + \Theta(n)$ when $n > 1$

## Recurrences



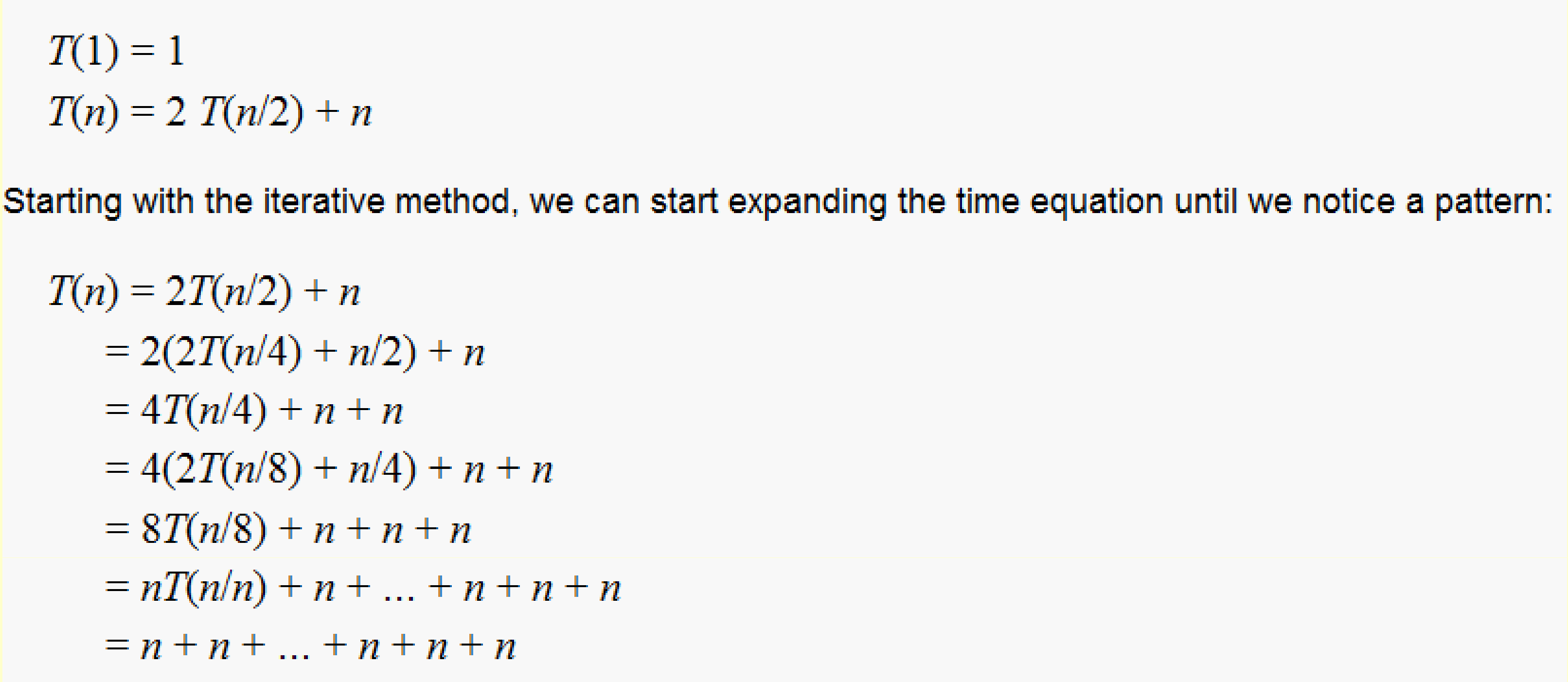
The expression:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + cn & n > 1 \end{cases}$$

is a *recurrence*.

- Recurrence: an equation that describes a function in terms of its value on smaller functions

# Solving Recurrence Using Iterative Method



$T(1) = 1$

$T(n) = 2\ T(n/2) + n$

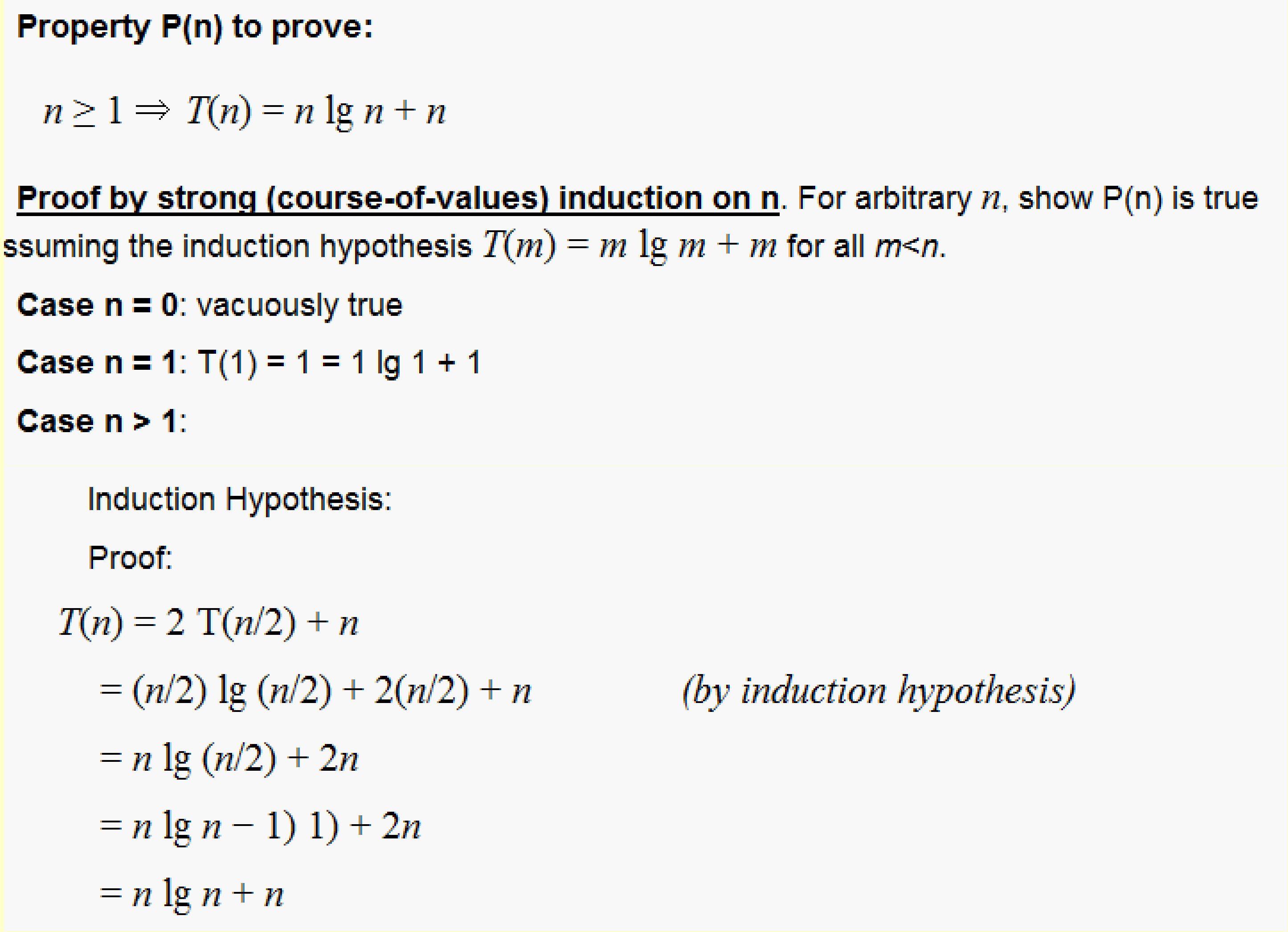
Starting with the iterative method, we can start expanding the time equation until we notice a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n \\
&= 4T(n/4) + n + n \\
&= 4(2T(n/8) + n/4) + n + n \\
&= 8T(n/8) + n + n + n \\
&= nT(n/n) + n + \ldots + n + n + n \\
&= n + n + \ldots + n + n + n
\end{aligned}
$$

Counting the number of repetitions of *n* in the sum at the end, we see that there are lg $n$ + 1 of them. Thus the running time is $n(\lg n + 1) = n \lg n + n$.

We observe that $n \lg n + n < n \lg n + n \lg n = 2n \lg n$ for $n>0$, so the running time is $O(n \lg n)$.

# Proof By Induction



**Property P(n) to prove:**

$$n \geq 1 \Rightarrow T(n) = n \lg n + n$$

**Proof by strong (course-of-values) induction on n**. For arbitrary $n$, show P(n) is true ssuming the induction hypothesis $T(m) = m \lg m + m$ for all $m$<$n$.

**Case n = 0**: vacuously true

**Case n = 1**: T(1) = 1 = 1 lg 1 + 1

**Case n > 1**:

Induction Hypothesis:

Proof:

$$T(n) = 2\,\mathrm{T}(n/2) + n$$

$$= (n/2) \lg (n/2) + 2(n/2) + n \qquad \textit{(by induction hypothesis)}$$

$$= n \lg (n/2) + 2n$$

$$= n \lg n - 1)\ 1) + 2n$$

$$= n \lg n + n$$

## Recurrence relations master theorem

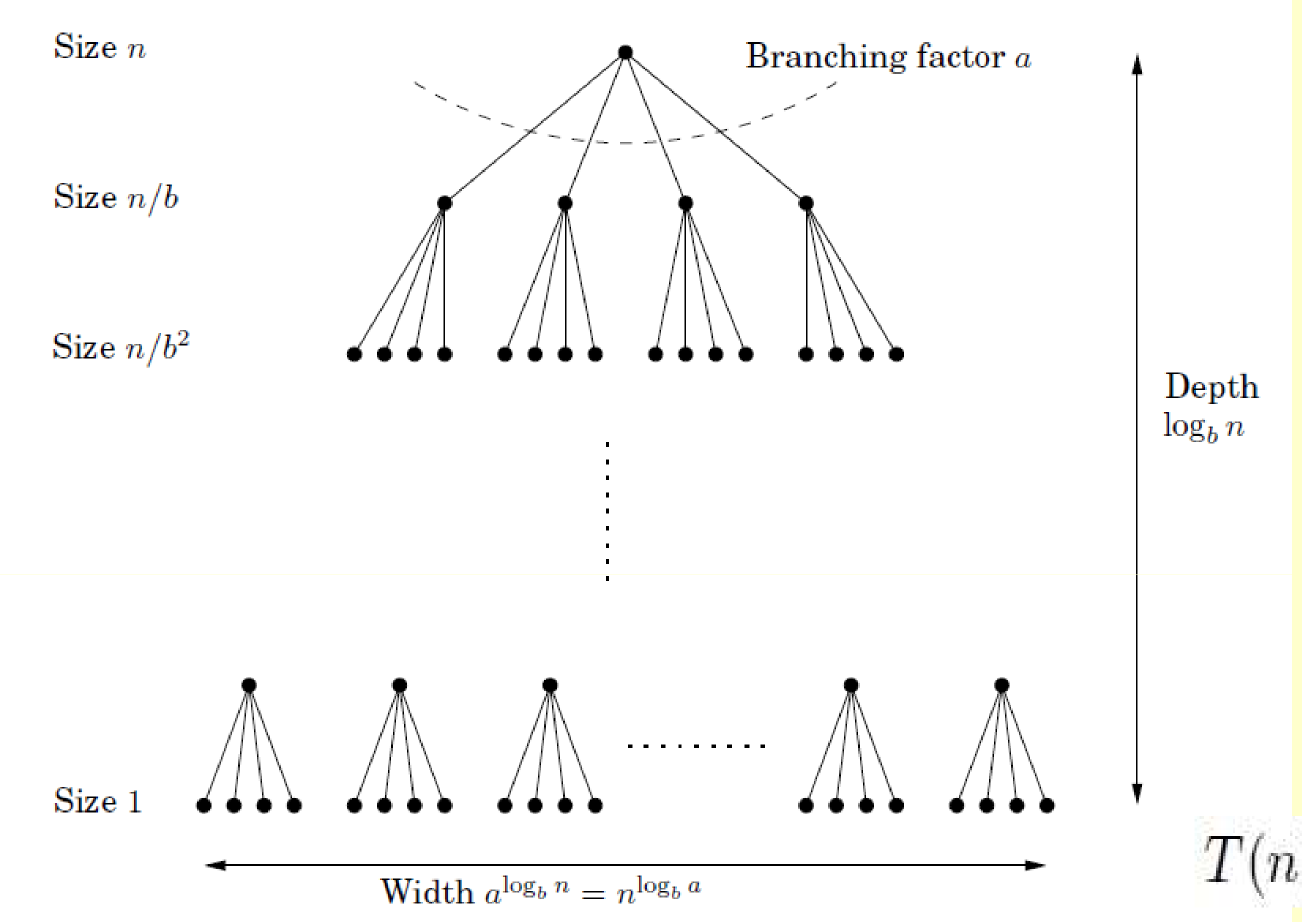
Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size *n* by recursively solving, say, *a* sub-problems of size *n/b* and then combining these answers in O(*n^d*) time, for some *a; b; d > 0*

Their running time can therefore be captured by the equation:

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

We can derive a closed-form solution to this general recurrence so that we no longer have to solve it explicitly in each new instance.

# General Recursive Analysis



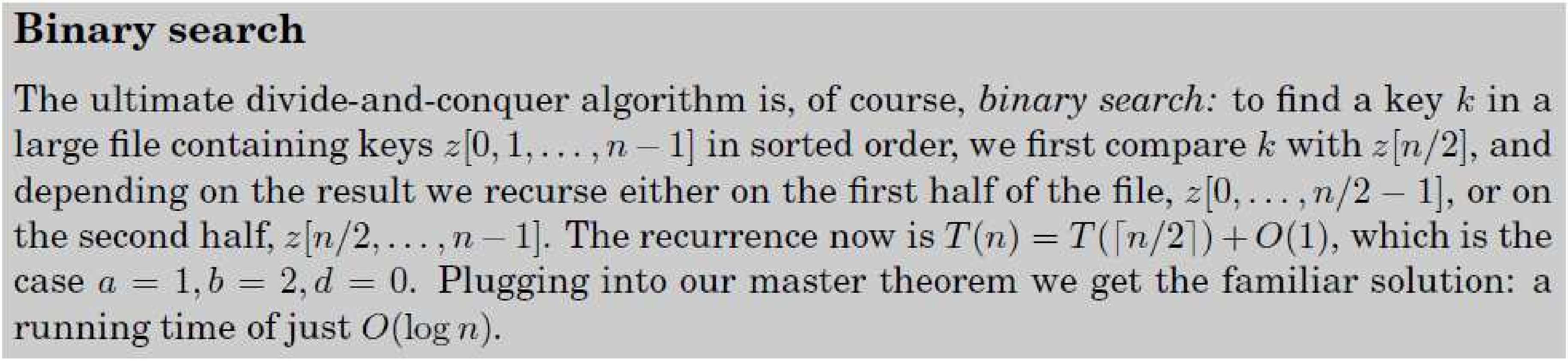
$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

$$\Longrightarrow$$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a\ . \end{cases}$$

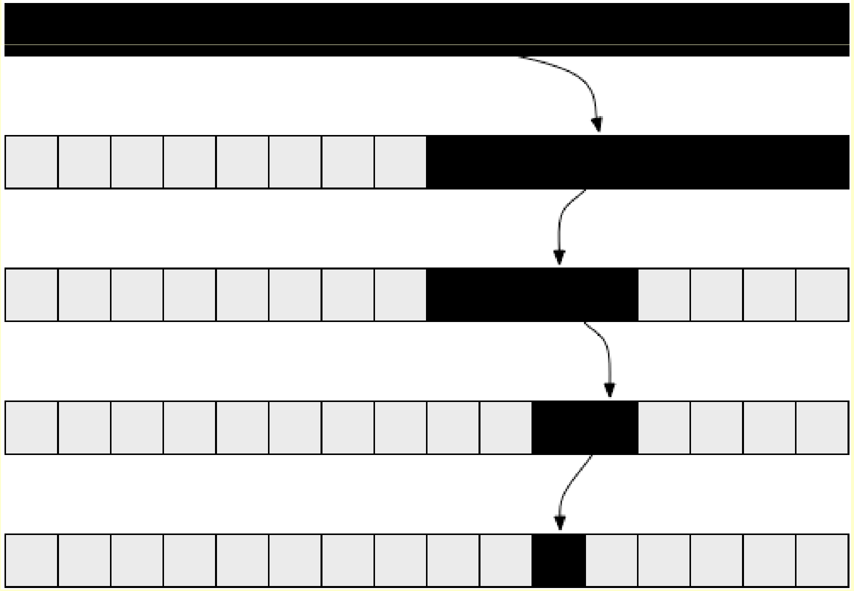
**TO DO**: Can you prove this theorem

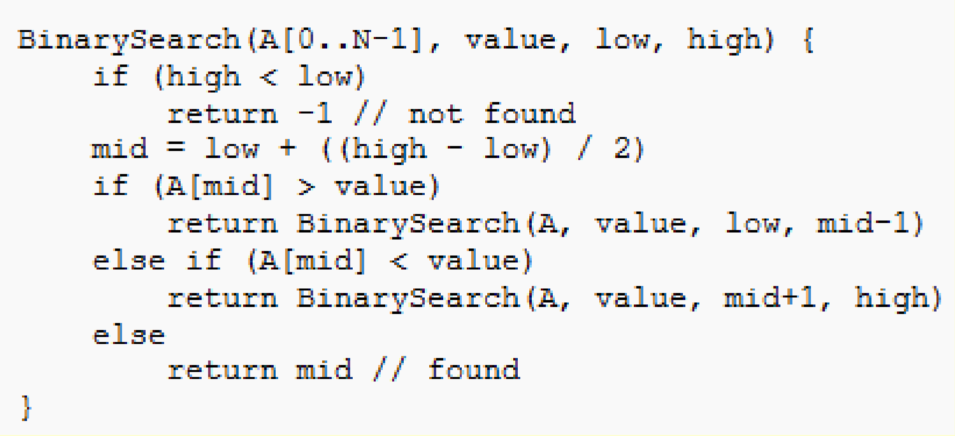
# Binary Search – Example of Applying the Master Theorem



## Binary search

The ultimate divide-and-conquer algorithm is, of course, *binary search:* to find a key $k$ in a large file containing keys $z[0,1,\ldots,n-1]$ in sorted order, we first compare $k$ with $z[n/2]$, and depending on the result we recurse either on the first half of the file, $z[0,\ldots,n/2-1]$, or on the second half, $z[n/2,\ldots,n-1]$. The recurrence now is $T(n) = T(\lceil n/2 \rceil) + O(1)$, which is the case $a = 1, b = 2, d = 0$. Plugging into our master theorem we get the familiar solution: a running time of just $O(\log n)$.





```
BinarySearch(A[0..N-1], value, low, high) {
    if (high < low)
        return -1 // not found
    mid = low + ((high - low) / 2)
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid-1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid // found
}
```

## Greedy Algorithms

Greedy algorithms are simple and straightforward. They are shortsighted in their approach in the sense that they make choices on the basis of information at hand without worrying about the effect these choices may have in the future. They are easy to invent, easy to implement and most of the time quite efficient. Many problems cannot be solved correctly by greedy approach. Greedy algorithms are often used to solve optimization problems

### Greedy Approach

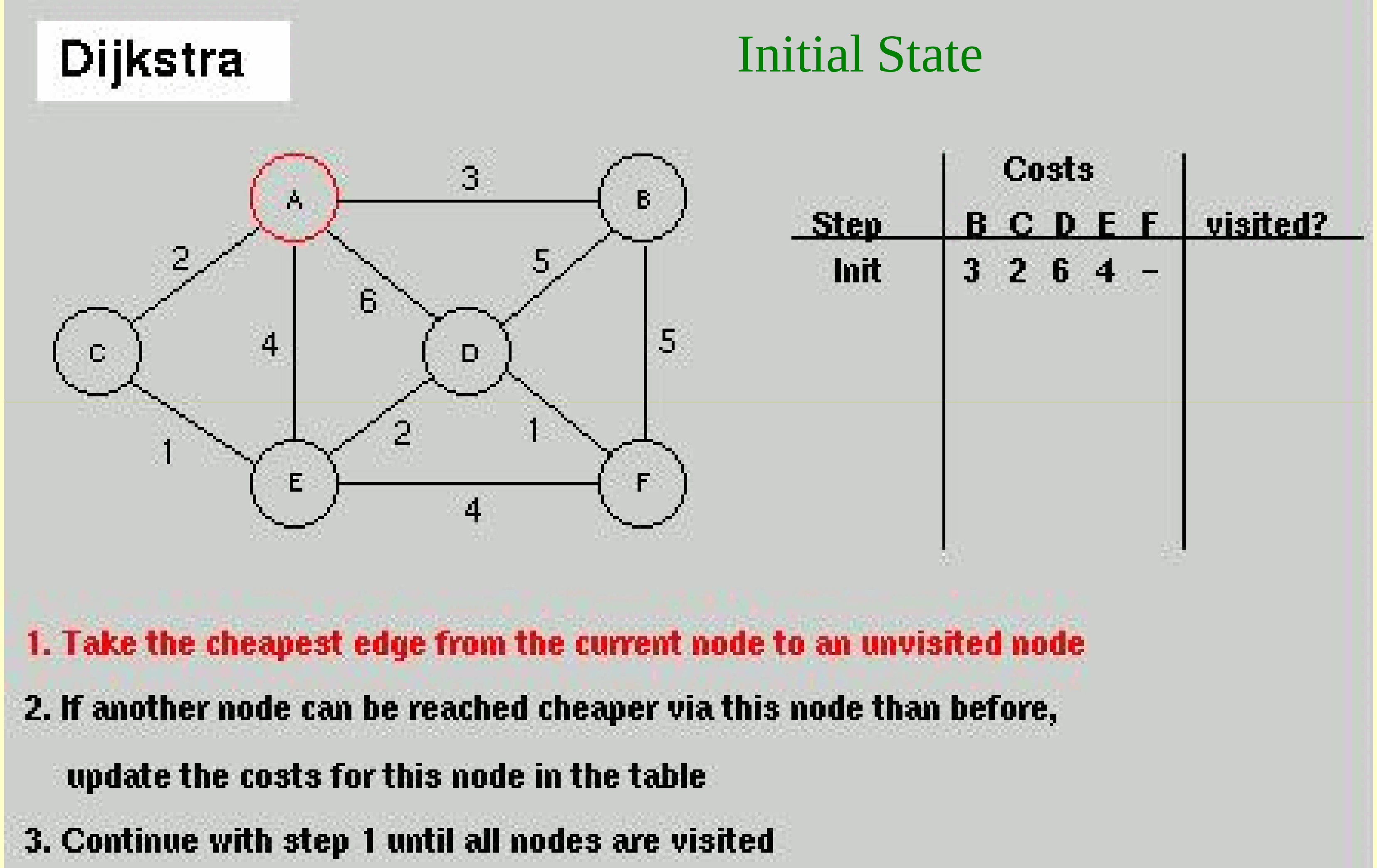
Greedy Algorithm works by making the choice that seems most promising at any moment; it never reconsiders this choice, whatever situation may arise later.

**Greedy-Choice Property:**

It says that a globally optimal solution can be arrived at by making a locally optimal choice.

**Complexity Analysis** – the analysis is usually specific to the algorithm (applying generic reasoning techniques)

# Dijkstra’s Shortest Path Algorithm is a typical example of **greediness**



Initial State

|  | Costs |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Step | B | C | D | E | F | visited? |
| Init | 3 | 2 | 6 | 4 | – |  |

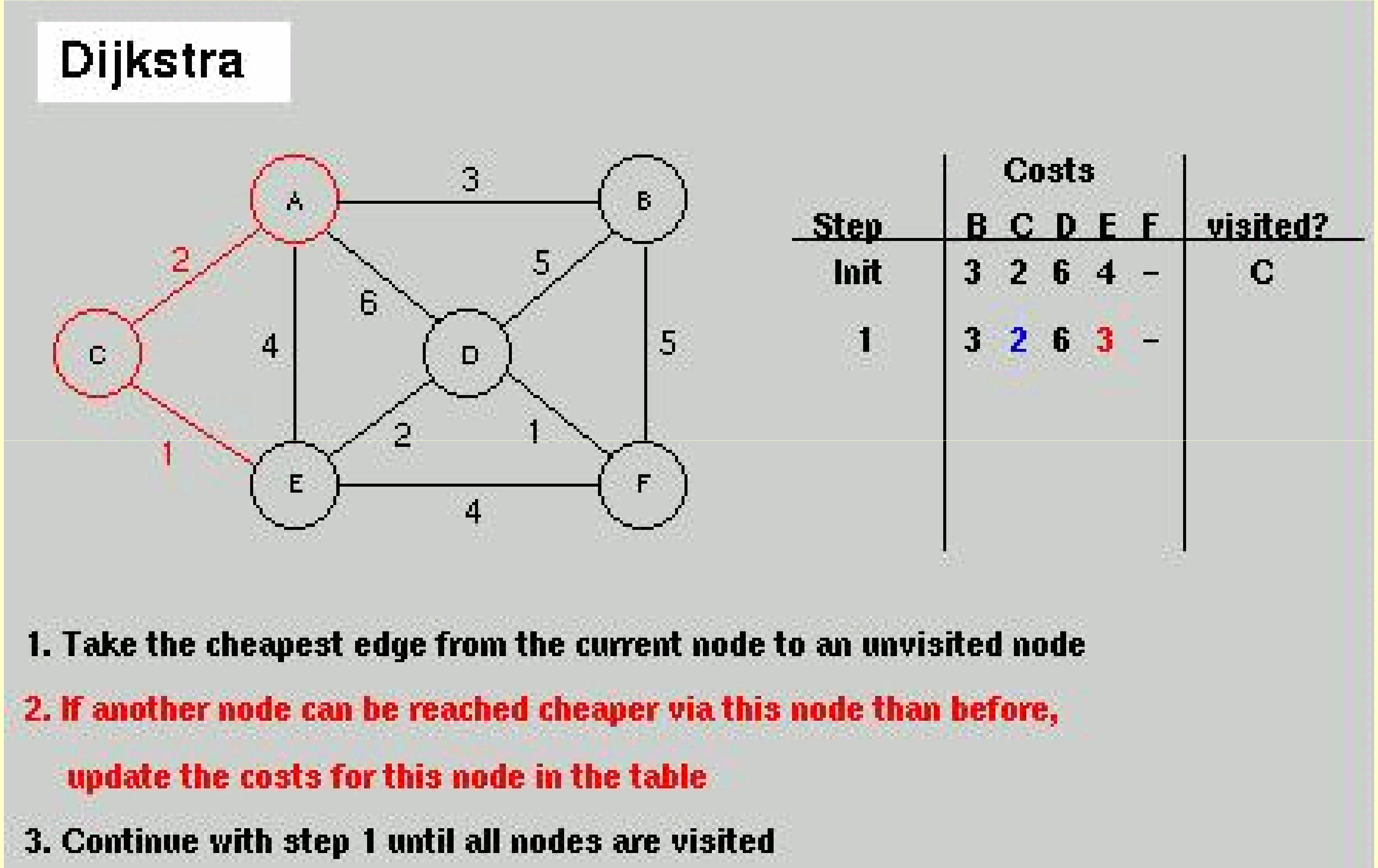
1. Take the cheapest edge from the current node to an unvisited node

2. If another node can be reached cheaper via this node than before,

   update the costs for this node in the table

3. Continue with step 1 until all nodes are visited

# Step 1



## Dijkstra

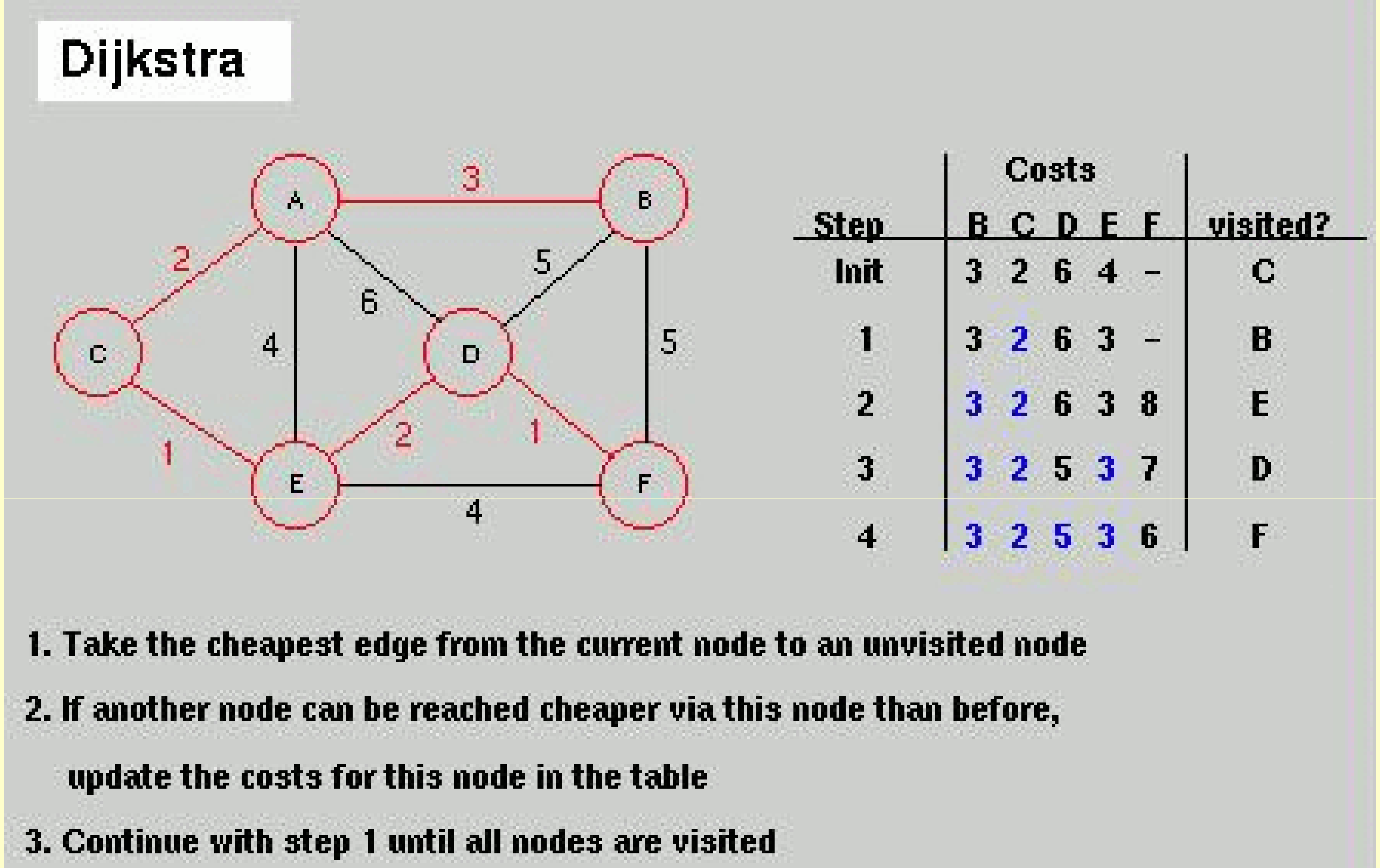
| Step | Costs |  |  |  |  | visited? |
|---|---|---|---|---|---|---|
|  | B | C | D | E | F |  |
| Init | 3 | 2 | 6 | 4 | – | C |
| 1 | 3 | 2 | 6 | 3 | – |  |

1. Take the cheapest edge from the current node to an unvisited node

2. If another node can be reached cheaper via this node than before, update the costs for this node in the table

3. Continue with step 1 until all nodes are visited

# Final Step

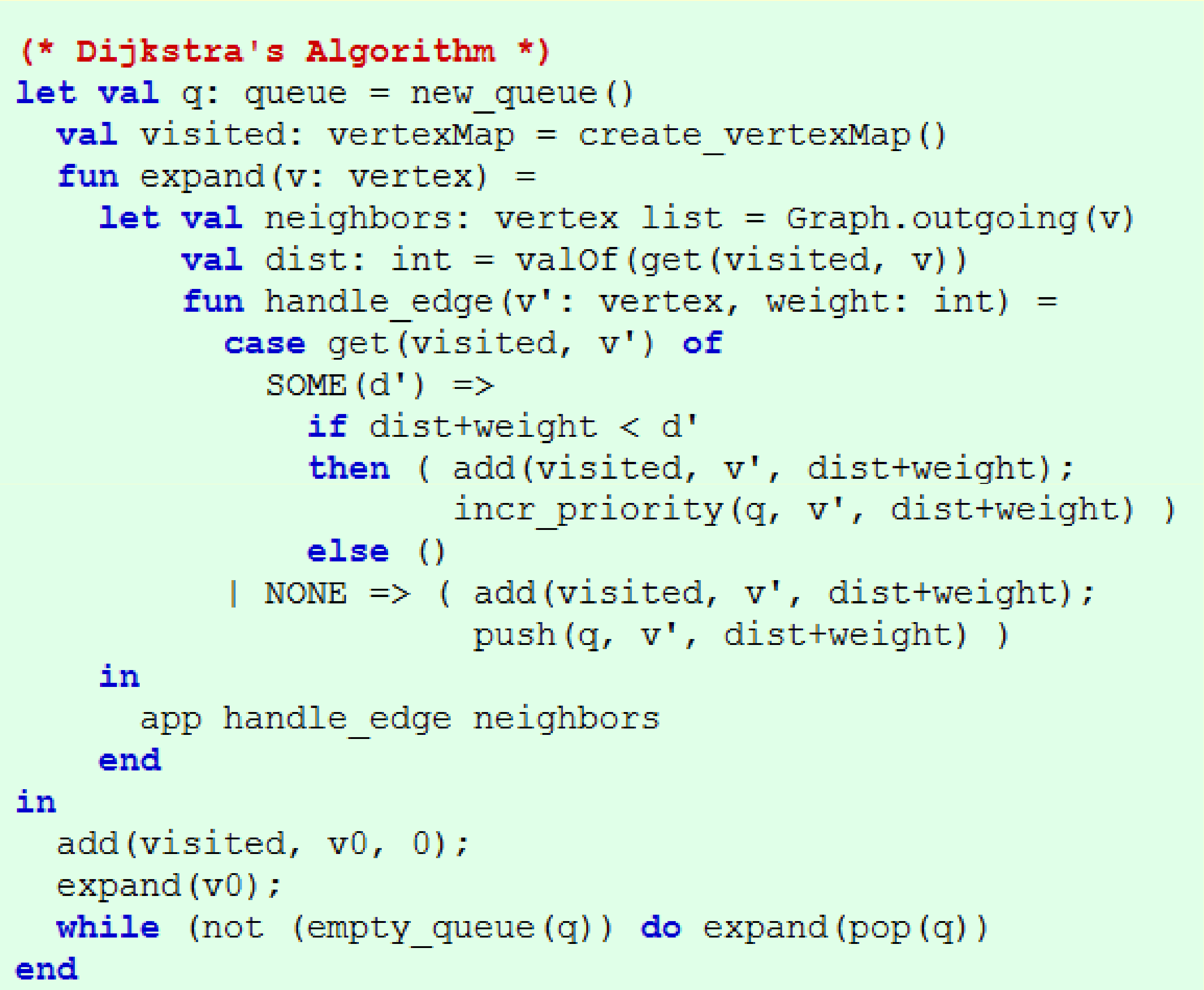


Dijkstra

| Step | Costs |  |  |  |  | visited? |
|---|---|---|---|---|---|---|
|  | B | C | D | E | F |  |
| Init | 3 | 2 | 6 | 4 | – | C |
| 1 | 3 | 2 | 6 | 3 | – | B |
| 2 | 3 | 2 | 6 | 3 | 8 | E |
| 3 | 3 | 2 | 5 | 3 | 7 | D |
| 4 | 3 | 2 | 5 | 3 | 6 | F |

1. Take the cheapest edge from the current node to an unvisited node
2. If another node can be reached cheaper via this node than before,
   update the costs for this node in the table
3. Continue with step 1 until all nodes are visited

# Shortest Path In A Graph – typical implementation



```
(* Dijkstra's Algorithm *)
let val q: queue = new_queue()
  val visited: vertexMap = create_vertexMap()
  fun expand(v: vertex) =
    let val neighbors: vertex list = Graph.outgoing(v)
        val dist: int = valOf(get(visited, v))
        fun handle_edge(v': vertex, weight: int) =
          case get(visited, v') of
            SOME(d') =>
              if dist+weight < d'
              then ( add(visited, v', dist+weight);
                     incr_priority(q, v', dist+weight) )
              else ()
          | NONE => ( add(visited, v', dist+weight);
                      push(q, v', dist+weight) )
    in
      app handle_edge neighbors
    end
in
  add(visited, v0, 0);
  expand(v0);
  while (not (empty_queue(q)) do expand(pop(q))
end
```

# Shortest Path Algorithm (Informal) Analysis

Every time the main loop executes, one vertex is extracted from the queue.

Assuming that there are *V* vertices in the graph, the queue may contain $O(V)$ vertices. Each pop operation takes $O(\lg V)$ time assuming the heap implementation of priority queues. So the total time required to execute the main loop itself is $O(V \lg V)$.

In addition, we must consider the time spent in the function expand, which applies the function handle_edge to each outgoing edge. Because expand is only called once per vertex, handle_edge is only called once per edge.

It might call push(v'), but there can be at most *V* such calls during the entire execution, so the total cost of that case arm is at most $O(V \lg V)$. The other case arm may be called $O(E)$ times, however, and each call to increase_priority takes $O(\lg V)$ time with the heap implementation.

Therefore the total run time is $O(V \lg V + E \lg V)$, which is $O(E \lg V)$ because *V* is $O(E)$ assuming a connected graph.

# Complexity: Some PBL

**Purse problem – bag of coins, required to pay an exact price**

The complexity analysis, is to be based on the fundamental operations of your chosen machine/language, used to implement the function Pay, specified below.

Input:
*Bag* of integer coins, Target integer Price
Output:
*empty bag* to signify that price cannot be paid exactly
or
A smallest bag of coins taken from the original bag and whose sum is equal to the price to be paid.

EG Pay ([1,1,2,3,3,4,18], 6) = [2,4] or [3,3]
Pay ([1,1,2,4,18], 15) = []

**TO DO:** Implement the function Pay, execute tests and analyze run times.

## Complexity Classes: P vs NP

The class **P** consists of all those *decision problems* that can be solved on a deterministic sequential machine – like a Turing machine, and a typical PC - in an amount of time that is polynomial in the size of the input

The class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine.

*NP does not stand for "non-polynomial"*. There are many complexity classes that are much harder than NP.

Arguably, the biggest open question in theoretical computer science concerns the relationship between those two classes:

Is **P** equal to **NP**?

We have already seen some problems in P so now let us consider NP

## Some well-known problems in NP

- k-clique: Given a graph, does it have a size k clique? (i.e. complete subgraph on k vertices)

- k-independent set: Given a graph, does it have a size k independent set? (i.e. k vertices with no edge between them)

- k-coloring: Given a graph, can the vertices be colored with k colors such that adjacent vertices get different colors?

- Satisfiability (SAT): Given a boolean expression, is there an assignment of truth values (T or F) to variables such that the expression is satisfied (evaluated to T)?

- Travel salesman problem (TSP): Given an edge-weighted complete graph and an integer x, is there a Hamiltonian cycle of length at most x?

- Hamiltonian Path (or Hamiltonian Cycle): Given a graph G does it have a Hamiltonian path? (i.e. a path that goes through every vertex exactly once)

# NP-Completeness

A problem *X* is *hard* for a class of problems *C* if every problem in *C* can be *reduced* to *X*

Forall  complexity classes C, if a problem X is in C and hard for C, then X is said to be *complete* for C

A problem is NP-complete if it is NP and no other NP problem is more than a *polynomial factor harder*.

Informally, a problem is NP-complete if answers can be verified quickly, and *a quick algorithm to solve this problem can be used to solve all other NP problems quickly.*

The class of NP-complete problems contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P.

Note: there are many other interesting complexity classes, but we do not have time to study them further

# Complexity Classes: NP-Complete

The concept of *NP-complete* was introduced in 1971 by Stephen Cook in *The complexity of theorem-proving procedures*, though the term *NP-complete* did not appear anywhere in his paper.

Lander (1975) - *On the structure of polynomial time reducibility* - showed the existence of problems outside P and NP-complete

# Complexity Analysis – Many Open Questions: Factoring Example

It is not known exactly which complexity classes contain the decision version of the integer factorization problem.

It is known to be in both NP and co-NP. This is because both YES and NO answers can be trivially verified given the prime factors

It is suspected to be outside of all three of the complexity classes P, NP-complete, and co-NP-complete.

Many people have tried to find classical polynomial-time algorithms for it and failed, and therefore it is widely suspected to be outside P.

In contrast, the decision problem "is N a composite number?" appears to be much easier than the problem of actually finding the factors of N. Specifically, the former can be solved in polynomial time (in the number n of digits of N).

In addition, there are a number of probabilistic algorithms that can test primality very quickly in practice if one is willing to accept the small possibility of error.

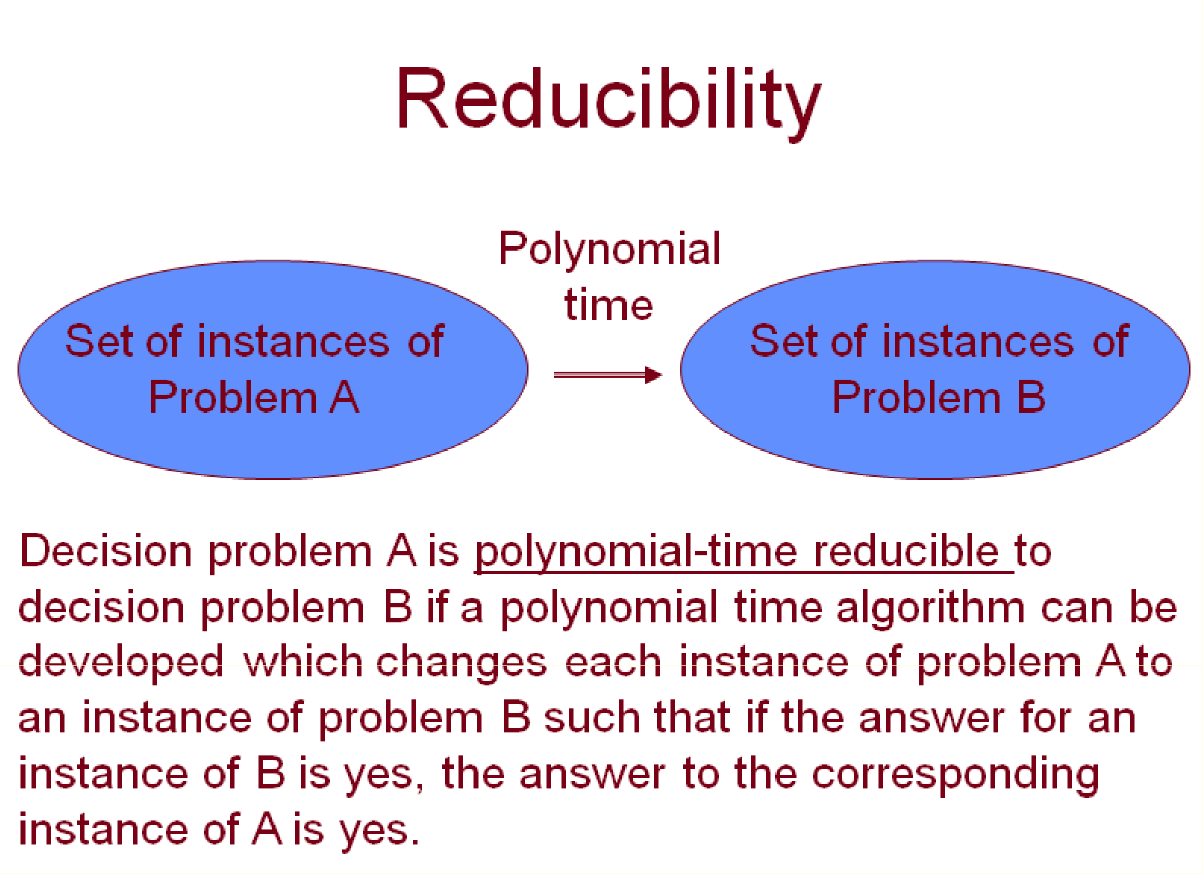
# Proving NP-Completeness of a problem – general approach

Approach:

- Restate problem as a decision problem
- Demonstrate that the decision problem is in the class NP
- Show that a problem known to be NP-Complete can be reduced to the current problem in polynomial time.

This technique for showing that a problem is in the class NP-Complete requires that we have one NP-Complete problem to begin with.

Boolean (circuit) satisifiability was the first problem to be shown to be in NP-Complete.



## Reducibility

Polynomial time

Set of instances of Problem A

Set of instances of Problem B

Decision problem A is polynomial-time reducible to decision problem B if a polynomial time algorithm can be developed which changes each instance of problem A to an instance of problem B such that if the answer for an instance of B is yes, the answer to the corresponding instance of A is yes.

In 1972, Richard Karp *published"Reducibility Among Combinatorial Problems*", showing – using this reduction technique starting on the satisifiability result - that many diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are NP-complete.