7. Sorting and Searching Algorithms

7.1. Efficiency of Algorithms

- Worst case efficiency

  is the *maximum* number of steps that an algorithm can take for *any* collection of data values.

- Best case efficiency

  is the *minimum* number of steps that an algorithm can take *any* collection of data values.

- Average case efficiency

  - the efficiency averaged on all possible inputs
  - must assume a distribution of the input
  - we normally assume uniform distribution (all keys are equally probable)

If the input has size *n*, efficiency will be a *function of n*

7.1. Efficiency of Algorithms contd…

- We are interested in analyzing the efficiency of an algorithm
  - involves determining the quantity of computer resources consumed by the algorithm
- These resources include
  - the amount of memory and
  - the amount of computational time
- The efficiency of a given algorithm is determined the resources required, as the size of the algorithm grows.

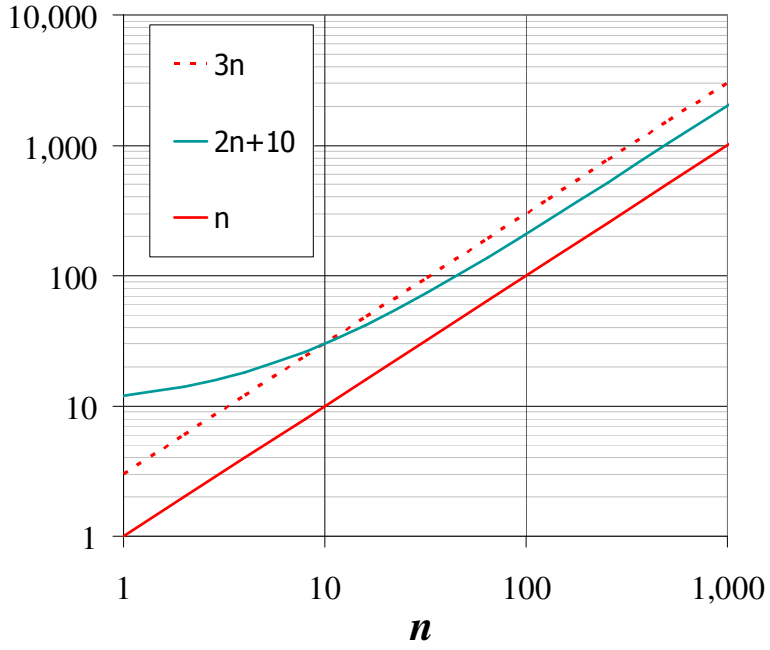
7.1.1. The Big-O Notation

- We can say that a function is “of the order of n", which can be written as O(n) to describe the upper bound on the number of operations
- This is called Big-Oh notation
- Some common orders are:
  - O(1) constant (the size of n has no effect)
  - O(log n) logarithmic
  - O(n log n)
  - O(n^2) quadratic
  - O(n^3) cubic
  - O(2^n) exponential

7.1.2. Formal Big-O Definition

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$f(n) \leq cg(n)$ for $n \geq n_0$



Example: $2n + 10$ is $O(n)$

$2n + 10 \leq cn$

$(c - 2)\, n \geq 10$

$n \geq 10/(c - 2)$

Pick $c = 3$ and $n0 = 10$

7.1.3. Big-O and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

7.1.4. Big-O Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

7.1.5 Examples

- We say that $n^4 + 100n^2 + 10n + 50$ is of the order of n^4 or $O(n^4)$
- We say that $10n^3 + 2n^2$ is $O(n^3)$
- We say that $n^3 - n^2$ is $O(n^3)$
- We say that 10 is $O(1)$,
- We say that 1273 is $O(1)$

7.1.6. Big-O Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

7.1.7. Relatives of Big-O

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$
- **little-oh**
  - f(n) is $o(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$
- **little-omega**
  - f(n) is $\omega(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

7.1.8. Average vs. Worst Case

- Algorithm may run faster on some inputs than it does on the others
- Average case refers to the running time of an algorithm as an *average taken over all inputs* of a same size
- Worst case refers to the running time of an algorithm as the *maximum taken over all inputs* of a same size

7.2. Searching Algorithms

- Searching algorithms are closely related to the concept of dictionaries.
- Dictionaries are data structures that support search, insert, and delete operations.
- One of the most effective representations is a hash table. Typically, a simple function is applied to the key to determine its place in the dictionary. Another efficient search algorithms on sorted tables is binary search.
- If the dictionary is not sorted then heuristic methods of dynamic reorganization of the dictionary are of great value. One of the simplest are cache-based methods: several recently used keys are stored a special data structure that permits fast search (for example is always sorted). For example keeping the last N recently found values at the top of the table (or list) dramatically improved performance. Other cache-based approach are also possible. In the simplest form the cache can be merged with the dictionary:
  - **move-to-front method** : A *heuristic* that moves the target of a *search* to the *head* of a *list* so it is found faster next time.
  - **transposition method** : Search an *array* or *list* by checking items one at a time. If the value is found, swap it with its predecessor so it is found faster next time.

7.2.1. Binary search trees

- **binary search tree** (**BST**) is a binary tree data structure which has the following properties:
  - each node (item in the tree) has a key value;
  - a total order (linear order) is defined on these key values;
  - the left subtree of a node contains only values less than the parent node's key value;
  - the right subtree of a node contains only values greater than or equal to the parent node's key value.
- The major advantage of binary search trees over the other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.
- Binary search trees can choose to allow or disallow duplicate values, depending on the implementation.
- Binary search trees are a fundamental data structures used to construct more abstract data structures such as sets, multisets, and associative arrays.

7.2.2. B-trees

- B-trees are multiway trees, commonly used in external storage, in which nodes correspond to blocks on the disk. As in other trees, the algorithms find their way down the tree, reading one block at each level. B-trees provide searching, insertion, and deletion of records in O(logN) time. This is quite fast and works even for very large files. However, the programming is not trivial.

7.2.3.1. **Breadth-first search**

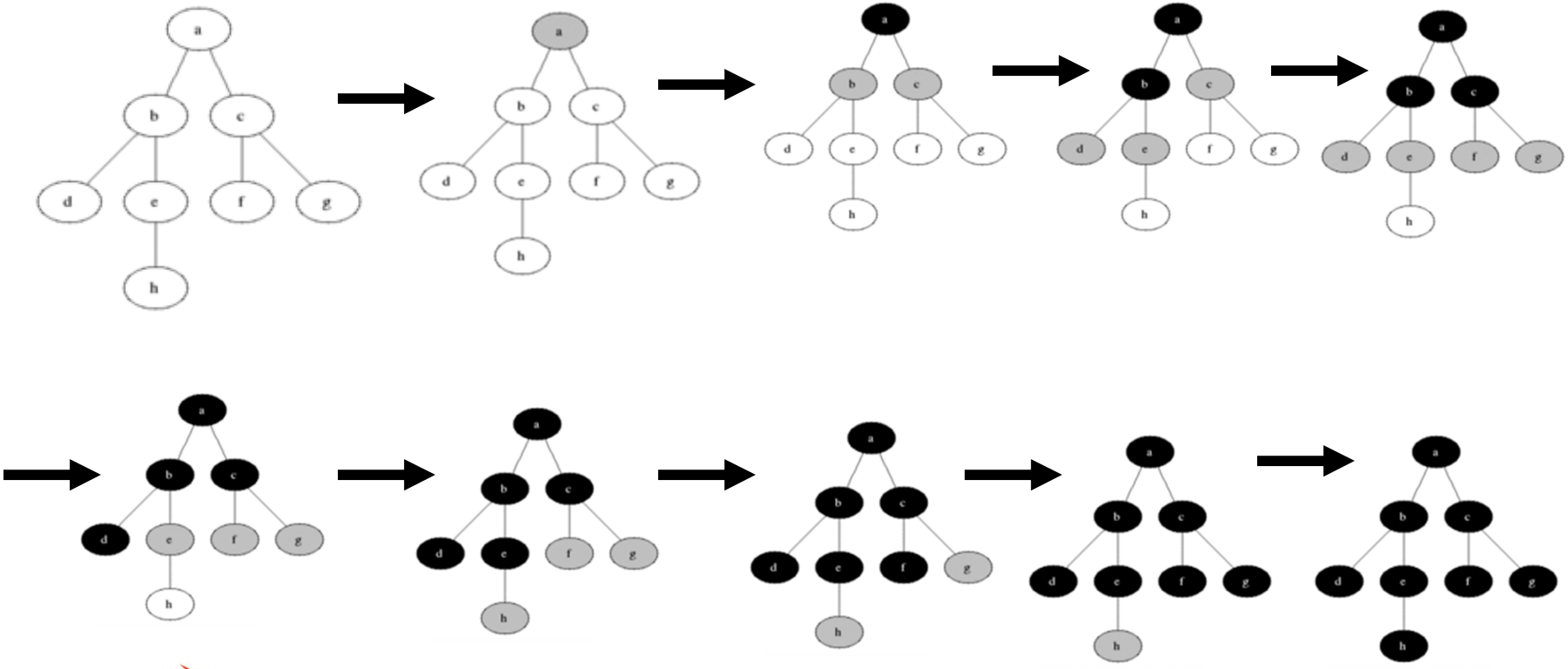
- **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

7.2.3.1. **Breadth-first search**

- **How it works**
  - BFS is an uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a heuristic.
  - From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

7.2.3.1. Breadth-first search

- **How it works**



7.2.3.1. **Breadth-first search**

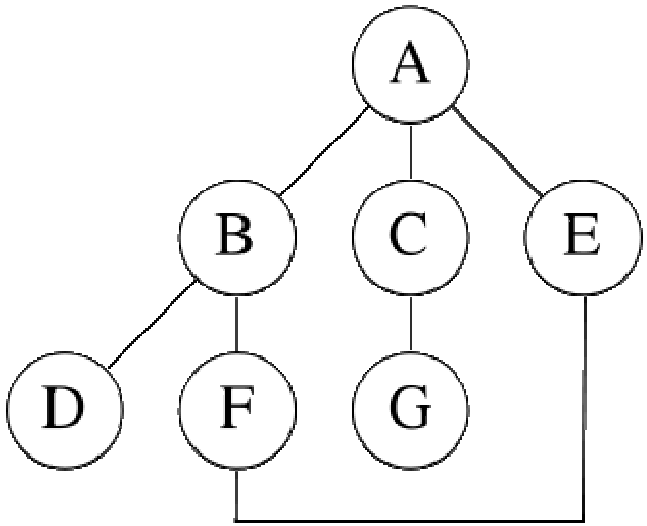
- **Applications of BFS**
  - Breadth-first search can be used to solve many problems in graph theory, for example:
  - Finding all connected components in a graph.
  - Finding all nodes within one connected component
  - Copying Collection, Cheney's algorithm
  - Finding the shortest path between two nodes *u* and *v* (in an unweighted graph)
  - Testing a graph for bipartiteness
  - (Reverse) Cuthill–McKee mesh numbering

7.2.3.2. **Depth-first search**

- **Depth-first search** (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.
- Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for exploration.

7.2.3.2. **Depth-first search**

- **How it works**



7.2.3.2. **Depth-first search**

- a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.
- Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.
- Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:
  - 0: A
  - 1: A (repeated), B, C, E
- (Note that iterative deepening has now seen C, when a conventional depth-first search did not.)
  - 2: A, B, D, F, C, G, E, F
- (Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)
  - 3: A, B, D, F, E, C, G, E, F, B
- For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

7.2.4. Java Implementations

Java implemetation – Depth-first search

```
class StackX
{
private final int SIZE = 20;
private int[] st;
private int top;
public StackX() // constructor
{
st = new int[SIZE]; // make array
top = -1;
}
public void push(int j) // put item on stack
{ st[++top] = j; }
public int pop() // take item off stack
{ return st[top--]; }
public int peek() // peek at top of stack
{ return st[top]; }
public boolean isEmpty() // true if nothing on stack
{ return (top == -1); }
} // end class StackX
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
class Vertex
{
public char label; // label (e.g. 'A')
public boolean wasVisited;
// ----------------
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
// ----------------
} // end class Vertex
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private StackX theStack;
// -----------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
theStack = new StackX();
} // end constructor
// -----------------
```

7.2.4. Java Implementations

```
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// ------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// ------------------
```

7.2.4. Java Implementations

```
public void dfs() // depth-first search
{ // begin at vertex 0
vertexList[0].wasVisited = true; // mark it
displayVertex(0); // display it
theStack.push(0); // push it
while( !theStack.isEmpty() ) // until stack empty,
{
// get an unvisited vertex adjacent to stack top
int v = getAdjUnvisitedVertex( theStack.peek() );
if(v == -1) // if no such vertex,
theStack.pop();
else // if it exists,
{
vertexList[v].wasVisited = true; // mark it
displayVertex(v); // display it
theStack.push(v); // push it
}
} // end while
// stack is empty, so we're done
for(int j=0; j<nVerts; j++) // reset flags
vertexList[j].wasVisited = false;
} // end dfs
// ----------------
```

7.2.4. Java Implementations

```
// returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v)
{
for(int j=0; j<nVerts; j++)
if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
return j;
return -1;
} // end getAdjUnvisitedVert()
// ------------------
} // end class Graph
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
class DFSApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for dfs)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1); // AB
theGraph.addEdge(1, 2); // BC
theGraph.addEdge(0, 3); // AD
theGraph.addEdge(3, 4); // DE
System.out.print("Visits: ");
theGraph.dfs(); // depth-first search
System.out.println();
} // end main()
} // end class DFSApp
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

- **Breath-first search**

```
class Queue
{
private final int SIZE = 20;
private int[] queArray;
private int front;
private int rear;
public Queue() // constructor
{
queArray = new int[SIZE];
front = 0;
rear = -1;
}
public void insert(int j) // put item at rear of queue
{
if(rear == SIZE-1)
rear = -1;
queArray[++rear] = j;
}
```

7.2.4. Java Implementations

```
public int remove() // take item from front of queue
{
int temp = queArray[front++];
if(front == SIZE)
front = 0;
return temp;
}
public boolean isEmpty() // true if queue is empty
{
return ( rear+1==front || (front+SIZE-1==rear) );
}
} // end class Queue
//////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
public boolean wasVisited;
// ------------------------------------------------------------
-
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
// ------------------------------------------------------------
-
} // end class Vertex
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private Queue theQueue;
// -----------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
theQueue = new Queue();
} // end constructor
// ------------------------------------------------------
```

7.2.4. Java Implementations

```
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
// ------------------------------------------------------------
-
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// ------------------------------------------------------------
-
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// ------------------------------------------------------------
```

7.2.4. Java Implementations

```
public void bfs() // breadth-first search
{ // begin at vertex 0
vertexList[0].wasVisited = true; // mark it
displayVertex(0); // display it
theQueue.insert(0); // insert at tail
int v2;
while( !theQueue.isEmpty() ) // until queue empty,
{
int v1 = theQueue.remove(); // remove vertex at head
// until it has no unvisited neighbors
while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
{ // get one,
vertexList[v2].wasVisited = true; // mark it
displayVertex(v2); // display it
theQueue.insert(v2); // insert it
} // end while
} // end while(queue not empty)
// queue is empty, so we're done
for(int j=0; j<nVerts; j++) // reset flags
vertexList[j].wasVisited = false;
} // end bfs()
// --------------------------------
```

7.2.4. Java Implementations

```
// returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v)
{
for(int j=0; j<nVerts; j++)
if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
return j;
return -1;
} // end getAdjUnvisitedVert()
// ------------------------------------------------------------
-
} // end class Graph
////////////////////////////////////////////////////////////////
```

7.2.4. Java Implementations

```
class BFSApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for dfs)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1); // AB
theGraph.addEdge(1, 2); // BC
theGraph.addEdge(0, 3); // AD
theGraph.addEdge(3, 4); // DE
System.out.print("Visits: ");
theGraph.bfs(); // breadth-first search
System.out.println();
} // end main()
} // end class BFSApp
```

7.3. Sorting algorithms

- a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:
  - The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
  - The output is a permutation, or reordering, of the input.

7.3.1. Insertion Sort

- Insertion Sort orders a list in the same way we would order a hand of playing cards.
- Compare the first two numbers, placing the smallest one in the first position.
- Compare the third number to the second number. If the third number is larger, then the first three numbers are in order. If not, then swap them. Now compare the numbers in positions one and two and swap them if necessary.
- Proceed in this manner until reaching the end of the list.

7.3.1.1. Analyzing Insertion Sort

- Using Insertion Sort, the number of comparisons on a list of size n varies, because, if the numbers are already in order, further comparisons are avoided. So we will find the average number of comparisons.

- Using limits and probability, we find that the average number of comparisons is $(n-1)(n/4) + k$, where k increases for larger values of n.

7.3.2. Selection Sort

- Scan the list and put the smallest number in the first position.
- Disregard the first position, which is now the smallest number, and put the second smallest number in the second position.
- Proceed in this manner until reaching the end of the list.

7.3.2. Selection Sort

Sorting Times
(by Selection Sort on a 386 microprocessor running at 20 mHz).

- Note that as the list size doubles, the time increases about four-fold.

- A little arithmetic shows that the time to sort 128,000 numbers would be over 12 hours!

List Size	Seconds
1000	2.69
2000	11.78
4000	47.51
8000	190.70

7.3.2.1. Analyzing Selection Sort

- We will assume that the computer spends time only on comparisons and swaps. However, since the average number of swaps is extremely difficult to compute, we will focus only on comparisons.

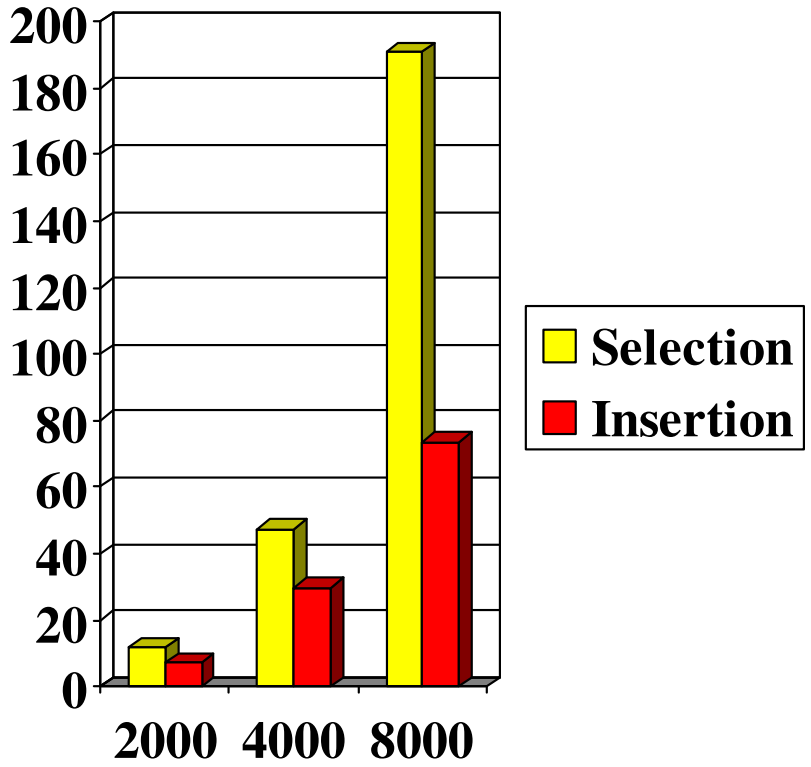
- On a list of size 4, we would compare the first position to the other 3 numbers, the second position to the 2 remaining numbers, and the third position to the 1 remaining number. This is a total of 3+2+1 = 6 comparisons.

- So, for a list of size n, there will be n-1 + n-2 + … + 2 + 1 comparisons. In other words, (n-1)(n/2) comparisons.

7.3.2.2. Selection Sort vs. Insertion Sort

(Time in Seconds on a 386 microprocessor running at 20 mHz).

- Insertion Sort is about twice as fast as Selection Sort.
- But note that as the list size doubles, the time for both sorts increases about four-fold.



7.3.2.3. The Order of an Algorithm

- To compare two algorithms, we will take the limit of the number of comparisons of the first algorithm divided by the number of comparisons for the second algorithm as n (the number of items in the list) approaches infinity.

- When taking the limit of (ln N) / N or other similar cases, L'Hopital's Rule becomes a handy tool.

- Two algorithms have the same **ORDER** if the limit is greater than zero and less than infinity. Otherwise, they have different orders.

7.3.2.4. Order of Insertion & Selection Sorts

- Comparing Insertion Sort to Selection Sort, we take the limit of [(n-1)(n/4) + k] / [(n-1)(n/2)] as n approaches infinity. As n grows larger, k does not grow fast enough to affect the limit, and so the limit = 1/2.

- Since the fastest growing term in both Insertion & Selection Sort is n squared, we say they both have order n squared.
- Is there a sorting algorithm with a smaller (and thus faster) order?

7.3.3. Bubble sort

- *Bubble sort* is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. While simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes. Its average case and worst case are both O(n^2).

7.3.4. Shell sort

- *Shell sort* was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements (sets with less than 1000 or so elements). Another advantage of this algorithm is that it requires relatively small amounts of memory.

7.3.5. Merge sort

- *Merge sort* takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(*n* log *n*).

7.3.6. Quicksort

- Choose an element out of the list as a pivot. A good process to select a pivot is to compare the first, middle, and last elements and choose the middle value.
- Compare every other element in the list to the pivot and create two lists, one list where every element is smaller than the pivot and one where every element is larger.
- Now split each of these lists into smaller lists.
- Continue in this way until the small lists have only one or two elements and we can sort them with at most one comparison each.

7.3.6.1. Selection, Insertion, & Quicksort

(Times in seconds on a 386 microprocessor running at 20 mHz).

List Size	Selection	Insertion	Quicksort
1000	2.69	1.73	0.11
2000	11.78	7.46	0.22
3000	47.51	29.98	0.44
4000	190.70	73.47	0.96

71.3.7. Heap sort

- *Heapsort* is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes *O(log n)* time, instead of *O(n)* for a linear scan as in simple selection sort. This allows Heapsort to run in *O(n log n)* time.

7.3.8. Radix sort

- *Radix sort* is an algorithm that sorts a list of fixed-size numbers of length k in $O(n \cdot k)$ time by treating them as bit strings. We first sort the list by the least significant bit while preserving their relative order using a stable sort. Then we sort them by the next bit, and so on from right to left, and the list will end up sorted. Most often, the counting sort algorithm is used to accomplish the bitwise sorting, since the number of values a bit can have is small.

7.3.9. Java Implementation

- **Bubble sort**

```
void bubbleSort(int a[], int n)
    /* Sorts in increasing order the array A of the size N */
    {
      int k;
      int bound = n-1;
      int t;
      int last_swap;

      while (bound) {
        last_swap = 0;
        for ( k=0; k<bound; k++ )
           t = a[k]; /* t is a maximum of A[0]..A[k] */
           if ( t > a[k+1] ) {
             a[k] = a[k+1]; a[k+1] = t; /*swap*/
             last_swap = k; /* mark the last swap position */
           }//if
        }//for
        bound=last_swap; /*elements after bound already sorted */
      }//while
    } // bubbleSort
```